Spis treści

Rozdział 1

R0: Wstęp

1.1 Cel

Celem tego kursu jest zapoznanie czytelnika z kilkoma rzeczami:

- programowaniem funkcyjnym w duchu Haskella i rodziny ML, przeciwstawionym programowaniu imperatywnemu
- dowodzeniem twierdzeń, które jest:
  - formalne, gdzie "formalny" znaczy "zweryfikowany przez komputer"
  - interaktywne, czyli umożliwiające dowolne wykonywanie i cofanie kroków dowodu oraz sprawdzenie jego stanu po każdym kroku
  - (pół)automatyczne, czyli takie, w którym komputer może wyręczyć użytkownika w wykonywaniu trywialnych i żmudnych, ale koniecznych kroków dowodu
- matematyką opartą na logice konstruktywnej, teorii typów i teorii kategorii oraz na ich zastosowaniach do dowodzenia poprawności programów funkcyjnych i w szeroko pojętej informatyce

W tym krótkim wstępie postaramy się spojrzeć na powyższe cele z perspektywy historycznej, a nie dydaktycznej. Nie przejmuj się zatem, jeżeli nie rozumiesz jakiegoś pojęcia lub terminu — czas na dogłębne wyjaśnienia przyjdzie w kolejnych rozdziałach.

1.2 Wybór

Istnieje wiele środków, które pozwoliłyby nam osiągnąć postawione cele, a jako że nie sposób poznać ich wszystkich, musimy dokonać wyboru.

Wśród dostępnych języków programowania jest wymieniony już Haskell, ale nie pozwala on na dowodzenie twierdzeń (a poza tym jest sprzeczny, jeżeli zinterpretujemy go jako system logiczny), a także jego silniejsze potomstwo, jak Idris czy Agda, w których możemy dowodzić, ale ich wsparcie dla interaktywności oraz automatyzacji jest marne.

Wśród asystentów dowodzenia (ang. proof assistants) mamy do wyboru takich zawodników, jak polski system Mizar, który nie jest jednak oparty na teorii typów, Lean, który niestety jest jeszcze w fazie rozwoju, oraz Coq. Nasz wybór padnie właśnie na ten ostatni język.

1.3 Programowanie i dowodzenie

1.3.1 Alan Turing i jego maszyna

Teoretyczna nauka o obliczeniach powstała niedługo przed wynalezieniem pierwszych komputerów. Od samego początku definicji obliczalności oraz modeli obliczeń było wiele. Choć pokazano później, że wszystkie są równoważne, z konkurencji między nimi wyłonił się niekwestionowany zwycięzca — maszyna Turinga, wynaleziona przez Alana... (zgadnij jak miał na nazwisko).

Maszyna Turinga nazywa się maszyną nieprzypadkowo — jest mocno "hardware'owym" modelem obliczeń. Idea jest dość prosta: maszyna ma nieskończenie długą taśmę, przy pomocy której może odczytywać i zapisywać symbole oraz manipulować nimi według pewnych reguł.

W czasach pierwszych komputerów taki "sprzętowy" sposób myślenia przeważył i wyznaczył kierunek rozwoju języków programowania, który dominuje do dziś. Kierunek ten jest imperatywny; program to w jego wyobrażeniu ciąg instrukcji, których rolą jest zmiana obecnego stanu pamięci na inny.

Ten styl programowania sprawdził się w tym sensie, że istnieje na świecie cała masa różnych systemów informatycznych zaprogramowanych w językach imperatywnych, które jakoś działają... Nie jest on jednak doskonały. Wprost przeciwnie — jest:

- trudny w analizie (trudno przewidzieć, co robi program, jeżeli na jego zachowanie wpływ ma cały stan programu)
- trudny w urównoleglaniu (trudno wykonywać jednocześnie różne części programu, jeżeli wszystkie mogą modyfikować wspólny globalny stan)

1.3.2 Alonzo Church i rachunek

Innym modelem obliczeń, nieco bardziej abstrakcyjnym czy też "software'owym" jest rachunek , wymyślony przez Alonzo Churcha. Nie stał się tak wpływowy jak maszyny Turinga, mimo że jest równie prosty — opiera się jedynie na dwóch operacjach:

- -abstrakcji, czyli związaniu zmiennej wolnej w wyrażeniu, co czyni z niego funkcję
- aplikacji funkcji do argumentu, która jest realizowana przez podstawienie argumentu za zmienną związaną

Nie bój się, jeśli nie rozumiesz; jestem marnym bajkopisarzem i postaram się wyjaśnić wszystko później, przy użyciu odpowiednich przykładów.

Oryginalny rachunek nie był typowany, tzn. każdą funkcję można "wywołać" z każdym argumentem, co może prowadzić do bezsensownych pomyłek. Jakiś czas później wymyślono typowany rachunek , w którym każdy term (wyrażenie) miał swój "typ", czyli metkę, która mówiła, jakiego jest rodzaju (liczba, funkcja etc.).

Następnie odkryto, że przy pomocy typowanego rachunku można wyrazić intuicjonistyczny rachunek zdań oraz reprezentować dowody przeprowadzone przy użyciu dedukcji naturalnej. Tak narodziła się "korespondencja Curry'ego-Howarda", która stwierdza między innymi, że pewne systemy logiczne odpowiadają pewnym rodzajom typowanego rachunku , że zdania logiczne odpowiadają typom, a dowody — programom.

1.3.3 Martin-Löf, Coquand, CoC, CIC i Coq

Kolejnego kroku dokonał Jean-Yves Girard, tworząc System F — typowany, polimorficzny rachunek , który umożliwia reprezentację funkcji generycznych, działających na argumentach dowolnego typu w ten sam sposób (przykładem niech będzie funkcja identycznościowa). System ten został również odkryty niezależnie przez Johna Reynoldsa.

Następna gałąź badań, która przyczyniła się do obecnego kształtu języka Coq, została zapoczątkowana przez szwedzkiego matematyka imieniem Per Martin-Löf. W swojej intuicjonistycznej teorii typów (blisko spokrewnionej z rachunkiem) wprowadził on pojęcie typu zależnego. Typy zależne, jak się okazało, odpowiadają intuicjonistycznemu rachunkowi predykatów — i tak korespondencja Curry'ego-Howarda rozrastała się...

Innymi rozszerzeniami typowanego rachunku były operatory typów (ang. type operators), czyli funkcje biorące i zwracające typy. Te trzy ścieżki rozwoju (polimorfizm, operatory typów i typy zależne) połączył w rachunku konstrukcji (ang. Calculus of Constructions, w skrócie CoC) Thierry Coquand, jeden z twórców języka Coq, którego pierwsza wersja była oparta właśnie o rachunek konstrukcji.

Zwieńczeniem tej ścieżki rozwoju były typy induktywne, również obecne w teorii typów Martina-Löfa. Połączenie rachunku konstrukcji i typów induktywnych dało rachunek induktywnych konstrukcji (ang. Calculus of Inductive Constructions, w skrócie CIC), który jest obecną podstawą teoretyczną języka Coq (po drobnych rozszerzeniach, takich jak dodanie typów koinduktywnych oraz hierarchii uniwersów, również pożyczonej od Martina-Löfa).

1.4 Filozofia i matematyka

1.4.1 Konstruktywizm

Po co to wszystko, zapytasz? Czy te rzeczy istnieją tylko dlatego, że kilku dziwnym ludziom się nudziło? Nie do końca. Przyjrzyjmy się pewnemu wesołemu twierdzeniu i jego smutnemu dowodowi.

Twierdzenie: istnieją takie dwie liczby niewymierne a i b, że a ˆ b (a podniesione do potęgi b) jest liczbą wymierną.

Dowód: jeżeli $\sqrt{2}^{\sqrt{2}} jestniewymierny, toniecha = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2}. Wtedy a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2} * \sqrt{}}$

Fajny dowód, co? To teraz dam ci zagadkę: podaj mi dwie niewymierne liczby a i b takie, że a ˆ b jest wymierne. Pewnie zerkasz teraz do dowodu, ale zaraz... cóż to? Jak to możliwe, że ten wredny dowód udowadnia istnienie takich liczb, mimo że nie mówi wprost, co to za liczby?

Tym właśnie jest niekonstruktywizm - możesz pokazać, że coś istnieje, ale bez wskazywania konkretnego obiektu. Możesz np. pokazać, że równanie ma rozwiązanie i wciąż nie wiedzieć, co to za rozwiązanie. Niewesoło, prawda?

Podobnego zdania był dawno temu holenderski matematyk L. E. J. Brouwer. Obraził się on więc na tego typu dowody i postanowił zrobić swoją własną logikę i oprzeć na niej swoją własną, lepszą matematykę. Powstała w ten sposób logika konstruktywna okazała się być mniej więcej tym samym, co wspomniany wyżej rachunek lambda, choć Brouwer jeszcze o tym nie wiedział.

1.4.2 Praktyka

W międzyczasie na osiągnięciach wymienionych wyżej panów zaczęto budować wieżę z kości słoniowej. Chociaż nigdy nie dosięgnie ona nieba (można pokazać, że niektóre problemy są niemożliwe do rozwiązania matematycznie ani za pomocą komputerów), to po jakimś czasie zaczęła być przydatna.

W połowie XIX wieku postawiono problem, który można krótko podsumować tak: czy każdą mapę polityczną świata da się pomalować czterema kolorami w taki sposób, aby sąsiednie kraje miały inne kolory?

Przez bardzo długi czas próbowano go rozwiązywać na różne sposoby, ale wszystkie one zawodziły. Po ponad stu latach prób problem rozwiązali Appel i Haken pokazując, że każdą mapę da się pomalować czterema kolorami. Popełnili oni jednak grzech bardzo ciężki, gdyż użyli do tego komputerów.

Programy, które napisali, by udowodnić twierdzenie, wiele razy okazały się błędne i musiały być wielokrotnie poprawiane. Sprawiło to, że część matematyków nie uznała ich dowodu, gdyż nie umieli oni ręcznie sprawdzić poprawności wszystkich tych pomocniczych programów.

Po upływie kolejnych 30 lat dowód udało się sformalizować w Coqu, co ostatecznio zamknęło sprawę. Morał płynący z tej historii jest dość prosty:

- niektóre twierdzenia można udowodnić jedynie sprawdzając dużą ilość przypadków, co jest trudne dla ludzi
- można przy dowodzeniu korzystać z komputerów i nie musi to wcale podważać wiary w słuszność dowodu, a może ją wręcz wzmocnić

1.4.3 Homofobia... ekhm, homotopia, czyli quo vadimus?

To jednak nie koniec niebezpiecznych związków matematyków z komputerami.

Nie tak dawno temu w odległej galaktyce (a konkretniej w Rosji) był sobie matematyk nazwiskiem Voevodsky.

TODO

1.5 Literatura

1.5.1 Książki

Mimo, iż Coq liczy sobie dobre 27 lat, książek na jego temat zaczęło przybywać dopiero od kilku. Z dostępnych pozycji polecenia godne są:

- Software Foundations — trzytomowa seria dostępna za darmo tutaj: https://softwarefoundations.cis.u
W jej skład wchodzą:
  - Logical Foundations, której głównym autorem jest Benjamin Pierce — bardzo przystępne acz niekompletne wprowadzenie do Coqa. Omawia podstawy programowania funkcyjnego, rekursję i indukcję strukturalną, polimorfizm, podstawy logiki i prostą automatyzację.
  - Programming Language Foundations, której głównym autorem jest Benjamin Pierce — wprowadzenie do teorii języków programowania. Omawia definiowanie ich składni i semantyki, dowodzenie ich własności oraz podstawy systemów typów i proste optymalizacje. Zawiera też kilka rozdziałów na temat bardziej zaawansowanej automatyzacji.
  - Verified Functional Algorithms, której autorem jest Andrew Appel — jak sama nazwa wskazuje skupia się ona na algorytmach, adaptowaniu ich do realiów języków funkcyjnych oraz weryfikacją poprawności ich działania. Nie jest ona jeszcze dopracowana, ale pewnie zmieni się to w przyszłości.
- Coq'Art, której autorami są Yves Bertot oraz Pierre Castéran — książka nieco szerzej opisująca język Coq, poświęca sporo miejsca rachunkowi konstrukcji i aspektom teoretycznym. Zawiera także rozdziały dotyczące automatyzacji, silnej specyfikacji, koindukcji, zaawansowanej rekurencji i refleksji. Wersja francuska jest dostępna za darmo pod adresem https://www.labri.fr/perso/casteran/CoqArt/ Wersję angielską można za darmo pobrać z rosyjskich stron z książkami, ale broń Boże tego nie rób! Piractwo to grzech.
- Certified Programming with Dependent Types autorstwa Adama Chlipali — książka dla zaawansowanych, traktująca o praktycznym użyciu typów zależnych oraz kładąca bardzo mocny nacisk na automatyzację, dostępna za darmo tu: adam.chlipala.net/cpdt

- Mathematical Components Book, dostępna za darmo tutaj: https://math-comp.github.io/mcb/book. to książka dotycząca biblioteki o nazwie Mathematical Components. Zawiera ona wprowadzenia do Coqa, ale poza tym opisuje też dwie inne rzeczy:
  - Metodologię dowodzenie zwaną *small scale reflection* (ang. reflekcja na małą skalę), która pozwala wykorzystać w dowodach maksimum możliwości obliczeniowych Coqa, a dzięki temu uprościć dowody i zorganizować twierdzenia w logiczny sposób
  - Język taktyk Ssreflect, którego bazą jest `Ltac`, a który wprowadza w stosunku do niego wiele ulepszeń i udogodnień, umożliwiając między innymi sprawne zastosowanie metodologii *small scale reflection* w praktyce
- Manual, dostępny pod adresem https://coq.inria.fr/refman/, nie jest wprawdzie zbyt przyjazny do czytania ciurkiem, ale można tu znaleźć wiele wartościowych informacji. Gdyby ktoś jednak pokusił się o przeczytanie go od deski do deski, polecam następującą kolejności rozdziałów: 4 -> (5) -> 1 -> 2 -> 17 -> 29 -> 13 -> 12 -> (3) -> (6) -> 7 -> 8 -> 9 -> 10 -> 21 -> 22 -> 25 -> 26 -> 27 -> 18 -> 19 -> 20 -> 24 -> 23 -> (11) -> (14) -> (15) -> (16) -> (28) -> (30), gdzie nawiasy okrągłe oznaczają rozdziały opcjonalne (niezbyt ciekawe lub nieprzydatne)
- Formal Reasoning About Programs — powstająca książka Adama Chlipali. Nie wiem o czym jest i nie polecam czytać dopóki jest oznaczona jako draft. Dostępna tu: http://adam.chlipala.net/frap/

Zalecana kolejność czytania: SF, część 1 -> (Coq'Art) -> (MCB) -> SF, część 2 i 3 -> CPDT -> Manual

1.5.2 Blogi

W Internecie można też dokopać się do blogów, na których przynajmniej część postów dotyczy Coqa. Póki co nie miałem czasu wszystkich przeczytać i wobec tego większość linków wrzucam w ciemno:

- http://www.cis.upenn.edu/~aarthur/poleiro/ (znajdziesz tu posty na temat parsowania, kombinatorycznej teorii gier, czytelnego strukturyzowania dowodu, unikania automatycznego generowania nazw, przeszukiwania, algorytmów sortowania oraz dowodzenia przez reflekcję).
- http://coq-blog.clarus.me/
- https://gmalecha.github.io/
- http://seb.mondet.org/blog/index.html (znajdziesz tu 3 posty na temat silnych specyfikacji)

- http://gallium.inria.fr/blog/ (znajdziesz tu posty na temat mechanizmu ewaluacji, inwersji, weryfikacji parserów oraz pisania pluginów do Coqa; większość materiału jest już dość leciwa)
- http://ilyasergey.net/pnp/
- https://homes.cs.washington.edu/~jrw12/#blog
- http://osa1.net/tags/coq
- http://coqhott.gforge.inria.fr/blog/

1.5.3 Inne

Coq ma też swój subreddit na Reddicie (można tu znaleźć różne rzeczy, w tym linki do prac naukowych) oraz tag na StackOverflow, gdzie można zadawać i odpowiadać na pytania:

- https://www.reddit.com/r/Coq/
- https://stackoverflow.com/questions/tagged/coq

1.6 Sprawy techniczne

Kurs ten tworzę z myślą o osobach, które potrafią programować w jakimś języku imperatywnym oraz znają podstawy logiki klasycznej, ale będę się starał uczynić go jak najbardziej zrozumiałym dla każdego. Polecam nie folgować sobie i wykonywać wszystkie ćwiczenia w miarę czytania, a cały kod koniecznie przepisywać ręcznie, bez kopiowania i wklejania. Poza ćwiczeniami składającymi się z pojedynczych twierdzeń powinny się też pojawić miniprojekty, które będą polegać na formalizacji jakiejś drobnej teorii lub zastosowaniu nabytej wiedzy do rozwiązania jakiegoś typowego problemu.

Język Coq można pobrać z jego strony domowej: coq.inria.fr

Z tej samej strony można pobrać CoqIDE, darmowe IDE stworzone specjalnie dla języka Coq. Wprawdzie z Coqa można korzystać w konsoli lub przy użyciu edytora Proof General, zintegrowanego z Emacsem, ale w dalszej części tekstu będę zakładał, że użytkownik korzysta właśnie z CoqIDE.

Gdyby ktoś miał problemy z CoqIDE, lekką alternatywą jest ProofWeb: http://proofweb.cs.ru.nl/index.

Uwaga: kurs powstaje w czasie rzeczywistym, więc w niektórych miejscach możesz natknąć się na znacznik TODO, który informuje, że dany fragment nie został jeszcze skończony.

Rozdział 2

R1: Logika

Naszą przygodę z Coqiem rozpoczniemy od skoku na głęboką wodę, czyli nauki dowodzenia twierdzeń w logice konstruktywnej przy pomocy taktyk. Powiemy sobie także co nieco o automatyzacji i cechach różniących logikę konstruktywną od klasycznej oraz dowiemy się, czym jest dedukcja naturalna.

Coq składa się w zasadzie z trzech języków:

- język termów nazywa się Gallina. Służy do pisania programów oraz podawania twierdzeń
- język komend nazywa się vernacular ("potoczny"). Służy do interakcji z Coqiem, takich jak np. wyszukanie wszystkich obiektów związanych z podaną nazwą
- język taktyk nazywa się Ltac. Służy do dowodzenia twierdzeń.

2.1 Typy i termy

`Section` *constructive_propositional_logic*.

Mechanizm sekcji nie będzie nas na razie interesował. Użyjemy go, żeby nie zaśmiecać głównej przestrzeni nazw.

`Hypothesis` P Q R : `Prop`.

Zapis $x : A$ oznacza, że term x jest typu A. `Prop` to typ zdań logicznych, więc komendę tę można odczytać następująco: niech P, Q i R będą zdaniami logicznymi.

Czym są termy? Są to twory o naturze syntaktycznej (składniowej), reprezentujące funkcje, typy, zdania logiczne, predykaty, relacje etc. Polskim słowem o najbliższym znaczeniu jest słowo "wyrażenie". Zamiast prób definiowania termów, co byłoby problematyczne, zobaczmy przykłady:

- 2 — stałe są termami
- P — zmienne są termami

- `Prop` — typy są termami
- `fun` $x : nat \Rightarrow x + 2$ — -abstrakcje (funkcje) są termami
- $f\ x$ — aplikacje funkcji do argumentu są termami
- `if` *true* `then` 5 `else` 2 — konstrukcja if-then-else jest termem

Nie są to wszystkie występujące w Coqu rodzaje termów — jest ich nieco więcej.

Kolejnym fundamentalnym pojęciem jest pojęcie typu. W Coqu każdy term ma dokładnie jeden, niezmienny typ. Czym są typy? Intuicyjnie można powiedzieć, że typ to rodzaj metki, która dostarcza nam informacji dotyczących danego termu.

Dla przykładu, stwierdzenie $x : nat$ informuje nas, że x jest liczbą naturalną, dzięki czemu wiemy, że możemy użyć go jako argumentu dodawania: term $x + 1$ jest poprawnie typowany (ang. well-typed), tzn. $x + 1 : nat$, a więc możemy skonkludować, że $x + 1$ również jest liczbą naturalną.

Innym przykładem niech będzie stwierdzenie $f : nat \to nat$, które mówi nam, że f jest funkcją, która bierze liczbę naturalną i zwraca liczbę naturalną. Dzięki temu wiemy, że term $f\ 2$ jest poprawnie typowany i jest liczbą naturalną, tzn. $f\ 2 : nat$, zaś term $f\ f$ nie jest poprawnie typowany, a więc próba jego użycia, a nawet napisania byłaby błędem.

Typy są tworami absolutnie kluczowymi. Informują nas, z jakimi obiektami mamy do czynienia i co możemy z nimi zrobić, a Coq pilnuje ścisłego przestrzegania tych reguł. Dzięki temu wykluczona zostaje możliwość popełnienia całej gamy różnych błędów, które występują w językach nietypowanych, takich jak dodanie liczby do ciągu znaków.

Co więcej, system typów Coqa jest jednym z najsilniejszych, jakie dotychczas wymyślono, dzięki czemu umożliwia nam wiele rzeczy, których prawie żaden inny język programowania nie potrafi, jak np. reprezentowanie skomplikowanych obiektów matematycznych i dowodzenie twierdzeń.

```
Check 2.
(*  ===> 2 : nat *)

Check P.
(*  ===> P : Prop *)
```

Uwaga techniczna: w komentarzach postaci `(*  ===> *)` będę przedstawiać wyniki wypisywane przez komendy.

Typ każdego termu możemy sprawdzić przy pomocy komendy `Check`. Jest ona nie do przecenienia. Jeżeli nie rozumiesz, co się dzieje w trakcie dowodu lub dlaczego Coq nie chce zaakceptować jakiejś definicji, użyj komendy `Check`, żeby sprawdzić, z jakimi typami masz do czynienia.

2.2 Typy a zbiory

Z filozoficznego punktu widzenia należy stanowczo odróżnić typy od zbiorów, znanych chociażby z teorii zbiorów ZF, która jest najpowszechniej używaną podstawą współczesnej matematyki:

  - zbiory są materialne, podczas gdy typy są strukturalne. Dla przykładu, zbiory {1, 2} oraz {2, 3} mają przecięcie równe {2}, które to przecięcie jest podzbiorem każdego z nich. W przypadku typów jest inaczej — dwa różne typy są zawsze rozłączne i żaden typ nie jest podtypem innego
  - relacja “x $\in A''jest semantyczna, tzn. jest zdaniem logicznym i wymaga dowodu. Relacja “x : A''jest syntaktyczna, a wicnie jest zdaniem logicznym i nie wymaga dowodu ^- Coq jest w stanie sprawdzi$

- zbiór to kolekcja obiektów, do której można włożyć cokolwiek. Nowe zbiory mogą być formowane ze starych w sposób niemal dowolny (aksjomaty są dość liberalne). Typ to kolekcja obiektów o takiej samej wewnętrznej naturze. Zasady formowania nowych typów ze starych są bardzo ścisłe
- teoria zbiorów mówi, jakie obiekty istnieją (np. aksjomat zbioru potęgowego mówi, że dla każdego zbioru istnieje zbiór wszystkich jego podzbiorów). Teoria typów mówi, w jaki sposób obiekty mogą być konstruowane — różnica być może ciężko dostrzegalna dla niewprawionego oka, ale znaczna

2.3 Logika klasyczna i konstruktywna

Jak udowodnić twierdzenie, by komputer mógł zweryfikować nasz dowód? Jedną z metod dowodzenia używanych w logice klasycznej są tabelki prawdy. Są one metodą skuteczną, gdyż działają zawsze i wszędzie, ale nie są wolne od problemów.

Pierwszą, praktyczną przeszkodą jest rozmiar tabelek — rośnie on wykładniczo wraz ze wzrostem ilości zmiennych zdaniowych, co czyni tę metodę skrajnie niewydajną i obliczeniożerną, a więc niepraktyczną dla twierdzeń większych niż zabawkowe.

Druga przeszkoda, natury filozoficznej, i bardziej fundamentalna od pierwszej to poczynione implicite założenie, że każde zdanie jest prawdziwe lub fałszywe, co w logice konstruktywnej jest nie do końca prawdą, choć w logice klasycznej jest słuszne. Wynika to z różnych interpretacji prawdziwości w tych logikach.

Dowód konstruktywny to taki, który polega na skonstruowaniu pewnego obiektu i logika konstruktywna dopuszcza jedynie takie dowody. Logika klasyczna, mimo że również dopuszcza dowody konstruktywne, standardy ma nieco luźniejsze i dopuszcza również dowód polegający na pokazaniu, że nieistnienie jakiegoś obiektu jest sprzeczne. Jest to sposób dowodzenia fundamentalnie odmienny od poprzedniego, gdyż sprzeczność nieistnienia jakiegoś obiektu nie daje nam żadnej wskazówki, jak go skonstruować.

Dobrym przykładem jest poszukiwanie rozwiązań równania: jeżeli udowodnimy, że nieistnienie rozwiązania jest sprzeczne, nie znaczy to wcale, że znaleźliśmy rozwiązanie. Wiemy tylko, że jakieś istnieje, ale nie wiemy, jak je skonstruować.

2.4 Dedukcja naturalna i taktyki

Ważną konkluzją płynącą z powyższych rozważań jest fakt, że logika konstruktywna ma interpretację obliczeniową — każdy dowód można interpretować jako pewien program. Odnosząc się do poprzedniego przykładu, konstruktywny dowód faktu, że jakieś równanie ma rozwiązanie, jest jednocześnie programem, który to rozwiązanie oblicza.

Wszystko to sprawia, że dużo lepszym, z naszego punktu widzenia, stylem dowodzenia będzie *dedukcja naturalna* — styl oparty na małej liczbie aksjomatów, zaś dużej liczbie reguł wnioskowania. Reguł, z których każda ma swą własną interpretację obliczeniową, dzięki czemu dowodząc przy ich pomocy będziemy jednocześnie konstruować pewien program. Sprawdzenie, czy dowód jest poprawny, będzie się sprowadzało do sprawdzenia, czy program ten jest poprawnie typowany (co Coq może zrobić automatycznie), zaś wykonanie tego programu skonstruuje obiekt, który będzie "świadkiem" prawdziwości twierdzenia.

Jako, że każdy dowód jest też programem, w Coqu dowodzić można na dwa diametralnie różne sposoby. Pierwszy z nich polega na "ręcznym" skonstruowaniu termu, który reprezentuje dowód — ten sposób dowodzenia przypomina zwykłe programowanie.

Drugim sposobem jest użycie taktyk. Ten sposób jest rozszerzeniem opisanego powyżej systemu dedukcji naturalnej. Taktyki nie są tym samym, co reguły wnioskowania — regułom odpowiadają jedynie najprostsze taktyki. Język taktyk Coqa, Ltac, pozwala z prostych taktyk budować bardziej skomplikowane przy użyciu konstrukcji podobnych do tych, których używa się do pisania "zwykłych" programów.

Taktyki konstruują dowody, czyli programy, jednocześnie same będąc programami. Innymi słowy: taktyki to programy, które piszą inne programy.

Ufff... jeżeli twój mózg jeszcze nie eksplodował, to czas wziąć się do konkretów!

2.5 Konstruktywny rachunek zdań

Nadszedł dobry moment na to, żebyś odpalił CoqIDE. Sesja interaktywna w CoqIDE przebiega następująco: edytujemy plik z rozszerzeniem .v wpisując komendy. Po kliknięciu przycisku "Forward one command" (strzałka w dół) Coq interpretuje kolejną komendę, a po kliknięciu "Backward one command" (strzałka w górę) cofa się o jedną komendę do tyłu. Ta interaktywność, szczególnie w trakcie przeprowadzania dowodu, jest bardzo mocnym atutem Coqa — naucz się ją wykorzystywać, dokładnie obserwując skutki działania każdej komendy.

W razie problemów z CoqIDE poszukaj pomocy w manualu: coq.inria.fr/refman/Reference-Manual018.html

2.5.1 Implikacja

Zacznijmy od czegoś prostego: pokażemy, że P implikuje P.

```
Lemma impl_refl : P → P.
Proof.
  intro dowód_na_to_że_P_zachodzi.
```

```
  exact dowód_na_to_że_P_zachodzi.
Qed.
```

Słowo kluczowe `Lemma` obwieszcza, że chcemy podać twierdzenie. Musi mieć ono nazwę (tutaj *impl_refl*). Samo twierdzenie podane jest po dwukropku — twierdzenie jest typem, a jego udowodnienie sprowadza się do skonstruowania termu tego typu. Zauważmy też, że każda komenda musi kończyć się kropką.

Twierdzenia powinny mieć łatwe do zapamiętania oraz sensowne nazwy, które informują (z grubsza), co właściwie chcemy udowodnić. Nazwa *impl_refl* oznacza, że twierdzenie wyraża fakt, że implikacja jest zwrotna.

Dowody będziemy zaczynać komendą `Proof`. Jest ona opcjonalna, ale poprawia czytelność, więc warto ją stosować.

Jeżeli każesz Coqowi zinterpretować komendę zaczynającą się od `Lemma`, po prawej stronie ekranu pojawi się stan aktualnie przeprowadzanego dowodu.

Od góry mamy: ilość podcelów (rozwiązanie wszystkich kończy dowód) — obecnie 1, kontekst (znajdują się w nim obiekty, które możemy wykorzystać w dowodzie) — obecnie mamy w nim zdania P, Q i R; kreskę oddzielającą kontekst od aktualnego celu, obok niej licznik, który informuje nas, nad którym podcelem pracujemy — obecnie 1/1, oraz aktualny cel — dopiero zaczynamy, więc brzmi tak samo jak nasze twierdzenie.

Taktyki mogą wprowadzać zmiany w celu lub w kontekście, w wyniku czego rozwiązują lub generują nowe podcele. Taktyka może zakończyć się sukcesem lub zawieść. Dokładne warunki sukcesu lub porażnki zależą od konkretnej taktyki.

Taktyka `intro` działa na cele będące implikacją $\to$ i wprowadza jedną hipotezę z celu do kontekstu jeżeli to możliwe; w przeciwnym przypadku zawodzi. W dowodach słownych lub pisanych na kartce/tablicy użyciu taktyki `intro` odpowiadałoby stwierdzenie "załóżmy, że P jest prawdą", "załóżmy, że P zachodzi" lub po prostu "załóżmy, że P".

Szczegółem, który odróżnia dowód w Coqu (który dalej będziemy zwać "dowodem formalnym") od dowodu na kartce/tablicy/słownie (zwanego dalej "dowodem nieformalnym"), jest fakt, że nie tylko sama hipoteza, ale też dowód ("świadek") jej prawdziwości, musi mieć jakąś nazwę — w przeciwnym wypadku nie bylibyśmy w stanie się do nich odnosić. Dowodząc na tablicy, możemy odnieść się do jej zawartości np. poprzez wskazanie miejsca, w stylu "dowód w prawym górnym rogu tablicy". W Coqu wszelkie odniesienia działają identycznie jak odniesienia do zmiennych w każdym innym języku programowania — przy pomocy nazwy.

Upewnij się też, że dokładnie rozumiesz, co taktyka `intro` wprowadziła do kontekstu. Nie było to zdanie P — ono już się tam znajdowało, o czym świadczyło stwierdzenie P : `Prop` — cofnij stan dowodu i sprawdź, jeżeli nie wierzysz. Hipotezą wprowadzoną do kontekstu był obiekt, którego nazwę podaliśmy taktyce jako argument, tzn. *dowód_na_to_że_P_zachodzi*, który jest właśnie tym, co głosi jego nazwa — "świadkiem" prawdziwości P. Niech nie zmyli cię użyte na początku rozdziału słowo kluczowe `Hypothesis`.

Taktyka `exact` rozwiązuje cel, jeżeli term podany jako argument ma taki sam typ, jak cel, a w przeciwnym przypadku zawodzi. Jej użyciu w dowodzie nieformalnym odpowiada stwierdzenie "mamy w założeniach dowód na to, że P, który nazywa się x, więc x dowodzi tego, że P".

Pamiętaj, że cel jest zdaniem logicznym, czyli typem, a hipoteza jest dowodem tego zdania, czyli termem tego typu. Przyzwyczaj się do tego utożsamienia typów i zdań oraz dowodów i programów/termów — jest to wspomniana we wstępie korespondencja Curry'ego-Howarda, której wiele wcieleń jeszcze zobaczymy.

Dowód kończy się zazwyczaj komendą `Qed`, która go zapisuje.

```
Lemma impl_refl' : P → P.
Proof.
  intro. assumption.
Qed.
```

Zauważmy, że w Coqowych nazwach można używać apostrofu. Zgodnie z konwencją nazwa pokroju *x'* oznacza, że *x'* jest w jakiś sposób blisko związany z *x*. W tym wypadku używamy go, żeby podać inny dowód udowodnionego już wcześniej twierdzenia. Nie ma też nic złego w pisaniu taktyk w jednej linijce (styl pisania jak zawsze powinien maksymalizować czytelność).

Jeżeli użyjemy taktyki `intro` bez podawania nazwy hipotezy, zostanie użyta nazwa domyślna (dla wartości typu `Prop` jest to *H*; jeżeli ta nazwa jest zajęta, zostanie użyte *H0*, *H1*...). Domyślne nazwy zazwyczaj nie są dobrym pomysłem, ale w prostych dowodach możemy sobie na nie pozwolić.

Taktyka `assumption` (ang. "założenie") sama potrafi znaleźć nazwę hipotezy, która rozwiązuje cel. Jeżeli nie znajdzie takiej hipotezy, to zawodzi. Jej użycie w dowodzenie nieformalnym odpowiada stwierdzeniu "P zachodzi na mocy założenia".

```
Print impl_refl'.
(*  ===> impl_refl' = fun H : P => H
     : P -> P *)
```

Wspomnieliśmy wcześniej, że zdania logiczne są typami, a ich dowody termami. Używając komendy `Print` możemy wyświetlić definicję podanego termu (nie każdego, ale na razie się tym nie przejmuj). Jak się okazuje, dowód naszej trywialnej implikacji jest funkcją. Jest to kolejny element korespondencji Curry'ego-Howarda.

Po głębszym namyśle nie powinien nas on dziwić: implikację można interpretować wszakże jako funkcję, która bierze dowód poprzednika i zwraca dowód następnika. Wykonanie funkcji odpowiada tutaj procesowi wywnioskowania konkluzji z przesłanki.

Wspomnieliśmy także, że każda taktyka ma swoją własną interpretację obliczeniową. Jaki był więc udział taktyk `intro` i `exact` w konstrukcji naszego dowodu? Dowód implikacji jest funkcją, więc możemy sobie wyobrazić, że na początku dowodu term wyglądał tak: `fun` ?1 ⇒ ?2 (symbole ?1 i ?2 reprezentują fragmenty dowodu, których jeszcze nie skonstruowaliśmy). Taktyka `intro` wprowadza zmienną do kontekstu i nadaje jej nazwę, czemu odpowiada zastąpienie w naszym termie ?1 przez $H : P$. Możemy sobie wyobrazić, że po użyciu taktyki intro term wygląda tak: `fun` $H : P \Rightarrow$?2. Użycie taktyki `exact` (lub `assumption`) dało w efekcie zastępienie ?2 przez *H*, czyli szukany dowód zdania *P*. Ostatecznie term przybrał postać `fun` $H : P \Rightarrow H$. Ponieważ nie ma już żadnych brakujących elementów, dowód kończy się. Gdy użyliśmy komendy `Qed` Coq zweryfikował, czy aby na pewno term skonstruowany przez taktyki jest poprawnie typowany, a następnie zaakceptował nasz dowód.

```
Lemma modus_ponens :
  (P → Q) → P → Q.
Proof.
  intros. apply H. assumption.
Qed.
```

Implikacja jest operatorem łączącym w prawo (ang. right associative), więc wyrażenie $(P \to Q) \to P \to Q$ to coś innego, niż $P \to Q \to P \to Q$ — w pierwszym przypadku jedna z hipotez jest implikacją

Wprowadzanie zmiennych do kontekstu pojedynczo może nie być dobrym pomysłem, jeżeli jest ich dużo. Taktyka `intros` pozwala nam wprowadzić do kontekstu zero lub więcej zmiennych na raz, a także kontrolować ich nazwy. Taktyka ta nigdy nie zawodzi. Jej odpowiednik w dowodach nieformalnych oraz interpretacja obliczeniowa są takie, jak wielokrotnego (lub zerokrotnego) użycia taktyki `intro`.

Taktyka `apply` pozwala zaaplikować hipotezę do celu, jeżeli hipoteza jest implikacją, której konkluzją jest cel. W wyniku działania tej taktyki zostanie wygenerowana ilość podcelów równa ilości przesłanek, a stary cel zostanie rozwiązany. W kolejnych krokrach będziemy musieli udowodnić, że przesłanki są prawdziwe. W naszym przypadku hipotezę H typu $P \to Q$ zaaplikowaliśmy do celu Q, więc zostanie wygenerowany jeden podcel P.

Interpretacją obliczeniową taktyki `apply` jest, jak sama nazwa wskazuje, aplikacja funkcji. Nie powinno nas to wcale dziwić — wszak ustaliliśmy przed chwilą, że implikacje są funkcjami. Możemy sobie wyobrazić, że po użyciu taktyki `intros` nasz proofterm (będę tego wyrażenia używał zamiast rozwlekłego "term będący dowodem") wyglądał tak: `fun` $(H : P \to Q)$ $(H0 : P) \Rightarrow$?1. Taktyka `apply` H przekształca brakujący fragment dowodu ?1 we fragment, w którym również czegoś brakuje: H ?2 — tym czymś jest argument. Pasujący argument znaleźliśmy przy pomocy taktyki assumption, więc ostatecznie proofterm ma postać `fun` $(H : P \to Q)$ $(H0 : P) \Rightarrow H\ H0$.

Reguła wnioskowania modus ponens jest zdecydowanie najważniejszą (a w wielu systemach logicznych jedyną) regułą wnioskowania. To właśnie ona odpowiada za to, że w systemie dedukcji naturalnej dowodzimy "od tyłu" — zaczynamy od celu i aplikujemy hipotezy, aż dojdziemy do jakiegoś zdania prawdziwego.

Nadszedł czas na pierwsze ćwiczenia. Zanim przejdziesz dalej, postaraj się je wykonać — dzięki temu upewnisz się, że zrozumiałeś w wystarczającym stopniu omawiane w tekście zagadnienia. Postaraj się nie tylko udowodnić poniższe twierdzenia, ale także zrozumieć (a póki zadania są proste — być może także przewidzieć), jaki proofterm zostanie wygenerowany. Powodzenia!

Ćwiczenie (implikacja) Udowodnij poniższe twierdzenia.

```
Lemma impl_trans :
  (P → Q) → (Q → R) → (P → R).

Lemma impl_permute :
```

$(P \to Q \to R) \to (Q \to P \to R)$.

```
Lemma impl_dist :
  (P → Q → R) → ((P → Q) → (P → R)).
```

Ćwiczenie (bez `apply`) Udowodnij następujące twierdzenie bez używania taktyki `apply`.

```
Lemma modus_ponens' :
  (P → Q) → P → Q.
```

2.5.2 Fałsz

```
Lemma ex_falso : False → P.
Proof.
  intro. inversion H.
Qed.
```

False to zdanie zawsze fałszywe, którego nie można udowodnić. Nie istnieje żaden term tego typu, więc jeżeli taki term znajdzie się w naszym kontekście, to znaczy, że uzyskaliśmy sprzeczność. Jeżeli użyjemy taktyki `inversion` na hipotezie, która jest typu *False*, obecny podcel zostanie natychmiast rozwiązany.

Nazwa *ex_falso* pochodzi od łacińskiego wyrażenia "ex falso sequitur quodlibet", które znaczy "z fałszu wynika cokolwiek zechcesz".

Uzasadnienie tej reguły wnioskowania w logice klasycznej jest dziecinnie proste: skoro fałsz to prawda, to w tabelce prawdy dla tego zdania w kolumnie wynikowej wszystkie zera (fałsz) możemy zastąpić jedynkami (prawda), otrzymując zdanie prawdziwe.

W logice konstruktywnej takie uzasadnienie oczywiście nie przejdzie, gdyż ustaliliśmy już, że nie możemy o dowolnym zdaniu powiedzieć, że jest albo prawdziwe, albo fałszywe, gdyż nie jesteśmy w stanie tak ogólnego faktu udowodnić. Nie będziemy na razie uzasadniać tej reguły ani wnikać w szczegóły działania taktyki `inversion` — dowiemy się tego już niedługo.

2.5.3 Prawda

```
Lemma truth : True.
Proof.
  trivial.
Qed.
```

True to zdanie zawsze prawdziwe. Jego udowodnienie nie jest zbyt trudne — możemy to zrobić np. przy pomocy taktyki `trivial`, która, jak sama nazwa wskazuje, potrafi sama rozwiązywać proste cele.

```
Print truth.
(*  ===> truth = I : True *)
```

Jeżeli przyjrzymy się skonstruowanemu prooftermowi, dostrzeżemy term o nazwie *I*. Jest to jedyny dowód zdania *True*. Jego nazwa nie niesie ze sobą żadnego głębszego znaczenia, ale jego istnienie jest konieczne — pamiętajmy, że udowodnienie zdania sprowadza się do skonstruowania termu odpowiedniego typu. Nie inaczej jest w przypadku zdania zawsze prawdziwego — musi istnieć jego dowód, a żeby móc się do niego odnosić, musi też mieć jakąś nazwę.

Zdanie *True*, w przeciwieństwie do *False*, nie jest zbyt użyteczne ani często spotykane, ale czasem się przydaje.

2.5.4 Negacja

```
Check ¬P.
(*  ===> ~ P : Prop *)
```

W Coqu negację zdania P oznaczamy przez $\neg P$. Symbol $\neg$ nie jest jednak nazwą negacji — nazwy nie mogą być symbolami. Jest to jedynie notacja, która ma uczynić zapis krótszym i bardziej podobnym do tego używanego na codzień. Niesie to jednak za sobą pewne konsekwencje — nie możemy np. użyć komendy `Print ~.`, żeby wyświetlić definicję negacji. Jak więc poznać nazwę, kryjącą się za jakąś notacją?

```
Locate "~".
(*  ===> "~ x" := not x ... *)
```

Możemy to zrobić przy pomocy komendy `Locate`. Wyświetla ona, do jakich nazw odwołuje się dana notacja. Negacja w Coqu nazywa się *not*.

W logice klasycznej negację zdania P można zinterpretować po prostu jako spójnik zdaniowy tworzący nowe zdanie, którego wartość logiczna jest przeciwna do wartości zdania P.

Jeżeli uważnie czytałeś fragmenty dotyczące logiki klasycznej i konstruktywnej, dostrzegasz już zapewne, że taka definicja nie przejdzie w logice konstruktywnej, której interpretacja opiera się na dowodach, a nie wartościach logicznych. Jak więc konstruktywnie zdefiniować negację?

Zauważmy, że jeżeli zdanie P ma dowód, to nie powinien istnieć żaden dowód jego negacji, $\neg P$. Uzyskanie takiego dowodu oznaczałoby sprzeczność, a więc w szczególności możliwość udowodnienia *False*. Jak to spostrzeżenie przekłada się na Coqową praktykę? Skoro znamy już nazwę negacji, *not*, możemy sprawdzić jej definicję:

```
Print not.
(*  ===> not = fun A : Prop => A -> False
     : Prop -> Prop *)
```

Definicja negacji w Coqu opiera się właśnie na powyższym spostrzeżeniu: jest to funkcja, która bierze zdanie A, a zwraca zdanie $A \to$ *False*, które możemy odczytać jako "A prowadzi do sprzeczności". Jeżeli nie przekonuje cię to rozumowanie, przyjrzyj się uważnie poniższemu twierdzeniu.

```
Lemma P_notP : ¬P → P → False.
```

```
Proof.
  intros HnotP HP.
  unfold not in HnotP.
  apply HnotP.
  assumption.
Qed.
```

Taktyka `unfold` służy do odwijania definicji. W wyniku jej działania nazwa zostanie zastąpiona przez jej definicję, ale tylko w celu. Jeżeli podana nazwa do niczego się nie odnosi, taktyka zawiedzie. Aby odwinąć definicję w hipotezie, musimy użyć taktyki `unfold` *nazwa* `in` *hipoteza*, a jeżeli chcemy odwinąć ją wszędzie — `unfold` *nazwa* `in` *.

Twierdzenie to jest też pewnym uzasadnieniem definicji negacji: jest ona zdefiniowana tak, aby uzyskanie fałszu z dwóch sprzecznych przesłanek było jak najprostsze.

```
Lemma P_notP' : ¬P → P → 42 = 666.
Proof.
  intros. cut False.
    inversion 1.
    apply H. assumption.
Qed.
```

Taktyką, która czasem przydaje się w dowodzeniu negacji i radzeniu sobie z *False*, jest `cut`. Jeżeli nasz cel jest postaci G, to taktyka `cut` P rozwiąże go i wygeneruje nam w zamian dwa podcele postaci $P \to G$ oraz P. Nieformalnie odpowiada takiemu rozumowaniu: "cel G wynika z P; P zachodzi".

Udowodnić *False* $\to 42 = 666$ moglibyśmy tak jak poprzednio: wprowadzić hipotezę *False* do kontekstu przy pomocy `intro`, a potem użyć na niej `inversion`. Możemy jednak zrobić to nieco szybciej. Jeżeli cel jest implikacją, to taktyka `inversion` 1 działa tak samo, jak wprowadzenie do kontekstu jednej przesłanki i użycie na niej zwykłego `inversion`.

Drugi podcel również moglibyśmy rozwiązać jak poprzednio: odwinąć definicję negacji, zaaplikować odpowiednią hipotezę, a potem zakończyć przy pomocy `assumption`. Nie musimy jednak wykonywać pierwszego z tych kroków — Coq jest w stanie zorientować się, że $\neg P$ jest tak naprawdę implikacją, i zaaplikować hipotezę H bez odwijania definicji negacji. W ten sposób oszczędzamy sobie trochę pisania, choć ktoś mógłby argumentować, że zmniejszamy czytelność dowodu.

Uwaga dotycząca stylu kodowania: postaraj się zachować 2 spacje wcięcia na każdy poziom zagłębienia, gdzie poziom zagłębienia zwiększa się o 1, gdy jakaś taktyka wygeneruje więcej niż 1 podcel. Tutaj taktyka `cut` wygenerowała nam 2 podcele, więc dowody obydwu zaczniemy od nowej linii po dwóch dodatkowych spacjach. Rozwiązanie takie znacznie zwiększa czytelność, szczególnie w długich dowodach.

Interpretacja obliczeniowa negacji wynika wprost z interpretacji obliczeniowej implikacji. Konstruktywna negacja różni się od tej klasycznej, o czym przekonasz się w ćwiczeniu.

Ćwiczenie (negacja) Udowodnij poniższe twierdzenia.

Lemma *not_False* : ¬*False*.

Lemma *not_True* : ¬ *True* → *False*.

Ćwiczenie (podwójna negacja) Udowodnij poniższe twierdzenia. Zastanów się, czy można udowodnić ~~*P* → *P*.

Lemma *dbl_neg_intro* : *P* → ~~*P*.

Lemma *double_neg_elim_irrefutable* :
~~ (~~ *P* → *P*).

Ćwiczenie (potrójna negacja) Udowodnij poniższe twierdzenie. Jakie są różnice między negacją, podwójną negacją i potrójną negacją?

Lemma *triple_neg_rev* : ~~~*P* → ¬*P*.

2.5.5 Koniunkcja

```
Lemma and_intro : P → Q → P ∧ Q.
Proof.
  intros. split.
    assumption.
    assumption.
Qed.
```

Symbol ∧ oznacza koniunkcję dwóch zdań logicznych i podobnie jak ¬ jest jedynie notacją (koniunkcja w Coqu nazywa się *and*).

W logice klasycznej koniunkcja jest prawdziwa, gdy obydwa jej człony są prawdziwe. W logice konstruktywnej sytuacja jest analogiczna, choć subtelnie różna: aby udowodnić koniunkcję, musimy udowodnić każdy z jej dwóch członów osobno.

Koniunkcji w Coqu dowodzimy przy pomocy taktyki `split`. Jako że musimy udowodnić oddzielnie oba jej człony, zostały dla nas wygenerowane dwa nowe podcele — jeden dla lewego członu, a drugi dla prawego. Ponieważ stary cel został rozwiązany, to do udowodnienia pozostają nam tylko te dwa nowe podcele.

```
Lemma and_proj1 : P ∧ Q → P.
Proof.
  intro H. destruct H. assumption.
Qed.
```

Aby udowodnić koniunkcję, użyliśmy taktyki `split`, która rozbiła ją na dwa osobne podcele. Jeżeli koniunkcją jest jedną z naszych hipotez, możemy posłużyć się podobnie działającą taktyką `destruct`, która dowód koniunkcji rozbija na osobne dowody obu jej członów. W naszym przypadku hipoteza $H : P \wedge Q$ zostaje rozbita na hipotezy $H : P$ oraz $H0 : Q$. Zauważ, że nowe hipotezy dostały nowe, domyślne nazwy.

Lemma *and_proj1'* : *P* ∧ *Q* → *P*.

```
Proof.
  intro HPQ. destruct HPQ as [HP HQ]. assumption.
Qed.
```

Podobnie jak w przypadku taktyki `intro`, domyślne nazwy nadawane przez taktykę `destruct` często nie są zbyt fortunne. Żeby nadać częściom składowym rozbijanej hipotezy nowe nazwy, możemy użyć tej taktyki ze składnią `destruct` *nazwa* `as` *wzorzec*. Ponieważ koniunkcja składa się z dwóch członów, *wzorzec* będzie miał postać [*nazwa1 nazwa2*].

Interpretacja obliczeniowa koniunkcji jest bardzo prosta: koniunkcja to uporządkowana para zdań, zaś dowód koniunkcji to uporządkowana para dowodów — pierwszy jej element dowodzi pierwszego członu koniunkcji, a drugi element — drugiego członu koniunkcji.

Ćwiczenie (koniunkcja) Udowodnij poniższe twierdzenia.

`Lemma` *and_proj2* : $P \land Q \to Q$.

`Lemma` *and3_intro* : $P \to Q \to R \to P \land Q \land R$.

`Lemma` *and3_proj* : $P \land Q \land R \to Q$.

`Lemma` *noncontradiction* : ~$(P \land \neg P)$.

2.5.6 Równoważność zdaniowa

Równoważność zdaniowa jest w Coqu oznaczana $\leftrightarrow$. Symbol ten, jak (prawie) każdy jest jedynie notacją — równoważność nazywa się *iff*. Jest to skrót od ang. "if and only if". Po polsku zdanie $P \leftrightarrow Q$ możemy odczytać jako "P wtedy i tylko wtedy, gdy Q".

```
Print iff.
(*  ===> fun A B : Prop => (A -> B) /\ (B -> A)
     : Prop -> Prop -> Prop *)
```

Jak widać, równoważność $P \leftrightarrow Q$ to koniunkcja dwóch implikacji $P \to Q$ oraz $Q \to P$. W związku z tym nie powinno nas dziwić, że pracuje się z nią tak samo jak z koniunkcją. Tak jak nie musieliśmy odwijać definicji negacji, żeby zaaplikować ją jak rasową impikcję, tak też nie musimy odwijać definicji równoważności, żeby posługiwać się nią jak prawdziwą koniunkcją. Jej interpretacja obliczeniowa wywodzi się z interpretacji obliczeniowej koniunkcji oraz implikacji.

`Lemma` *iff_intro* : $(P \to Q) \to (Q \to P) \to (P \leftrightarrow Q)$.

```
Proof.
  intros. split.
    intro. apply H. assumption.
    intro. apply H0. assumption.
Qed.
```

Do rozbijania równoważności będących celem służy, tak jak w przypadku koniunkcji, taktyka `split`.

`Lemma` *iff_proj1* : $(P \leftrightarrow Q) \to (P \to Q)$.

```
Proof.
  intros. destruct H as [HPQ HQP].
  apply HPQ. assumption.
Qed.
```

Równoważność znajdującą się w kontekście możemy zaś, tak jak koniunkcje, rozbijać taktyką `destruct`. Taką samą postać ma również wzorzec, służący w klauzuli `as` do nadawania nazw zmiennym.

Ćwiczenie (równoważność zdaniowa) Udowodnij poniższe twierdzenia.

Lemma *iff_refl* : $P \leftrightarrow P$.

Lemma *iff_symm* : $(P \leftrightarrow Q) \rightarrow (Q \leftrightarrow P)$.

Lemma *iff_trans*: $(P \leftrightarrow Q) \rightarrow (Q \leftrightarrow R) \rightarrow (P \leftrightarrow R)$.

Lemma *iff_not* : $(P \leftrightarrow Q) \rightarrow (\tilde{}P \leftrightarrow \neg Q)$.

Lemma *curry_uncurry* : $(P \rightarrow Q \rightarrow R) \leftrightarrow (P \wedge Q \rightarrow R)$.

2.5.7 Dysjunkcja

```
Lemma or_left : P → P ∨ Q.
Proof.
  intro. left. assumption.
Qed.
```

Symbol $\vee$ oznacza dysjunkcję dwóch zdań logicznych. W języku polskim czasem używa się też określenia "alternatywa", ale będziemy się tego wystrzegać, rezerwując to słowo dla czegoś innego. Żeby dowieść dysjunkcji $P \vee Q$, musimy udowonić albo lewy, albo prawy jej człon. Taktyki `left` oraz `right` pozwalają nam wybrać, którego z nich chcemy dowodzić.

```
Lemma or_comm_impl : P ∨ Q → Q ∨ P.
Proof.
  intro. destruct H as [p | q].
    right. assumption.
    left. assumption.
Qed.
```

Zauważmy, że użycie taktyki `destruct` zmieniło nam ilość celów. Wynika to z faktu, że nie wiemy, który człon hipotezy $P \vee Q$ jest prawdziwy, więc dla każdego przypadku musimy przeprowadzić osobny dowód. Inaczej wygląda też wzorzec służący do rozbicia tej hipotezy — w przypadku dysjunkcji ma on postać [*nazwa1* | *nazwa2*].

Interpretacja obliczeniowa dysjunkcji jest następująca: jest to suma rozłączna dwóch zdań. Dowód dysjunkcji to dowód jednego z jej członów z dodatkową informacją o tym, który to człon.

To ostatnie stwierdzenie odróżnia dysjunkcję konstruktywną od klasycznej: klasyczna dysjunkcja to stwierdzenie "któres z tych dwóch zdań jest prawdziwe (lub oba)", zaś konstruktywna to stwierdzenie "lewy człon jest prawdziwy albo prawy człon jest prawdziwy (albo oba, ale i tak dowodzimy tylko jednego)". Jest to znaczna różnica — w przypadku logiki klasycznej nie wiemy, który człon jest prawdziwy.

Ćwiczenie (dysjunkcja) Udowodnij poniższe twierdzenia.

`Lemma` *or_right* : $Q \to P \lor Q$.

`Lemma` *or_big* : $Q \to P \lor Q \lor R$.

`Lemma` *or3_comm_impl* : $P \lor Q \lor R \to R \lor Q \lor P$.

Ćwiczenie (dysjunkcja i implikacja) Udowodnij poniższe twierdzenie. Następnie zastanów się, czy odwrotna implikacja również zachodzi.

`Lemma` *or_impl* : $\neg P \lor Q \to (P \to Q)$.

2.6 Konstruktywny rachunek kwantyfikatorów

`End` *constructive_propositional_logic*.

Komenda `End` zamyka sekcję, którą otworzyliśmy na samym początku tego rozdziału. Zdania P, Q i R znikają z dostępnej dla nas przestrzeni nazw, dzięki czemu uniknęliśmy jej zaśmiecenia. Nasze twierdzenia wciąż są jednak dostępne (sprawdź to).

Zajmiemy się teraz konstruktywnym rachunkiem kwantyfikatorów. Jest on rozszerzeniem omówionego przed chwilą konstruktywnego rachunku zdań o kwantyfikatory, które pozwolą nam wyrażać takie zależności jak "każdy" oraz "istnieje", oraz o predykaty i relacje, które mózemy interpretować odpowiednio jako właściwości obiektów oraz zależności między obiektami.

2.6.1 Kwantyfikacja uniwersalna

Zobaczmy o co chodzi na znanym nam już przykładzie zwrotności implikacji:

```
Lemma impl_refl'' : ∀ P : Prop, P → P.
Proof.
  intros. assumption.
Qed.
```

∀ oznacza kwantyfikację uniwersalną. Możemy ten symbol odczytywać "dla każdego". Zasięg kwantyfikatora rozciąga się od przecinka aż do kropki. Wobec tego treść naszego twierdzenia możemy odczytać "dla każdego zdania logicznego P, P implikuje P".

Kwantyfikator uniwersalny jest w swej naturze bardzo podobny do implikacji — zmienne, których dotyczy, możemy wprowadzić do kontekstu przy pomocy taktyki `intro`. W dowodzie nieforamlnym użyciu taktyki `intro` P na celu kwantyfikowanym uniwersalnie odpowiadałoby stwierdzenie "niech P będzie dowolnym zdaniem logicznym".

Zauważ, że używając taktyki `intros`, możemy wprowadzić do kontekstu jednocześnie zmienne kwantyfikowane uniwersalnie oraz przesłanki występujące po lewej stronie implikacji. To wszystko powinno nasunąć nam myśl, że kwantyfikacja uniwersalna i implikacja są ze sobą blisko związane.

```
Print impl_refl''.
(*  ===> impl_refl'' = fun (P : Prop) (H : P) => H
     : forall P : Prop, P -> P *)
```

Rzeczywiście: dowodem naszego zdania jest coś, co na pierwszy rzut oka wygląda jak funkcja. Jeżeli jednak przyjrzysz się jej uważnie, dostrzeżesz że nie może być to zwykła funkcja — typ zwracanej wartości H różni się w zależności od argumentu P. Jeżeli za P wstawimy $1 = 1$, to H będzie dowodem na to, że $1 = 1$. Jeżeli za P wstawimy $2 = 2$, to H będzie dowodem na to, że $2 = 2$. Zauważ, że $1 = 1$ oraz $2 = 2$ to dwa różne zdania, a zatem są to także różne typy.

Dowód naszego zdania nie może być zatem zwykłą funkcją — gdyby nią był, zawsze zwracałby wartości tego samego typu. Jest on funkcją zależną, czyli taką, której przeciwdziedzina zależy od dziedziny. Funkcja zależna dla każdego argumentu może zwracać wartości różnego typu.

Ustaliliśmy więc, że kwantyfikacja uniwersalna jest pewnym uogólnieniem implikacji, zaś jej interpretacją obliczeniową jest funkcja zależna, czyli pewne uogólnienie zwykłej funkcji, która jest interpretacją obliczeniową implikacji.

```
Lemma general_to_particular :
  ∀ P : nat → Prop,
    (∀ n : nat, P n) → P 42.
Proof.
  intros. apply H.
Restart.
  intros. specialize (H 42). assumption.
Qed.
```

Podobnie jak zwykłe funkcje, funkcje zależne możemy aplikować do celu za pomocą taktyki `apply`. Możliwy jest też inny sposób rozumowania, nieco bardziej przypominający rozumowania "w przód": przy pomocy taktyki `specialize` możemy zainstancjować n w naszej hipotezie H, podając jej pewną liczbę naturalną. Wtedy nasza hipoteza H z ogólnej, z kwantyfikacją po wszystkich liczbach naturalnych, zmieni się w szczególną, dotyczącą tylko podanej przez nas liczby.

Komenda *Restart* pozwala nam zacząć dowód od nowa w dowolnym jego momencie. Jej użycie nie jest wymagane, by ukończyć powyższy dowód — spróbuj wstawić w jej miejsce `Qed`. Użyłem jej tylko po to, żeby czytelnie zestawić ze sobą sposoby rozumowania w przód i w tył dla kwantyfikacji uniwersalnej.

```
Lemma and_proj1'' :
```

```
  ∀ (P Q : nat → Prop),
    (∀ n : nat, P n ∧ Q n) → (∀ n : nat, P n).
Proof.
  intros P Q H k. destruct (H k). assumption.
Qed.
```

W powyższym przykładzie próba użycia taktyki `destruct` na hipotezie H zawiodłaby — H nie jest produktem. Żeby rozbić tę hipotezę, musielibyśmy najpierw wyspecjalizować ją dla interesującego nas k, a dopiero potem rozbić. Możemy jednak zrobić to w nieco krótszy sposób — pisząc `destruct` $(H\ k)$. Dzięki temu "w locie" przemienimy H z hipotezy ogólnej w szczególną, dotycząca tylko k, a potem rozbijemy. Podobnie poprzednie twierdzenie moglibyśmy udowodnić szybciej, jeżeli zamiast `specialize` i `assumption` napisalibyśmy `destruct` $(H\ 42)$ (choć i tak najszybciej jest oczywiście użyć `apply` H.

Ćwiczenie (kwantyfikacja uniwersalna) Udowodnij poniższe twierdzenie. Co ono oznacza? Przeczytaj je na głos. Zinterpretuj je, tzn. sparafrazuj.

```
Lemma all_dist :
  ∀ (A : Type) (P Q : A → Prop),
    (∀ x : A, P x ∧ Q x) ↔
    (∀ x : A, P x) ∧ (∀ x : A, Q x).
```

2.6.2 Kwantyfikacja egzystencjalna

Zdania egzystencjalne to zdania postaci "istnieje obiekt x, który ma właściwość P". W Coqu prezentują się tak:

```
Lemma ex_example1 :
  ∃ n : nat, n = 0.
Proof.
  ∃ 0. trivial.
Qed.
```

Kwantyfikacja egzystencjalna jest w Coqu zapisywana jako ∃ (exists). Aby udowodnić zdanie kwantyfikowane egzystencjalnie, musimy skonstruować obiekt, którego istnienie postulujemy, oraz udowodnić, że ma deklarowaną właściwość. Jest to wymóg dużo bardziej restrykcyjny niż w logice klasycznej, gdzie możemy zadowolić się stwierdzeniem, że nieistnienie takiego obiektu jest sprzeczne.

Powyższe twierdzenie możemy odczytać "istnieje liczba naturalna, która jest równa 0". W dowodzenie nieformalnym użyciu taktyki ∃ odpowiada stwierdzenie: "liczbą posiadającą tę właściwość jest 0". Następnie pozostaje nam udowodnić, iż rzeczywiście $0 = 0$, co jest trywialne.

```
Lemma ex_example2 :
  ¬ ∃ n : nat, 0 = S n.
Proof.
```

```
  intro. destruct H as [n H]. inversion H.
Qed.
```

Gdy zdanie kwantyfikowane egzystencjalnie znajdzie się w naszych założeniach, możemy je rozbić i uzyskać wspomniany w nim obiekt oraz dowód wspominianej właściwości. Nie powinno nas to dziwić — skoro zakładamy, że zdanie to jest prawdziwe, to musiało zostać ono udowodnione w sposób opisany powyżej — właśnie poprzez wskazanie obiektu i udowodnienia, że ma daną własność.

Myślę, że dostrzegasz już pewną prawidłowość:

- udowodnienie koniunkcji wymaga udowodnienia obydwu członów z osobna, więc dowód koniunkcji można rozbić na dowody poszczególnych członów (podobna sytuacja zachodzi w przypadku równoważności)
- udowodnienie dysjunkcji wymaga udowodnienia któregoś z członów, więc dowód dysjunkcji można rozbić, uzyskując dwa osobne podcele, a w każdym z nich dowód jednego z członów tej dysjunkcji
- udowodnienie zdania egzystencjalnego wymaga wskazania obiektu i podania dowodu żądanej własności, więc dowód takiego zdania możemy rozbić, uzyskując ten obiekt i dowód jego własności

Takie konstruowanie i dekonstruowanie dowodów (i innych termów) będzie naszym chlebem powszednim w logice konstruktywnej i w Coqu. Wynika ono z samej natury konstrukcji: zasady konstruowania termów danego typu są ściśle określone, więc możemy dokonywać dekonstrukcji, która polega po prostu na sprawdzeniu, jakimi zasadami posłużono się w konstrukcji. Nie przejmuj się, jeżeli wydaje ci się to nie do końca jasne — więcej dowiesz się już w kolejnym rozdziale.

Ostatnią wartą omówienia sprawą jest interpretacja obliczeniowa kwantyfikacji egzystencjalnej. Jest nią para zależna, tzn. taka, w której typ drugiego elementu może zależeć od pierwszego — pierwszym elementem pary jest obiekt, a drugim dowód, że ma on pewną własność. Zauważ, że podstawiając 0 do $\exists\ n : nat,\ n = 0$, otrzymamy zdanie $0 = 0$, które jest innym zdaniem, niż $1 = 0$ (choćby dlatego, że pierwsze jest prawdziwe, a drugie nie). Interpretacją obliczeniową taktyki $\exists$ jest wobec tego podanie pierwszego elementu pary, a podanie drugiego to po prostu przeprowadzenie reszty dowodu.

"Zależność" jest tutaj tego samego rodzaju, co w przypadku produktu zależnego — tam typ wyniku mógł być różny w zależność od wartości, jaką funkcja bierze na wejściu, a w przypadku sumy zależnej typ drugiego elementu może być różny w zależności od tego, jaki jest pierwszy element.

Nie daj się zwieść niefortunnemu nazewnictwu: produkt zależny $\forall\ x : A, B$, którego elementami są funkcje zależne, jest uogólnieniem typu funkcyjnego $A \to B$, którego elementami są zwykłe funkcje, zaś suma zależna $\exists\ x : A, B$, której elementami są pary zależne, jest uogólnieniem produktu $A \times B$, którego elementami są zwykłe pary.

Ćwiczenie (kwantyfikacja egzystencjalna) Udowodnij poniższe twierdzenie.

```
Lemma ex_or_dist :
  ∀ (A : Type) (P Q : A → Prop),
    (∃ x : A, P x ∨ Q x) ↔
    (∃ x : A, P x) ∨ (∃ x : A, Q x).
```

2.7 Paradoks golibrody

Języki naturalne, jakimi ludzie posługują się w życiu codziennym (polski, angielski suahili, język indian Navajo) zawierają spory zestaw spójników oraz kwantyfikatorów ("i", "a", "oraz", "lub", "albo", "jeżeli ... to", "pod warunkiem, że ", "wtedy", i wiele innych).

Należy z całą stanowczością zaznaczyć, że te spójniki i kwantyfikatory, a w szczególności ich intuicyjna interpretacja, znacznie różnią się od analogicznych spójników i kwantyfikatorów logicznych, które mieliśmy okazję poznać w tym rozdziale. Żeby to sobie uświadomić, zapoznamy się z pewnego rodzaju "paradoksem".

```
Theorem barbers_paradox :
  ∀ (man : Type) (barber : man)
    (shaves : man → man → Prop),
      (∀ x : man, shaves barber x ↔ ¬ shaves x x) → False.
```

Twierdzenie to formułowane jest zazwyczaj tak: nie istnieje człowiek, który goli wszystkich tych (i tylko tych), którzy sami siebie nie golą.

Ale cóż takiego znaczy to przedziwne zdanie? Czy matematyka dają nam magiczną, aprioryczną wiedzę o fryzjerach?

Odczytajmy je poetycko. Wyobraźmy sobie pewne miasteczko. Typ *man* będzie reprezentował jego mieszkańców. Niech term *barber* typu *man* oznacza hipotetycznego golibrodę. Hipotetycznego, gdyż samo użycie jakiejś nazwy nie powoduje automatycznie, że nazywany obiekt istnieje (przykładów jest masa, np. jednorożce, sprawiedliwość społeczna).

Mamy też relację *shaves*. Będziemy ją interpretować w ten sposób, że *shaves a b* zachodzi, gdy *a* goli brodę *b*. Nasza hipoteza ∀ *x* : *man*, *shaves barber x* ↔ ¬ *shaves x x* jest zawoalowanym sposobem podania następującej definicji: golibrodą nazwiemy te osoby, który golą wszystkie te i tylko te osoby, które same siebie nie golą.

Póki co sytuacja rozwija się w całkiem niekontrowersyjny sposób. Żeby zburzyć tę sielankę, możemy zadać sobie następujące pytanie: czy golibroda rzeczywiście istnieje? Dziwne to pytanie, gdy na każdym rogu ulicy można spotkać fryzjera, ale nie dajmy się zwieść.

W myśl rzymskich sentencji "quis custodiet ipsos custodes?" ("kto będzie pilnował strażników?") oraz "medice, cure te ipsum!" ("lekarzu, wylecz sam siebie!") możemy zadać dużo bardziej konkretne pytanie: kto goli brody golibrody? A idąc jeszcze krok dalej: czy golibroda goli sam siebie?

Rozstrzygnięcie jest banalne i wynika wprost z definicji: jeśli golibroda (*barber*) to ten, kto goli (*shaves barber x*) wszystkich tych i tylko tych (∀ *x* : *man*), którzy sami siebie nie golą ($\neg$ *shaves x x*), to podstawiając *barber* za *x* otrzymujemy sprzeczność: *shaves barber barber* zachodzi wtedy i tylko wtedy, gdy $\neg$ *shaves barber barber*.

Tak więc golibroda, zupełnie jak Święty Mikołaj, nie istnieje. Zdanie to nie ma jednak wiele wspólnego ze światem rzeczywistym: wynika ono jedynie z takiej a nie innej, przyjętej przez nas całkowicie arbitralnie definicji słowa "golibroda". Można to łatwo zobrazować, przeformułowywując powyższe twierdzenie z użyciem innych nazw:

$\texttt{Lemma}$ *barbers_paradox'* :
  $\forall$ (A : $\texttt{Type}$) (x : A) (P : $A \to A \to \texttt{Prop}$),
    ($\forall$ y : A, P x $y \leftrightarrow \neg$ P y y) $\to$ *False*.

Nieistnienie "golibrody" i pokrewny mu paradoks pytania "czy golibroda goli sam siebie?" jest konsekwencją wyłącznie formy powyższego zdania logicznego i nie mówi nic o rzeczywistoświatych golibrodach.

Paradoksalność całego "paradoksu" bierze się z tego, że typom, zmiennym i relacjom specjalnie nadano takie nazwy, żeby zwykły człowiek bez głębszych dywagacji nad definicją słowa "golibroda" przjął, że golibroda istnieje. Robiąc tak, wpada w sidła pułapki zastawionej przez logika i zostaje trafiony paradoksalną konkluzją: golibroda nie istnieje.

2.8 Paradoks pieniądza i kebaba

Przestrzegłem cię już przed nieopatrznym interpretowaniem zdań języka naturalnego za pomocą zdań logiki formalnej. Gdybyś jednak wciąż był skłonny to robić, przyjrzyjmy się kolejnemu "paradoksowi".

$\texttt{Lemma}$ *copy* :
  $\forall$ P : $\texttt{Prop}$, $P \to P \land P$.

Powyższe niewinnie wyglądające twierdzenie mówi nam, że P implikuje P i P. Spróbujmy przerobić je na paradoks, wymyślając jakąś wesołą interpretację dla P.

Niech zdanie P znaczy "mam złotówkę". Wtedy powyższe twierdzenie mówi, że jeżeli mam złotówkę, to mam dwa złote. Widać, że jeżeli jedną z tych dwóch złotówek znów wrzucimy do twierdzenia, to będziemy mieli już trzy złote. Tak więc jeżeli mam złotówkę, to mam dowolną ilość pieniędzy.

Dla jeszcze lepszego efektu powiedzmy, że za 10 złotych możemy kupić kebaba. W ostatecznej formie nasze twierdzenie brzmi więc: jeżeli mam złotówkę, to mogę kupić nieograniczoną ilość kebabów.

Jak widać, logika formalna (przynajmniej w takiej postaci, w jakiej ją poznajemy) nie nadaje się do rozumowania na temat zasobów. Zasobów, bo tym właśnie są pieniądze i kebaby. Zasoby to byty, które można przetwarzać, przemieszczać i zużywać, ale nie można ich kopiować i tworzyć z niczego. Powyższe twierdzenie dobitnie pokazuje, że zdania logiczne nie mają nic wspólnego z zasobami, gdyż ich dowody mogą być bez ograniczeń kopiowane.

Ćwiczenie (formalizacja paradoksu) UWAGA TODO: to ćwiczenie wymaga znajomości rozdziału 2, w szczególności indukcji i rekursji na liczbach naturalnych.

Zdefiniuj funkcję *andn* : *nat* → `Prop` → `Prop`, taką, że *andn n P* to n-krotna koniunkcja zdania P, np. *andn* 5 P to $P \land P \land P \land P \land P$. Następnie pokaż, że P implikuje *andn n* P dla dowolnego n.

Na końcu sformalizuj resztę paradoksu, tzn. zapisz jakoś, co to znaczy mieć złotówkę i że za 10 złotych można kupić kebaba. Wywnioskuj stąd, że mając złotówkę, możemy kupić dowolną liczbę kebabów.

Szach mat, Turcjo bankrutuj!

2.9 Kombinatory taktyk

Przyjrzyjmy się jeszcze raz twierdzeniu *iff_intro* (lekko zmodernizowanemu przy pomocy kwantyfikacji uniwersalnej).

```
Lemma iff_intro' :
  ∀ P Q : Prop,
    (P → Q) → (Q → P) → (P ↔ Q).
Proof.
  intros. split.
    intro. apply H. assumption.
    intro. apply H0. assumption.
Qed.
```

Jego dowód wygląda dość schematycznie. Taktyka `split` generuje nam dwa podcele będące implikacjami — na każdym z osobna używamy następnie `intro`, a kończymy `assumption`. Jedyne, czym różnią się dowody podcelów, to nazwa aplikowanej hipotezy.

A co, gdyby jakaś taktyka wygenerowała nam 100 takich schematycznych podcelów? Czy musielibyśmy przechodzić przez mękę ręcznego dowodzenia tych niezbyt ciekawych przypadków? Czy da się powyższy dowód jakoś skrócić lub zautomatyzować?

Odpowiedź na szczęście brzmi "tak". Z pomocą przychodzą nam kombinatory taktyk (ang. tacticals), czyli taktyki, które mogą przyjmować jako argumenty inne taktyki. Dzięki temu możemy łączyć proste taktyki w nieco bardziej skomplikowane lub jedynie zmieniać niektóre aspekty ich zachowań.

2.9.1 ; (średnik)

```
Lemma iff_intro'' :
  ∀ P Q : Prop,
    (P → Q) → (Q → P) → (P ↔ Q).
Proof.
  split; intros; [apply H | apply H0]; assumption.
Qed.
```

Najbardziej podstawowym kombinatorem jest ; (średnik). Zapis *t1*; *t2* oznacza "użyj na obecnym celu taktyki *t1*, a następnie na wszystkich podcelach wygenerowanych przez *t1* użyj taktyki *t2*".

Zauważmy, że taktyka `split` działa nie tylko na koniunkcjach i równoważnościach, ale także wtedy, gdy są one konkluzją pewnej implikacji. W takich przypadkach taktyka `split` przed rozbiciem ich wprowadzi do kontekstu przesłanki implikacji (a także zmienne związane kwantyfikacją uniwersalną), zaoszczędzając nam użycia wcześniej taktyki `intros`.

Wobec tego, zamiast wprowadzać zmienne do kontekstu przy pomocy `intros`, rozbijać cel `split`em, a potem jeszcze w każdym podcelu z osobna wprowadzać do kontekstu przesłankę implikacji, możemy to zrobić szybciej pisząc `split; intros`.

Drugie użycie średnika jest uogólnieniem pierwszego. Zapis *t*; [*t1* | *t2* | ... | *tn*] oznacza "użyj na obecnym podcelu taktyki *t*; następnie na pierwszym wygenerowanym przez nią podcelu użyj taktyki *t1*, na drugim *t2*, etc., a na n-tym użyj taktyki *tn*". Wobec tego zapis *t1*; *t2* jest jedynie skróconą formą *t1*; [*t2* | *t2* | ... | *t2*].

Użycie tej formy kombinatora ; jest uzasadnione tym, że w pierwszym z naszych podcelów musimy zaaplikować hipotezę *H*, a w drugim *H0* — w przeciwnym wypadku nasza taktyka zawiodłaby (sprawdź to). Ostatnie użycie tego kombinatora jest identyczne jak pierwsze — każdy z podcelów kończymy taktyką `assumption`.

Dzięki średnikowi dowód naszego twierdzenia skurczył się z trzech linijek do jednej, co jest wyśmienitym wynikiem — trzy razy mniej linii kodu to trzy razy mniejszy problem z jego utrzymaniem. Fakt ten ma jednak również i swoją ciemną stronę. Jest nią utrata interaktywności — wykonanie taktyki przeprowadza dowód od początku do końca.

Znalezienie odpowiedniego balansu między automatyzacją i interaktywnością nie jest sprawą łatwą. Dowodząc twierdzenia twoim pierwszym i podstawowym celem powinno być zawsze jego zrozumienie, co oznacza dowód mniej lub bardziej interaktywny, nieautomatyczny. Gdy uda ci się już udowodnić i zrozumieć dane twierdzenie, możesz przejść do automatyzacji. Proces ten jest analogiczny jak w przypadku inżynierii oprogramowania — najpierw tworzy się działający prototyp, a potem się go usprawnia.

Praktyka pokazuje jednak, że naszym ostatecznym celem powinna być pełna automatyzacja, tzn. sytuacja, w której dowód każdego twierdzenia (poza zupełnie banalnymi) będzie się sprowadzał, jak w powyższym przykładzie, do użycia jednej, specjalnie dla niego stworzonej taktyki. Ma to swoje uzasadnienie:

- zrozumienie cudzych dowodów jest zazwyczaj dość trudne, co ma spore znaczenie w większych projektach, w których uczestniczy wiele osób, z których część odchodzi, a na ich miejsce przychodzą nowe

- przebrnięcie przez dowód interaktywny, nawet jeżeli ma walory edukacyjne i jest oświecające, jest zazwyczaj czasochłonne, a czas to pieniądz

- skoro zrozumienie dowodu jest trudne i czasochłonne, to będziemy chcieli unikać jego zmieniania, co może nastąpić np. gdy będziemy chcieli dodać do systemu jakąś funkcjonalność i udowodnić, że po jej dodaniu system wciąż działa poprawnie

Ćwiczenie (średnik) Poniższe twierdzenia udowodnij najpierw z jak największym zrozumieniem, a następnie zautomatyzuj tak, aby całość była rozwiązywana w jednym kroku przez pojedynczą taktykę.

```
Lemma or_comm_ex :
  ∀ P Q : Prop, P ∨ Q → Q ∨ P.
```

```
Lemma diamond :
  ∀ P Q R S : Prop,
    (P → Q) ∨ (P → R) → (Q → S) → (R → S) → P → S.
```

2.9.2 || (alternatywa)

```
Lemma iff_intro''' :
  ∀ P Q : Prop,
    (P → Q) → (Q → P) → (P ↔ Q).
Proof.
  split; intros; apply H0 || apply H; assumption.
Qed.
```

Innym przydatnym kombinatorem jest ||, który będziemy nazywać alternatywą. Żeby wyjaśnić jego działanie, posłużymy się pojęciem postępu. Taktyka dokonuje postępu, jeżeli wygenerowany przez nią cel różni się od poprzedniego celu. Innymi słowy, taktyka nie dokonuje postępu, jeżeli nie zmienia obecnego celu. Za pomocą `progress` *t* możemy sprawdzić, czy taktyka *t* dokona postępu na obecnym celu.

Taktyka *t1* || *t2* używa na obecnym celu *t1*. Jeżeli *t1* dokona postępu, to *t1* || *t2* będzie miało taki efekt jak *t1* i skończy się sukcesem. Jeżeli *t1* nie dokona postępu, to na obecnym celu zostanie użyte *t2*. Jeżeli *t2* dokona postępu, to *t1* || *t2* będzie miało taki efekt jak *t2* i skończy się sukcesem. Jeżeli *t2* nie dokona postępu, to *t1* || *t2* zawiedzie i cel się nie zmieni.

W naszym przypadku w każdym z podcelów wygenerowanych przez `split`; `intros` próbujemy zaaplikować najpierw *H0*, a potem *H*. Na pierwszym podcelu `apply` *H0* zawiedzie (a więc nie dokona postępu), więc zostanie użyte `apply` *H*, które zmieni cel. Wobec tego `apply` *H0* || `apply` *H* na pierwszym podcelu będzie miało taki sam efekt, jak użycie `apply` *H*. W drugim podcelu `apply` *H0* skończy się sukcesem, więc tu `apply` *H0* || `apply` *H* będzie miało taki sam efekt, jak `apply` *H0*.

2.9.3 `idtac`, `do` oraz `repeat`

```
Lemma idtac_do_example :
  ∀ P Q R S : Prop,
    P → S ∨ R ∨ Q ∨ P.
Proof.
  idtac. intros. do 3 right. assumption.
Qed.
```

`idtac` to taktyka identycznościowa, czyli taka, która nic nic robi. Sama w sobie nie jest zbyt użyteczna, ale przydaje się do czasem do tworzenia bardziej skomplikowanych taktyk.

Kombinator `do` pozwala nam użyć danej taktyki określoną ilość razy. `do` n t na obecnym celu używa t. Jeżeli t zawiedzie, to `do` n t również zawiedzie. Jeżeli t skończy się sukcesem, to na każdym podcelu wygenerowanym przez t użyte zostanie `do` (n - 1) t. W szczególności `do` 0 t działa jak `idtac`, czyli kończy się sukcesem nic nie robiąc.

W naszym przypadku użycie taktyki `do 3 right` sprawi, że przy wyborze członu dysjunkcji, którego chcemy dowodzić, trzykrotnie pójdziemy w prawo. Zauważmy, że taktyka `do 4 right` zawiodłaby, gdyż po 3 użyciach `right` cel nie byłby już dysjunkcją, więc kolejne użycie `right` zawiodłoby, a wtedy cała taktyka `do 4 right` również zawiodłaby.

```
Lemma repeat_example :
  ∀ P A B C D E F : Prop,
    P → A ∨ B ∨ C ∨ D ∨ E ∨ F ∨ P.
Proof.
  intros. repeat right. assumption.
Qed.
```

Kombinator `repeat` powtarza daną taktykę, aż ta rozwiąże cel, zawiedzie, lub nie zrobi postępu. Formalnie: `repeat` t używa na obecnym celu taktyki t. Jeżeli t rozwiąże cel, to `repeat` t kończy się sukcesem. Jeżeli t zawiedzie lub nie zrobi postępu, to `repeat` t również kończy się sukcesem. Jeżeli t zrobi jakiś postęp, to na każdym wygenerowaym przez nią celu zostanie użyte `repeat` t.

W naszym przypadku `repeat right` ma taki efekt, że przy wyborze członu dysjunkcji wybieramy człon będący najbardziej na prawo, tzn. dopóki cel jest dysjunkcją, aplikowana jest taktyka `right`, która wybiera prawy człon. Kiedy nasz cel przestaje być dysjunkcją, taktyka `right` zawodzi i wtedy taktyka `repeat right` kończy swoje działanie sukcesem.

2.9.4 `try` i `fail`

```
Lemma iff_intro4 :
  ∀ P Q : Prop,
    (P → Q) → (Q → P) → (P ↔ Q).
Proof.
  split; intros; try (apply H0; assumption; fail);
  try (apply H; assumption; fail).
Qed.
```

`try` jest kombinatorem, który zmienia zachowanie przekazanej mu taktyki. Jeżeli t zawiedzie, to `try` t zadziała jak `idtac`, czyli nic nie zrobi i skończy się sukcesem. Jeżeli t skończy się sukcesem, to `try` t również skończy się sukcesem i będzie miało taki sam efekt, jak t. Tak więc, podobnie jak `repeat`, `try` nigdy nie zawodzi.

`fail` jest przeciwieństwem `idtac` — jest to taktyka, która zawsze zawodzi. Sama w sobie jest bezużyteczna, ale w tandemie z `try` oraz średnikiem daje nam pełną kontrolę nad tym, czy taktyka zakończy się sukcesem, czy zawiedzie, a także czy dokona postępu.

Częstym sposobem użycia `try` i `fail` jest `try` (t; `fail`). Taktyka ta na obecnym celu używa t. Jeżeli t rozwiąże cel, to `fail` nie zostanie wywołane i całe `try` (t; `fail`) zadziała tak jak t, czyli rozwiąże cel. Jeżeli t nie rozwiąże celu, to na wygenerowanych podcelach wywoływane zostanie `fail`, które zawiedzie — dzięki temu t; `fail` również zawiedzie, nie dokonując żadnych zmian w celu (nie dokona postępu), a całe `try` (t; `fail`) zakończy się sukcesem, również nie dokonując w celu żadnych zmian. Wobec tego działanie `try` (t; `fail`) można podsumować tak: "jeżeli t rozwiąże cel to użyj jej, a jeżeli nie, to nic nie rób".

Postaraj się dokładnie zrozumieć, jak opis ten ma się do powyższego przykładu — spróbuj usunąć jakieś `try`, `fail` lub średnik i zobacz, co się stanie.

Oczywiście przykład ten jest bardzo sztuczny — najlepszym pomysłem udowodnienia tego twierdzenia jest użycie ogólnej postaci średnika t; *t1* | ... | *tn*, tak jak w przykładzie *iff_intro''*. Idiom `try` (t; `fail`) najlepiej sprawdza się, gdy użycie średnika w ten sposób jest niepraktyczne, czyli gdy celów jest dużo, a rozwiązać automatycznie potrafimy tylko część z nich. Możemy użyć go wtedy, żeby pozbyć się prostszych przypadków nie zaśmiecając sobie jednak kontekstu w pozostałych przypadkach. Idiom ten jest też dużo bardziej odporny na przyszłe zmiany w programie, gdyż użycie go nie wymaga wiedzy o tym, ile podcelów zostanie wygenerowanych.

Przedstawione kombinatory są najbardziej użyteczne i stąd najpowszechniej używane. Nie są to jednak wszystkie kombinatory — jest ich znacznie więcej. Opisy taktyk i kombinatorów z biblioteki standardowej znajdziesz tu: https://coq.inria.fr/refman/tactic-index.html

2.10 Zadania

Poniższe zadania stanowią (chyba) kompletny zbiór praw rządzących logikami konstruktywną i klasyczną (w szczególności, niektóre z zadań mogą pokrywać się z ćwiczeniami zawartymi w tekście). Wróć do nich za jakiś czas, gdy czas przetrzebi trochę twoją pamięć (np. za tydzień).

Rozwiąż wszystkie zadania dwukrotnie: raz ręcznie, zaś za drugim razem w sposób zautomatyzowany.

2.10.1 Konstruktywny rachunek zdań

`Section` *exercises_propositional*.

`Hypotheses` P Q R S : `Prop`.

Komenda `Hypotheses` formalnie działa jak wprowadzenie aksjomatu, który w naszym przypadku brzmi "P, Q, R i S są zdaniami logicznymi". Taki aksjomat jest rzecz jasna zupełnie niegroźny, ale z innymi trzeba uważać — gdybyśmy wprowadzili aksjomat $1 = 2$, to popadlibyśmy w sprzeczność i nie moglibyśmy ufać żadnym dowodom, które przeprowadzamy.

Przemienność

`Lemma` *and_comm* :
  $P \wedge Q \rightarrow Q \wedge P$.

`Lemma` *or_comm* :
  $P \vee Q \rightarrow Q \vee P$.

Łączność

`Lemma` *and_assoc* :
  $P \wedge (Q \wedge R) \leftrightarrow (P \wedge Q) \wedge R$.

`Lemma` *or_assoc* :
  $P \vee (Q \vee R) \leftrightarrow (P \vee Q) \vee R$.

Rozdzielność

`Lemma` *and_dist_or* :
  $P \wedge (Q \vee R) \leftrightarrow (P \wedge Q) \vee (P \wedge R)$.

`Lemma` *or_dist_and* :
  $P \vee (Q \wedge R) \leftrightarrow (P \vee Q) \wedge (P \vee R)$.

`Lemma` *imp_dist_imp* :
  $(P \rightarrow Q \rightarrow R) \leftrightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$.

Kuryfikacja i dekuryfikacja

`Lemma` *curry* :
  $(P \wedge Q \rightarrow R) \rightarrow (P \rightarrow Q \rightarrow R)$.

`Lemma` *uncurry* :
  $(P \rightarrow Q \rightarrow R) \rightarrow (P \wedge Q \rightarrow R)$.

Prawa de Morgana

`Lemma` *deMorgan_1* :
  $\tilde{}(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$.

`Lemma` *deMorgan_2* :
  $\neg P \vee \neg Q \rightarrow \tilde{}(P \wedge Q)$.

Niesprzeczność i zasada wyłączonego środka

`Lemma` *noncontradiction'* :
  $\tilde{}(P \wedge \neg P)$.

Lemma *noncontradiction_v2* :
  $\neg (P \leftrightarrow \neg P)$.

Lemma *em_irrefutable* :
  ~~ $(P \lor \neg P)$.

Elementy neutralne i anihilujące

Lemma *and_false_annihilation* :
  $P \land False \leftrightarrow False$.

Lemma *or_false_neutral* :
  $P \lor False \leftrightarrow P$.

Lemma *and_true_neutral* :
  $P \land True \leftrightarrow P$.

Lemma *or_true_annihilation* :
  $P \lor True \leftrightarrow True$.

Inne

Lemma *or_imp_and* :
  $(P \lor Q \to R) \leftrightarrow (P \to R) \land (Q \to R)$.

Lemma *and_not_imp* :
  $P \land \neg Q \to$ ~$(P \to Q)$.

Lemma *or_not_imp* :
  $\neg P \lor Q \to (P \to Q)$.

Lemma *contraposition* :
  $(P \to Q) \to$ (~$Q \to \neg P)$.

Lemma *absurd* :
  $\neg P \to P \to Q$.

Lemma *impl_and* :
  $(P \to Q \land R) \to ((P \to Q) \land (P \to R))$.

End *exercises_propositional*.

Check *and_comm*.

```
(*  ===> forall P Q : Prop, P /\ Q -> Q /\ P *)
```

W praktyce komenda `Hypothesis` służy do tego, żeby za dużo nie pisać — po zamknięciu sekcji komendą `End`, Coq doda do każdego twierdzenia znajdującego się w tej sekcji kwantyfikację uniwersalną po hipotezach zadeklarowanych przy pomocy `Hypothesis`. W naszym przypadku Coq dodał do *and_comm* kwantyfikację po P i Q, mimo że nie napisaliśmy jej explicite.

2.10.2 Konstruktywny rachunek kwantyfikatorów

Section *QuantifiersExercises*.

Variable A : Type.
Hypotheses $P\ Q : A \to$ Prop.

Projekcje

Lemma *forall_and_proj1* :
$(\forall\ x : A, P\ x \land Q\ x) \to (\forall\ x : A, P\ x)$.

Lemma *forall_and_proj2* :
$(\forall\ x : A, P\ x \land Q\ x) \to (\forall\ x : A, Q\ x)$.

Rozdzielność

Lemma *forall_dist_and* :
$(\forall\ x : A, P\ x \land Q\ x) \leftrightarrow$
$(\forall\ x : A, P\ x) \land (\forall\ x : A, Q\ x)$.

Lemma *exists_dist_or* :
$(\exists\ x : A, P\ x \lor Q\ x) \leftrightarrow$
$(\exists\ x : A, P\ x) \lor (\exists\ x : A, Q\ x)$.

Lemma *ex_dist_and* :
$(\exists\ x : A, P\ x \land Q\ x) \to$
$(\exists\ y : A, P\ y) \land (\exists\ z : A, Q\ z)$.

Inne

Lemma *forall_or_imp* :
$(\forall\ x : A, P\ x) \lor (\forall\ x : A, Q\ x) \to$
$\forall\ x : A, P\ x \lor Q\ x$.

Lemma *forall_imp_imp* :
$(\forall\ x : A, P\ x \to Q\ x) \to$
$(\forall\ x : A, P\ x) \to (\forall\ x : A, Q\ x)$.

Lemma *forall_inhabited_nondep* :
$\forall\ R :$ Prop, $A \to ((\forall\ x : A, R) \leftrightarrow R)$.

Lemma *forall_or_nondep* :
$\forall\ R :$ Prop,
$(\forall\ x : A, P\ x) \lor R \to (\forall\ x : A, P\ x \lor R)$.

Lemma *forall_nondep_impl* :
$\forall\ R :$ Prop,

$(\forall\ x : A,\ R \to P\ x) \leftrightarrow (R \to \forall\ x : A,\ P\ x)$.

End *QuantifiersExercises*.

2.10.3 Klasyczny rachunek zdań (i kwantyfikatorów)

Section *ClassicalExercises*.

Require Import *Classical*.

Hypotheses $P\ Q\ R\ S$: Prop.

Komenda Require Import pozwala nam zaimportować żądany moduł z biblioteki standardowej Coqa. Dzięki temu będziemy mogli używać zawartych w nim definicji, twierdzeń etc.

Classical to moduł, który pozwala przeprowadzać rozumowania w logice klasycznej. Deklaruje on jako aksjomaty niektóre tautologie logiki klasycznej, np. zasadę wyłączonego środka, która tutaj nazywa się *classic*.

Check *classic*.

```
(*  ===> forall P : Prop, P \/ ~ P *)
```

Lemma *imp_and_or* : $(P \to Q \lor R) \to ((P \to Q) \lor (P \to R))$.

Lemma *deMorgan_2_conv* : $\tilde{}(P \land Q) \to \neg P \lor \neg Q$.

Lemma *not_imp* : $\tilde{}(P \to Q) \to P \land \neg Q$.

Lemma *imp_not_or* : $(P \to Q) \to (\tilde{}P \lor Q)$.

Lemma *material_implication* : $(P \to Q) \leftrightarrow (\tilde{}P \lor Q)$.

Lemma *contraposition_conv* : $(\tilde{}Q \to \neg P) \to (P \to Q)$.

Lemma *excluded_middle* : $P \lor \neg P$.

Lemma *peirce* : $((P \to Q) \to P) \to P$.

End *ClassicalExercises*.

2.11 Paradoks pijoka

Theorem *drinkers_paradox* :
  $\forall$ (*man* : Type) (*drinks* : *man* $\to$ Prop) (*random_guy* : *man*),
    $\exists$ *drinker* : *man*, *drinks drinker* $\to$
      $\forall$ *x* : *man*, *drinks x*.

Na zakończenie zwróćmy swą uwagę ku kolejnemu paradoksowi, tym razem dotyczącemu logiki klasycznej. Z angielska zwie się on "drinker's paradox", ja zaś ku powszechnej wesołości używał będę nazwy "paradoks pijoka".

Zazwyczaj jest on wyrażany mniej więcej tak: w każdym barze jest taki człowiek, że jeżeli on pije, to wszyscy piją. Jak to możliwe? Czy matematyka stwierdza istnienie magicznych ludzi zdolnych popaść swoich barowych towarzyszy w alkoholizm?

Oczywiście nie. W celu osiągnięcia oświecenia w tej kwestii prześledźmy dowód tego faktu (jeżeli nie udało ci się go wymyślić, pomyśl jeszcze trochę).

Ustalmy najpierw, jak ma się formalne brzmienie twierdzenia do naszej poetyckiej parafrazy dwa akapity wyżej. Początek "w każdym barze" możemy pominąć i sformalizować sytuację w pewnym konkretnym barze. Nie ma to znaczenia dla prawdziwości tego zdania.

Sytuację w barze modelujemy za pomocą typu *man*, które reprezentuje klientów baru, predykatu *drinks*, interpretowanego tak, że *drinks* x oznacza, że x pije. Pojawia się też osoba określona tajemniczym mianem *random_guy*. Jest ona konieczna, gdyż nasza poetycka parafraza czyni jedno założenie implicite: mianowicie, że w barze ktoś jest. Jest ono konieczne, gdyż gdyby w barze nie było nikogo, to w szczególności nie mogłoby tam być nikogo, kto spełnia jakieś dodatkowe warunki.

I tak docieramy do sedna sprawy: istnieje osoba, którą będziemy zwać pijokiem ($\exists$ *drinker* : *man*), taka, że jeżeli ona pije (*drinks drinker*), to wszyscy piją ($\forall$ x : *man*, *drinks* x).

Dowód jest banalny i opiera się na zasadzie wyłączonego środka, w Coqu zwanej *classic*. Dzięki niej możemy sprowadzić dowód do analizy dwóch przypadków.

Przypadek 1: wszyscy piją. Cóż, skoro wszyscy piją, to wszyscy piją. Pozostaje nam wskazać pijoka: mógłby to być ktokolwiek, ale z konieczności zostaje nim *random_guy*, gdyż do żadnego innego klienta baru nie możemy się odnieść.

Przypadek 2: nieprawda, że wszyscy piją. Parafrazując: istnieje ktoś, kto nie pije. Jest to obserwacja kluczowa. Skoro kolo przyszedł do baru i nie pije, to z automatu jest podejrzany. Uczyńmy go więc, wbrew zdrowemu rozsądkowi, naszym pijokiem.

Pozostaje nam udowodnić, że jeżeli pijok pije, to wszyscy piją. Załóżmy więc, że pijok pije. Wiemy jednak skądinąd, że pijok nie pije. Wobec tego mamy sprzeczność i wszyscy piją (a także jedzą naleśniki z betonem serwowane przez gadające ślimaki i robią dużo innych dziwnych rzeczy — wszakże *ex falso quodlibet*).

Gdzież więc leży paradoksalność całego paradoksu? Wynika ona w znacznej mierze ze znaczenia słowa "jeżeli". W mowie potocznej różni się ono znacznie od tzw. implikacji materialnej, w Coqu reprezentowanej (ale tylko przy założeniu reguły wyłączonego środka) przez implikację ($\rightarrow$).

Określenie "taka osoba, że jeżeli ona pije, to wszyscy piją" przeciętny człowiek interpretuje w kategoriach przyczyny i skutku, a więc przypisuje rzeczonej osobie magiczną zdolność zmuszania wszystkich do picia, tak jakby posiadała zdolność wznoszenia toastów za pomocą telepatii.

Jest to błąd, gdyż zamierzonym znaczeniem słowa jeżeli jest tutaj (ze względu na kontekst matematyczny) implikacja materialna. W jednym z powyższych ćwiczeń udowodniłeś, że w logice klasycznej mamy tautologię $P \rightarrow Q \leftrightarrow \neg P \lor Q$, a więc że implikacja jest prawdziwa gdy jej przesłanka jest fałszywa lub gdy jej konkluzja jest prawdziwa.

Do paradoksalności paradoksu swoje cegiełki dokładają też reguły logiki klasycznej (wyłączony środek) oraz logiki konstruktywnej (*ex falso quodlibet*), których w użyliśmy w dowodzie, a które dla zwykłego człowieka nie muszą być takie oczywiste.

Ćwiczenie (logika klasyczna) W powyższym dowodzie logiki klasycznej użyłem conajmniej dwukrotnie. Zacytuj wszystkie fragmenty dowodu wykorzystujące logikę klasyczną.

Ćwiczenie (niepusty bar) Pokaż, że założenie o tym, że w barze jest conajmniej jeden klient, jest konieczne. Co więcej, pokaż że stwierdzenie "w barze jest taki klient, że jeżeli on pije, to wszyscy piją" jest równoważne stwierdzeniu "w barze jest jakiś klient".

Które z tych dwóch implikacji wymagają logiki intuicjonistycznej, a które klasycznej?

```
Lemma dp_nonempty :
  ∀ (man : Type) (drinks : man → Prop),
    (∃ drinker : man, drinks drinker →
      ∀ x : man, drinks x) ↔
    (∃ x : man, True).
```

2.12 Ściąga

https://www.inf.ed.ac.uk/teaching/courses/tspl/cheatsheet.pdf

Zauważyłem palącą potrzebę istnienia krótkiej ściągi, dotyczącą podstaw logiki. Oto i ona:

- *True* to zdanie zawsze prawdziwe. Można je udowodnić za pomocą taktyki `trivial`. Można je też rozbić za pomocą `destruct`, ale nie jest to zbyt użyteczne.
- *False* to zdanie zawsze fałszywe. Można je udowodnić tylko jeżeli w kontekście już mamy jakiś inny (zazwyczaj zakamuflowany) dowód *False*. Można je rozbić za pomocą taktyki `destruct`, co kończy dowód, bo z fałszu wynika wszystko.
- $P \land Q$ to koniunkcja zdań P i Q. Aby ją udowodnić, używamy taktyki `split` i dowodzimy osobno P, a osobno Q. Jeżeli mamy w kontekście dowód na $P \land Q$, to za pomocą taktyki `destruct` możemy z niego wyciągnąć dowody na P i na Q.
- $P \lor Q$ to dysjunkcja zdań P i Q. Aby ją udowodnić, używamy taktyki `left` lub `right`, a następnie dowodzimy odpowiednio P albo Q. Jeżeli mamy w kontekście dowód $H : P \lor Q$, to możemy go rozbić za pomocą taktyki `destruct` H, co odpowiada rozumowaniu przez przypadki: musimy pokazać, że cel jest prawdziwy zarówno, gdy prawdziwe jest tylko P, jak i wtedy, gdy prawdziwe jest jedynie Q
- $P \to Q$ to zdanie "P implikuje Q". Żeby je udowodnić, używamy taktyki `intro` lub `intros`, które wprowadzają do kontekstu dowód na P będący założeniem. Jeżeli mamy w kontekście dowód $H : P \to Q$, to możemy dowieść Q za pomocą taktyki `apply` H, a następnie będziemy musieli udowodnić P. Jeżeli mamy w kontekście $H : P \to Q$ oraz $p : P$, to możemy uzyskać dowód $p : Q$ za pomocą taktyki `apply` H `in` p. Możemy uzyskać $H : Q$ za pomocą `specialize` $(H\ p)$

- $\neg$ P to negacja zdania P. Faktycznie jest to notacja na *not* P, które to samo jest skrótem oznaczającym $P \to$ *False*. Z negacją radzimy sobie za pomocą taktyki `unfold` *not* albo `unfold` *not* `in` ..., a następnie postępujemy jak z implikacją.

- $P \leftrightarrow Q$ to równoważność zdań P i Q. Jest to notacja na *iff* P Q, które jest skrótem od $(P \to Q) \land (Q \to P)$. Radzimy sobie z nią za pomocą taktyk `unfold` *iff* oraz `unfold` *iff* `in` ...

- $\forall$ x : A, P x to zdanie mówiące "dla każdego x typu A zachodzi P x". Postępujemy z nim tak jak z implikacją, która jest jego specjalnym przypadkiem.

- $\exists$ x : A, P x to zdanie mówiące "istnieje taki x typu A, który spełnia P". Dowodzimy go za pomocą taktyki $\exists$ a, a następnie musimy pokazać P a. Jeżeli mamy taki dowód w kontekście, możemy rozbić go na a i P a za pomocą taktyki `destruct`.

2.13 Konkluzja

W niniejszym rozdziale zapoznaliśmy się z logiką konstruktywną. Poznaliśmy jej składnię, interpretację obliczeniową, nauczyliśmy się dowodzić w systemie dedukcji naturalnej oraz dowiedzieliśmy się, jak to wszystko zrealizować w Coqu. Poznaliśmy też kombinatory taktyk, dzięki którym możemy skrócić i uprościć nasze formalne dowody.

Zapoznaliśmy się też z logiką klasyczną i jej interpretacją. Poznaliśmy też dwa paradoksy związane z różnicami w interpretacji zdań w języku naturalnym oraz zdań matematycznych. Jeden z paradoksów dobrze pokazał nam w praktyce, na czym polega różnica między logiką konstruktywną i klasyczną.

Skoro potrafimy już co nieco dowodzić, a także wiemy, że nasze metody nie nadają się do rozumowania o pieniądzach ani kebabach, nadszedł czas zapoznać się z jakimiś bytami, o których moglibyśmy czegoś dowieść — w następnym rozdziale zajmiemy się na poważnie typami, programami i obliczeniami oraz udowadnianiem ich właściwości.

Rozdział 3

R2: Indukcja i rekursja

W poprzednim rozdziale dowiedzieliśmy się już co nieco o typach, a także spotkaliśmy kilka z nich oraz kilka sposobów tworzenia nowych typów ze starych (takich jak np. koniunkcja; pamiętaj, że zdania są typami). W tym rozdziale dowiemy się o nich nieco więcej: spotkamy się z ich sortami oraz uniwersami, w których żyją; dowiemy się, jak definiować nowe typy przy pomocy indukcji oraz jak użyć rekursji do tworzenia funkcji, które konstruują i dekonstruują ich termy.

3.1 Sorty

Jeżeli przeczytałeś uważnie sekcję "Typy i termy" z poprzedniego rozdziału, zauważyłeś zapewne stwierdzenie, że typy są termami. W połączeniu ze stwierdzeniem, że każdy term ma swój typ, zrodzić musi się pytanie: jakiego typu są typy? Zacznijmy od tego, że żeby uniknąć używania mało poetyckiego określenia "typy typów", typy typów nazywamy sortami.

`Prop`, jak już wiesz, jest sortem zdań logicznych. Jeżeli $x : A$ oraz A : `Prop` (tzn. A jest sortu `Prop`), to typ A możemy interpretować jako zdanie logiczne, a term x jako jego dowód. Na przykład I jest dowodem zdania *True*, tzn. I : *True*, zaś term 42 nie jest dowodem *True*, gdyż 42 : *nat*.

```
Check True.
(*  ===> True : Prop *)

Check I.
(*  ===> I : True *)

Check 42.
(*  ===> 42 : nat *)
```

O ile jednak każde zdanie logiczne jest typem, nie każdy typ jest zdaniem — przykładem niech będą liczby naturalne *nat*. Sortem *nat* jest `Set`. Niech nie zmyli cię ta nazwa: `Set` nie ma nic wspólnego ze zbiorami znanymi choćby z teorii zbiorów ZF.

`Set` jest sortem, w którym żyją specyfikacje. Jeżeli $x : A$ oraz A : `Set` (tzn. sortem A jest `Set`), to A możemy interpretować jako specyfikację pewnej klasy programów, a term x jako program, który tę specyfikację spełnia (implementuje). Na przykład $2 + 2$ jest programem, ktory spełnia specyfikację *nat*, tzn. $2 + 2$: *nat*, zaś `fun` n : *nat* $\Rightarrow$ n nie spełnia specyfikacji *nat*, gdyż `fun` n : *nat* $\Rightarrow$ n : *nat* $\rightarrow$ *nat*.

```
Check nat.
(*  ===> nat : Set *)

Check 2 + 2.
(*  ===> 2 + 2 : nat *)

Check fun n : nat => n.
(*  fun n : nat => n : nat -> nat *)
```

Oczywiście w przypadku typu *nat* mówiene o specyfikacji jest trochę na wyrost, gdyż określenie "specyfikacja" kojarzy nam się z czymś, co określa właściwości, jakie powinien mieć spełniający ją program. O takich specyfikacjach dowiemy się więcej w kolejnych rozdziałach. Choć każda specyfikacja jest typem, to rzecz jasna nie każdy typ jest specyfikacją — niektóre typy są przecież zdaniami.

3.2 Hierarchia uniwersów

Uwaga: ta sekcja jest czysto teoretyczna. Jeżeli boisz się uprawiania teorii dla samej teorii, możesz ją pominąć.

Jeżeli czytasz uważnie, to pewnie wciąż czujesz niedosyt — wszakże sorty, jako typy, także są termami. Jakie są więc typy/sorty sortów? Przekonajmy się.

```
Check Prop.
(*  ===> Prop : Type *)

Check Set.
(*  ===> Set : Type *)
```

`Prop` oraz `Set` są typu/sortu `Type`, który bywa też nazywany uniwersum. To stwierdzenie wciąż jednak pewnie nie zaspakaja twojej ciekawości. Pójdźmy więc po nitce do kłębka.

```
Check Type.
(*  ===> Type : Type *)
```

Zdaje się, że osiągnęliśmy kłębek i że `Type` jest typu `Type`. Rzeczywistość jest jednak o wiele ciekawsza. Gdyby rzeczywiście zachodziło `Type` : `Type`, doszłoby do paradoksu znanego jako paradoks Girarda (którego omówienie jednak pominiemy). Prawda jest inna.

```
(*  Set Printing Universes. *)
```

Uwaga: powyższa komenda zadziała jedynie w konsoli (program coqtop). Aby osiągnąć ten sam efekt w CoqIDE, zaznacz opcję View > Display universe levels.

```
Check Type.
(*  ===> Type (*  Top.7 *)  : Type (*  (Top.7)+1 *)  *)
```

Co oznacza ten dziwny napis? Otóż w Coqu mamy do czynienia nie z jednym, ale z wieloma (a nawet nieskończenie wieloma) uniwersami. Uniwersa te są numerowane liczbami naturalnymi: najniższe uniwersum ma numer 0, a każde kolejne o jeden większy. Wobec tego hierarchia uniwersów wygląda tak (użyta notacja nie jest tą, której używa Coq; została wymyślona ad hoc):

- `Set` jest typu/sortu `Type`(0)
- `Type`(0) jest typu/sortu `Type`(1)
- w ogólności, `Type`(i) jest typu/sortu `Type`($i + 1$)

Aby uniknąć paradoksu, definicje odnoszące się do typów żyjących na różnych poziomach hierarchii muszą same bytować w uniwersum na poziomie wyższym niż każdy z tych, do których się odwołują. Aby to zapewnić, Coq musi pamiętać, na którym poziomie znajduje każde użycie `Type` i odpowiednio dopasowywać poziom hierarchii, do którego wrzucone zostaną nowe definicje.

Co więcej, w poprzednim rozdziale dopuściłem się drobnego kłamstewka twierdząc, że każdy term ma dokładnie jeden typ. W pewnym sensie nie jest tak, gdyż powyższa hierarcha jest *kumulatywna* — znaczy to, że jeśli A : `Type`(i), to także A : `Type`(j) dla i $<$ j. Tak więc każdy typ, którego sortem jest `Type`, nie tylko nie ma unikalnego typu/sortu, ale ma ich nieskończenie wiele.

Brawo! Czytając tę sekcję, dotarłeś do króliczej nory i posiadłeś wiedzę tajemną, której prawie na pewno nigdy ani nigdzie nie użyjesz. Możemy zatem przejść do meritum.

3.3 Typy induktywne

W Coqu są trzy główne rodzaje typów: produkt zależny, typy induktywne i typy koinduktywne. Z pierwszym z nich już się zetknęliśmy, drugi poznamy w tym rozdziale, trzeci na razie pominiemy.

Typ induktywny definiuje się przy pomocy zbioru konstruktorów, które służą, jak sama nazwa wskazuje, do budowania termów tego typu. Konstruktory te są funkcjami (być może zależnymi), których przeciwdziedziną jest definiowany typ, ale niczego nie obliczają — nadają jedynie termom ich "kształt". W szczególności, nie mają nic wspólnego z konstruktorami w takich językach jak C++ lub Java — nie mogą przetwarzać swoich argumentów, alokować pamięci, dokonywać operacji wejścia/wyjścia etc.

Tym, co jest ważne w przypadku konstruktorów, jest ich ilość, nazwy oraz ilość i typy przyjmowanych argumentów. To te cztery rzeczy decydują o tym, jakie "kształty" będą miały termy danego typu, a więc i czym będzie sam typ. W ogolności każdy term jest skończonym, ukorzenionym drzewem, którego kształt zależy od charakterystyki konstruktorów tak:

- każdy konstruktor to inny rodzaj węzła (nazwa konstruktora to nazwa węzła)
- konstruktory nierekurencyjne to liście, a rekurencyjne — węzły wewnętrzne

- argumenty konstruktorów to dane przechowywane w danym węźle

Typ induktywny można wyobrażać sobie jako przestrzeń zawierającą te i tylko te drzewa, które można zrobić przy pomocy jego konstruktorów. Nie przejmuj się, jeżeli opis ten wydaje ci się dziwny — sposób definiowania typów induktywnych i ich wartości w Coqu jest diametralnie różny od sposobu definiowania klas i obiektów w językach imperatywnych i wymaga przyzwyczajenia się do niego. Zobaczmy, jak powyższy opis ma się do konkretnych przykładów.

3.3.1 Enumeracje

Najprostszym rodzajem typów induktywnych są enumeracje, czyli typy, których wszystkie konstruktory są stałymi.

```
Inductive bool : Set :=
    | true : bool
    | false : bool.
```

Definicja typu induktywnego ma następującą postać: najpierw występuje słowo kluczowe `Inductive`, następnie nazwa typu, a po dwukropku sort (`Set`, `Prop` lub `Type`). Następnie wymieniamy konstruktory typu — dla czytelności każdy w osobnej linii. Mają one swoje unikalne nazwy i są funkcjami, których przeciwdziedziną jest definiowany typ. W naszym przypadku mamy 2 konstruktory, zwane *true* oraz *false*, które są funkcjami zeroargumentowymi.

Definicję tę możemy udczytać następująco: "*true* jest typu *bool*, *false* jest typu *bool* i nie ma żadnych więcej wartości typu *bool*".

Uwaga: należy odróżnić symbole := oraz =. Pierwszy z nich służy do definiowania, a drugi do zapisywania równości.

Zapis *name* := *term* oznacza "niech od teraz *name* będzie inną nazwą dla *term*". Jest to komenda, a nie zdanie logiczne. Od teraz jeżeli natkniemy się na nazwę *name*, będziemy mogli odwinąć jej definicję i wstawić w jej miejsce *term*. Przykład: `Definition` *five* := 5. Antyprzykład: 2 := 5 (błąd składni).

Zapis $a = b$ oznacza "a jest równe b". Jest to zdanie logiczne, a nie komenda. Zdanie to rzecz jasna nie musi być prawdziwe. Przykład: $2 = 5$. Antyprzykład: *five* = 5 (jeżeli *five* nie jest zdefiniowane, to dostajemy komunikat w stylu "nie znaleziono nazwy *five*").

```
Definition negb (b : bool) : bool :=
match b with
    | true ⇒ false
    | false ⇒ true
end.
```

Definicja funkcji wygląda tak: najpierw mamy słowo kluczowe `Definition` (jeżeli funkcja nie jest rekurencyjna), następnie argumenty funkcji w postaci (*name* : `type`); po dwukropku przeciwdziedzina, a po symbolu := ciało funkcji.

Podstawowym narzędziem służącym do definiowania funkcji jest "dopasowywanie do wzorca" (ang. pattern matching; w dalszej części będę używał nazwy angielskiej). Pozwala ono sprawdzić, którego konstruktora użyto do zrobienia dopasowywanej wartości. Podobnym tworem występującym w językach imperatywnych jest instrukcja switch, ale pattern matching jest od niej dużo potężniejszy.

Nasza funkcja działa następująco: sprawdzamy, którym konstruktorem zrobiono argument *b* — jeżeli było to *true*, zwracamy *false*, a gdy było to *false*, zwracamy *true*.

Ćwiczenie (*andb* i *orb*) Zdefiniuj funkcje *andb* (koniunkcja boolowska) i *orb* (alternatywa boolowska).

Zanim odpalimy naszą funkcję, powinniśmy zadać sobie pytanie: w jaki sposób programy są wykonywane? W celu lepszego zrozumienia tego zagadnienia porównamy ewaluację programów napisanych w językach imperatywnych oraz funkcyjnych.

Rozważmy bardzo uproszczony model: interpreter wykonuje program, który nie dokonuje operacji wejścia/wyjścia, napisany w jakimś języku imperatywnym. W tej sytuacji działanie interpretera sprowadza się do tego, że czyta on kod programu instrukcja po instrukcji, a następnie w zależności od przeczytanej instrukcji odpowiednio zmienia swój stan.

W języku czysto funkcyjnym taki model ewaluacji jest niemożliwy, gdyż nie dysponujemy globalnym stanem. Zamiast tego, w Coqu wykonanie programu polega na jego redukcji. O co chodzi? Przypomnijmy najpierw, że program to term, którego typem jest specyfikacja, czyli typ sortu `Set`. Termy mogą być redukowane, czyli zamieniane na równoważne, ale prostsze, przy użyciu reguł redukcji. Prześledźmy wykonanie programu *negb true* krok po kroku.

```
Eval cbv delta in negb true.
(*  ===> = (fun b : bool => match b with
                   | true => false
                   | false => true
                   end) true
        : bool *)
```

Redukcja delta odwija definicje. Żeby użyć jakiejś redukcji, używamy komendy `Eval cbv` *redukcje* `in` *term*.

```
Eval cbv delta beta in negb true.
(*  ===> = match true with
           | true => false
           | false => true
           end
        : bool *)
```

Redukcja beta podstawia argument do funkcji.

```
Eval cbv delta beta iota in negb true.
(*  === = false : bool *)
```

Redukcja jota dopasowuje term do wzorca i zamienia go na term znajdujący się po prawej stronie ⇒ dla dopasowanego przypadku.

```
Eval cbv in negb true.
(*  ===> = false : bool *)
```

Żeby zredukować term za jednym zamachem, możemy pominąć nazwy redukcji występujące po `cbv` — w taki wypadku zostaną zaaplikowane wszystkie możliwe redukcje, czyli program zostanie wykonany. Do jego wykonania możemy też użyć komend `Eval simpl` oraz `Eval compute` (a od wersji Coqa 8.5 także `Eval` *cbn*).

Ćwiczenie (redukcja) Zredukuj "ręcznie" programy *andb false false* oraz *orb false true*.

Jak widać, wykonanie programu w Coqu jest dość toporne. Wynika to z faktu, że Coq nie został stworzony do wykonywania programów, a jedynie do ich definiowania i dowodzenia ich poprawności. Aby użyć programu napisanego w Coqu w świecie rzeczywistym, stosuje się zazwyczaj mechanizm ekstrakcji, który pozwala z programu napisanego w Coqu automatycznie uzyskać program w Scheme, OCamlu lub Haskellu, które są od Coqa dużo szybsze i dużo powszechniej używane. My jednak nie będziemy się tym przejmować.

Zdefiniowawszy naszą funkcję, możemy zadać sobie pytanie: czy definicja jest poprawna? Gdybyśmy pisali w jednym z języków imperatywnych, moglibyśmy napisać dla niej testy jednostkowe, polegające np. na tym, że generujemy losowo wejście funkcji i sprawdzamy, czy wynik posiada żądaną przez nas właściwość. Coq umożliwia nam coś dużo silniejszego: możemy wyrazić przy pomocy twierdzenia, że funkcja posiada interesującą nas własność, a następnie spróbować je udowodnić. Jeżeli nam się powiedzie, mamy całkowitą pewność, że funkcja rzeczywiście posiada żądaną własność.

```
Theorem negb_involutive :
  ∀ b : bool, negb (negb b) = b.
Proof.
  intros. destruct b.
    simpl. reflexivity.
    simpl. reflexivity.
Qed.
```

Nasze twierdzenie głosi, że funkcja *negb* jest inwolucją, tzn. że dwukrotne jej zaaplikowanie do danego argumentu nie zmienia go, lub też, innymi słowy, że *negb* jest swoją własną odwrotnością.

Dowód przebiega w następujący sposób: taktyką `intro` wprowadzamy zmienną *b* do kontekstu, a następnie używamy taktyki `destruct`, aby sprawdzić, którym konstruktorem została zrobiona. Ponieważ typ *bool* ma dwa konstruktory, taktyka ta generuje nam dwa podcele: musimy udowodnić twierdzenie osobno dla przypadku, gdy *b* = *true* oraz dla *b* = *false*. Potem przy pomocy taktyki `simpl` redukujemy (czyli wykonujemy) programy *negb* (*negb true*) i *negb* (*negb false*). Zauważ, że byłoby to niemożliwe, gdyby argument był postaci *b* (nie można wtedy zaaplikować żadnej redukcji), ale jest jak najbardziej możliwe, gdy jest on postaci *true* albo *false* (wtedy redukcja przebiega jak w przykładzie). Na koniec używamy taktyki `reflexivity`, która potrafi udowodnić cel postaci $a = a$.

`destruct` jest taktykowym odpowiednikiem pattern matchingu i bardzo często jest używany, gdy mamy do czynienia z czymś, co zostało za jego pomocą zdefiniowane.

Być może poczułeś dyskomfort spowodowany tym, że cel postaci $a = a$ można udowodnić dwoma różnymi taktykami (`reflexivity` oraz `trivial`) albo że termy można redukować na cztery różne sposoby (`Eval simpl`, `Eval compute`, `Eval cbv`, `Eval` *cbn*). Niestety, będziesz musiał się do tego przyzwyczaić — język taktyka Coqa jest strasznie zabałaganiony i niesie ze sobą spory bagaż swej mrocznej przeszłości. Na szczęście od niedawna trwają prace nad jego ucywilizowaniem, czego pierwsze efekty widać już od wersji 8.5. W chwilach desperacji uratować może cię jedynie dokumentacja:

- https://coq.inria.fr/refman/tactic-index.html
- https://coq.inria.fr/refman/Reference-Manual010.html

```
Theorem negb_involutive' :
  ∀ b : bool, negb (negb b) = b.
Proof.
  destruct b; simpl; reflexivity.
Qed.
```

Zauważmy, że nie musimy używać taktyki `intro`, żeby wprowadzić b do kontekstu: taktyka `destruct` sama jest w stanie wykryć, że nie ma go w kontekście i wprowadzić je tam przed rozbiciem go na konstruktory. Zauważmy też, że oba podcele rozwiązaliśmy w ten sam sposób, więc możemy użyć kombinatora ; (średnika), żeby rozwiązać je oba za jednym zamachem.

Ćwiczenie (logika boolowska) Udowodnij poniższe twierdzenia.

```
Theorem andb_assoc :
  ∀ b1 b2 b3 : bool,
    andb b1 (andb b2 b3) = andb (andb b1 b2) b3.

Theorem andb_comm :
  ∀ b1 b2 : bool,
    andb b1 b2 = andb b2 b1.

Theorem orb_assoc :
  ∀ b1 b2 b3 : bool,
    orb b1 (orb b2 b3) = orb (orb b1 b2) b3.

Theorem orb_comm :
  ∀ b1 b2 : bool,
    orb b1 b2 = orb b2 b1.

Theorem andb_true_elim :
  ∀ b1 b2 : bool,
    andb b1 b2 = true → b1 = true ∧ b2 = true.
```

Ćwiczenie (róża kierunków) `Module` *Directions.*

Zdefiniuj typ opisujący kierunki podstawowe (północ, południe, wschód, zachód - dodatkowe punkty za nadanie im sensownych nazw).

Zdefiniuj funkcje *turnL* i *turnR*, które reprezentują obrót o 90 stopni przeciwnie/zgodnie z ruchem wskazówek zegara. Sformułuj i udowodnij twierdzenia mówiące, że:

- obrót cztery razy w lewo/prawo niczego nie zmienia
- obrót trzy razy w prawo to tak naprawdę obrót w lewo (jest to tzw. pierwsze twierdzenie korwinizmu)
- obrót trzy razy w lewo to obrót w prawo (jest to tzw. drugie twierdzenie korwinizmu)
- obrót w prawo, a potem w lewo niczego nie zmienia
- obrót w lewo, a potem w prawo niczego nie zmienia
- każdy kierunek to północ, południe, wschód lub zachód (tzn. nie ma innych kierunków)

Zdefiniuj funkcję *opposite*, które danemu kierunkowi przyporządkowuje kierunek do niego przeciwny (czyli północy przyporządkowuje południe etc.). Wymyśl i udowodnij jakąś ciekawę specyfikację dla tej funkcji (wskazówka: powiąż ją z *turnL* i *turnR*).

Zdefiniuj funkcję *is_opposite*, która bierze dwa kierunki i zwraca *true*, gdy są one przeciwne oraz *false* w przeciwnym wypadku. Wymyśl i udowodnij jakąś specyfikację dla tej funkcji. Wskazówka: jakie są jej związku z *turnL*, *turnR* i *opposite*?

Pokaż, że funkcje *turnL*, *turnR* oraz *opposite* są injekcjami i surjekcjami (co to dokładnie znacz, dowiemy się później). Uwaga: to zadanie wymaga użyci taktyki `inversion`, która jest opisana w podrozdziale o polimorfizmie.

`Lemma` *turnL_inj* :
  $\forall\ x\ y : D,\ turnL\ x = turnL\ y \to x = y.$

`Lemma` *turnR_inj* :
  $\forall\ x\ y : D,\ turnR\ x = turnR\ y \to x = y.$

`Lemma` *opposite_inj* :
  $\forall\ x\ y : D,\ opposite\ x = opposite\ y \to x = y.$

`Lemma` *turnL_sur* :
  $\forall\ y : D,\ \exists\ x : D,\ turnL\ x = y.$

`Lemma` *turnR_sur* :
  $\forall\ y : D,\ \exists\ x : D,\ turnR\ x = y.$

`Lemma` *opposite_sur* :
  $\forall\ y : D,\ \exists\ x : D,\ opposite\ x = y.$

`End` *Directions.*

Ćwiczenie (różne enumeracje) TODO Zdefiniuj typy induktywne reprezentujące:

- dni tygodnia
- miesiące
- kolory podstawowe systemu RGB

Wymyśl do nich jakieś ciekawe funkcje i twierdzenia.

3.3.2 Konstruktory rekurencyjne

Powiedzieliśmy, że termy typów induktywnych są drzewami. W przypadku enumeracji jest to słabo widoczne, gdyż drzewa te są zdegenerowane — są to po prostu punkciki oznaczone nazwami konstruktorów. Efekt rozgałęzienia możemy uzyskać, gdy jeden z konstruktorów będzie rekurencyjny, tzn. gdy jako argument będzie przyjmował term typu, który właśnie definiujemy. Naszym przykładem będą liczby naturalne (choć i tutaj rozgałęzienie będzie nieco zdegenerowane - każdy term będzie mógł mieć co najwyżej jedno).

```
Module NatDef.
```

Mechanizm modułów jest podobny do mechanizmu sekcji i na razie nie będzie nas interesował — użyjemy go, żeby nie zaśmiecać głównej przestrzeni nazw (mechanizm sekcji w tym przypadku by nie pomógł).

```
Inductive nat : Set :=
    | O : nat
    | S : nat → nat.

Notation "0":= O.
```

Uwaga: nazwa pierwszego konstruktora to duża litera O, a nie cyfra 0 — cyfry nie mogą być nazwami. Żeby obejść tę niedogodność, musimy posłużyć się mechanizmem notacji — dzięki temu będziemy mogli pisać 0 zamiast O.

Definicję tę możemy odczytać w następujący sposób: "0 jest liczbą naturalną; jeżeli n jest liczbą naturalną, to S n również jest liczbą naturalną". Konstruktor S odpowiada tutaj dodawaniu jedynki: S 0 to 1, S (S 0) to 2, S (S (S 0)) to 3 i tak dalej.

```
Check (S (S (S 0))).
(*  ===> S (S (S 0)) : nat *)

End NatDef.

Check S (S (S 0)).
(*  ===> 3 : nat *)
```

Ręczne liczenie ilości S w każdej liczbie szybko staje się trudne nawet dla małych liczb. Na szczęście standardowa biblioteka Coqa udostępnia notacje, które pozwalają nam zapisywać liczby naturalne przy pomocy dobrze znanych nam cyfr arabskich. Żeby uzyskać do nich dostęp, musimy opuścić zdefiniowany przez nas moduł *NatDef*, żeby nasza definicja *nat* nie przysłaniała tej bibliotecznej. Zaczniemy za to nowy moduł, żebyśmy mogli swobodnie zredefiniować działania na liczbach naturalnych z biblioteki standardowej.

```
Module NatOps.

Fixpoint plus (n m : nat) : nat :=
match n with
    | 0 => m
    | S n' => S (plus n' m)
end.
```

W zapisie unarnym liczby naturalne możemy wyobrażać sobie jako kupki S-ów, więc dodawanie dwóch liczb sprowadza się do przerzucenia S-ów z jednej kupki na drugą.

Definiowanie funkcji dla typów z konstruktorami rekurencyjnymi wygląda podobnie jak dla enumeracji, ale występują drobne różnice: jeżeli będziemy używać rekurencji, musimy zaznaczyć to za pomocą słowa kluczowego `Fixpoint` (zamiast wcześniejszego `Definition`). Zauważmy też, że jeżeli funkcja ma wiele argumentów tego samego typu, możemy napisać (*arg1* ... *argN* : `type`) zamiast dłuższego (*arg1* : `type`) ... (*argN* : `type`).

Jeżeli konstruktor typu induktywnego bierze jakieś argumenty, to wzorce, które go dopasowują, stają się nieco bardziej skomplikowane: poza nazwą konstruktora muszą też dopasowywać argumenty — w naszym przypadku używamy zmiennej o nazwie *n'*, która istnieje tylko lokalnie (tylko we wzorcu dopasowania oraz po prawej stronie strzałki ⇒).

Naszą funkcję zdefiniowaliśmy przy pomocy najbardziej elementarnego rodzaju rekursji, jaki jest dostępny w Coqu: rekursji strukturalnej. W przypadku takiej rekursji wywołania rekurencyjne mogą odbywać się jedynie na termach strukturalnie mniejszych, niż obecny argument główny rekurencji. W naszym przypadku argumentem głównym jest n (bo on jest dopasowywany), zaś rekurencyjnych wywołań dokonujemy na *n'*, gdzie $n = S$ *n'*. *n'* jest strukturalnie mniejszy od S *n'*, gdyż składa się z jednego S mniej. Jeżeli wyobrazimy sobie nasze liczby jako kupki S-ów, to wywołania rekurencyjne możemy robić jedynie po zdjęciu z kupki co najmniej jednego S.

Ćwiczenie (rekursja niestrukturalna) Wymyśl funkcję z liczb naturalnych w liczby naturalne, która jest rekurencyjna, ale nie jest strukturalnie rekurencyjna. Precyzyjniej pisząc: później okaże się, że wszystkie formy rekurencji to tak naprawdę rekursja strukturalna pod przykrywką. Wymyśl taką definicję, która na pierwszy rzut oka nie jest strukturalnie rekurencyjna.

Podobnie jak w przypadku funkcji *negb*, tak i tym razem w celu sprawdzenia poprawności naszej definicji spróbujemy udowodnić, że posiada ona właściwości, których się spodziewamy.

```
Theorem plus_O_n :
  ∀ n : nat, plus 0 n = n.
Proof.
```

```
  intro. simpl. trivial.
Qed.
```

Tak prosty dowód nie powinien nas dziwić — wszakże twierdzenie to wynika wprost z definicji (spróbuj zredukować "ręcznie" wyrażenie $0 + n$).

```
Theorem plus_n_O_try1 :
  ∀ n : nat, plus n 0 = n.
Proof.
  intro. destruct n.
    trivial.
    simpl. f_equal.
Abort.
```

Tym razem nie jest już tak prosto — $n + 0 = n$ nie wynika z definicji przez prostą redukcję, gdyż argumentem głównym funkcji *plus* jest jej pierwszy argument, czyli n. Żeby móc zredukować to wyrażenie, musimy rozbić n. Pokazanie, że $0 + 0 = 0$ jest trywialne, ale drugi przypadek jest już beznadziejny. Ponieważ funkcje zachowują równość (czyli $a = b$ implikuje $f\ a = f\ b$), to aby pokazać, że $f\ a = f\ b$, wystarczyć pokazać, że $a = b$ — w ten właśnie sposób działa taktyka `f_equal`. Nie pomogła nam ona jednak — po jej użyciu mamy do pokazania to samo, co na początku, czyli $n + 0 = n$.

```
Theorem plus_n_O :
  ∀ n : nat, plus n 0 = n.
Proof.
  intro. induction n.
    trivial.
    simpl. f_equal. assumption.
Qed.
```

Do udowodnienia tego twierdzenia musimy posłużyć się indukcją. Indukcja jest sposobem dowodzenia właściwości typów induktywnych i funkcji rekurencyjnych, który działa mniej więcej tak: żeby udowodnić, że każdy term typu A posiada własność P, pokazujemy najpierw, że konstruktory nierekurencyjne posiadają tę własność dla dowolnych argumentów, a następnie, że konstruktory rekurencyjne zachowują tę własność.

W przypadku liczb naturalnych indukcja wygląda tak: żeby pokazać, że każda liczba naturalna ma własność P, najpierw należy pokazać, że zachodzi $P\ 0$, a następnie, że jeżeli zachodzi $P\ n$, to zachodzi także $P\ (S\ n)$. Z tych dwóch reguł możemy zbudować dowód na to, że $P\ n$ zachodzi dla dowolnego n.

Ten sposób rozumowania możemy zrealizować w Coqu przy pomocy taktyki `induction`. Działa ona podobnie do `destruct`, czyli rozbija podany term na konstruktory, ale w przypadku konstruktorów rekurencyjnych robi coś jeszcze — daje nam założenie indukcyjne, które mówi, że dowodzone przez nas twierdzenie zachodzi dla rekurencyjnych argumentów konstruktora. Właśnie tego było nam trzeba: założenie indukcyjne pozwala nam dokończyć dowód.

```
Theorem plus_comm :
```

```
  ∀ n m : nat, plus n m = plus m n.
Proof.
  induction n as [| n']; simpl; intros.
    rewrite plus_n_O. trivial.
    induction m as [| m'].
      simpl. rewrite plus_n_O. trivial.
      simpl. rewrite IHn'. rewrite ← IHm'. simpl. rewrite IHn'.
        trivial.
Qed.
```

Pojedyncza indukcja nie zawsze wystarcza, co obrazuje powyższy przypadek. Zauważmy, że przed użyciem `induction` nie musimy wprowadzać zmiennych do kontekstu — taktyka ta robi to sama, podobnie jak `destruct`. Również podobnie jak `destruct`, możemy przekazać jej wzorzec, którym nadajemy nazwy argumentom konstruktorów, na które rozbijany jest term.

W ogólności wzorzec ma postać [*a11* .. *a1n* | ... | *am1* .. *amk*]. Pionowa kreska oddziela argumenty poszczególnych konstruktorów: *a11* .. *a1n* to argumenty pierwszego konstruktora, zaś *am1* .. *amk* to argumenty m-tego konstruktora. *nat* ma dwa konstruktory, z czego pierwszy nie bierze argumentów, a drugi bierze jeden, więc nasz wzorzec ma postać [| *n'*]. Dzięki temu nie musimy polegać na domyślnych nazwach nadawanych argumentom przez Coqa, które często wprowadzają zamęt.

Jeżeli damy taktyce `rewrite` nazwę hipotezy lub twierdzenia, którego konkluzją jest $a = b$, to zamienia ona w obecnym podcelu wszystkie wystąpienia a na b oraz generuje tyle podcelów, ile przesłanek ma użyta hipoteza lub twierdzenie. W naszym przypadku użyliśmy udowodnionego uprzednio twierdzenia *plus_n_O*, które nie ma przesłanek, czego efektem było po prostu przepisanie *plus* m 0 na m.

Przepisywać możemy też w drugą stronę pisząc `rewrite` ←. Wtedy jeżeli konkluzją danego `rewrite` twierdzenia lub hipotezy jest $a = b$, to w celu wszystkie b zostaną zastąpione przez a.

Ćwiczenie (mnożenie) Zdefiniuj mnożenie i udowodnij jego właściwości.

```
Theorem mult_0_l :
  ∀ n : nat, mult 0 n = 0.

Theorem mult_0_r :
  ∀ n : nat, mult n 0 = 0.

Theorem mult_1_l :
  ∀ n : nat, mult 1 n = n.

Theorem mult_1_r :
  ∀ n : nat, mult n 1 = n.
```

Jeżeli ćwiczenie było za proste i czytałeś podrozdział o kombinatorach taktyk, to spróbuj udowodnić:

- dwa pierwsze twierdzenia używając nie więcej niż 2 taktyk
- trzecie bez użycia indukcji, używając nie więcej niż 4 taktyk
- czwarte używając nie więcej niż 4 taktyk

Wszystkie dowody powinny być nie dłuższe niż pół linijki.

Ćwiczenie (inne dodawanie) Dodawanie można alternatywnie zdefiniować także w sposób przedstawiony poniżej. Udowodnij, że ta definicja jest równoważna poprzedniej.

```
Fixpoint plus' (n m : nat) : nat :=
match m with
    | 0 => n
    | S m' => plus' (S n) m'
end.

Theorem plus'_n_0 :
  forall n : nat, plus' n 0 = n.

Theorem plus'_S :
  forall n m : nat, plus' (S n) m = S (plus' n m).

Theorem plus'_0_n :
  forall n : nat, plus' 0 n = n.

Theorem plus'_comm :
  forall n m : nat, plus' n m = plus' m n.

Theorem plus'_is_plus :
  forall n m : nat, plus' n m = plus n m.

End NatOps.
```

3.3.3 Typy polimorficzne i właściwości konstruktorów

Przy pomocy komendy `Inductive` możemy definiować nie tylko typy induktywne, ale także rodziny typów induktywnych. Jeżeli taka rodzina parametryzowana jest typem, to mamy do czynienia z polimorfizmem.

```
Inductive option (A : Type) : Type :=
    | Some : A -> option A
    | None : option A.
```

option jest rodziną typów, zaś samo *option A* dla ustalonego *A* jest typem, który reprezentuje możliwość istnienia wartości typu *A* (konstruktor *Some*) albo i nie (konstruktor *None*).

```
Check Some.
(*  ===> Some forall A : Type, A -> option A *)
```

```
Check Some nat 5.
(*  ===> Some nat 5 *)

Check None.
(*  ===> None forall A : Type, option A *)

Arguments Some [A] _.
Arguments None [A].
```

Jak widać typ A, będący parametrem *option*, jest też pierwszym argumentem każdego z konstruktorów. Pisanie go bywa uciążliwe, ale na szczęście Coq może sam wywnioskować jego wartość, jeżeli mu każemy. Komenda *Arguments* pozwala nam określić, które argumenty mają być domyślne — chcemy, aby argument A był domyślny, gdyż w przypadku konstruktura *Some* może być wywnioskowany z drugiego argumentu, a w przypadku *None* — zazwyczaj z kontekstu.

Konstruktory typów induktywnych mają kilka właściwości, o których warto wiedzieć. Po pierwsze, wartości zrobione za pomocą różnych konstruktorów są różne. Jest to konieczne, gdyż za pomocą pattern machingu możemy rozróżnić różne konstruktory — gdyby były one równe, uzyskalibyśmy sprzeczność.

```
Definition isSome {A : Type} (a : option A) : Prop :=
match a with
    | Some _ => True
    | None => False
end.
```

Pomocnicza funkcja *isSome* ma za zadanie sprawdzić, którym konstruktorem zrobiono wartość typu *option A*. Zapis $\{A$: `Type`$\}$ oznacza, że A jest argumentem domyślnym funkcji — Coq może go wywnioskować, gdyż zna typ argumentu a (jest nim *option A*). Zauważ też, że funkcja ta zwraca zdania logiczne, a nie wartości boolowskie.

```
Theorem some_not_none :
  ∀ (A : Type) (a : A), Some a ≠ None.
Proof.
  unfold not; intros. change False with (isSome (@None A)).
  rewrite ← H. simpl. trivial.
Qed.
```

Możemy użyć tej pomocniczej funkcji, aby udowodnić, że konstruktory *Some* i *None* tworzą różne wartości. Taktyka `change` *t1* `with` *t2* pozwala nam zamienić term *t1* na *t2* pod warunkiem, że są one konwertowalne (czyli jeden z nich redukuje się do drugiego). W naszym wypadku chcemy zastąpić *False* przez *isSome* (@*None A*), który redukuje się do *False* (spróbuj zredukować to wyrażenie ręcznie).

Użycie symbolu @ pozwala nam dla danego wyrażenia zrezygnować z próby automatycznego wywnioskowania argumentów domyślnych — w tym przypadku Coq nie potrafiłby wywnioskować argumentu dla konstruktora *None*, więc musimy podać ten argument ręcznie.

Następnie możemy skorzystać z równania *Some a* = *None*, żeby uzyskać cel postaci *isSome* (*Some a*). Cel ten redukuje się do *True*, którego udowodnienie jest trywialne.

```
Theorem some_not_none' :
  ∀ (A : Type) (a : A), Some a ≠ None.
Proof. inversion 1. Qed.
```

Cała procedura jest dość skomplikowana — w szczególności wymaga napisania funkcji pomocniczej. Na szczęście Coq jest w stanie sam wywnioskować, że konstruktory są różne. Możemy zrobić to przy pomocy znanej nam z poprzedniego rozdziału taktyki `inversion`. Zapis `inversion` 1 oznacza: wprowadź zmienne związane przez kwantyfikację uniwersaną do kontekstu i użyj taktyki `inversion` na pierwszej przesłance implikacji. W naszym przypadku implikacja jest ukryta w definicji negacji: *Some a* $\neq$ *None* to tak naprawdę *Some a* $=$ *None* $\rightarrow$ *False*.

```
Theorem some_inj :
  ∀ (A : Type) (x y : A),
    Some x = Some y → x = y.
Proof.
  intros. injection H. trivial.
Qed.
```

Kolejną właściwością konstruktorów jest fakt, że są one injekcjami, tzn. jeżeli dwa termy zrobione tymi samymi konstruktorami są równe, to argumenty tych konstruktorów też są równe.

Aby skorzystać z tej właściwości w dowodzie, możemy użyć taktyki `injection`, podając jej jako argument nazwę hipotezy. Jeżeli hipoteza jest postaci *C x1 ... xn* $=$ *C y1 ... yn*, to nasz cel *G* zostanie zastąpiony przez implikację *x1* $=$ *y1* $\rightarrow$ ... $\rightarrow$ *xn* $=$ *yn* $\rightarrow$ *G*. Po wprowadzeniu hipotez do kontekstu możemy użyć ich do udowodnienia *G*, zazwyczaj przy pomocy taktyki `rewrite`.

W naszym przypadku *H* miało postać *Some x* $=$ *Some y*, a cel $x = y$, więc `injection` *H* przekształciło cel do postaci $x = y \rightarrow x = y$, który jest trywialny.

```
Theorem some_inj' :
  ∀ (A : Type) (x y : A), Some x = Some y → x = y.
Proof.
  inversion 1. trivial.
Qed.
```

Taktyka `inversion` może nam pomóc również wtedy, kiedy chcemy skorzystać z injektywności konstruktorów. W zasadzie jest ona nawet bardziej przydatna — działa ona tak jak `injection`, ale zamiast zostawiać cel w postaci *x1* $=$ *y1* $\rightarrow$ ... $\rightarrow$ *G*, wprowadza ona wygenerowane hipotezy do kontekstu, a następnie przepisuje w celu wszystkie, których przepisanie jest możliwe. W ten sposób oszczędza nam ona nieco pisania.

W naszym przypadku *inverson* 1 dodała do kontekstu hipotezę $x = y$, a następnie przepisała ją w celu (który miał postać $x = y$), dając cel postaci $y = y$.

```
Theorem some_inj'' :
  ∀ (A : Type) (x y : A), Some x = Some y → x = y.
Proof.
```

```
  injection 1. intro. subst. trivial.
Qed.
```

Taktyką ułatwiającą pracę z `injection` oraz `inversion` jest `subst`. Taktyka ta wyszukuje w kontekście hipotezy postaci $a = b$, przepisuje je we wszystkich hipotezach w kontekście i celu, w których jest to możliwe, a następnie usuwa. Szczególnie często spotykana jest kombinacja `inversion` H; `subst`, gdyż `inversion` często generuje sporą ilość hipotez postaci $a = b$, które `subst` następnie "sprząta".

W naszym przypadku hipoteza $H0 : x = y$ została przepisana nie tylko w celu, dając $y = y$, ale także w hipotezie H, dając $H : Some\ y = Some\ y$.

Ćwiczenie (zero i jeden) Udowodnij poniższe twierdzenie bez używania taktyki `inversion`. Żeby było trudniej, nie pisz osobnej funkcji pomocniczej — zdefiniuj swoją funkcję bezpośrednio w miejscu, w którym chcesz jej użyć.

```
Theorem zero_not_one : 0 ≠ 1.
```

Dwie opisane właściwości, choć pozornie niewinne, a nawet przydatne, mają bardzo istotne i daleko idące konsekwencje. Powodują one na przykład, że nie istnieją typy ilorazowe. Dokładne znaczenie tego faktu omówimy później, zaś teraz musimy zadowolić się jedynie prostym przykładem w formie ćwiczenia.

```
Module rational.

Inductive rational : Set :=
    | mk_rational :
      ∀ (sign : bool) (numerator denominator : nat),
        denominator ≠ 0 → rational.

Axiom rational_eq :
  ∀ (s s' : bool) (p p' q q' : nat)
    (H : q ≠ 0) (H' : q' ≠ 0), p × q' = p' × q →
      mk_rational s p q H = mk_rational s' p' q' H'.
```

Typ *rational* ma reprezentować liczby wymierne. Znak jest typu *bool* — możemy interpretować, że *true* oznacza obecność znaku minus, a *false* brak znaku. Dwie liczby naturalne będą oznaczać kolejno licznik i mianownik, a na końcu żądamy jeszcze dowodu na to, że mianownik nie jest zerem.

Oczywiście typ ten sam w sobie niewiele ma wspólnego z liczbami wymiernymi — jest to po prostu trójka elementów o typach *bool*, *nat*, *nat*, z których ostatni nie jest zerem. Żeby rzeczywiście reprezentował liczby wymierne musimy zapewnić, że termy, które reprezentują te same wartości, są równe, np. 1/2 musi być równa 2/4.

W tym celu postulujemy aksjomat, który zapewni nam pożądane właściwości relacji równości. Komenda `Axiom` pozwala nam wymusić istnienie termu pożądanego typu i nadać mu nazwę, jednak jest szalenie niebezpieczna — jeżeli zapostulujemy aksjomat, który jest sprzeczny, jesteśmy zgubieni.

W takiej sytuacji całe nasze dowodzenie idzie na marne, gdyż ze sprzecznego aksjomatu możemy wywnioskować *False*, z *False* zaś możemy wywnioskować cokolwiek, o czym przekonaliśmy się w rozdziale pierwszym. Tak też jest w tym przypadku — aksjomat *rational_eq* jest sprzeczny, gdyż łamie zasadę injektywności konstruktorów.

Ćwiczenie (niedobry aksjomat) Udowodnij, że aksjomat *rational_eq* jest sprzeczny. Wskazówka: znajdź dwie liczby wymierne, które są równe na mocy tego aksjomatu, ale które można rozróżnić za pomocą dopasowania do wzorca.

`Theorem` *rational_eq_inconsistent* : *False*.

`End` *rational*.

3.3.4 Typy induktywne — (prawie) pełna moc

Połączenie funkcji zależnych, konstruktorów rekurencyjnych i polimorfizmu pozwala nam na opisywanie (prawie) dowolnych typów. Jednym z najbardziej podstawowych i najbardziej przydatnych narzędzi w programowaniu funkcyjnym (i w ogóle w życiu) są listy.

`Module` *MyList*.

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

Lista przechowuje wartości pewnego ustalonego typu A (a więc nie można np. trzymać w jednej liście jednocześnie wartości typu *bool* i *nat*) i może mieć jedną z dwóch postaci: może być pusta (konstruktor *nil*) albo składać się z głowy i ogona (konstruktor *cons*). Głowa listy to wartość typu A, zaś jej ogon to inna lista przechowująca wartości typu A.

```
Check nil.
(*  ===> nil : forall A : Type, list A *)

Check cons.
(*  ===> cons : forall A : Type, A -> list A -> list A *)

Arguments nil [A].
Arguments cons [A] _ _.
```

Jak już wspomnieliśmy, jeżeli typ induktywny ma argument (w naszym przypadku A : `Type`), to argument ten jest też pierwszym argumentem każdego z konstruktorów. W przypadku konstruktora *cons* podawanie argumentu A jest zbędne, gdyż kolejnym jego argumentem jest wartość tego typu. Wobec tego Coq może sam go wywnioskować, jeżeli mu każemy.

Robimy to za pomocą komendy *Arguments konstruktor argumenty*. Argumenty w nawiasach kwadratowych Coq będzie traktował jako domyślne, a te oznaczone podkreślnikiem trzeba będzie zawsze podawać ręcznie. Nazwa argumentu domyślnego musi być taka sama jak w definicji typu (w naszym przypadku w definicji *list* argument nazywał się A, więc tak też musimy go nazwać używając komendy *Arguments*). Musimy wypisać wszystkie argumenty danego konstruktora — ich ilość możemy sprawdzić np. komendą `Check`.

Warto w tym momencie zauważyć, że Coq zna typy wszystkich termów, które zostały skonstruowane — gdyby tak nie było, nie mógłby sam uzupełniać argumentów domyślnych, a komenda `Check` nie mogłaby działać.

```
Notation "[]":= nil.
Infix "::":= (cons) (at level 60, right associativity ).

Check [].
(*  ===> [] : list ?254 *)

Check 0 :: 1 :: 2 :: nil.
(*  ===> [0; 1; 2] : list nat *)
```

Nazwy *nil* i *cons* są zdecydowanie za długie w porównaniu do swej częstości występowania. Dzięki powyższym eleganckim notacjom zaoszczędzimy sobie trochę pisania. Jeżeli jednak notacje utrudniają nam np. odczytanie celu, który mamy udowodnić, możemy je wyłączyć odznaczając w CoqIDE View > Display Notations.

Wynik [] : *list* ?254 (lub podobny) wyświetlony przez Coqa dla [] mówi nam, że [] jest listą pewnego ustalonego typu, ale Coq jeszcze nie wie, jakiego (bo ma za mało informacji, bo wywnioskować argument domyślny konstruktora *nil*).

```
Notation "[ x ]":= (cons x nil).
Notation "[ x ; y ; .. ; z ]":=
    (cons x (cons y .. (cons z nil) .. )).

Check [5].
(*  ===> [5] : list nat *)

Check [0; 1; 2; 3].
(*  ===> [0; 1; 2; 3] : list nat *)
```

Zauważ, że system notacji Coqa jest bardzo silny — ostatnia notacja (ta zawierająca ..) jest rekurencyjna. W innych językach tego typu notacje są zazwyczaj wbudowane w język i ograniczają się do podstawowych typów, takich jak listy właśnie.

```
Fixpoint app {A : Type} (l1 l2 : list A) : list A :=
match l1 with
    | [] => l2
    | h :: t => h :: app t l2
end.

Notation ł1 ++ l2":= (app l1 l2).
```

Funkcje na listach możemy definiować analogicznie do funkcji na liczbach naturalnych. Zaczniemy od słowa kluczowego `Fixpoint`, gdyż będziemy potrzebować rekurencji. Pierwszym argumentem naszej funkcji będzie typ *A* — musimy go wymienić, bo inaczej nie będziemy mogli mieć argumentów typu *list A* (pamiętaj, że samo *list* jest rodziną typów, a nie typem). Zapis {*A* : `Type`} oznacza, że Coq ma traktować *A* jako argument domyślny — jest to szybszy sposób, niż użycie komendy *Arguments*.

Nasz funkcja ma za zadanie dokleić na końcu (ang. append) pierwszej listy drugą listę. Definicja jest dość intuicyjna: doklejenie jakiejś listy na koniec listy pustej daje pierwszą listę, a doklejenie listy na koniec listy mającej głowę i ogon jest doklejeniem jej na koniec ogona.

```
Eval compute in [1; 2; 3] ++ [4; 5; 6].
(*  ===> [1; 2; 3; 4; 5; 6] : list nat *)
```

Wynik działania naszej funkcji wygląda poprawnie, ale niech cię nie zwiodą ładne oczka — jedynym sposobem ustalenia poprawności naszego kodu jest udowodnienie, że posiada on pożądane przez nas właściwości.

```
Theorem app_nil_l :
  ∀ (A : Type) (l : list A), [] ++ l = l.
Proof.
  intros. simpl. reflexivity.
Qed.

Theorem app_nil_r :
  ∀ (A : Type) (l : list A), l ++ [] = l.
Proof.
  induction l as [| h t].
    simpl. reflexivity.
    simpl. rewrite IHt. reflexivity.
Qed.
```

Sposoby dowodzenia są analogiczne jak w przypadku liczb naturalnych. Pierwsze twierdzenie zachodzi na mocy samej definicji funkcji *app* i dowód sprowadza się do wykonania programu za pomocą taktyki `simpl`. Drugie jest analogiczne do twierdzenia *plus_n_0*, z tą różnicą, że w drugim celu zamiast `f_equal` posłużyliśmy się taktyką `rewrite`.

Zauważ też, że zmianie uległa postać wzorca przekazanego taktyce `induction` — teraz ma on postać [| *h t*], gdyż *list* ma 2 konstruktory, z których pierwszy, *nil*, nie bierze argumentów (argumenty domyślne nie są wymieniane we wzorcach), zaś drugi, *cons*, ma dwa argumenty — głowę, tutaj nazwaną *h* (jako skrót od ang. head) oraz ogon, tutaj nazwany *t* (jako skrót od ang. tail).

Ćwiczenie (właściwości funkcji *app*) Udowodnij poniższe właściwości funkcji *app*. Wskazówka: może ci się przydać taktyka `specialize`.

```
Theorem app_assoc :
  ∀ (A : Type) (l1 l2 l3 : list A),
    l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.

Theorem app_not_comm :
  ¬ ∀ (A : Type) (l1 l2 : list A), l1 ++ l2 = l2 ++ l1.
```

Ćwiczenie (*length*) Zdefiniuj funkcję *length*, która oblicza długość listy, a następnie udowodnij poprawność swojej implementacji.

```
Theorem length_nil :
  ∀ A : Type, length (@nil A) = 0.

Theorem length_cons :
  ∀ (A : Type) (h : A) (t : list A), length (h :: t) ≠ 0.

Theorem length_app :
  ∀ (A : Type) (l1 l2 : list A),
    length (l1 ++ l2) = length l1 + length l2.

End MyList.
```

3.3.5 Rekordy

W wielu językach programowania występują typy rekordów (ang. record types). Charakteryzują się one tym, że mają z góry określoną ilość pól o potencjalnie różnych typach. W językach imperatywnych rekordy wyewoluowały zaś w obiekty, które różnią się od rekordów tym, że mogą zawierać również funkcje, których dziedziną jest obiekt, w którym funkcja się znajduje.

W Coqu mamy do dyspozycji rekordy, ale nie obiekty. Trzeba tu po raz kolejny pochwalić siłę systemu typów Coqa — o ile w większości języków rekordy są osobnym konstruktem językowym, o tyle w Coqu mogą być one z łatwością reprezentowane przez typy induktywne z jednym konstruktorem (wraz z odpowiednimi projekcjami, które dekonstruują rekord).

```
Module rational2.

Record rational : Set :=
{
    sign : bool;
    numerator : nat;
    denominator : nat;
    denominator_not_zero : denominator ≠ 0
}.
```

Z typem induktywnym o jednym konstruktorze już się zetknęliśmy, próbując zdefiniować liczby wymierne. Powyższa definicja używająca rekordu ma drobną przewagę nad poprzednią, w której słowo kluczowe `Inductive` pada explicité:

- wygląda ładniej
- ma projekcje

```
Check sign.
(*  ===> sign : rational -> bool *)

Check denominator_not_zero.
(*  ===> denominator_not_zero
    : forall r : rational, denominator r <> 0 *)
```

Dzięki projekcjom mamy dostęp do poszczególnych pól rekordu bez konieczności jego dekonstruowania — nie musimy używać konstruktu `match` ani taktyki `destruct`, jeżeli nie chcemy. Często bywa to bardzo wygodne.

Projekcję *sign* możemy interpretować jako funkcję, która bierze liczbę wymierną r i zwraca jej znak, zaś projekcja *denominator_not_zero* mówi nam, że mianownik żadnej liczb wymiernej nie jest zerem.

Pozwa tymi wizualno-praktycznymi drobnostkami, obie definicje są równoważne (w szczególności, powyższa definicja, podobnie jak poprzednia, nie jest dobrą reprezentacją liczb wymiernych).

`End` *rational2*.

Ćwiczenie (kalendarz) Zdefiniuj typ induktywny reprezentujący datę i napisz ręcznie wszystkie projekcje. Następnie zdefiniuj rekord reprezentujący datę i zachwyć się tym, ile czasu i głupiego pisania zaoszczędziłbyś, gdybyś od razu użył rekordu...

3.3.6 Klasy

Mechanizmem ułatwiającym życie jeszcze bardziej niż rekordy są klasy. Niech nie zmyli cię ta nazwa — nie mają one nic wspólnego z klasami znanymi z języków imperatywnych. Bliżej im raczej do interfejsów, od których są zresztą dużo silniejsze.

W językach imperatywnych interfejs możemy zaimplementować zazwyczaj definiując nowy typ. W Coqu możemy uczynić typ instancją klasy w dowolnym miejscu — nawet jeżeli to nie my go zdefiniowaliśmy. Co więcej, instancjami klas mogą być nie tylko typy, ale dowolne termy. Klasy są w Coqu pełnoprawnym tworem — mogą mieć argumenty, zawierać inne klasy, być przekazywane jako argumenty do funkcji etc. Używa się ich zazwyczaj dwojako:

- zamiast rekordów (zwiększa to nieco czytelność)
- jako interfejsy

```
Class EqDec (A : Type) : Type :=
{
    eq_dec : A → A → bool;
    eq_dec_spec : ∀ x y : A, eq_dec x y = true ↔ x = y
}.
```

Nie będziemy po raz trzeci powtarzać (kulawej) definicji liczb wymiernych — użycie do tego klas zamiast rekordów sprowadza się do zamienienia słowa kluczowego `Record` na `Class` w poprzedniej definicji.

Przyjrzyjmmy się za to wykorzystaniu klasy w roli interfejsu. Argument A : `Type` po nazwie klasy mówi nam, że nasz interfejs będą mogły implementować typy. Dalej zapis : `Type` mówi nam, że nasza klasa jest typem — klasy, jako ulepszone rekordy, są typami induktywnymi z jednym konstruktorem.

Nasza klasa ma dwa pola, które będzie musiał podać użytkownik chcący uczynić swój typ jej instancją: funkcję *eq_dec* oraz jej specyfikację, która mówi nam, że *eq_dec* zwraca *true* wtedy i tylko wtedy, gdy jej argumenty są równe.

Wobec tego typy będące instancjami *EqDec* można interpretować jako typy, dla których równość elementów można sprawdzić za pomocą jakiegoś algorytmu. Nie wszystkie typy posiadają tę własność — problematyczne są szczególnie te, których elementy są w jakiś sposób "nieskończone".

```
Instance EqDec_bool : EqDec bool :=
{
    eq_dec := fun b b' : bool ⇒
      match b, b' with
        | true, true ⇒ true
        | false, false ⇒ true
        | _, _ ⇒ false
      end
}.
Proof.
  destruct x, y; split; trivial; inversion 1.
Defined.
```

Instancje klas definiujemy przy pomocy słowa kluczowego `Instance`. Jeżeli używamy klasy jako interfejsu, który implementować mogą typy, to zazwyczaj będziemy potrzebować tylko jednej instancji, więc jej nazwa będzie niemal identyczna jak jej typ (dzięki temu łatwo będzie ją zapamiętać).

Po symbolu := w nawiasach klamrowych definiujemy pola, które nie są dowodami. Całość, jako komenda, musi kończyć się kropką. Gdy klasa nie zawiera żadnych pól będących dowodami, definicja jest zakończona. W przeciwnym przypadku Coq przechodzi w tryb dowodzenia, w którym każdemu polu będącemu dowodem odpowiada jeden podcel. Po rozwiązaniu wszystkich podcelów instancja jest zdefiniowana.

W naszym przypadku klasa ma dwa pola — funkcję i dowód na to, że funkcja spełnia specyfikację — więc w nawiasach klamrowych musimy podać jedynie funkcję. Zauważmy, że nie musimy koniecznie definiować jej właśnie w tym miejscu — możemy zrobić to wcześniej, np. za pomocą komendy `Definition` albo `Fixpoint`, a tutaj odnieść się do niej używając jej nazwy. W przypadku bardziej skomplikowanych definicji jest to nawet lepsze wyjście, gdyż zyskujemy dzięki niemu kontrolę nad tym, w którym miejscu rozwinąć definicję, dzięki czemu kontekst i cel stają się czytelniejsze.

Ponieważ nasza klasa ma pole, które jest dowodem, Coq przechodzi w tryb dowodzenia. Dowód, mimo iż wymaga rozpatrzenia ośmiu przypadków, mieści się w jednej linijce — widać tutaj moc automatyzacji. Prześledźmy, co się w nim dzieje.

Najpierw rozbijamy wartości boolowskie x i y. Nie musimy wcześniej wprowadzać ich do kontekstu taktyką `intros`, gdyż `destruct` sam potrafi to zrobić. W wyniku tego dostajemy cztere podcele. W każdym z nich taktyką `split` rozbijamy równoważność na dwie implikacje. Sześć z nich ma postać $P \to P$, więc radzi sobie z nimi taktyka `trivial`. Dwie pozostałe mają przesłanki postaci $false = true$ albo $true = false$, które są sprzeczne na mocy omówionych wcześniej właściwości konstruktorów. Taktyką `inversion 1` wskazujemy, że pierwsza przesłanka implikacji zawiera taką właśnie sprzeczną równość termów zrobionych różnymi konstruktorami, a Coq załatwia za nas resztę.

Jeżeli masz problem z odczytaniem tego dowodu, koniecznie przeczytaj ponownie fragment rozdziału pierwszego dotyczący kombinatorów taktyk. Jeżeli nie potrafisz wyobrazić sobie podcelów generowanych przez kolejne taktyki, zastąp chwilowo średniki kropkami, a jeżeli to nie pomaga, udowodnij całe twierdzenie bez automatyzacji.

Dzięki takim ćwiczeniom prędzej czy później oswoisz się z tym sposobem dowodzenia, choć nie jest to sztuka prosta — czytanie cudzych dowodów jest równie trudne jak czytanie cudzych programów.

Prawie nigdy zresztą nowopowstałe dowody nie są od razu zautomatyzowane aż w takim stopniu — najpierw są przeprowadzone w części lub w całości ręcznie. Automatyzacja jest wynikiem dostrzeżenia w dowodzie pewnych powtarzających się wzorców. Proces ten przypomina trochę refaktoryzację kodu — gdy dostrzeżemy powtarzające się fragmenty kodu, przenosimy je do osobnych procedur. Analogicznie, gdy dostrzegamy powtarzające się fragmenty dowodu, łączymy je kombinatorami taktyk lub piszemy własne, zupełnie nowe taktyki (temat pisania własnych taktyk poruszę prędzej czy później).

Od teraz będę zakładał, że nie masz problemów ze zrozumieniem takich dowodów i kolejne przykładowe dowody będę pisał w bardziej zwratej formie.

Zauważ, że definicję instancji kończymy komendą `Defined`, a nie `Qed`, jak to było w przypadku dowodów twierdzeń. Wynika to z faktu, że Coq inaczej traktuje specyfikacje i programy, a inaczej zdania i dowody. W przypadku dowodu liczy się sam fakt jego istnienia, a nie jego treść, więc komenda `Qed` każe Coqowi zapamiętać jedynie, że twierdzenie udowodniono, a zapomnieć, jak dokładnie wyglądał proofterm. W przypadku programów takie zachowanie jest niedopuszczalne, więc `Defined` każe Coqowi zapamiętać term ze wszystkimi szczegółami. Jeżeli nie wiesz, której z tych dwóch komend użyć, użyj `Defined`.

Ćwiczenie (*EqDec*) Zdefiniuj instancje klasy *EqDec* dla typów *unit* oraz *nat*.

Ćwiczenie (równość funkcji) Czy możliwe jest zdefiniowanie instancji klasy *EqDec* dla typu:

- $bool \to bool$
- $bool \to nat$
- $nat \to bool$
- $nat \to nat$
- `Prop`

Jeżeli tak, udowodnij w Coqu. Jeżeli nie, zaargumentuj słownie.

```
Instance EqDec_option (A : Type) (_ : EqDec A) : EqDec (option A) :=
{
    eq_dec := fun opt1 opt2 : option A ⇒
      match opt1, opt2 with
        | Some a, Some a' ⇒ eq_dec a a'
        | None, None ⇒ true
        | _, _ ⇒ false
      end
}.
Proof.
  destruct x, y; split; trivial; try (inversion 1; fail); intro.
    apply (eq_dec_spec a a0) in H. subst. trivial.
    apply (eq_dec_spec a a0). inversion H. trivial.
Defined.
```

Instancje klas mogą przyjmować argumenty, w tym również instancje innych klas albo inne instancje tej samej klasy. Dzięki temu możemy wyrazić ideę interfejsów warunkowych.

W naszym przypadku typ *option A* może być instancją klasy *EqDec* jedynie pod warunkiem, że jego argument również jest instancją tej klasy. Jest to konieczne, gdyż porównywanie termów typu *option A* sprowadza się do porównywania termów typu *A*.

Zauważ, że kod *eq_dec a a'* nie odwołuje się do definiowanej właśnie funkcji *eq_dec* dla typu *option A* — odnosi się do funkcji *eq_dec*, której dostarcza nam instancja _ : *EqDec A*. Jak widać, nie musimy nawet nadawać jej nazwy — Coqa interesuje tylko jej obecność.

Na podstawie typów termów *a* i *a'*, które są Coqowi znane, potrafi on wywnioskować, że *eq_dec a a'* nie jest wywołaniem rekurencyjnym, lecz odnosi się do instancji innej niż obecnie definiowana. Coq może ją znaleźć i odnosić się do niej, mimo że my nie możemy (gdybyśmy chcieli odnosić się do tej instancji, musielibyśmy zmienić nazwę z _ na coś innego).

Przydatne komendy

Ponieważ w następnym zadaniu pojawia się stwierdzenie "znajdź", czas, aby opisać kilka przydatnych komend.

```
Check unit.
(*  ===> unit : Set *)

Print unit.
(*  ===> Inductive unit : Set := tt : unit *)
```

Przypomnijmy, że komenda `Check` wyświetla typ danego jej termu, a `Print` wypisuje jego definicję.

```
Search nat.
```

`Search` wyświetla wszystkie obiekty, które zawierają podaną nazwę. W naszym przypadku pokazały się wszystkie funkcje, w których sygnaturze występuje typ *nat*.

`SearchAbout` *nat*.

`SearchAbout` wyświetla wszystkie obiekty, które mają jakiś związek z daną nazwą. Zazwyczaj wskaże on nam dużo więcej obiektów, niż zwykłe `Search`, np. poza funkcjami, w których sygnaturze występuje *nat*, pokazuje też twierdzenia dotyczące ich właściwości.

`Search`*Pattern* (_ + _ = _).

`SearchPattern` jako argument bierze wzorzec i wyświetla wszystkie obiekty, które zawierają podterm pasujący do danego wzorca. W naszym przypadku pokazały się twierdzenia, w których występuje podterm mający po lewej dodawanie, a po prawej cokolwiek.

Dokładny opis wszystkich komend znajdziesz tutaj: https://coq.inria.fr/refman/command-index.html

Ćwiczenie (równość list) Zdefiniuj instancję klasy *EqDec* dla typu *list* A.

Ćwiczenie (równość funkcji 2) Niech A i B będą dowolnymi typami. Zastanów się, kiedy możliwe jest zdefiniowanie instancji klasy *EqDec* dla $A \rightarrow B$.

3.3.7 Ważne typy induktywne

`Module` *ImportantTypes*.

Typ pusty

`Inductive` *Empty_set* : `Set` := .

Empty_set jest, jak sama nazwa wskazuje, typem pustym. Żaden term nie jest tego typu. Innymi słowy: jeżeli jakiś term jest typu *Empty_set*, to mamy sprzeczność.

```
Definition create {A : Type} (x : Empty_set) : A :=
  match x with end.
```

Jeżeli mamy term typu *Empty_set*, to możemy w sposób niemal magiczny wyczarować term dowolnego typu A, używając pattern matchingu z pustym wzorcem.

Ćwiczenie (*create_unique*) Udowodnij, że powyższa funkcja jest unikalna.

```
Theorem create_unique :
  ∀ (A : Type) (f : Empty_set → A),
    (∀ x : Empty_set, create x = f x).
```

Ćwiczenie (*no_fun_from_nonempty_to_empty*) Pokaż, że nie istnieją funkcje z typu niepustego w pusty.

```
Theorem no_fun_from_nonempty_to_empty :
  ∀ (A : Type) (a : A) (f : A → Empty_set), False.
```

Singleton

`Inductive` $unit$: `Set` :=
  | tt : $unit$.

$unit$ jest typem, który ma tylko jeden term, zwany tt (nazwa ta jest wzięta z sufitu).

`Definition` $delete$ {A : `Type`} (a : A) : $unit$:= tt.

Funkcja *delete* jest w pewien sposób "dualna" do napotkanej przez nas wcześniej funkcji *create*. Mając term typu *Empty_set* mogliśmy stworzyć term dowolnego innego typu, zaś mając term dowolnego typu A, możemy "zapomnieć o nim" albo "skasować go", wysyłając go funkcją *delete* w jedyny term typu *unit*, czyli *tt*.

Uwaga: określenie "skasować" nie ma nic wspólnego z fizycznym niszczeniem albo dealokacją pamięci. Jest to tylko metafora.

Ćwiczenie (*delete_unique*) Pokaż, że funkcja *delete* jest unikalna.

`Theorem` $delete\_unique$:
  $\forall$ (A : `Type`) (f : $A \to unit$),
    ($\forall$ x : A, $delete\ x = f\ x$).

Produkt

`Inductive` $prod$ ($A\ B$: `Type`) : `Type` :=
  | $pair$: $A \to B \to prod\ A\ B$.

Arguments pair [A] [B] _ _.

Produkt typów A i B to typ, którego termami są pary. Pierwszy element pary to term typu A, a drugi to term typu B. Tym, co charakteryzuje produkt, są projekcje:

- fst : $\forall$ $A\ B$: `Type`, $prod\ A\ B \to A$ wyciąga z pary jej pierwszy element
- snd : $\forall$ $A\ B$: `Type`, $prod\ A\ B \to B$ wyciąga z pary jej drugi element

Ćwiczenie (projekcje) Zdefiniuj projekcje i udowodnij poprawność swoich definicji.

`Theorem` $proj\_spec$:
  $\forall$ ($A\ B$: `Type`) (p : $prod\ A\ B$),
    $p = pair\ (fst\ p)\ (snd\ p)$.

Suma

`Inductive` sum ($A\ B$: `Type`) : `Type` :=
  | inl : $A \to sum\ A\ B$
  | inr : $B \to sum\ A\ B$.

```
Arguments inl [A] [B] _.
Arguments inr [A] [B] _.
```

Suma A i B to typ, którego termy są albo termami typu A, zawiniętymi w konstruktor *inl*, albo termami typu B, zawiniętymi w konstruktor *inr*. Suma, w przeciwieństwie do produktu, zdecydowanie nie ma projekcji.

Ćwiczenie (sumy bez projekcji) Pokaż, że suma nie ma projekcji.

```
Theorem sum_no_fst :
  ∀ (proj : ∀ A B : Type, sum A B → A), False.

Theorem sum_no_snd :
  ∀ (proj : ∀ A B : Type, sum A B → B), False.

End ImportantTypes.
```

3.3.8 Typy puste

Typy puste to typy, które nie mają żadnych elementów. Z jednym z nich już się spotkaliśmy — był to *Empty_set*, który jest pusty, gdyż nie ma żadnych konstruktorów. Czy wszystkie typy puste to typy, które nie mają konstruktorów?

```
Inductive Empty : Type :=
    | c : Empty_set → Empty.

Theorem Empty_is_empty :
  ∀ empty : Empty, False.
Proof.
  intro. destruct empty. destruct e.
Qed.
```

Okazuje się, że nie. Pustość i niepustość jest kwestią czegoś więcej, niż tylko ilości konstruktorów. Powyższy przykład pokazuje dobitnie, że ważne są też typy argumentów konstruktorów. Jeżeli typ któregoś z argumentów konstruktora jest pusty, to nie można użyć go do zrobienia żadnego termu. Jeżeli każdy konstruktor typu T ma argument, którego typ jest pusty, to sam typ T również jest pusty.

Wobec powyższych rozważań możemy sformułować następujące kryterium: typ T jest niepusty, jeżeli ma co najmniej jeden konstruktor, który nie bierze argumentów, których typy są puste. Jakkolwiek jest to bardzo dobre kryterium, to jednak nie rozwiewa ono niestety wszystkich możliwych wątpliwości.

```
Inductive InfiniteList (A : Type) : Type :=
    | InfiniteCons : A → InfiniteList A → InfiniteList A.
```

Czy typ *InfiniteList A* jest niepusty? Skorzystajmy z naszego kryterium: ma on jeden konstruktor biorący dwa argumenty, jeden typu A oraz drugi typu *InfiniteList A*. W zależności od tego, czym jest A, może on być pusty lub nie — przyjmijmy, że A jest niepusty. W przypadku drugiego argumentu napotykamy jednak na problem: to, czy *InfiniteList A* jest niepusty zależy od tego, czy typ argumentu jego konstruktora, również *InfiniteList A*, jest niepusty. Sytuacja jest więc beznadziejna — mamy błędne koło.

Powyższy przykład pokazuje, że nasze kryterium może nie poradzić sobie z rekurencją. Jak zatem rozstrzygnąć, czy typ ten jest niepusty? Musimy odwołać się bezpośrednio do definicji i zastanowić się, czy możliwe jest skonstruowanie jakichś jego termów. W tym celu przypomnijmy, czym są typy induktywne:

- Typ induktywny to rodzaj planu, który pokazuje, w jaki sposób można konstruować jego termy, które są drzewami.
- Konstruktory to węzły drzewa. Ich nazwy oraz ilość i typy argumentów nadają drzewu kształt i znaczenie.
- Konstruktory nierekurencyjne to liście drzewa.
- Konstruktory rekurencyjne to węzły wewnętrzne drzewa.

Kluczowym faktem jest rozmiar termów: o ile rozgałęzienia mogą być potencjalnie nieskończone, o tyle wszystkie gałęzie muszą mieć skończoną długość. Pociąga to za sobą bardzo istotny fakt: typy mające jedynie konstruktory rekurencyjne są puste, gdyż bez użycia konstruktorów nierekurencyjnych możemy konstruować jedynie drzewa nieskończone (i to tylko przy nierealnym założeniu, że możliwe jest zakończenie konstrukcji liczącej sobie nieskończoność kroków).

```
Theorem InfiniteList_is_empty :
  ∀ A : Type, InfiniteList A → False.
Proof.
  intros A l. induction l as [h t]. exact IHt.
Qed.
```

Pokazanie, że *InfiniteList A* jest pusty, jest bardzo proste — wystarczy posłużyć się indukcją. Indukcja po l : *InfiniteList A* daje nam hipotezę indukcyjną *IHt* : *False*, której możemy użyć, aby natychmiast zakończyć dowód.

Zaraz, co właściwie się stało? Dlaczego dostaliśmy zupełnie za darmo hipotezę *IHt*, która jest szukanym przez nas dowodem? W ten właśnie sposób przeprowadza się dowody indukcyjne: zakładamy, że hipoteza P zachodzi dla termu t : *InfiniteList A*, a następnie musimy pokazać, że P zachodzi także dla termu *InfiniteCons h t*. Zazwyczaj P jest predykatem i wykonanie kroku indukcyjnego jest nietrywialne, w naszym przypadku jest jednak inaczej — postać P jest taka sama dla t oraz dla *InfiniteCons h t* i jest nią *False*.

Czy ten konfundujący fakt nie oznacza jednak, że *list A*, czyli typ zwykłych list, również jest pusty? Spróbujmy pokazać, że tak jest.

```
Theorem list_empty :
  ∀ (A : Type), list A → False.
Proof.
  intros A l. induction l as [| h t].
```

Focus 2. `exact` *IHt*.
`Abort`.

Pokazanie, że typ *list A* jest pusty, jest rzecz jasna niemożliwe, gdyż typ ten zdecydowanie pusty nie jest — w jego definicji stoi jak byk napisane, że dla dowolnego typu A istnieje lista termów typu A. Jest nią oczywiście @*nil A*.

Przyjrzyjmy się naszej próbie dowodu. Próbujemy posłużyć się indukcją w ten sam sposób co poprzednio. Taktyka `induction` generuje nam dwa podcele, gdyż *list* ma dwa konstruktory — pierwszy podcel dla *nil*, a drugi dla *cons*. Komenda *Focus* pozwala nam przełączyć się do wybranego celu, w tym przypadku celu nr 2, czyli gdy l jest postaci *cons h t*.

Sprawa wygląda identycznie jak poprzednio — za darmo dostajemy hipotezę *IHt* : *False*, której używamy do natychmiastowego rozwiązania naszego celu. Tym, co stanowi przeszkodę nie do pokonania, jest cel nr 1, czyli gdy l zrobiono za pomocą konstruktora *nil*. Ten konstruktor nie jest rekurencyjny, więc nie dostajemy żadnej hipotezy indukcyjnej. Lista l zostaje w każdym miejscu, w którym występuje, zastąpiona przez [], a ponieważ nie występuje nigdzie — znika. Musimy teraz udowodnić fałsz wiedząc jedynie, że A jest typem, co jest niemożliwe.

3.4 Induktywne zdania i predykaty

Wiemy, że słowo kluczowe `Inductive` pozwala nam definiować nowe typy (a nawet rodziny typów, jak w przypadku *option*). Wiemy też, że zdania są typami. Wobec tego nie powinno nas dziwić, że induktywnie możemy definiować także zdania, spójniki logiczne, predykaty oraz relacje.

3.4.1 Induktywne zdania

`Inductive` *false_prop* : `Prop` := .

`Inductive` *true_prop* : `Prop` :=
| *obvious_proof* : *true_prop*
| *tricky_proof* : *true_prop*
| *weird_proof* : *true_prop*
| *magical_proof* : *true_prop*.

Induktywne definicje zdań nie są zbyt ciekawe, gdyż pozwalają definiować jedynie zdania fałszywe (zero konstruktorów) lub prawdziwe (jeden lub więcej konstruktorów). Pierwsze z naszych zdań jest fałszywe (a więc rónoważne z *False*), drugie zaś jest prawdziwe (czyli równoważne z *True*) i to na cztery sposoby!

Ćwiczenie (induktywne zdania) `Theorem` *false_prop_iff_False* : *false_prop* $\leftrightarrow$ *False*.

`Theorem` *true_prop_iff_True* : *true_prop* $\leftrightarrow$ *True*.

3.4.2 Induktywne predykaty

Przypomnijmy, że predykaty to funkcje, których przeciwdziedziną jest sort `Prop`, czyli funkcje zwracające zdania logiczne. Predykat $P : A \to$ `Prop` można rozumieć jako właściwość, którą mogą posiadać termy typu A, zaś dla konkretnego $x : A$ zapis $P\ x$ interpretować można "term x posiada właściwóść P".

O ile istnieją tylko dwa rodzaje induktwynych zdań (prawdziwe i fałszywe), o tyle induktywnie zdefiniowane predykaty są dużo bardziej ciekawe i użyteczne, gdyż dla jednych termów mogą być prawdziwe, a dla innych nie.

```
Inductive even : nat → Prop :=
    | even0 : even 0
    | evenSS : ∀ n : nat, even n → even (S (S n)).
```

Predykat *even* ma oznaczać właściwość "bycia liczbą parzystą". Jego definicję można zinterpretować tak:

- "0 jest liczbą przystą"
- "jeżeli n jest liczbą parzystą, to $n + 2$ również jest liczbą parzystą"

Jak widać, induktywna definicja parzystości różni się od powszechnie używanej definicji, która głosi, że "liczba jest parzysta, gdy dzieli się bez reszty przez 2". Różnica jest natury filozoficznej: definicja induktywna mówi, jak konstruować liczby parzyste, podczas gdy druga, "klasyczna" definicja mówi, jak sprawdzić, czy liczba jest parzysta.

Przez wzgląd na swą konstruktywność, w Coqu induktywne definicje predykatów czy relacji są często dużo bardziej użyteczne od tych nieinduktywnych, choć nie wszystko można zdefiniować induktywnie.

```
Theorem zero_is_even : even 0.
Proof.
  apply even0.
Qed.
```

Jak możemy udowodnić, że 0 jest liczbą parzystą? Posłuży nam do tego konstruktor *even0*, który wprost głosi, że *even* 0. Nie daj się zwieść: *even0*, pisane bez spacji, jest nazwą konstruktora, podczas gdy *even* 0, ze spacją, jest zdaniem (czyli termem typu `Prop`), które można interpretować jako "0 jest liczbą parzystą".

```
Theorem two_is_even : even 2.
Proof.
  apply evenSS. apply even0.
Qed.
```

Jak możemy udowodnić, że 2 jest parzyste? Konstruktor *even0* nam nie pomoże, gdyż jego postać (*even* 0) nie pasuje do postaci naszego twierdzenia (*even* 2). Pozostaje nam jednak konstruktor *evenSS*.

Jeżeli przypomnimy sobie, że 2 to tak naprawdę S (S 0), natychmiast dostrzeżemy, że jego konkluzja pasuje do postaci naszego twierdzenia. Możemy go więc zaaplikować (pamiętaj, że konstruktory są jak zwykłe funkcje, tylko że niczego nie obliczają — nadają one typom ich kształty). Teraz wystarczy pokazać, że *even* 0 zachodzi, co już potrafimy.

```
Theorem four_is_even : even 4.
Proof.
  constructor. constructor. constructor.
Qed.
```

Jak pokazać, że 4 jest parzyste? Tą samą metodą, która pokazaliśmy, że 2 jest parzyste. 4 to S (S (S (S 0))), więc możemy użyć konstruktora *evenSS*. Zamiast jednak pisać `apply` *evenSS*, możemy użyć taktyki `constructor`. Taktyka ta działa na celach, w których chcemy skonstruować wartość jakiegoś typu induktywnego (a więc także gdy dowodzimy twierdzeń o induktywnych predykatach). Szuka ona konstruktora, który może zaaplikować na celu, i jeżeli znajdzie, to aplikuje go, a gdy nie — zawodzi.

W naszym przypadku pierwsze dwa użycia `constructor` aplikują konstruktor *evenSS*, a trzecie — konstruktor *even0*.

```
Theorem the_answer_is_even : even 42.
Proof.
  repeat constructor.
Qed.
```

A co, gdy chcemy pokazać, że 42 jest parzyste? Czy musimy 22 razy napisać `constructor`? Na szczęście nie — wystarczy posłużyć się kombinatorem `repeat` (jeżeli nie pamiętasz, jak działa, zajrzyj do rozdziału 1).

```
Theorem one_not_even_failed : ¬ even 1.
Proof.
  unfold not. intro. destruct H.
Abort.

Theorem one_not_even : ¬ even 1.
Proof.
  unfold not. intro. inversion H.
Qed.
```

A jak pokazać, że 1 nie jest parzyste? Mając w kontekście dowód na to, że 1 jest parzyste (H : *even* 1), możemy zastantowić się, w jaki sposób dowód ten został zrobiony. Nie mógł zostać zrobiony konstruktorem *even0*, gdyż ten dowodzi, że 0 jest parzyste, a przecież przekonaliśmy się już, że 0 to nie 1. Nie mógł też zostać zrobiony konstruktorem *evenSS*, gdyż ten ma w konkluzji *even* (S (S n)), podczas gdy 1 jest postaci S 0 — nie pasuje on do konkluzji *evenSS*, gdyż "ma za mało Sów".

Nasze rozumowanie prowadzi do wniosku, że za pomocą *even0* i *evenSS*, które są jedynymi konstruktorami *even*, nie można skonstruować *even* 1, więc 1 nie może być parzyste. Na podstawie wcześniejszych doświadczeń mogłoby się nam wydawać, że `destruct` załatwi sprawę, jednak tak nie jest — taktyka ta jest w tym przypadku upośledzona i nie potrafi nam pomóc. Zamiast tego możemy się posłużyć taktyką `inversion`. Działa ona dokładnie w sposób opisany w poprzednim akapicie.

```
Theorem three_not_even : ¬ even 3.
Proof.
  intro. inversion H. inversion H1.
Qed.
```

Jak pokazać, że 3 nie jest parzyste? Pomoże nam w tym, jak poprzednio, inwersja. Tym razem jednak nie załatwia ona sprawy od razu. Jeżeli zastanowimy się, jak można pokazać *even* 3, to dojdziemy do wniosku, że można to zrobić konstruktorem *evenSS*, gdyż 3 to tak naprawdę *S* (*S* 1). To właśnie robi pierwsza inwersja: mówi nam, że *H* : *even* 3 można uzyskać z zaaplikowania *evenSS* do 1, jeżeli tylko mamy dowód *H1* : *even* 1 na to, że 1 jest parzyste. Jak pokazać, że 1 nie jest parzyste, już wiemy.

Ćwiczenie (odd) Zdefiniuj induktywny predykat *odd*, który ma oznaczać "bycie liczbą nieparzystą" i udowodnij, że zachowuje się on jak należy.

```
Theorem one_odd : odd 1.
```

```
Theorem seven_odd : odd 7.
```

```
Theorem zero_not_odd : ¬ odd 0.
```

```
Theorem two_not_odd : ¬ odd 2.
```

3.4.3 Indukcja po dowodzie

```
Require Import Arith.
```

Biblioteka *Arith* zawiera różne definicje i twierdzenia dotyczące arytmetyki. Będzie nam ona potrzebna w tym podrozdziale.

Jak udowodnić, że suma liczb parzystych jest parzysta? Być może właśnie pomyślałeś o indukcji. Spróbujmy zatem:

```
Theorem even_sum_failed1 :
  ∀ n m : nat, even n → even m → even (n + m).
Proof.
  induction n as [| n']; simpl; intros.
    trivial.
    induction m as [| m']; rewrite plus_comm; simpl; intros.
      assumption.
      constructor. rewrite plus_comm. apply IHn'.
Abort.
```

Próbując jednak indukcji po *n*, a potem po *m*, docieramy do martwego punktu. Musimy udowodnić *even n'*, podczas gdy zachodzi *even* (*S n'*) (czyli *even n'* jest fałszywe). Wynika to z faktu, że przy indukcji n zwiększa się o 1 ($P\ n \to P\ (S\ n)$), podczas gdy w definicji *even* mamy konstruktor głoszący, że ($even\ n \to even\ (S\ (S\ n))$).

Być może w drugiej kolejności pomyślałeś o taktyce `destruct`: jeżeli sprawdzimy, w jaki sposób udowodniono *even n*, to przy okazji dowiemy się też, że n może być jedynie postaci 0 lub $S\ (S\ n')$. Dzięki temu powinniśmy uniknąć problemu z poprzedniej próby.

```
Theorem even_sum_failed2 :
  ∀ n m : nat, even n → even m → even (n + m).
Proof.
  intros n m Hn Hm. destruct Hn, Hm; simpl.
    constructor.
    constructor. assumption.
    rewrite plus_comm. simpl. constructor. assumption.
    rewrite plus_comm. simpl. do 2 constructor.
Abort.
```

Niestety, taktyka `destruct` okazała się za słaba. Predykat *even* jest induktywny, a zatem bez indukcji się nie obędzie. Rozwiązaniem naszych problemów nie będzie jednak indukcja po n lub m, lecz po dowodzie na to, że n jest parzyste.

```
Theorem even_sum :
  ∀ n m : nat, even n → even m → even (n + m).
Proof.
  intros n m Hn Hm. induction Hn as [| n' Hn'].
    simpl. assumption.
    simpl. constructor. assumption.
Qed.
```

Indukcja po dowodzie działa dokładnie tak samo, jak indukcja, z którą zetknęliśmy się dotychczas. Różni się od niej jedynie tym, że aż do teraz robiliśmy indukcję jedynie po termach, których typy były sortu `Set` lub `Type`. Indukcja po dowodzie to indukcja po termie, którego typ jest sortu `Prop`.

W naszym przypadku użycie `induction` *Hn* ma następujący skutek:

- W pierwszym przypadku *Hn* to po prostu konstruktor *even0*, a zatem n jest zerem.
- W drugim przypadku *Hn* to *evenSS n' Hn'*, gdzie n jest postaci $S\ (S\ n')$, zaś *Hn'* jest dowodem na to, że *n'* jest parzyste.

Taktyki `replace` i `assert`.

Przy następnych ćwiczeniach mogą przydać ci się taktyki `replace` oraz `assert`.

```
Theorem stupid_example_replace :
  ∀ n : nat, n + 0 = n.
Proof.
  intro. replace (n + 0) with (0 + n).
```

```
    trivial.
    apply plus_comm.
Qed.
```

Taktyka `replace` t `with` t' pozwala nam zastąpić w celu każde wystąpienie termu t termem t'. Jeżeli t nie ma w celu, to taktyka zawodzi, a w przeciwnym wypadku dodaje nam jeden podcel, w którym musimy udowodnić, że $t = t'$. Można też zastosować ją w hipotezie, pisząc `replace` t `with` t' `in` H.

```
Theorem stupid_example_assert :
  ∀ n : nat, n + 0 + 0 = n.
Proof.
  intro. assert (H : n + 0 = n).
    apply plus_0_r.
    do 2 rewrite H. trivial.
Qed.
```

Taktyka `assert` $(x : A)$ dodaje do kontekstu term x typu A oraz generuje jeden dodatkowy podcel, w którym musimy skonstruować x. Zawodzi ona, jeżeli nazwa x jest już zajęta.

Ćwiczenie (właściwości *even*) Udowodnij poniższe twierdzenia. Zanim zaczniesz, zastanów się, po czym należy przeprowadzić indukcję: po wartości, czy po dowodzie?

```
Theorem double_is_even :
  ∀ n : nat, even (2 × n).

Theorem even_is_double :
  ∀ n : nat, even n → ∃ k : nat, n = 2 × k.
```

3.4.4 Definicje stałych i spójników logicznych

W rozdziale pierwszym dowiedzieliśmy się, że produkt zależny (typ, którego termami są funkcje zależne), a więc i implikacja, jest typem podstawowym/wbudowanym oraz że negacja jest zdefiniowana jako implikowanie fałszu. Teraz, gdy wiemy już co nieco o typach induktywnych, nadszedł czas by zapoznać się z definicjami spójników logicznych (i nie tylko).

```
Module MyConnectives.
```

Prawda i fałsz

```
Inductive False : Prop := .
```

Fałsz nie ma żadnych konstruktorów, a zatem nie może zostać w żaden sposób skonstruowany, czyli udowodniony. Jego definicja jest bliźniaczo podobna do czegoś, co już kiedyś widzieliśmy — tym czymś był *Empty_set*, czyli typ pusty. Nie jest to wcale przypadek. Natknęliśmy się (znowu) na przykład korespondencji Curry'ego-Howarda.

Przypomnijmy, że głosi ona (w sporym uproszczeniu), iż sorty `Prop` i `Set`/`Type` są do siebie bardzo podobne. Jednym z tych podobieństw było to, że dowody implikacji są funkcjami. Kolejnym jest fakt, że *False* jest odpowiednikiem *Empty_set*, od którego różni się tym, że żyje w `Prop`, a nie w `Set`.

Ta definicja rzuca też trochę światła na sposób wnioskowania "ex falso quodlibet" (z fałszu wynika wszystko), który poznaliśmy w rozdziale pierwszym.

Użycie taktyki `destruct` lub `inversion` na termie dowolnego typu induktywnego to sprawdzenie, którym konstruktorem term ten został zrobiony — generują one dokładnie tyle podcelów, ile jest możliwych konstruktorów. Użycie ich na termie typu *False* generuje zero podcelów, co ma efekt natychmiastowego zakończenia dowodu. Dzięki temu mając dowód *False* możemy udowodnić cokolwiek.

```
Inductive True : Prop :=
  | I : True.
```

True jest odpowiednikiem *unit*, od którego różni się tym, że żyje w `Prop`, a nie w `Set`. Ma dokładnie jeden dowód, który w Coqu nazwano, z zupełnie nieznanych powodów (zapewne dla hecy), *I*.

Koniunkcja i dysjunkcja

```
Inductive and (P Q : Prop) : Prop :=
  | conj : P → Q → and P Q.
```

Dowód koniunkcji zdań P i Q to para dowodów: pierwszy element pary jest dowodem P, zaś drugi dowodem Q. Koniunkcja jest odpowiednkiem produktu, od którego różni się tym, że żyje w `Prop`, a nie w `Type`.

```
Inductive or (P Q : Prop) : Prop :=
  | or_introl : P → or P Q
  | or_intror : Q → or P Q.
```

Dowód dysjunkcji zdań P i Q to dowód P albo dowód Q wraz ze wskazaniem, którego zdania jest to dowód. Dysjunkcja jest odpowiednikiem sumy, od której różni się tym, że żyje w `Prop`, a nie w `Type`.

```
End MyConnectives.
```

3.4.5 Równość

```
Module MyEq.
```

Czym jest równość? To pytanie stawiało sobie wielu filozofów, szczególnie politycznych, a także ekonomistów. Odpowiedź na nie jest jednym z największych osiągnięć matematyki w dziejach: równość to jeden z typów induktywnych, które możemy zdefiniować w Coqu.

```
Inductive eq {A : Type} (x : A) : A → Prop :=
  | eq_refl : eq x x.
```

Spróbujmy przeczytać tę definicję: dla danego typu A oraz termu x tego typu, $eq\ x$ jest predykatem, który ma jeden konstruktor głoszący, że $eq\ x\ x$ zachodzi. Choć definicja taka brzmi obco i dziwacznie, ma ona swoje uzasadnienie (które niestety poznamy dopiero w przyszłości).

```
Theorem eq_refl_trivial : eq 42 42.
Proof.
  apply eq_refl.
Qed.
```

Poznane przez nas dotychczas taktyki potrafiące udowadniać proste równości, jak `trivial` czy `reflexivity` działają w ten sposób, że po prostu aplikują na celu *eq_refl*. Nazwa *eq_refl* to skrót od ang. "reflexivity of equality", czyli "zwrotność równości" — jest to najważniejsza cecha równości, która oznacza, że każdy term jest równy samemu sobie.

```
Theorem eq_refl_nontrivial : eq (1 + 41) 42.
Proof.
  constructor.
Qed.
```

Mogłoby wydawać się, że zwrotność nie wystarcza do udowadniania "nietrywialnych" równości pokroju $1 + 41 = 42$, jednak tak nie jest. Dlaczego *eq_refl* odnosi na tym celu sukces skoro $1 + 41$ oraz 42 zdecydowanie różnią się postacią? Odpowiedź jest prosta: typ *eq* w rzeczywistości owija jedynie równość pierwotną, wbudowaną w samo jądro Coqa, którą jest konwertowalność.

```
Theorem eq_refl_alpha :
  ∀ A : Type, eq (fun x : A ⇒ x) (fun y : A ⇒ y).
Proof.
  intro. change (fun x : A ⇒ x) with (fun y : A ⇒ y).
  apply eq_refl.
Qed.

Theorem eq_refl_beta :
  ∀ m : nat, eq ((fun n : nat ⇒ n + n) m) (m + m).
Proof.
  intro. simpl. apply eq_refl.
Qed.

Definition ultimate_answer : nat := 42.

Theorem eq_refl_delta : eq ultimate_answer 42.
Proof.
  unfold ultimate_answer. apply eq_refl.
Qed.

Theorem eq_refl_iota :
  eq 42 (match 0 with | 0 ⇒ 42 | _ ⇒ 13 end).
Proof.
```

```
  simpl. apply eq_refl.
Qed.
```

Przypomnijmy, co już wiemy o redukcjach:

- konwersja alfa pozwala nam zmienić nazwę zmiennej związanej w funkcji anonimowej nową, jeżeli ta nie jest jeszcze używana. W naszym przykładzie zamieniamy x w `fun` $x : A \Rightarrow x$ na y, otrzymując `fun` $y : A \Rightarrow y$ — konwersja jest legalna. Jednak w funkcji `fun` $x\ y : nat \Rightarrow x + x$ nie możemy użyć konwersji alfa, żeby zmienić nazwę x na y, bo y jest już używana (tak nazywa się drugi argument).

- Redukcja beta zastępuje argumentem każde wystąpienie zmiennej związanej w funkcji anonimowej. W naszym przypadku redukcja ta zamienia (`fun` $n : nat \Rightarrow n + n$) m na $m + m$ — w miejsce n wstawiamy m.

- Redukcja delta odwija definicje. W naszym przypadku zdefiniowaliśmy, że *ultimate_answer* oznacza 42, więc redukcja delta w miejsce *ultimate_answer* wstawia 42.

- Redukcja jota wykonuje pattern matching. W naszym przypadku 0 jest termem, który postać jest znana (został on skonstruowany konstruktorem 0) i który pasuje do wzorca $|\ 0 \Rightarrow 42$, a zatem redukcja jota zamienia całe wyrażenie od `match` aż do `end` na 42.

Termy x i y są konwertowalne, gdy za pomocą konwersji alfa oraz redukcji beta, delta i jota można zredukować x do y lub y do x.

Uważny czytelnik zada sobie w tym momencie pytanie: skoro równość to konwertowalność, to jakim cudem równe są termy $0 + n$ oraz $n + 0$, które przecież nie są konwertowalne?

TODO: udzielić odpowiedzi na to pytanie.

```
End MyEq.
```

3.4.6 Indukcja wzajemna

Jest jeszcze jeden rodzaj indukcji, o którym dotychczas nie mówiliśmy: indukcja wzajemna (ang. mutual induction). Bez zbędnego teoretyzowania zbadajmy sprawę na przykładzie klasyków polskiej literatury:

Smok to wysuszony zmok

Zmok to zmoczony smok

Stanisław Lem

Idea stojąca za indukcją wzajemną jest prosta: chcemy przez indukcję zdefiniować jednocześnie dwa obiekty, które mogą się nawzajem do siebie odwoływać.

W owym definiowaniu nie mamy rzecz jasna pełnej swobody — obowiązują te same kryteria co w przypadku zwykłych, "pojedynczych" definicji typów induktywnych. Wobec tego zauważyć należy, że definicja słowa "smok" podana przez Lema jest według Coqowych standardów nieakceptowalna, gdyż jeżeli w definicji *smoka* rozwiniemy definicję *zmoka*, to otrzymamy

Smok ty wysuszony zmoczony smok

Widać gołym okiem, iż próba zredukowania (czyli obliczenia) obiektu *smok* nigdy się nie skończy. Jak już wiemy, niekończące się obliczenia w logice odpowiadają sprzeczności, a zatem ani *smoki*, ani *zmoki* w Coqowym świecie nie istnieją.

Nie znaczy to bynajmniej, że wszystkie definicje przez indukcję wzajemną są w Coqu niepoprawne, choć należy przyznać, że są dość rzadko używane. Czas jednak abyśmy ujrzeli pierwszy prawdziwy przkład indukcji wzajemnej.

```
Module MutInd.

Inductive even : nat → Prop :=
    | even0 : even 0
    | evenS : ∀ n : nat, odd n → even (S n)

with odd : nat → Prop :=
    | oddS : ∀ n : nat, even n → odd (S n).
```

Aby zrozumieć tę definicję, zestawmy ją z naszą definicją parzystości z sekcji *Induktywne predykaty*.

Zdefiniowaliśmy tam predykat bycia liczbą parzystą tak:

- 0 jest parzyste
- jeżeli n jest parzyste, to $n + 2$ też jest parzyste

Tym razem jednak nie definiujemy jedynie predykatu "jest liczbą parzystą". Definiujemy jednocześnie dwa predykaty: "jest liczbą parzystą" oraz "jest liczbą nieparzystą", które odwołują się do siebi nawzajm. Definicja brzmi tak:

- 0 jest parzyste
- jeżeli n jest nieparzyste, to $n + 1$ jest parzyste
- jeżeli n jest parzyste, to $n + 1$ jest nieparzyste

Czy definicja taka rzeczywiście ma sens? Sprawdźmy to:

- 0 jest parzyste na mocy definicji
- jeżeli 0 jest parzyste (a jest), to 1 jest nieparzyste
- jeżeli 1 jest nieparzyste (a jest), to 2 jest parzyste
- i tak dalej, ad infinitum

Jak widać, za pomocą naszej wzajemnie induktywnej definicji *even* można wygenerować wszystkie liczby parzyste (i tylko je), tak więc nowe *even* jest równoważne staremu *even* z sekcji *Induktywne predykaty*. Podobnie *odd* może wygenerować wszystkie liczby nieparzyste i tylko je.

Ćwiczenie (upewniające) Upewnij się, że powyższy akapit nie kłamie.

```
Lemma even_0 : even 0.
```

```
Lemma odd_1 : odd 1.
```

```
Lemma even_2 : even 2.
```

```
Lemma even_42 : even 42.
```

```
Lemma not_odd_0 : ¬ odd 0.
```

```
Lemma not_even_1 : ¬ even 1.
```

Ćwiczenie (właściwości *even* i *odd*) Udowodnij podstawowe właściwości *even* i *odd*.

```
Lemma even_SS :
  ∀ n : nat, even n → even (S (S n)).
```

```
Lemma odd_SS :
  ∀ n : nat, odd n → odd (S (S n)).
```

```
Lemma even_plus :
  ∀ n m : nat, even n → even m → even (n + m).
```

Jeśli poległeś przy ostatnim zadaniu — nie przejmuj się. Specjalnie dobrałem złośliwy przykład.

W tym momencie należy sobie zadać pytanie: jak dowodzić właściwości typów wzajemnie induktywnych? Aby udzielić odpowiedzi, spróbujmy udowodnić *even_plus* za pomocą indukcji po n, a potem prześledźmy, co poszło nie tak.

```
Lemma even_plus_failed_1 :
  ∀ n m : nat, even n → even m → even (n + m).
Proof.
  induction n; intros.
    assumption.
    simpl. constructor. inversion H; subst.
Abort.
```

Nasza indukcja po n zawiodła, gdyż nasza hipoteza indukcyjna ma w konkluzji *even* $(n + m)$, podczas gdy nasz cel jest postaci *odd* $(n + m)$. Zauważmy, że teoretycznie cel powinno dać się udowodnić, jako że mamy hipotezy *even* m oraz *odd* n, a suma liczby parzystej i nieparzystej jest nieparzysta.

Nie zrażajmy się jednak i spróbujmy indukcji po dowodzie *even* n.

```
Lemma even_plus_failed_2 :
  ∀ n m : nat, even n → even m → even (n + m).
Proof.
  induction 1; simpl; intro.
    assumption.
    constructor.
```

```
Abort.
```

Nasza indukcja po dowodzie hipotezy *even n* zawiodła, i to z kretesem, gdyż w kontekście nie mamy nawet żadnej hipotezy indukcyjnej! Co właściwie się stało?

```
Check even_ind.
(*  ===> even_ind :
     forall P : nat -> Prop,
     P 0 -> (forall n : nat, odd n -> P (S n)) ->
       forall n : nat, even n -> P n *)
```

Jak widać, w naszej hipotezie "indukcyjnej" wygenerowanej przez Coqa w ogóle nie ma żadnej indukcji. Jest tam jedynie odwołanie do predykatu *odd*...

Zauważmy jednak, że naszym celem znów było *odd* $(n + m)$, a hipotezy *odd n* oraz *even m* sprawiają, że w teorii powinno dać się ten cel udowodnić, gdyż suma liczby parzystej i nieparzystej jest nieparzysta.

Mogłoby się zdawać, że cierpimy na niedopasowanie (próba 1) lub brak (próba 2) hipotez indukcyjnych. Wydaje się też, że skoro w obydwu próbach zatrzymaliśmy się na celu *odd* $(n + m)$, to pomocne mogłoby okazać się poniższe twierdzenie.

```
Lemma odd_even_plus_failed :
  ∀ n m : nat, odd n → even m → odd (n + m).
Proof.
  induction n; intros.
    inversion H.
    simpl. constructor. inversion H; subst.
Abort.
```

Niestety — nie dla psa kiełbasa, gdyż natykamy się na problemy bliźniaczo podobne do tych, które napotkaliśmy w poprzednim twierdzeniu: nasza hipoteza indukcyjna ma w konkluzji *odd* $(n + m)$, podczas gdy nasz cel jest postaci *even* $(n + m)$.

Próba przepchnięcia lematu za pomocą indukcji po dowodzie hipotezy *odd n* także nie zadziała, z tych samych powodów dla których indukcja po *even n* nie pozwoliła nam udowodnić *even_plus*. Zauważmy jednak, że cel jest udowadnialny, gdyż jako hipotezy mamy *even n* oraz *even m*, a suma dwóch liczb parzystych jest parzysta.

Wydaje się, że wpadliśmy w błędne koło i jesteśmy w matni, bez wyjścia, bez nadziei, bez krzty szans na powodzenie: w dowodzie *even_plus* potrzebujemy lematu *odd_even_plus*, ale nie możemy go udowodnić, gdyż w dowodzie *odd_even_plus* wymagane jest użycie lematu *even_plus*. Ehhh, gdybyśmy tak mogli udowodnić oba te twierdzenia na raz...

Eureka!

Zauważ, że w naszych dotychczasowych dowodach przez indukcję posługiwaliśmy się zwykłą, "pojedynczą" indukcją. Była ona wystarczająca, gdyż mieliśmy do czynienia jedynie ze zwykłymi typami induktywnymi. Tym razem jednak jest inaczej: w ostatnich trzech dowodach chcieliśmy użyć "pojedynczej" indukcji do udowodnienia czegoś na temat predykatów wzajemnie induktywnych.

Jest to ogromny zgrzyt. Do dowodzenia właściwości typów wzajemnie induktywnych powinniśmy użyć... o zgrozo, jak mogliśmy to przeoczyć, przecież to takie oczywiste... indukcji wzajemnej!

Najprostszy sposób przeprowadzenia tego dowodu wygląda tak:

```
Theorem even_plus :
  ∀ n m : nat, even n → even m → even (n + m)
with odd_even_plus :
  ∀ n m : nat, odd n → even m → odd (n + m).
Proof.
  assumption.
  assumption.
Fail Qed.
Restart.
  destruct n as [| n']; simpl; intros.
    assumption.
    constructor. apply odd_even_plus.
      inversion H. assumption.
      assumption.
  destruct n as [| n']; simpl; intros.
    inversion H.
    constructor. apply even_plus.
      inversion H. assumption.
      assumption.
Qed.
```

Co tu się właściwie stało? Pierwsze dwie linijki są takie same jak poprzednio: stwierdzamy, że będziemy dowodzić twierdzenia o podanej nazwie i postaci. Następnie mamy słowo kluczowe `with`, które pełni tu rolę podobną jak w definicjach przez indukcję wzajemną: podając po nim nazwę i postać twierdzenia mówimy Coqowi, że chcemy dowodzić tego twierdzenia (*odd_even_plus*) jednocześnie z poprzednim (*even_plus*).

Dotychczas po rozpoczęciu dowodu ukazywał nam się jeden cel. Tym razem, jako że dowodzimy dwóch twierdzeń jednocześnie, mamy przed sobą dwa cele. W kontekście mamy też od razu dwie hipotezy indukcyjne. Musimy na nie bardzo uważać: dotychczas hipotezy indukcyjne pojawiały się dopiero w kroku indukcyjnym i sposób ich użycia był oczywisty. Tym razem jest inaczej — jako, że mamy je od samego początku, możemy natychmiast użyć ich do "udowodnienia" naszych twierdzeń.

Niestety, takie "udowodnienie" odpowiada wywołaniu rekurencyjnemu na argumencie, który nie jest strukturalnie mniejszy (coś jak $f\ x := f\ x$). Fakt ten obrazuje wiadomość o błędzie, jaką Coq daje nam po tej próbie:

```
(*  ===> Error: Cannot guess decreasing argument of fix. *)
```

Zaczynamy dowód od nowa, tym razem już bez oszukiwania. Musimy udowodnić każdy z naszych celów osobno, ale możemy korzystać z obydwu hipotez indukcyjnych. W obydwu celach zaczynamy od analizy przypadków, czyli rozbicia n, i rozwiązania przypadku bazowego. Rozbicie n dało nam n', które jest strukturalnie mniejsze od n, a zatem możemy bez obaw użyć naszej hipotezy indukcyjnej. Reszta jest trywialna.

```
Theorem even_double :
  ∀ n : nat, even (2 × n).
Proof.
  induction n as [| n']; simpl in *; constructor.
  rewrite ← plus_n_O in *. rewrite plus_comm. simpl. constructor.
    assumption.
Qed.
```

```
End MutInd.
```

3.5 Różne

3.5.1 Rodziny typów induktywnych

Słowo kluczowe `Inductive` pozwala nam definiować nie tylko typy induktywne, ale także rodziny typów induktywnych — i to nawet na dwa sposoby. W tym podrozdziale przyjrzymy się obu z nich oraz różnicom między nimi, a także ich wadom i zaletom. Przyjrzyjmy się raz jeszcze typowi *option*:

```
Print option.
(*  ===> Inductive option (A : Type) : Type :=
            | Some : A -> option A
            | None : option A *)
```

```
Check Some.
(*  ===> Some : forall A : Type, A -> option A *)
```

```
Check @None.
(*  ===> @None : forall A : Type, option A *)
```

Definiując rodzinę typów *option*, umieściliśmy argument będący typem w nawiasach okrągłych tuż po nazwie definiowanego typu, a przed : `Type`. Definiując konstruktory, nie napisaliśmy nigdzie $\forall$ A : `Type`, ..., a mimo tego komenda `Check` jasno pokazuje, że typy obydwu konstruktorów zaczynają się od takiej właśnie kwantyfikacji.

(Przypomnijmy, że w przypadku *None* argument A jest domyślny, więc wyświetlenie pełnego typu tego konstruktora wymagało użycia symbolu @, który oznacza "wyświetl wszystkie argumenty domyślne").

W ogólności, definiowanie rodziny typów T jako T $(x1 : A1)$ ... $(xN : AN)$ ma następujący efekt:

- kwantyfikacja $\forall$ $(x1 : A1)$ ... $(xN : AN)$ jest dodawana na początek każdego konstruktora

- w konkluzji konstruktora *T* musi wystąpić zaaplikowany do tych argumentów, czyli jako *T x1 ... xN* — wstawienie innych argumentów jest błędem

Fail `Inductive` *option'* (A : `Type`) : `Type` :=
  | *Some'* : $A \to$ *option'* A
  | *None'* : $\forall$ B : `Type`, *option'* B.

Próba zdefiniowania typu *option'* kończy się następującym komunikatem o błędzie:

```
(*  Error: Last occurrence of óption'" must have Ą" as 1st argument in
   "forall B : Type, option' B". *)
```

Drugi sposób zdefiniowania rodziny typów *option* przedstawiono poniżej. Tym razem zamiast umieszczać argument A : `Type` po nazwie definiowanego typu, deklarujemy, że typem *option'* jest `Type` $\to$ `Type`.

`Inductive` *option'* : `Type` $\to$ `Type` :=
  | *Some'* : $\forall$ A : `Type`, $A \to$ *option'* A
  | *None'* : $\forall$ B : `Type`, *option'* B.

Taki zabieg daje nam większą swobodę: w każdym konstruktorze z osobna musimy explicité umieścić kwantyfikację po argumencie sortu `Type`, dzięki czemu różne konstruktory mogą w konkluzji mieć *option'* zaaplikowany do różnych argumentów.

`Check` *Some'*.

```
(*  ===> Some' : forall A : Type, A -> option' A *)
```

`Check` *None'*.

```
(*  ===> None' : forall B : Type, option' B *)
```

Zauważmy jednak, że definicje *option* i *option'* są równoważne — typ konstruktora *None'* różni się od typu *None* jedynie nazwą argumentu (A dla *None*, B dla *None'*).

Jak zatem rozstrzygnąć, który sposób definiowania jest "lepszy"? W naszym przypadku lepszy jest sposób pierwszy, odpowiadający typowi *option*, gdyż jest bardziej zwięzły. Nie jest to jednak jedyne kryterium.

`Check` *option_ind*.

```
(*  ===> option_ind :
            forall (A : Type) (P : option A -> Prop),
            (forall a : A, P (Some a)) -> P None ->
            forall o : option A, P o *)
```

`Check` *option'_ind*.

```
(*  ===> option'_ind :
            forall P : forall T : Type, option' T -> Prop,
            (forall (A : Type) (a : A), P A (Some' A a)) ->
            (forall B : Type, P B (None' B)) ->
            forall (T : Type) (o : option' T), P T o *)
```

Dwa powyższe termy to reguły indukcyjne, wygenerowane automatycznie przez Coqa dla typów *option* oraz *option'*. Reguła dla *option* jest wizualnie krótsza, co, jak dowiemy się w przyszłości, oznacza zapewne, że jest prostsza, zaś prostsza reguła indukcyjna oznacza łatwiejsze dowodzenie przez indukcję. Jest to w zasadzie najmocniejszy argument przemawiający za pierwszym sposobem zdefiniowania *option*.

Powyższe rozważania nie oznaczają jednak, że sposób pierwszy jest zawsze lepszy — sposób drugi jest bardziej ogólny i istnieją rodziny typów, których zdefiniowanie sposobem pierwszym jest niemożliwe. Klasycznym przykładem jest rodzina typów *vec*.

```
Inductive vec (A : Type) : nat → Type :=
    | vnil : vec A 0
    | vcons : ∀ n : nat, A → vec A n → vec A (S n).
```

Konstruktor *vnil* reprezentuje listę pustą, której długość wynosi rzecz jasna 0, zaś *vcons* reprezentuje listę składająca się z głowy i ogona o długości *n*, której długość wynosi oczywiście *S n*.

vec reprezetuje listy o długości znanej statycznie (tzn. Coq zna długość takiej listy już w trakcie sprawdzania typów), dzięki czemu możemy obliczać ich długość w czasie stałym (po prostu odczytując ją z typu danej listy).

Zauważ, że w obu konstruktorach argumenty typu *nat* są różne, a zatem zdefiniowanie tego typu jako *vec* (*A* : `Type`) (*n* : *nat*) ... byłoby niemożliwe.

Przykład ten pokazuje nam również, że przy definiowaniu rodzin typów możemy dowolnie mieszać sposoby pierwszy i drugi — w naszym przypadku argument *A* : `Type` jest wspólny dla wszystkich konstruktorów, więc umieszczamy go przed ostatnim :, zaś argument typu *nat* różni się w zależności od konstruktora, a zatem umieszczamy go po ostatnim :.

Ćwiczenie Zdefiniuj następujące typy (zadbaj o to, żeby wygenerowana reguła indukcyjna była jak najkrótsza):

- typ drzew binarnych przechowujących elementy typu *A*
- typ drzew binarnych przechowujących elementy typu *A*, których wysokość jest znana statycznie
- typ heterogenicznych drzew binarnych (mogą one przechowywać elementy różnych typów)
- typ heterogenicznych drzew binarnych, których wysokość jest znana statycznie

3.5.2 Indukcja wzajemna a indeksowane rodziny typów

```
Module MutualIndution_vs_InductiveFamilies.
```

TODO: napisać tu coś.

```
Inductive even : nat → Prop :=
    | even0 : even 0
```

```
    | evenS : ∀ n : nat, odd n → even (S n)

with odd : nat → Prop :=
    | oddS : ∀ n : nat, even n → odd (S n).

Inductive even_odd : bool → nat → Prop :=
    | even0' : even_odd true 0
    | evenS' : ∀ n : nat, even_odd false n → even_odd true (S n)
    | oddS' : ∀ n : nat, even_odd true n → even_odd false (S n).

Definition even' := even_odd true.
Definition odd' := even_odd false.

Lemma even_even' :
  ∀ n : nat, even n → even' n

with odd_odd' :
  ∀ n : nat, odd n → odd' n.

Lemma even'_even :
  ∀ n : nat, even' n → even n

with odd'_odd :
  ∀ n : nat, odd' n → odd n.

End MutualIndution_vs_InductiveFamilies.
```

3.5.3 Sumy zależne i podtypy

W Coqu, w przeciwieństwie do wielu języków imperatywnych, nie ma mechanizmu subtypowania, a każde dwa typy są ze sobą rozłączne. Nie jest to problemem, gdyż subtypowanie możemy zasymulować za pomocą sum zależnych, a te zdefiniować możemy induktywnie.

```
Module sigma.

Inductive sigT (A : Type) (P : A → Type) : Type :=
    | existT : ∀ x : A, P x → sigT A P.
```

Typ *sigT* reprezentuje sumę zależną, której elementami są pary zależne. Pierwszym elementem pary jest x, który jest typu A, zaś drugim elementem pary jest term typu $P\ x$. Suma zależna jest wobec tego pewnym uogólnieniem produktu.

Niech cię nie zmyli nazewnictwo:

- Suma jest reprezentowana przez typ *sum* A B. Jej elementami są elementy A zawinięte w konstruktor *inl* oraz elementy B zawinięte w konstruktor *inr*. Reprezentuje ideę "lub/albo". Typ B nie może zależeć od typu A.

- Produkt jest reprezentowany przez typ *prod* A B. Jego elementami są pary elementów A i B. Reprezentuje on ideę "i/oraz". Typ B nie może zależeć od typu A.

- Uogólnieniem produktu jest suma zależna. Jest ona reprezentowana przez typ *sigT* A P. Jej elementami są pary zależne elementów A i P x, gdzie $x : A$ jest pierwszym elementem pary. Reprezentuje ona ideę “i/oraz”, gdzie typ P x może zależeć od elementu x typu A.

- Typ funkcji jest reprezentowany przez $A \rightarrow B$. Jego elementami są termy postaci `fun` $x : A \Rightarrow$ .... Reprezentują ideę “daj mi coś typu A, a ja oddam ci coś typu B”. Typ B nie może zależeć od typu A.

- Uogólnieniem typu funkcji jest produkt zależny $\forall\ x : A, B\ x$. Jego elementami są termu postaci `fun` $x : A \Rightarrow$ .... Reprezentuje on ideę “daj mi x typu A, a ja oddam ci coś typu B x”. Typ B x może zależeć od typu elementu x typu A.

sigT jest najogólniejszą postacią pary zależnej — A jest typem, a P rodziną typów. Mimo swej ogólności jest używany dość rzadko, gdyż najbardziej przydatną postacią sumy zależnej jest typ *sig*:

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
  | exist : ∀ x : A, P x → sig A P.
Arguments exist [A] [P] _ _.
```

Typ *sig* A P można interpretować jako typ składający się z tych elementów A, które spełniają predykat P. Formalnie jest to para zależna, której pierwszym elementem jest term typu A, zaś drugim dowód na to, że spełnia on predykat P.

```
Definition even_nat : Type := sig nat even.

Definition even_four : even_nat := exist 4 four_is_even.
```

Typ *even_nat* reprezentuje parzyste liczby naturalne, zaś term *even_four* to liczba 4 wraz z załączonym dowodem faktu, że 4 jest parzyste.

Interpretacja typu *sig* sprawia, że jest on wykorzystywany bardzo często do podawania specyfikacji programów — pozwala on dodać do wyniku zwracanego przez funkcję informację o jego właściwościach. W przypadku argumentów raczej nie jest używany, gdyż prościej jest po prostu wymagać dowodów żądanych właściwości w osobnych argumentach niż pakować je w *sig* po to, żeby i tak zostały później odpakowane.

```
Definition even_42 : sig nat even.
Proof.
  apply (exist 42). repeat constructor.
Defined.
```

Definiowanie wartości typu *sig* jest problematyczne, gdyż zawierają one dowody. Napisanie definicji “ręcznie”, explicité podając proofterm, nie wchodzi w grę. Innym potencjalnym rozwiązaniem jest napisanie dowodu na boku, a następnie użycie go we właściwej definicji, ale jest ono dłuższe niż to konieczne.

Przypomnijmy sobie, czym są taktyki. Dowody to termy, których typy są sortu `Prop`, a taktyki służą do konstruowania tych dowodów. Ponieważ dowody nie różnią się (prawie) niczym od programów, taktyk można użyć także do pisania programów. Taktyki to metaprogramy (napisane w języku Ltac), które piszą programy (w języku termów Coqa, zwanym Gallina).

Wobec tego trybu dowodzenia oraz taktyk możemy używać nie tylko do dowodzenia, ale także do definiowania i to właśnie uczyniliśmy w powyższym przykładzie. Skonstruowanie termu typu *sig nat even*, czyli parzystej liczby naturalnej, odbyło się w następujący sposób.

Naszym celem jest początkowo *sig nat even*, czyli typ, którego element chcemy skonstrować. Używamy konstruktora *exist*, który w naszym przypadku jest typu $\forall\ x : nat,\ even\ n \to sig\ nat\ even$. Wobec tego *exist* 42 jest typu *even* 42 $\to$ *sig nat even*, a jego zaaplikowanie skutkować będzie zamianą naszego celu na *even* 42. Następnie dowodzimy tego faktu, co kończy proces definiowania.

Ćwiczenie Zdefiniuj predykat *sorted*, który jest spełniony, gdy jego argument jest listą posortowaną. Następnie zdefiniuj typ list liczb naturalnych posortowanych według relacji $\leq$ i skonstruuj term tego typu odpowiadający liście [42; 666; 1337].

`End` *sigma*.

3.5.4 Kwantyfikacja egzystencjalna

Znamy już pary zależne i wiemy, że mogą służyć do reprezentowania podtypów, których w Coqu brak. Czas zatem uświadomić sobie kolejny fragment korespondencji Curry'ego-Howarda, a mianowicie definicję kwantyfikacji egzystencjalnej:

`Module` *ex*.

`Inductive` *ex* $(A$: `Type`$)$ $(P : A \to$ `Prop`$)$: `Prop` :=
  | *ex_intro* : $\forall\ x : A,\ P\ x \to ex\ A\ P$.

ex to kolejne wcielenia sumy zależnej. Porównaj dokładnie tę definicję z definicją *sigT* oraz *sig*. *ex* jest niemal identyczne jak *sig*: jest to para zależna, której pierwszym elementem jest term $x : A$, a drugim dowód na to, że $P\ x$ zachodzi. *ex* jednak, w przeciwieństwie do *sig*, żyje w `Prop`, czyli jest zdaniem — nie liczą się konkretne postaci jego termów ani ich ilość, a jedynie fakt ich istnienia. To sprawia, że *ex* jest doskonałym kandydatem do reprezentowania kwantyfikacji egzystencjalnej.

Ćwiczenie Udowodnij, że dla każdej liczby naturalnej n istnieje liczba od niej większa. Następnie zastanów się, jak działa taktyka $\exists$.

`Theorem` *exists_greater* :
  $\forall\ n : nat,\ ex\ nat$ (`fun` $k : nat \Rightarrow n < k$).

`End` *ex*.

3.5.5 W-typy

TODO: napisz coś

```
Inductive W (A : Type) (B : A → Type) : Type :=
    | sup : ∀ x : A, (B x → W A B) → W A B.

Arguments sup {A B} _ _.

Definition boolW : Type :=
  W bool (fun _ ⇒ Empty_set).

Definition trueW : boolW :=
  sup true (fun e : Empty_set ⇒ match e with end).

Definition falseW : boolW :=
  sup false (fun e : Empty_set ⇒ match e with end).

Definition notW : boolW → boolW :=
  W_rect bool (fun _ ⇒ Empty_set) (fun _ ⇒ boolW)
        (fun b _ _ ⇒ if b then falseW else trueW).

Definition bool_boolW (b : bool) : boolW :=
  if b then trueW else falseW.

Definition boolW_bool : boolW → bool :=
  W_rect bool (fun _ ⇒ Empty_set) (fun _ ⇒ bool) (fun b _ _ ⇒ b).

Lemma boolW_bool_notW :
  ∀ b : boolW,
    boolW_bool (notW b) = negb (boolW_bool b).

Lemma boolW_bool__bool_boolW :
  ∀ b : bool,
    boolW_bool (bool_boolW b) = b.

Lemma bool_boolW__bool_boolW :
  ∀ b : boolW,
    bool_boolW (boolW_bool b) = b.

Definition natW : Type :=
  W bool (fun b : bool ⇒ if b then Empty_set else unit).

Definition zeroW : natW :=
  sup true (fun e : Empty_set ⇒ match e with end).

Definition succW (n : natW) : natW :=
  sup false (fun u : unit ⇒ n).

Definition doubleW : natW → natW :=
  W_rect _ (fun b : bool ⇒ if b then Empty_set else unit) (fun _ ⇒ natW)
    (fun a ⇒
      match a with
          | true ⇒ fun _ _ ⇒ zeroW
          | false ⇒ fun _ g ⇒ succW (succW (g tt))
      end).
```

```
Definition natW_nat :=
  W_rect _ (fun b : bool ⇒ if b then Empty_set else unit) (fun _ ⇒ nat)
    (fun a ⇒
      match a with
          | true ⇒ fun _ _ ⇒ 0
          | false ⇒ fun _ g ⇒ S (g tt)
      end).

Fixpoint nat_natW (n : nat) : natW :=
match n with
    | 0 ⇒ zeroW
    | S n' ⇒ succW (nat_natW n')
end.

Lemma natW_nat_doubleW :
  ∀ n : natW,
    natW_nat (doubleW n) = 2 × natW_nat n.

Lemma natW_nat__nat_natW :
  ∀ n : nat,
    natW_nat (nat_natW n) = n.

Lemma nat_natW__nat_natW :
  ∀ n : natW,
    nat_natW (natW_nat n) = n.
```

3.6 Wyższe czary

Najwyższy czas nauczyć się czegoś tak zaawansowanego, że nawet w Coqu (pełnym przecież dziwnych rzeczy) tego nie ma i nie zapowiada się na to, że będzie. Mam tutaj na myśli mechanizm definiwania typów, którego nazwa jest wybitnie mało oświecająca: indukcja-indukcja.

```
Unset Elimination Schemes.
```

Powyższa komenda mówi Coqowi, żeby nie generował automatycznie reguł indukcji. Przyda nam się ona, by uniknąć konfliktów nazw z regułami, które będziemy pisać ręcznie.

3.6.1 Przypomnienie

Zanim jednak wyjaśnimy, co to za stfur, przypomnijmy sobie różne, coraz bardziej innowacyjne sposoby definiowania przez indukcję. Przypomnimy sobie też, jak sformułować ich reguły rekursji oraz indukcji.

Enumeracje

```
Module enum.

Inductive I : Type :=
    | c0 : I
    | c1 : I
    | c2 : I.
```

Najprymitywniejszymi z typów induktywnych są enumeracje. Definiując je, wymieniamy po prostu wszystkie ich elementy.

```
Definition I_case_nondep_type : Type :=
  ∀ P : Type, P → P → P → I → P.
```

Reguła definiowania przez przypadki jest banalnie prosta: jeżeli w jakimś inny typie P uda nam się znaleźć po jednym elemencie dla każdego z elementów naszego typu I, to możemy zrobić funkcję $I \to P$.

```
Definition I_case_nondep : I_case_nondep_type :=
  fun (P : Type) (c0' c1' c2' : P) (i : I) ⇒
  match i with
      | c0 ⇒ c0'
      | c1 ⇒ c1'
      | c2 ⇒ c2'
  end.
```

Regułę zdefiniować możemy za pomocą dopasowania do wzorca.

```
Definition I_case_dep_type : Type :=
  ∀ (P : I → Type) (c0' : P c0) (c1' : P c1) (c2' : P c2),
    ∀ i : I, P i.
```

Zależną regułę definiowania przez przypadki możemy uzyskać z poprzedniej uzależniając przeciwdziedzinę P od dziedziny.

```
Definition I_case_dep : I_case_dep_type :=
  fun (P : I → Type) (c0' : P c0) (c1' : P c1) (c2' : P c2) (i : I) ⇒
  match i with
      | c0 ⇒ c0'
      | c1 ⇒ c1'
      | c2 ⇒ c2'
  end.
```

Definicja, jak widać, jest taka sama jak poprzednio, więc obliczeniowo obie reguły zachowują się tak samo. Różnica leży jedynie w typach - druga reguła jest ogólniejsza.

```
End enum.
```

Konstruktory rekurencjne

```
Module rec.

Inductive I : Type :=
    | x : I → I
    | D : I → I.
```

Typy induktywne stają się naprawdę induktywne, gdy konstruktory mogą brać argumenty typu, który właśnie definiujemy. Dzięki temu możemy tworzyć type, które mają nieskończenie wiele elementów, z których każdy ma kształt takiego czy innego drzewa.

```
Definition I_rec_type : Type :=
  ∀ P : Type, (P → P) → (P → P) → I → P.
```

Typ reguły rekursji (czyli rekursora) tworzymy tak jak dla enumeracji: jeżeli w typie P znajdziemy rzeczy o takim samym kształcie jak konstruktory typu I, to możemy zrobić funkcję $I \to P$. W naszym przypadku oba konstruktory mają kształt $I \to I$, więc do zdefiniowania naszej funkcji musimy znaleźć odpowiadające im rzeczy typu $P \to P$.

```
Fixpoint I_rec (P : Type) (x' : P → P) (D' : P → P) (i : I) : P :=
match i with
    | x i' ⇒ x' (I_rec P x' D' i')
    | D i' ⇒ D' (I_rec P x' D' i')
end.
```

Definicja rekursora jest prosta. Jeżeli wyobrazimy sobie $i : I$ jako drzewo, to węzły z etykietką x zastępujemy wywołaniem funkcji *x'*, a węzły z etykietką D zastępujemy wywołaniami funkcji D.

```
Definition I_ind_type : Type :=
  ∀ (P : I → Type)
    (x' : ∀ i : I, P i → P (x i))
    (D' : ∀ i : I, P i → P (D i)),
      ∀ i : I, P i.
```

Reguła indukcji (czyli induktor - cóż za piękna nazwa!) powstaje z reguły rekursji przez uzależnienie przeciwdziedziny P od dziedziny I.

```
Fixpoint I_ind (P : I → Type)
  (x' : ∀ i : I, P i → P (x i)) (D' : ∀ i : I, P i → P (D i))
  (i : I) : P i :=
match i with
    | x i' ⇒ x' i' (I_ind P x' D' i')
    | D i' ⇒ D' i' (I_ind P x' D' i')
end.
```

Podobnie jak poprzednio, implementacja reguły indukcji jest identyczna jak rekursora, jedynie typy są bardziej ogólnej.

Uwaga: nazywam reguły nieco inaczej niż te autogenerowane przez Coqa. Dla Coqa reguła indukcji dla I to nasze I_ind z $P : I \to$ `Type` zastąpionym przez $P : I \to$ `Prop`, zaś Coqowe I_rec odpowiadałoby naszemu I_ind dla $P : I \to$ `Set`.

Jeżeli smuci cię burdel nazewniczy, to nie przejmuj się - kiedyś będzie lepiej. Klasyfikacja reguł jest prosta:

- reguły mogą być zależne lub nie, w zależności od tego czy P zależy od I
- reguły mogą być rekurencyjne lub nie
- reguły mogą być dla sortu `Type`, `Prop` albo nawet `Set`

```
End rec.
```

Parametry

```
Module param.

Inductive I (A B : Type) : Type :=
    | c0 : A → I A B
    | c1 : B → I A B
    | c2 : A → B → I A B.

Arguments c0 {A B} _.
Arguments c1 {A B} _.
Arguments c2 {A B} _ _.
```

Kolejną innowacją są parametry, których głównym zastosowaniem jest polimorfizm. Dzięki parametrom możemy za jednym zamachem (tylko bez skojarzeń z Islamem!) zdefiniować nieskończenie wiele typów, po jednym dla każdego parametru.

```
Definition I_case_nondep_type : Type :=
  ∀ (A B P : Type) (c0' : A → P) (c1' : B → P) (c2' : A → B → P),
    I A B → P.
```

Typ rekursora jest oczywisty: jeżeli znajdziemy rzeczy o kształtach takich jak konstruktory I z I zastąpionym przez P, to możemy zrobić funkcję $I \to P$. Jako, że parametry są zawsze takie samo, możemy skwantyfikować je na samym początku.

```
Definition I_case_nondep
  (A B P : Type) (c0' : A → P) (c1' : B → P) (c2' : A → B → P)
  (i : I A B) : P :=
match i with
    | c0 a ⇒ c0' a
    | c1 b ⇒ c1' b
    | c2 a b ⇒ c2' a b
end.
```

Implementacja jest banalna.

```
Definition I_case_dep_type : Type :=
  ∀ (A B : Type) (P : I A B → Type)
    (c0' : ∀ a : A, P (c0 a))
    (c1' : ∀ b : B, P (c1 b))
    (c2' : ∀ (a : A) (b : B), P (c2 a b)),
      ∀ i : I A B, P i.
```

A regułę indukcję uzyskujemy przez uzależnienie *P* od *I*.

```
Definition I_case_dep
  (A B : Type) (P : I A B → Type)
  (c0' : ∀ a : A, P (c0 a))
  (c1' : ∀ b : B, P (c1 b))
  (c2' : ∀ (a : A) (b : B), P (c2 a b))
  (i : I A B) : P i :=
match i with
    | c0 a ⇒ c0' a
    | c1 b ⇒ c1' b
    | c2 a b ⇒ c2' a b
end.
```

```
End param.
```

Indukcja wzajemna

```
Module mutual.
```

```
Inductive Smok : Type :=
    | Wysuszony : Zmok → Smok

with Zmok : Type :=
    | Zmoczony : Smok → Zmok.
```

Indukcja wzajemna pozwala definiować na raz wiele typów, które mogą odwoływać się do siebie nawzajem. Cytując klasyków: smok to wysuszony zmok, zmok to zmoczony smok.

```
Definition Smok_case_nondep_type : Type :=
  ∀ S : Type, (Zmok → S) → Smok → S.
```

```
Definition Zmok_case_nondep_type : Type :=
  ∀ Z : Type, (Smok → Z) → Zmok → Z.
```

Reguła niezależnej analizy przypadków dla *Smok*a wygląda banalnie: jeżeli ze *Zmok*a potrafimy wyprodukować *S*, to ze *Smok*a też. Dla *Zmok*a jest analogicznie.

```
Definition Smok_case_nondep
  (S : Type) (Wy : Zmok → S) (smok : Smok) : S :=
match smok with
```

```
    | Wysuszony zmok => Wy zmok
end.

Definition Zmok_case_nondep
  (Z : Type) (Zm : Smok -> Z) (zmok : Zmok) : Z :=
match zmok with
    | Zmoczony smok => Zm smok
end.
```

Implementacja jest banalna.

```
Definition Smok_rec_type : Type :=
  forall S Z : Type, (Z -> S) -> (S -> Z) -> Smok -> S.

Definition Zmok_rec_type : Type :=
  forall S Z : Type, (Z -> S) -> (S -> Z) -> Zmok -> Z.
```

Typ rekursora jest jednak nieco bardziej zaawansowany. Żeby zdefiniować funkcję typu *Smok* → *S*, musimy mieć nie tylko rzeczy w kształcie konstruktorów *Smok*a, ale także w kształcie konstruktorów *Zmok*a, gdyż rekurencyjna struktura obu typów jest ze sobą nierozerwalnie związana.

```
Fixpoint Smok_rec
  (S Z : Type) (Wy : Z -> S) (Zm : S -> Z) (smok : Smok) : S :=
match smok with
    | Wysuszony zmok => Wy (Zmok_rec S Z Wy Zm zmok)
end

with Zmok_rec
  (S Z : Type) (Wy : Z -> S) (Zm : S -> Z) (zmok : Zmok) : Z :=
match zmok with
    | Zmoczony smok => Zm (Smok_rec S Z Wy Zm smok)
end.
```

Implementacja wymaga rekursji wzajemnej, ale poza nie jest jakoś wybitnie groźna.

```
Definition Smok_ind_type : Type :=
  forall (S : Smok -> Type) (Z : Zmok -> Type)
    (Wy : forall zmok : Zmok, Z zmok -> S (Wysuszony zmok))
    (Zm : forall smok : Smok, S smok -> Z (Zmoczony smok)),
      forall smok : Smok, S smok.

Definition Zmok_ind_type : Type :=
  forall (S : Smok -> Type) (Z : Zmok -> Type)
    (Wy : forall zmok : Zmok, Z zmok -> S (Wysuszony zmok))
    (Zm : forall smok : Smok, S smok -> Z (Zmoczony smok)),
      forall zmok : Zmok, Z zmok.

Fixpoint Smok_ind
  (S : Smok -> Type) (Z : Zmok -> Type)
```

```
  (Wy : ∀ zmok : Zmok, Z zmok → S (Wysuszony zmok))
  (Zm : ∀ smok : Smok, S smok → Z (Zmoczony smok))
  (smok : Smok) : S smok :=
match smok with
    | Wysuszony zmok ⇒ Wy zmok (Zmok_ind S Z Wy Zm zmok)
end

with Zmok_ind
  (S : Smok → Type) (Z : Zmok → Type)
  (Wy : ∀ zmok : Zmok, Z zmok → S (Wysuszony zmok))
  (Zm : ∀ smok : Smok, S smok → Z (Zmoczony smok))
  (zmok : Zmok) : Z zmok :=
match zmok with
    | Zmoczony smok ⇒ Zm smok (Smok_ind S Z Wy Zm smok)
end.
```

Mając rekursor, napisanie typu reguły indukcji jest banalne, podobnie jak jego implementacja.

```
End mutual.
```

Indeksy

```
Module index.

Inductive I : nat → Type :=
    | c0 : bool → I 0
    | c42 : nat → I 42.
```

Ostatnią poznaną przez nas innowacją są typy indeksowane. Tutaj również definiujemy za jednym zamachem (ekhem...) dużo typów, ale nie są one niezależne jak w przypadku parametrów, lecz mogą od siebie wzajemnie zależeć. Słowem, tak naprawdę definiujemy przez indukcję funkcję typu *A_1* → ... → *A_n* → `Type`/`Prop`, gdzie *A_i* to indeksy.

```
Definition I_case_very_nondep_type : Type :=
  ∀ (P : Type) (c0' : bool → P) (c42' : nat → P),
    ∀ n : nat, I n → P.

Definition I_case_very_nondep
  (P : Type) (c0' : bool → P) (c42' : nat → P)
  {n : nat} (i : I n) : P :=
match i with
    | c0 b ⇒ c0' b
    | c42 n ⇒ c42' n
end.
```

Możliwych reguł analizy przypadków mamy tutaj trochę więcej niż w przypadku parametrów. W powyższej regule *P* nie zależy od indeksu *n* : *nat*...

```
Definition I_case_nondep_type : Type :=
  ∀ (P : nat → Type) (c0' : bool → P 0) (c42' : nat → P 42),
    ∀ n : nat, I n → P n.

Definition I_case_nondep
  (P : nat → Type) (c0' : bool → P 0) (c42' : nat → P 42)
  {n : nat} (i : I n) : P n :=
match i with
    | c0 b ⇒ c0' b
    | c42 n ⇒ c42' n
end.
```

... a w powyższej tak. Jako, że indeksy zmieniają się pomiędzy konstruktorami, każdy z nich musi kwantyfikować je osobno (co akurat nie jest potrzebne w naszym przykładzie, gdyż jest zbyt prosty).

```
Definition I_case_dep_type : Type :=
  ∀ (P : ∀ n : nat, I n → Type)
    (c0' : ∀ b : bool, P 0 (c0 b))
    (c42' : ∀ n : nat, P 42 (c42 n)),
      ∀ (n : nat) (i : I n), P n i.

Definition I_case_dep
  (P : ∀ n : nat, I n → Type)
  (c0' : ∀ b : bool, P 0 (c0 b))
  (c42' : ∀ n : nat, P 42 (c42 n))
  (n : nat) (i : I n) : P n i :=
match i with
    | c0 b ⇒ c0' b
    | c42 n ⇒ c42' n
end.
```

Ogólnie reguła jest taka: reguła niezależna (pierwsza) nie zależy od niczego, a zależna (trzecia) zależy od wszystkiego. Reguła druga jest pośrednia - ot, take ciepłe kluchy.

```
End index.
```

3.6.2 Indukcja-indukcja

```
Module ind_ind.
```

Po powtórce nadszedł czas nowości. Zacznijmy od nazwy, która jest iście kretyńska: indukcja-indukcja. Każdy rozsądny człowiek zgodzi się, że dużo lepszą nazwą byłoby coś w stylu "indukcja wzajemna indeksowana".

Ta alternatywna nazwa rzuca sporo światła: indukcja-indukcja to połączenie i uogólnienie mechanizmów definiowania typów wzajemnie induktywnych oraz indeksowanych typów induktywnych.

Typy wzajemnie induktywne mogą odnosić się do siebie nawzajem, ale co to dokładnie znaczy? Ano to, że konstruktory każdego typu mogą brać argumenty wszystkch innych typów definiowanych jednocześnie z nim. To jest clou całej sprawy: konstruktory.

A co to ma do typów indeksowanych? Ano, zastanówmy się, co by się stało, gdybyśmy chcieli zdefiniować przez wzajemną indukcję typ A oraz rodzinę typów $B : A \to$ `Type`. Otóż nie da się: konstruktory A mogą odnosić się do B i vice-versa, ale A nie może być indeksem B.

Indukcja-indukcja to coś, co... tam taram tam tam... pozwala właśnie na to: możemy jednocześnie zdefiniować typ i indeksowaną nim rodzinę typów. I wszystko to ukryte pod taką smutną nazwą... lobby teoriotypowe nie chciało, żebyś się o tym dowiedział.

Czas na przykład!

Fail

```
Inductive slist {A : Type} (R : A → A → Prop) : Type :=
    | snil : slist R
    | scons : ∀ (h : A) (t : slist A), ok h t → slist A

with ok {A : Type} {R : A → A → Prop} : A → slist R → Prop :=
    | ok_snil : ∀ x : A, ok x snil
    | ok_scons :
        ∀ (h : A) (t : slist A) (p : ok h t) (x : A),
          R x h → ok x (scons h t p).
(*  ===> The reference slist was not found in the current environment. *)
```

Jako się już wcześniej rzekło, indukcja-indukcja nie jest wspierana przez Coqa - powyższa definicja kończy się informacją o błędzie: Coq nie widzi *slist* kiedy czyta indeksy *ok* właśnie dlatego, że nie dopuszcza on możliwości jednoczesnego definiowania rodziny (w tym wypadku relacji) *ok* wraz z jednym z jej indeksów, *slist*.

Będziemy zatem musieli poradzić sobie z przykładem jakoś inaczej - po prostu damy go sobie za pomocą aksjomatów. Zanim jednak to zrobimy, omówimy go dokładniej, gdyż deklarowanie aksjomatów jest niebezpieczne i nie chcemy się pomylić.

Zamysłem powyższego przykładu było zdefiniowanie typu list posortowanych *slist R*, gdzie R pełni rolę relacji porządku, jednocześnie z relacją *ok* : $A \to$ *slist* $R \to$ `Prop`, gdzie *ok x l* wyraża, że dostawienie x na początek listy posortowanej l daje listę posortowaną.

Przykład jest oczywiście dość bezsensowny, bo dokładnie to samo można osiągnąć bez używania indukcji-indukcji - wystarczy najpierw zdefiniować listy, a potem relację bycia listą posortowaną, a na koniec zapakować wszystko razem. Nie będziemy się tym jednak przejmować.

Definicja *slist R* jest następująca:

- *snil* to lista pusta
- *scons* robi posortowaną listę z głowy h i ogona t pod warunkiem, że dostanie też dowód zdania *ok h t* mówiącego, że można dostawić h na początek listy t

Definicja *ok* też jest banalna:

- każdy $x : A$ może być dostawiony do pustej listy
- jeżeli mamy listę $scons\ h\ t\ p$ oraz element x, o którym wiemy, że jest mniejszy od h, tzn. $R\ x\ h$, to x może zostać dostawiony do listy $scons\ h\ t\ p$

Jak powinny wyglądać reguły rekursji oraz indukcji? Na szczęście wciąż działają schematy, które wypracowaliśmy dotychczas.

Reguła rekursji mówi, że jeżeli znajdziemy w typie P coś o kształcie $slist\ R$, a w relacji Q coś o kształcie ok, to możemy zdefiniować funkcję $slist\ R \to P$ oraz $\forall\ (x : A)\ (l : slist\ R),\ ok\ x\ l \to Q$.

Regułę indukcji można uzyskać dodając tyle zależności, ile tylko zdołamy unieść.

Zobaczmy więc, jak zrealizować to wszystko za pomocą aksjomatów.

```
Axioms
  (slist : ∀ {A : Type}, (A → A → Prop) → Type)
  (ok : ∀ {A : Type} {R : A → A → Prop}, A → slist R → Prop).
```

Najpierw musimy zadeklarować *slist*, gdyż wymaga tego typ *ok*. Obie definicje wyglądają dokładnie tak, jak nagłówki w powyższej definicji odrzuconej przez Coqa.

Widać też, że gdybyśmy chcieli zdefiniować rodziny A i B, które są nawzajem swoimi indeksami, to nie moglibyśmy tego zrobić nawet za pomocą aksjomatów. Rodzi to pytanie o to, które dokładnie definicje przez indukcję-indukcję są legalne. Odpowiedź brzmi: nie wiem, ale może kiedyś się dowiem.

```
Axioms
  (snil : ∀ {A : Type} {R : A → A → Prop}, slist R)
  (scons :
    ∀ {A : Type} {R : A → A → Prop} (h : A) (t : slist R),
      ok h t → slist R)
  (ok_snil :
    ∀ {A : Type} {R : A → A → Prop} (x : A), ok x (@snil _ R))
  (ok_scons :
    ∀
      {A : Type} {R : A → A → Prop}
      (h : A) (t : slist R) (p : ok h t)
      (x : A), R x h → ok x (scons h t p)).
```

Następnie definiujemy konstruktory: najpierw konstruktory *slist*, a potem *ok*. Musimy to zrobić w tej kolejności, bo konstruktor *ok_snil* odnosi się do *snil*, a *ok_scons* do *scons*.

Znowu widzimy, że gdyby konstruktory obu typów odnosiły się do siebie nawzajem, to nie moglibyśmy zdefiniować takiego typu aksjomatycznie.

```
Axiom
  (ind : ∀
    (A : Type) (R : A → A → Prop)
```

$$
\begin{array}{l}
(P : \mathit{slist}\ R \to \mathtt{Type}) \\
(Q : \forall\ (h : A)\ (t : \mathit{slist}\ R),\ \mathit{ok}\ h\ t \to \mathtt{Type}) \\
(\mathit{Psnil} : P\ \mathit{snil}) \\
(\mathit{Pscons} : \\
\quad \forall\ (h : A)\ (t : \mathit{slist}\ R)\ (p : \mathit{ok}\ h\ t), \\
\qquad P\ t \to Q\ h\ t\ p \to P\ (\mathit{scons}\ h\ t\ p)) \\
(\mathit{Qok\_snil} : \forall\ x : A,\ Q\ x\ \mathit{snil}\ (\mathit{ok\_snil}\ x)) \\
(\mathit{Qok\_scons} : \\
\quad \forall \\
\qquad (h : A)\ (t : \mathit{slist}\ R)\ (p : \mathit{ok}\ h\ t) \\
\qquad (x : A)\ (H : R\ x\ h), \\
\qquad\quad P\ t \to Q\ h\ t\ p \to Q\ x\ (\mathit{scons}\ h\ t\ p)\ (\mathit{ok\_scons}\ h\ t\ p\ x\ H)), \\
\{f : (\forall\ l : \mathit{slist}\ R,\ P\ l)\ \& \\
\{g : (\forall\ (h : A)\ (t : \mathit{slist}\ R)\ (p : \mathit{ok}\ h\ t),\ Q\ h\ t\ p)\ \& \\
\quad f\ \mathit{snil} = \mathit{Psnil} \land \\
\quad (\forall\ (h : A)\ (t : \mathit{slist}\ R)\ (p : \mathit{ok}\ h\ t), \\
\qquad f\ (\mathit{scons}\ h\ t\ p) = \mathit{Pscons}\ h\ t\ p\ (f\ t)\ (g\ h\ t\ p)) \land \\
\quad (\forall\ x : A, \\
\qquad g\ x\ \mathit{snil}\ (\mathit{ok\_snil}\ x) = \mathit{Qok\_snil}\ x) \land \\
\quad (\forall \\
\qquad (h : A)\ (t : \mathit{slist}\ R)\ (p : \mathit{ok}\ h\ t) \\
\qquad (x : A)\ (H : R\ x\ h), \\
\qquad\quad g\ x\ (\mathit{scons}\ h\ t\ p)\ (\mathit{ok\_scons}\ h\ t\ p\ x\ H) = \\
\qquad\quad \mathit{Qok\_scons}\ h\ t\ p\ x\ H\ (f\ t)\ (g\ h\ t\ p)) \\
\ \}\}).
\end{array}
$$

Ugh, co za potfur. Spróbujmy rozłożyć go na czynniki pierwsze.

Przede wszystkim, żeby za dużo nie pisać, zobaczymy tylko regułę indukcji. Teoretycznie powinny to być dwie reguły (tak jak w przypadku *Smok*a i *Zmok*a) - jedna dla *slist* i jedna dla *ok*, ale żeby za dużo nie pisać, możemy zapisać je razem.

Typ A i relacja R są parametrami obu definicji, więc skwantyfikowane są na samym początku. Nasza reguła pozwala nam zdefiniować przez wzajemną rekursję dwie funkcje, $f : \forall\ l : \mathit{slist}\ R,\ P\ l$ oraz $g : \forall\ (h : A)\ (t : \mathit{slist}\ R)\ (p : \mathit{ok}\ h\ t),\ Q\ h\ t\ p$. Tak więc P to kodziedzina f, a Q - g.

Teraz potrzebujemy rozważyć wszystkie możliwe przypadki - tak jak przy pattern matchingu. Przypadek *snil* jest dość banalny. Przypadek *scons* jest trochę cięższy. Przede wszystkim chcemy, żeby konkluzja była postaci $P\ (\mathit{scons}\ h\ t\ p)$, ale jak powinny wyglądać hipotezy indukcyjne?

Jedyna słuszna odpowiedź brzmi: odpowiadają one typom wszystkich możliwych wywołań rekurencyjnych f i g na strukturalnych podtermach $\mathit{scons}\ h\ t\ p$. Jedynymi typami spełniającymi te warunki są $P\ t$ oraz $Q\ h\ t\ p$, więc dajemy je sobie jako hipotezy indukcyjne.

Przypadki dla Q wyglądają podobnie: *ok_snil* jest banalne, a dla *ok_scons* konkluzja musi być jedynej słusznej postaci, a hipotezami indukcyjnymi jest wszystko, co pasuje.

W efekcie otrzymujemy dwie funkcje, f i g. Tym razem następuje jednak mały twist: ponieważ nasza definicja jest aksjomatyczna, zagwarantować musimy sobie także reguły obliczania, które dotychczas były zamilaczne, bo wynikały z definicji przez dopasowanie do wzorca. Teraz wszystkie te "dopasowania" musimy napisać ręcznie w postaci odpowiednio skwantyfikowanych równań. Widzimy więc, że *Psnil*, *Pscons*, *Qok_snil* i *Qok_scons* odpowiadają klauzulom w dopasowaniu do wzorca.

Ufff... udało się. Tak spreparowaną definicją aksjomatyczną możemy się jako-tako posługiwać:

```
Definition rec'
  {A : Type} {R : A → A → Prop}
  (P : Type) (snil' : P) (scons' : A → P → P) :
  {f : slist R → P &
    f snil = snil' ∧
    ∀ (h : A) (t : slist R) (p : ok h t),
      f (scons h t p) = scons' h (f t)
  }.
Proof.
  destruct
  (
    ind
    A R
    (fun _ ⇒ P) (fun _ _ _ ⇒ True)
    snil' (fun h _ _ t _ ⇒ scons' h t)
    (fun _ ⇒ I) (fun _ _ _ _ _ _ _ _ ⇒ I)
  )
  as (f & g & H1 & H2 & H3 & H4).
  ∃ f. split.
    exact H1.
    exact H2.
Defined.
```

Możemy na przykład dość łatwo zdefiniować niezależnych rekursor tylko dla *slist*, nie odnoszący się w żaden sposób do *ok*. Widzimy jednak, że "programowanie" w taki aksjomatyczny sposób jest dość ciężkie - zamiast eleganckich dopasowań do wzorca musimy ręcznie wpisywać argumenty do reguły indukcyjnej.

```
Require Import List.
Import ListNotations.
```

```
Definition toList'
  {A : Type} {R : A → A → Prop} :
  {f : slist R → list A &
    f snil = [] ∧
    ∀ (h : A) (t : slist R) (p : ok h t),
```

```
    f (scons h t p) = h :: f t
  }.
Proof.
  exact (rec' (list A) [] cons).
Defined.

Definition toList
  {A : Type} {R : A → A → Prop} (l : slist R) : list A :=
match @toList' A R with
    | existT _ f _ ⇒ f l
end.
```

Używająnie takieg rekursora jest już dużo prostsze, co ilustruje powyższy przykład funkcji, która zapomina o tym, że lista jest posortowana i daje nam zwykłą listę.

Przykładowe posortowane listy wyglądają tak:

```
Definition slist_01 : slist le :=
  scons 0
    (scons 1
      snil
      (ok_snil 1))
    (ok_scons 1 snil (ok_snil 1) 0 (le_S 0 0 (le_n 0))).
```

Niezbyt piękna, prawda?

```
Compute toList slist_01.
```

Utrapieniem jest też to, że nasza funkcja się nie oblicza. Jest tak, bo została zdefiniowana za pomocą reguły indukcji, która jest aksjomatem. Aksjomaty zaś, jak wiadomo (albo i nie - TODO) się nie obliczają.

Wyniku powyższego wywołania nie będę nawet wklejał, gdyż jest naprawdę ohydny.

```
Lemma toList_slist_01_result :
  toList slist_01 = [0; 1].
Proof.
  unfold toList, slist_01.
  destruct toList' as (f & H1 & H2).
  rewrite 2!H2, H1. reflexivity.
Qed.
```

Najlepsze, co możemy osiągnąć, mając taką definicję, to udowodnienie, że jej wynik faktycznie jest taki, jak się spodziewamy.

Cóż, to by było na tyle w temacie indukcji-indukcji. Brak Coqowego wsparcia dla niej niestety wydatnie ogranicza praktyczne możliwości posługiwania się nią. Jednak uszy do góry - istnieją już języki, które sobie z nią radzą. Jednym z nich jest wspomniana we wstępie Agda, którą można znaleźć tu: https://agda.readthedocs.io/en/v2.6.0.1/

Ćwiczenie Zdefiniuj dla list posortowanych funkcję *slen*, która liczy ich długość. Udowodnij oczywiste twierdzenie wiążące ze sobą *slen*, *toList* oraz *length*.

Ćwiczenie Udowodnij, że przykład faktycznie jest bez sensu: zdefiniuje relację *sorted* : $(A \to A \to$ `Prop`$) \to$ *list* $A \to$ `Prop`, gdzie *sorted* R l oznacza, że lista l jest posortowana według porządku R. Używając *sorted* zdefiniuj typ list posortowanych *slist'* R, a następnie znajdź dwie funkcje f : *slist* $R \to$ *slist'* R i f : *slist'* $R \to$ *slist* R, które są swoimi odwrotnościami.

Ćwiczenie Żeby przekonać się, że przykład był naprawdę bezsensowny, zdefiniuj rodzinę typów *blist* : $(A \to A \to$ `Prop`$) \to A \to$ `Type`, gdzie elementami *blist* R x są listy posortowane, których elementy są R-większe od x. Użyj *blist* do zdefiniowania typu *slist''* R, a następnie udowodnij, że *slist* R i *slist''* R są sobie równoważne.

`End` *ind_ind*.

Ćwiczenie Typ stert binarnych *BHeap* R, gdzie $R : A \to A \to$ `Prop` jest relacją porządku, składa się z drzew, które mogą być albo puste, albo być węzłem przechowującym wartość v : A wraz z dwoma poddrzewami l r : *BHeap* R, przy czym v musi być R-większe od wszystkich elementów l oraz r.

Użyj indukcji-indukcji, żeby zdefiniować jednocześnie typ *BHeap* R oraz relację *ok*, gdzie *ok* v h zachodzi, gdy v jest R-większe od wszystkich elementów h.

Najpierw napisz pseudodefinicję, a potem przetłumacz ją na odpowiedni zestaw aksjomatów.

Następnie użyj swojej aksjomatycznej definicji, aby zdefiniować funkcję *mirror*, która tworzy lustrzane odbicie sterty h : *BHeap* R.

Ćwiczenie Typ drzew wyszukiwań binarnych *BST* R, gdzie $R : A \to A \to$ `Prop` jest relacją porządku, składa się z drzew, które mogą być albo puste, albo być węzłem przechowującym wartość v : A wraz z dwoma poddrzewami l r : *BST* R, przy czym v musi być R-większe od wszystkich elemtnów l oraz R-mniejsze od wszystkich elementów r.

Użyj indukcji-indukcji, żeby zdefiniować jednocześnie typ *BST* R wraz z odpowiednimi relacjami zapewniającymi poprawność konstrukcji węzła. Wypróbuj trzy podejścia:

- jest jedna relacja, *oklr*, gdzie *oklr* v l r oznacza, że z v, l i r można zrobić węzeł
- są dwie relacje, *okl* i *okr*, gdzie *okl* v l oznacza, że v jest R-większe od wszystkich elementów l, zaś *okr* v r, że v jest R-mniejsze od wszystkich elementów r
- jest jedna relacja, *ok*, gdzie *ok* v t oznacza, że v jest R-mniejsze od wszystkich elementów t

Najpierw napisz pseudodefinicję, a potem przetłumacz ją na odpowiedni zestaw aksjomatów.

3.6.3 Indukcja-rekursja

A oto kolejny potfur do naszej kolekcji: indukcja-rekursja. Nazwa, choć brzmi tak głupio, jak "indukcja-indukcja", niesie ze sobą jednak dużo więcej wyobraźni: indukcja-rekursja pozwala nam jednocześnie definiować typy induktywne oraz operujące na nich funkcje rekurencyjne.

Co to dokładnie znaczy? Dotychczas nasz modus operandi wyglądał tak, że najpierw definiowaliśmy jakiś typ induktywny, a potem przez rekursję definiowaliśmy operujące na nim funkcje, np:

- najpierw zdefiniowaliśmy typ *nat*, a potem dodawanie, mnożenie etc.
- najpierw zdefiniowaliśmy typ *list* A, a potem *app*, *rev* etc.

Dlaczego mielibyśmy chcieć definiować typ i funkcję jednocześnie? Dla tego samego, co zawsze, czyli zależności - indukcja-rekursja pozwala, żeby definicja typu odnosiła się do funkcji, która to z kolei jest zdefiniowana przez rekursję strukturalną po argumencie o tym typie.

Zobaczmy dobrze nam już znany bezsensowny przykład, czyli listy posortowane, tym razem zaimplementowane za pomocą indukcji-rekursji.

```
(*
   Inductive slist {A : Type} (R : A -> A -> bool) : Type :=
    | snil : slist R
    | scons : forall (h : A) (t : slist R), ok h t = true -> slist R

   with

   Definition ok
  {A : Type} {R : A -> A -> bool} (x : A) (t : slist R) : bool :=
   match t with
    | snil => true
    | scons h _ _ => R x h
   end.
   *)
```

Coq niestety nie wspiera indukcji-rekursji, a próba napisania powyższej definicji kończy się błędem składni. Podobnie jak poprzednio, pomożemy sobie za pomocą aksjomatów, jednak najpierw prześledźmy definicję.

Typ slist działa następująco:

- R to jakiś porządek. Zauważ, że tym razem $R : A \to A \to bool$, a więc porządek jest rozstrzygalny
- *snil* to lista pusta
- *scons* h t p to lista z głową h i ogonem t, zaś p : *ok* h t = *true* to dowód na to, że dostawienie h przed t daje listę posortowaną.

Tym razem jednak *ok* nie jest relacją, lecz funkcją zwracającą *bool*, która działa następująco:

- dla *snil* zwróć *true* - każde $h : A$ można dostawić do listy pustej
- dla *scons* h _ _ zwróć wynik porównania x z h

Istotą mechanizmu indukcji-rekursji w tym przykładzie jest to, że *scons* wymaga dowodu na to, że funkcja *ok* zwraca *true*, podczas gdy funkcja ta jest zdefiniowana przez jednocześnie z typem *slist*.

Użycie indukkcji-rekursji do zaimplementowania *slist* ma swoje zalety: dla konkretnych list (złożonych ze stałych, a nie ze zmiennych) dowody $ok\ h\ t = true$ będą postaci *eq_refl*, bo *ok* po prostu obliczy się do *true*. W przypadku indukcji-indukcji dowody na $ok\ h\ t$ były całkiem sporych rozmiarów drzewami. Innymi słowy, udało nam się zastąpić część termu obliczeniami. Ten intrygujący motyw jeszcze się w przyszłości pojawi, gdy omawiać będziemy dowód przez reflekcję.

Dosyć gadania! Zobaczmy, jak zakodować powyższą definicję za pomocą aksjomatów.

```
Axioms
  (slist : ∀ {A : Type}, (A → A → bool) → Type)
  (ok : ∀ {A : Type} {R : A → A → bool} (h : A) (t : slist R), bool)
  (snil :
    ∀ {A : Type} {R : A → A → bool}, slist R)
  (scons :
    ∀
      {A : Type} {R : A → A → bool}
      (h : A) (t : slist R), ok h t = true → slist R)
  (ok_snil :
    ∀ {A : Type} {R : A → A → bool} (x : A),
      ok x (@snil _ R) = true)
  (ok_scons :
    ∀
      {A : Type} {R : A → A → bool}
      (x h : A) (t : slist R) (p : ok h t = true),
        ok x (scons h t p) = R x h).
```

Najpierw musimy zadeklarować *slist*, a następnie *ok*, gdyż typ *ok* zależy od *slist*. Obie definicje wyglądają dokładnie tak, jak nagłówki w powyższej definicji odrzuconej przez Coqa.

Następnym krokiem jest zadeklarowanie konstruktorów *slist* oraz równań definiujących funkcję *ok*.

```
Axiom
  ind : ∀
    (A : Type) (R : A → A → bool)
    (P : slist R → Type)
```

```
    (Psnil : P snil)
    (Pscons :
      ∀ (h : A) (t : slist R) (p : ok h t = true),
        P t → P (scons h t p)),
    {f : ∀ l : slist R, P l |
      f snil = Psnil ∧
      (∀ (h : A) (t : slist R) (p : ok h t = true),
        f (scons h t p) = Pscons h t p (f t))}.
```

Innym zyskiem z użycia indukcji-rekursji jest postać reguły indukcyjnej: jest ona dużo prostsza, niż w przypadku indukcji-indukcji, gdyż teraz definiujemy tylko jeden typ, zaś towarzysząca mu funkcja nie wymaga w regule niczego specjalnego - po prostu pojawia się w niej tam, gdzie spodziewalibyśm się jej po definicji *slist*, ale nie robi niczego ponad to.

```
Definition rev
  {A : Type} {R : A → A → bool}
  (l : slist R) : slist (fun x y ⇒ negb (R x y)).
Proof.
  revert l.
  refine (proj1_sig (ind
    A R
    (fun _ ⇒ slist (fun x y ⇒ negb (R x y)))
    _ _)).
    exact snil.
    intros h t p.
      refine (proj1_sig (ind
        A (fun x y ⇒ negb (R x y))
        (fun _ ⇒ slist (fun x y ⇒ negb (R x y)))
        _ _)).
        exact (scons h snil (ok_snil h)).
        intros. apply (scons h0 t0). assumption.
Defined.

Fixpoint leb (n m : nat) : bool :=
match n, m with
    | 0, _ ⇒ true
    | _, 0 ⇒ false
    | S n', S m' ⇒ leb n' m'
end.

Goal @rev _ leb (scons 2 snil (ok_snil 2)) = scons 2 snil (ok_snil 2).
Proof.
  unfold rev.
  destruct (ind _) as (f & Hf1 & Hf2). cbn in *.
  rewrite Hf2, Hf1.
```

```
  destruct (ind _) as (g & Hg1 & Hg2). cbn in *.
  rewrite Hg1. reflexivity.
Qed.

Require Import JMeq.

Lemma rev_rev :
  ∀ (A : Type) (R : A → A → bool) (l : slist R),
    JMeq (rev (rev l)) l.
Proof.
  intros A R.
  refine (proj1_sig (ind
    A R
    (fun l ⇒ JMeq (rev (rev l)) l)
    _ _)).
Abort.
```

3.6.4 Jeszcze straszniejszy potfur

Rozdział 4

R2ipół

W poprzednim rozdziale dość dogłębnie zapoznaliśmy się z mechanizmem definiowania induktywnych typów i rodzin typów. Nauczyliśmy się też definiować funkcje operujące na ich elementach za pomocą dopasowania do wzorca oraz rekursji.

Indukcja i rekursja są ze sobą bardzo ściśle powiązane. Obie opierają się na autoreferencji, czyli odnoszeniu się do samego siebie:

- liczba naturalna to zero lub następnik liczby naturalnej
- długość listy złożonej z głowy i ogona to jeden plus długość ogona

Można użyć nawet mocniejszego stwierdzenia: indukcja i rekursja są dokładnie tym samym zjawiskiem. Skoro tak, dlaczego używamy na jego opisanie dwóch różnych słów? Cóż, jest to zaszłość historyczna, jak wiele innych, które napotkaliśmy. Rozróżniamy zdania i typy/specyfikacje, relacje i rodziny typów, dowody i termy/programy etc., choć te pierwsze są specjalnymi przypadkami tych drugich. Podobnie indukcja pojawiła się po raz pierwszy jako technika dowodzenia faktów o liczbach naturalnych, zaś rekursja jako technika pisania programów.

Dla jasności, terminów tych będziemy używać w następujący sposób:

- indukcja będzie oznaczać metodę definiowania typów oraz metodę dowodzenia
- rekursja będzie oznaczać metodę definiowania funkcji

W tym rozdziale zbadamy dokładniej rekursję: poznamy różne jej rodzaje, zobaczymy w jaki sposób za jej pomocą można zrobić własne niestandardowe reguły indukcyjne, poznamy rekursję (i indukcję) dobrze ufundowaną oraz zobaczymy, w jaki sposób połączyć indukcję i rekursję, by móc dowodzić poprawności pisanych przez nas funkcji wciśnięciem jednego przycisku (no, prawie).

4.1 Rodzaje rekursji

Funkcja może w swej definicji odwoływać się do samej siebie na różne sposoby. Najważniejszą klasyfikacją jest klasyfikacja ze względu na dozwolone argumenty w wywołaniu rekurencyjnym:

- Rekursja strukturalna to taka, w której funkcja wywołuje siebie na argumentach będących podtermami argumentów z obecnego wywołania.
- Rekursja dobrze ufundowana to taka, w której funkcja wywołuje siebie jedynie na argumentach "mniejszych", gdzie o tym, które argumenty są mniejsze, a które większe, decyduje pewna relacja dobrze ufundowana. Intuicyjnie relacja dobrze ufundowana jest jak drabina: schodząc po drabinie w dół kiedyś musimy schodzenie zakończyć. Nie możemy schodzić w nieskończoność.

Mniej ważną klasyfikacją jest klasyfikacja ze względu na... cóż, nie wiem jak to ładnie nazwać:

- Rekursja bezpośrednia to taka, w której funkcja f wywołuje siebie samą bezpośrednio.
- Rekursja pośrednia to taka, w której funkcja f wywołuje jakąś inną funkcję g, która wywołuje f. To, że f nie wywołuje samej siebie bezpośrednio nie oznacza wcale, że nie jest rekurencyjna.
- W szczególności, rekursja wzajemna to taka, w której funkcja f wywołuje funkcję g, a g wywołuje f.
- Rzecz jasna rekursję pośrednią oraz wzajemną można uogólnić na dowolną ilość funkcji.

Oczywiście powyższe dwie klasyfikacje to tylko wierzchołek góry lodowej, której nie ma sensu zdobywać, gdyż naszym celem jest posługiwanie się rekursją w praktyce, a nie dzielenie włosa na czworo. Wobec tego wszystkie inne rodzaje rekursji (albo nawet wszystkie możliwe rodzaje w ogóle) będziemy nazywać rekursją ogólną.

Z rekursją wzajemną zapoznaliśmy się już przy okazji badania indukcji wzajemnej w poprzednim rozdziale. W innych funkcyjnych językach programowania używa się jej zazwyczaj ze względów estetycznych, by móc elegancko i czytelnie pisać kod, ale jak widzieliśmy w Coqu jest ona bardzo upierdliwa, więc raczej nie będziemy jej używać. Skupmy się zatem na badaniu rekursji strukturalnej, dobrze ufundowanej i ogólnej.

Ćwiczenie Przypomnij sobie podrozdział o indukcji wzajemnej. Następnie wytłumacz, jak przetłumaczyć definicję funkcji za pomocą rekursji wzajemnej na definicję, która nie używa rekursji wzajemnej.

4.2 Rekursja ogólna

W Coqu rekursja ogólna nie jest dozwolona. Powód jest banalny: prowadzi ona do sprzeczności. W celu zobrazowania spróbujmy zdefiniować za pomocą taktyk następującą funkcję rekurencyjną:

```
Fixpoint loop (u : unit) : False.
Proof.
  apply loop. assumption.
Fail Qed.
Abort.
```

Przyjrzyjmy się uważnie definicji funkcji *loop*. Mimo, że udało nam się ujrzeć znajomy napis "No more subgoals", próba użycia komendy `Qed` kończy się błędem.

Fakt, że konstruujemy funkcję za pomocą taktyk, nie ma tu żadnego znaczenia, lecz służy jedynie lepszemu zobrazowaniu, dlaczego rekursja ogólna jest grzechem. Dokładnie to samo stałoby się, gdybyśmy próbowali zdefiniować *loop* ręcznie:

```
Fail Fixpoint loop (u : unit) : False := loop u.
```

Gdyby tak się nie stało, możliwe byłoby skonstruowanie dowodu *False*:

```
Fail Definition the_universe_explodes : False := loop tt.
```

Aby chronić nas przed tą katastrofą, Coq nakłada na rekurencję ograniczenie: argument główny wywołania rekurencyjnego musi być strukturalnym podtermem argumentu głównego obecnego wywołania. Innymi słowy, dozwolona jest jedynie rekursja strukturalna.

To właśnie napisane jest w komunikacie o błędzie, który dostajemy, próbując przeforsować powyższe definicje:

```
(*  Recursive definition of loop is ill-formed.
   In environment
   loop : unit -> False
   u : unit
   Recursive call to loop has principal argument equal to
   "u" instead of a subterm of "u".
   Recursive definition is: "fun u : unit => loop u". *)
```

Wywołanie rekurencyjne *loop* jest nielegalne, gdyż jego argumentem jest *u*, podczas gdy powinien być nim jakiś podterm *u*.

Zanim jednak dowiemy się, czym jest argument główny, czym są podtermy i jak dokładnie Coq weryfikuje poprawność naszych definicji funkcji rekurencyjnych, wróćmy na chwilę do indukcji. Jak się zaraz okaże, nielegalność rekursji ogólnej wymusza również pewne ograniczenia w definicjach induktywnych.

Ćwiczenie Ograniczenia nakładane przez Coqa sprawiają, że wszystkie napisane przez nas funkcje rekurencyjne muszą się kiedyś zatrzymać i zwrócić ostateczny wynik swojego działania. Tak więc nie możemy w Coqu pisać funkcji nieterminujących, czyli takich, które się nie zatrzymują.

Rozważ bardzo interesujące pytanie filozoficzne: czy funkcje, które nigdy się nie zatrzymują (lub nie zatrzymują się tylko dla niektórych argumentów) mogą być w ogóle do czegokolwiek przydatne?

Nie daj się wpuścić w maliny.

4.3 Ścisła pozytywność

Poznana przez nas dotychczas definicja typów induktywnych jest niepełna, gdyż pominęliśmy kryterium ścisłej pozytywności. Rozważmy następujący typ:

Fail `Inductive` *wut* $(A$: `Type`$)$: `Type` :=
  | C : $($*wut* $A \to A) \to$ *wut* A.

Uwaga: poprzedzenie komendą *Fail* innej komendy oznajmia Coqowi, że spodziewamy się, iż komenda zawiedzie. Coq akceptuje komendę *Fail* c, jeżeli komenda c zawodzi, i wypisuje komunikat o błędzie. Jeżeli komenda c zakończy się sukcesem, komenda *Fail* c zwróci błąd.

Komenda *Fail* jest przydatna w sytuacjach takich jak obecna, gdy chcemy zilustrować fakt, że jakaś komenda zawodzi.

```
(*  Error: Non strictly positive occurrence of "wut"
   in "(wut A -> A) -> wut A". *)
```

Żeby zrozumieć ten komunikat o błędzie, musimy najpierw przypomnieć sobie składnię konstruktorów. Konstruktory typu induktywnego T będą mieć (w dość sporym uproszczeniu) postać *arg1* $\to$ ... $\to$ *argN* $\to T$ — są to funkcje biorące pewną (być może zerową) ilość argumentów, a ich przeciwdziedziną jest definiowany typ T.

Jeżeli definiowany typ T nie występuje nigdzie w typach argumentów *arg1* ... *argN*, sytuacja jest klarowna i wszystko jest w porządku. W przeciwnym wypadku, w zależności od postaci typów argumentów, mogą pojawić się problemy.

Jeżeli typ któregoś z argumentów jest równy T, nie ma problemu — jest to po prostu argument rekurencyjny. Jeżeli jest on postaci $A \to T$ dla dowolnego typu A, również nie ma problemu — dzięki argumentom o takich typach możemy reprezentować np. drzewa o nieskończonym współczynniku rozgałęzienia. Mówimy, że w $A \to T$ typ T występuje w pozycji (ściśle) pozytywnej.

Problem pojawia się dopiero, gdy typ argumentu jest postaci $T \to A$ lub podobnej (np. $A \to T \to B$, $T \to T \to A \to B$ etc.). W takich przypadkach mówimy, że typ T występuje na pozycji negatywnej (albo "nie-ściśle-pozytywnej").

Pierwszym, stosunkowo błahym problemem jest fakt, że typy łamiące kryterium ścisłej pozytywności nie mają modeli teoriozbiorowych — znaczy to po prostu, że nie można reprezentować ich w teorii zbiorów za pomocą żadnych zbiorów. Dla wielu matematyków stanowi to problem natury praktycznej (są przyzwyczajeni do teorii zbiorów) lub filozoficznej.

Problem ten wynika z faktu, że konstruktory typów induktywnych są injekcjami, zaś typy argumentów, w których definiowany typ występuje na pozycji negatywnej, są "za duże". Np. w przypadku typu *wut bool* konstruktor C jest injekcją z *wut bool* $\to$ *bool* w *wut bool*. Gdybyśmy chcieli interpretować typy jako zbiory, to zbiór *wut bool* $\to$ *bool* jest "za duży", by można było go wstrzyknąć do *wut bool*, gdyż jest w bijekcji ze zbiorem potęgowym *wut bool*, a w teorii zbiorów powszechnie wiadomo, że nie ma injekcji ze zbioru potęgowego jakiegoś zbioru do niego samego.

Nie przejmuj się, jeżeli nie rozumiesz powyższego paragrafu — nie jest to główny powód obowiązywania kryterium ścisłej pozytywności, wszak jako buntownicy zajmujący się teorią typów nie powinniśmy zbytnio przejmować się teorią zbiorów.

Prawdziwy powód jest inny: dopuszczenie typów łamiących kryterium ścisłej pozytywności prowadzi do sprzeczności. Gdyby były one legalne, legalna byłaby również poniższa definicja:

Fail `Definition` *y* (*A* : `Type`) : *A* :=
  `let` *f* := (`fun` *x* : *wut A* ⇒ `match` *x* `with` | *C f'* ⇒ *f' x* `end`)
  `in` *f* (*C f*).

Jak widać, gdyby definicja typu *wut* została dopuszczona, moglibyśmy uzyskać zapętlający się program umożliwiający nam stworzenie elementu dowolnego typu i to bez użycia słowa kluczowego `Fixpoint` (program ten jest nazywany zazwyczaj kombinatorem Y, ang. Y combinator). Stąd już niedaleko do popadnięcia w zupełną sprzeczność:

Fail `Definition` *santa_is_a_pedophile* : *False* := *y False*.

Ćwiczenie `(* Inductive T : Type := *)`

Rozstrzygnij, czy następujące konstruktory spełniają kryterium ścisłej pozytywności. Następnie sprawdź w Coqu, czy udzieliłeś poprawnej odpowiedzi.

- | *C1* : *T*
- | *C2* : *bool* $\to$ *T*
- | *C3* : *T* $\to$ *T*
- | *C4* : *T* $\to$ *nat* $\to$ *T*
- | *C5* : ∀ *A* : `Type`, *T* $\to$ *A* $\to$ *T*
- | *C6* : ∀ *A* : `Type`, *A* $\to$ *T* $\to$ *T*
- | *C7* : ∀ *A* : `Type`, (*A* $\to$ *T*) $\to$ *T*
- | *C8* : ∀ *A* : `Type`, (*T* $\to$ *A*) $\to$ *T*
- | *C9* : (∀ *x* : *T*, *T*) $\to$ *T*

- | *C10* : $(\forall\ (A : \texttt{Type})\ (x :\ T),\ A) \to T$
- | *C11* : $\forall\ A\ B\ C : \texttt{Type},\ A \to (\forall\ x :\ T,\ B) \to (C \to\ T) \to\ T$

4.4 Rekursja strukturalna

Wiemy już, że rekursja ogólna prowadzi do sprzeczności, a jedyną legalną formą rekursji jest rekursja strukturalna. Funkcje rekurencyjne, które dotychczas pisaliśmy, były strukturalnie rekurencyjne, więc potrafisz już całkiem sprawnie posługiwać się tym rodzajem rekursji. Pozostaje nam zatem zbadać jedynie techniczne detale dotyczące sposobu realizacji rekursji strukturalnej w Coqu. W tym celu przyjrzyjmy się ponownie definicji dodawania:

```
Print plus.
(*  plus =
   fix plus (n m : nat) {struct n} : nat :=
     match n with
     | 0 => m
     | S p => S (plus p m)
     end
        : nat -> nat -> nat *)
```

Możemy zaobserwować parę rzeczy. Pierwsza, techniczna sprawa: po = widzimy nieznany nam konstrukt `fix`. Pozwala on tworzyć anonimowe funkcje rekruencyjne, tak jak `fun` pozwala tworzyć anonimowe funkcje nierekurencyjne. Funkcje zdefiniowane komendami `Fixpoint` i `Definition` są w jęzku termów Coqa reprezentowane odpowiednio za pomocą `fix` i `fun`.

Po drugie: za listą argumentów, a przed zwracanym typem, występuje adnotacja {`struct` *n*}. Wskazuje ona, który z argumentów funkcji jest argumentem głównym. Dotychczas gdy definiowaliśmy funkcje rekurencyjne nigdy nie musieliśmy jej pisać, bo Coq zawsze domyślał się, który argument ma być główny. W poetyckiej polszczyźnie argument główny możemy wskazać mówiąc np., że "funkcja plus zdefiniowana jest przez rekursję po pierwszym argumencie" albo "funkcja plus zdefinowana jest przez rekursję po n".

Czym jest argument główny? Spróbuję wyjasnić to w sposób operacyjny:

- jako argument główny możemy wskazać dowolny argument, którego typ jest induktywny
- Coq wymusza na nas, aby argumentem głównym wywołania rekurencyjnego był podterm argumentu głównego z obecnego wywołania

Dlaczego taki zabieg chroni nas przed sprzecznością? Przypomnij sobie, że termy typów induktywnych muszą być skończone. Parafrazując: są to drzewa o skończonym rozmiarze. Ich podtermy są od nich mniejsze, więc w kolejnych wywołaniach rekurencyjnych argument główny będzie malał, aż w końcu jego rozmiar skurczy się do zera. Wtedy rekursja zatrzyma się, bo nie będzie już żadnych podtermów, na których można by zrobić wywołanie rekurencyjne.

Żeby lepiej zrozumieć ten mechanizm, zbadajmy najpierw relację bycia podtermem dla typów induktywnych. Relację tę opisują dwie proste zasady:

- po pierwsze, jeżeli dany term został zrobiony jakimś konstruktorem, to jego podtermami są rekurencyjne argumenty tego konstruktora. Przykład: 0 jest podtermem S 0, zaś *nil* podtermem *cons* 42 *nil*.

- po drugie, jeżeli *t1* jest podtermem *t2*, a *t2* podtermem *t3*, to *t1* jest podtermem *t3* — własność ta nazywa się przechodniością. Przykład: S 0 jest podtermem S (S 0), a zatem 0 jest podtermem S (S 0). Podobnie *nil* jest podtermem *cons* 666 (*cons* 42 *nil*)

Ćwiczenie Zdefiniuj relacje bycia podtermem dla liczb naturalnych i list.

Udowodnij, że przytoczone wyżej przykłady nie są oszustwem. Komenda `Goal` jest wygodna, gdyż używając jej nie musimy nadawać twierdzeniu nazwy. Użycie `Qed` zapisze twierdzenie jako *Unnamed_thm*, *Unnamed_thm0*, *Unnamed_thm1* etc.

```
Goal subterm_nat 0 (S 0).

Goal subterm_list nil (cons 42 nil).
```

Ćwiczenie Udowodnij, że relacje *subterm_nat* oraz *subterm_list* są antyzwrotne i przechodnie. Uwaga: to może być całkiem trudne.

```
Lemma subterm_nat_antirefl :
  ∀ n : nat, ¬ subterm_nat n n.

Lemma subterm_nat_trans :
  ∀ a b c : nat,
    subterm_nat a b → subterm_nat b c → subterm_nat a c.

Lemma subterm_list_antirefl :
  ∀ (A : Type) (l : list A), ¬ subterm_list l l.

Lemma subterm_list_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    subterm_list l1 l2 → subterm_list l2 l3 →
      subterm_list l1 l3.
```

Jak widać, podtermy liczby naturalnej to liczby naturalne, które są od niej mniejsze, zaś podtermy listy to jej ogon, ogon ogona i tak dalej. Zero i lista pusta nie mają podtermów, gdyż są to przypadki bazowe, pochodzące od konstruktorów, które nie mają argumentów rekurencyjnych.

Dla każdego typu induktywnego możemy zdefiniować relację bycia podtermem podobną do tych dla liczb naturalnych i list. Zauważmy jednak, że nie możemy za jednym zamachem zdefiniować relacji bycia podtermem dla wszystkich typów induktywnych, gdyż nie możemy w Coqu powiedzieć czegoś w stylu "dla wszystkich typów induktywnych". Możemy powiedzieć jedynie "dla wszystkich typów".

Coq nie generuje jednak automatycznie takiej relacji, gdy definiujemy nowy typ induktywny. W jaki zatem sposób Coq sprawdza, czy jeden term jest podtermem drugiego? Otóż... w sumie, to nie sprawdza. Rzućmy okiem na następujący przykład:

```
Fail Fixpoint weird (n : nat) : unit :=
match n with
    | 0 => tt
    | S n' => weird 0
end.
```

Definicja zdaje się być poprawna: 0 to przypadek bazowy, a gdy *n* jest postaci *S n'*, wywołujemy funkcję rekurencyjnie na argumencie 0. 0 jest podtermem *S n'*, a zatem wszystko powinno być dobrze. Dostajemy jednak następujący komunikat o błędzie:

```
(*  Recursive definition of weird is ill-formed.
   In environment
   weird : nat -> unit
   n : nat
   n' : nat
   Recursive call to weird has principal argument equal to
   "0" instead of ń'". *)
```

Komunikat ten głosi, że argumentem głównym wywołania rekurencyjnego jest 0, podczas gdy powinno być nim *n'*. Wynika stąd jasno i wyraźnie, że jedynymi legalnymi argumentami w wywołaniu rekurencyjnym są te podtermy argumentu głównego, które zostają ujawnione w wyniku dopasowania do wzorca. Coq nie jest jednak głupi - jest głupszy, niż ci się wydaje, o czym świadczy poniższy przykład.

```
Fail Fixpoint fib (n : nat) : nat :=
match n with
    | 0 => 0
    | 1 => 1
    | S (S n') => fib n' + fib (S n')
end.
```

Funkcja ta próbuje policzyć n-tą liczbę Fibonacciego: https://en.wikipedia.org/wiki/Fibonacci_number ale słabo jej to wychodzi, gdyż dostajemy następujący błąd:

```
(*  Recursive definition of fib is ill-formed.
   In environment
   fib : nat -> nat
   n : nat
   n0 : nat
```

```
n' : nat
Recursive call to fib has principal argument equal to
Ŝ n'" instead of one of the following variables:
ń0" ń'". *)
```

Mimo, że *S n'* jest podtermem *S* (*S n'*), który pochodzi z dopasowania do wzorca, to Coq nie jest w stanie zauważyć tego faktu. W komunikacie o błędzie pojawia się za to tajemnicza zmienna *n0*, której w naszym kodzie nigdzie nie ma. Sposobem na poradzenie sobie z problemem jest pokazanie Coqowi palcem, o co nam chodzi:

```
Fixpoint fib (n : nat) : nat :=
match n with
    | 0 ⇒ 0
    | 1 ⇒ 1
    | S (S n' as n'') ⇒ fib n' + fib n''
end.
```

Tym razem Coq widzi, że *S n'* jest podtermem *S* (*S n'*), gdyż explicite nadaliśmy temu termowi nazwę *n''*, używając do tego klauzli `as`.

Ufff... udało nam się przebrnąć przez techniczne detale działania rekursji strukturalnej. Mogłoby się wydawać, że jest ona mechanizmem bardzo upośledzonym, ale z doświadczenia wiesz już, że w praktyce omówione wyżej problemy występują raczej rzadko.

Mogłoby się też wydawać, że skoro wywołania rekurencyjne możemy robić tylko na bezpośrednich podtermach dopasowanych we wzorcu, to nie da się zdefiniować prawie żadnej ciekawej funkcji. Jak zobaczymy w następnym podrozdziale, wcale tak nie jest - dzięki pewnej sztuczce (która jest jednocześnie fundamentalną własnością wszystkich możliwych wszechświatów) za pomocą rekursji strukturalnej można wyrazić rekursję dobrze ufundowaną, która na pierwszy rzut oka jest dużo potężniejsza i daje nam wiele możliwości definiowania różnych ciekawych funkcji.

Ćwiczenie (dzielenie) Zdefiniuj funkcję *div*, która implementuje dzielenie całkowitoliczbowe. Żeby uniknąć problemów z dzieleniem przez 0, *div n m* będziemy interpretować jako *n* podzielone przez *S m*, czyli np. *div n* 0 to n/1, *div n* 1 to n/2 etc. Uwaga: to ćwiczenie pojawia się właśnie w tym miejscu nieprzypadkowo.

4.5 Rekursja dobrze ufundowana

Typy induktywne są jak domino - każdy term to jedna kostka, indukcja i rekursja odpowiadają zaś temu co tygryski lubią najbardziej, czyli reakcji łańcuchowej przewracającej wszystkie kostki.

Typ *unit* to jedna biedna kostka, zaś *bool* to już dwie biedne kostki - *true* i *false*. W obu przypadkach nie dzieje się nic ciekawego - żeby wszystkie kostki się przewróciły, musimy pchnąć palcem każdą z osobna.

Typ *nat* jest już ciekawszy - są dwa rodzaje kostek, 0 i S, a jeżeli pchniemy kostkę 0 i między kolejnymi kostkami jest odpowiedni odstęp, to równy szlaczek kolejnych kostek przewracać się będzie do końca świata.

Podobnie dla typu *list* A mamy dwa rodzaje kostek - *nil* i *cons*, ale kostki rodzaju *cons* mają różne kolory - są nimi elementy typu A. Podobnie jak dla *nat*, jeżeli pchniemy kostkę *nil* i odstępy między kolejnymi kostkami są odpowiednie, to kostki będą przewracać się w nieskończoność. Tym razem jednak zamiast jednego szaroburego szlaczka będzie multum kolorowych szlaczków o wspólnych początkach (no chyba, że $A = unit$ - wtedy dostaniemy taki sam bury szlaczek jak dla *nat*).

Powyższe malownicze opisy przewracających się kostek domino bardziej przywodzą mi na myśl indukcję, niż rekursję, chociaż wiemy już, że jest to w sumie to samo. Przyjmują one perspektywę "od przodu" - jeżeli przewrócimy początkową kostkę i niczego nie spartaczyliśmy, kolejne kostki będą przewracać się już same.

Możemy jednak przyjąć inny sposób patrzenia - perspektywę "od tyłu". W tej perspektywie kostka początkowa przewraca się, jeżeli zostanie pchnięta palcem, zaś każda dalsza kostka przewraca się, jeżeli przewracają się wszystkie kostki bezpośrednio ją poprzedzające.

Jeszcze jeden drobny detal: żeby w ogóle móc pchnąć kostki początkowe (w liczbie mnogiej, bo rzecz jasna może być więcej niż jedna), musimy najpierw ustalić, które kostki są tymi początkowymi! Na szczęście nie jest to trudne - są to oczywiście te, których nie poprzedzają inne kostki.

I tutaj następuje pewien trik logiczno-językowo-wyobraźniowy: możemy o kostkach początkowych myśleć, że przewracają się, gdy przewrócą się wszystkie kostki je poprzedzające, których oczywiście nie ma, a nasz palec wyobrażać sobie po prostu jako fizyczną realizację tej pustej prawdy.

W ten oto wesoły sposób udało nam się uzyskać definicję elementu dostępnego oraz relacji dobrze ufundowanej.

```
Inductive Acc {A : Type} (R : A → A → Type) (x : A) : Prop :=
  | Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x.
```

Kostki domina reprezentuje typ A, zaś relacja R to sposób ułożenia kostek, a x to pewna konkretna kostka domina. Konstruktor *Acc_intro* mówi, że kostka x przewraca się, gdy przewracają się wszystkie kostki ją poprzedzające.

To samo nieco mniej bajkowym językiem: element x typu A jest dostępny w relacji R, jeżeli każdy poprzedzający go element y typu A również jest dostępny.

```
Definition well_founded {A : Type} (R : A → A → Type) : Prop :=
  ∀ x : A, Acc R x.
```

Układ kostek reprezentowany przez R przewraca się w całości, jeżeli każda kostka domina przewraca się z osobna.

Mniej poetycko: relacja jest dobrze ufundowana, jeżeli każdy element typu A jest dostępny.

Uwaga: typem naszej "relacji" nie jest $A \to A \to$ `Prop`, lecz $A \to A \to$ `Type`, a zatem R jest tak naprawdę indeksowaną rodziną typów. Różnica między relacją i rodziną typów jest taka, że relacja, gdy dostanie argumenty, zwraca zdanie, czyli coś typu `Prop`, a rodzina typów, gdy dostanie argumenty, zwraca typ, czyli coś typu `Type`. Tak więc pojęcie rodziny typów jest ogólniejsze niż pojęcie relacji. Ta ogólność przyda się nam za kilka chwil aby nie musieć pisać wszystkiego dwa razy.

Ćwiczenie (dostępność i ufundowanie) Sprawdź, czy relacje $\leq$, $<$ są dobrze ufundowane.

Pokaż, że relacja dobrze ufundowana jest antyzwrotna oraz zinterpretuj ten fakt (tzn. powiedz, o co tak naprawdę chodzi w tym stwierdzeniu).

```
Lemma wf_antirefl :
  ∀ (A : Type) (R : A → A → Prop),
    well_founded R → ∀ x : A, ¬ R x x.
```

Sprawdź, czy dobrze ufundowane są relacje *le'* i *lt'*. Uwaga: pierwsze zadanie jest bardzo łatwe, drugie jest piekielnie trudne. Jeżeli nie potrafisz rozwiązać go formalnie w Coqu, zrób to na kartce nieformalnie - będzie dużo łatwiej.

```
Definition le' (f g : nat → nat) : Prop :=
  ∀ n : nat, f n ≤ g n.
```

```
Definition lt' (f g : nat → nat) : Prop :=
  ∀ n : nat, f n < g n.
```

Sprawdź, czy dobrze ufundowana jest następująca relacja porządku: wszystkie liczby parzyste są mniejsze niż wszystkie liczby nieparzyste, zaś dwie liczby o tej samej parzystości porównujemy według zwykłego porządku $<$.

Wiemy już, co to znaczy, że kostka domina jest dostępna (każda kostka ją poprzedzająca jest dostępna, co formalnie wyraża predykat *Acc*) oraz że poprawny układ kostek to taki, w którym każda kostka jest dostępna (co formalnie wyraża predykat *well_founded*). Możemy teraz przejść do tego, do czego dążyliśmy od samego początku: udowodnić, że jeżeli poprawnie ustawimy kostki, to wszystkie się przewrócą.

```
Theorem well_founded_rect :
  ∀
    (A : Type) (R : A → A → Type)
    (wf : well_founded R) (P : A → Type),
      (∀ x : A, (∀ y : A, R y x → P y) → P x) →
        ∀ x : A, P x.
Proof.
  intros A R wf P IH x.
  Check Acc_rect.
  apply Acc_rect with R.
    intros y H1 H2. apply IH. assumption.
    apply wf.
```

```
Defined.

Theorem well_founded_ind :
  ∀
    (A : Type) (R : A → A → Prop)
    (wf : well_founded R) (P : A → Type),
      (∀ x : A, (∀ y : A, R y x → P y) → P x) →
        ∀ x : A, P x.
Proof.
  intros A R wf P H x.
  apply (well_founded_rect _ _ wf _ H).
Qed.
```

4.6 Indukcja funkcyjna

Rozdział 5

R3: Ltac — język taktyk

Matematycy uważają, że po osiągnięciu pewnego poziomu zaawansowania i obycia (nazywanego zazwyczaj “mathematical maturity”) skrupulatne rozpisywanie każdego kroku dowodu przestaje mieć sens i pozwalają sobie zarzucić je na rzecz bardziej wysokopoziomowego opisu rozumowania.

Myślę, że ta sytuacja ma miejsce w twoim przypadku — znasz już sporą część języka termów Coqa (zwanego Gallina) i potrafisz dowodzić różnych właściwości programów. Doszedłeś do punktu, w którym ręczne klepanie dowodów przestaje być produktywne, a staje się nudne i męczące.

Niestety, natura dowodu formalnego nie pozwala nam od tak po prostu pominąć mało ciekawych kroków. Czy chcemy czy nie, aby Coq przyjął dowód, kroki te muszą zostać wykonane. Wcale nie znaczy to jednak, że to my musimy je wykonać — mogą zrobić to za nas programy.

Te programy to oczywiście taktyki. Większość prymitywnych taktyk, jak `intro`, `destruct`, czy `assumption` już znamy. Choć nie wiesz o tym, używaliśmy też wielokrotnie taktyk całkiem zaawansowanych, takich jak `induction` czy `inversion`, bez których nasze formalne życie byłoby drogą przez mękę.

Wszystkie one są jednak taktykami wbudowanymi, danymi nam z góry przez Coqowych bogów i nie mamy wpływu na ich działanie. Jeżeli nie jesteśmy w stanie zrobić czegoś za ich pomocą, jesteśmy zgubieni. Czas najwyższy nauczyć się pisać własne taktyki, które pomogą nam wykonywać mało ciekawe kroki w dowodach, a w dalszej perspektywie także przeprowadzać bardziej zaawansowane rozumowania zupełnie automatycznie.

W tym rozdziale poznamy podstawy języka `Ltac`, który służy do tworzenia własnych taktyk. Jego składnię przedstawiono i skrupulatnie opisano tu: https://coq.inria.fr/refman/ltac.html

Choć przykład znaczy więcej niż 0x3E8 stron manuala i postaram się dokładnie zilustrować każdy istotny moim zdaniem konstrukt języka `Ltac`, to i tak polecam zapoznać się z powyższym linkiem.

Upewnij się też, że rozumiesz dokładnie, jak działają podstawowe kombinatory taktyk, które zostały omówione w rozdziale 1, gdyż nie będziemy omawiać ich drugi raz.

5.1 Zarządzanie celami i selektory

Dowodząc (lub konstruując cokolwiek za pomocą taktyk) mamy do rozwiązania jeden lub więcej celów. Cele są ponumerowane i domyślnie zawsze pracujemy nad tym, który ma numer 1.

Jednak wcale nie musi tak być — możemy zaznaczyć inny cel i zacząć nad nim pracować. Służy do tego komenda *Focus*. Cel o numerze n możemy zaznaczyć komendą *Focus n*. Jeżeli to zrobimy, wszystkie pozostałe cele chwilowo znikają. Do stanu domyślnego, w którym pracujemy nad celem nr 1 i wszystkie cele są widoczne możemy wrócić za pomocą komendy *Unfocus*.

```
Goal ∀ P Q R : Prop, P ∧ Q ∧ R → R ∧ Q ∧ P.
Proof.
  repeat split.
  Focus 3.
  Unfocus.
  Focus 2.
Abort.
```

Komenda *Focus* jest użyteczna głównie gdy któryś z dalszych celów jest łatwiejszy niż obecny. Możemy wtedy przełączyć się na niego, rozwiązać go i wyniesione stąd doświadczenie przenieść na trudniejsze cele. Jest wskazane, żeby po zakończeniu dowodu zrefaktoryzować go tak, aby komenda *Focus* w nim nie występowała.

Nie jest też tak, że zawsze musimy pracować nad celem o numerze 1. Możemy pracować na dowolnym zbiorze celów. Do wybierania celów, na które chcemy zadziałać taktykami, służą selektory. Jest ich kilka i mają taką składnię:

- *n*: *t* — użyj taktyki t na n-tym celu. 1: *t* jest równoważne *t*.
- *a-b*: *t* — użyj taktyki t na wszystkich celach o numerach od a do b
- *a_1-b_1*, ..., *a_n-b_n*: *t* — użyj taktyki *t* na wszystkich celach o numerach od a_1 do b_1, ..., od a_n do b_n (zamiast a_i-b_i możemy też użyć pojedynczej liczby)
- *all*: *t* - użyj *t* na wszystkich celach
- zamiast *t*, w powyższych przypadkach możemy też użyć wyrażenia > *t_1* | ... | *t_n*, które aplikuje taktykę *t_i* do i-tego celu zaznaczonego danym selektorem

```
Goal ∀ P Q R : Prop, P ∧ Q ∧ R → R ∧ Q ∧ P.
Proof.
  destruct 1 as [H [H' H'']]. repeat split.
  3: assumption. 2: assumption. 1: assumption.
Restart.
  destruct 1 as [H [H' H'']]. repeat split.
  1-2: assumption. assumption.
```

```
Restart.
  destruct 1 as [H [H' H'']]. repeat split.
  1-2, 3: assumption.
Restart.
  destruct 1 as [H [H' H'']]. repeat split.
  1-3: assumption.
Restart.
  destruct 1 as [H [H' H'']]. repeat split.
  all: assumption.
Restart.
  destruct 1 as [H [H' H'']]. repeat split.
  all: [> assumption | assumption | assumption].
Qed.
```

Zauważmy, że powyższe selektory działają jedynie, gdy zostaną umieszczone przed wszystkimi taktykami, których dotyczą. Próba użycia ich jako argumenty dla innych taktyk jest błędem.

Dla przykładu, w czwartym z powyższych dowodów nie możemy napisać `repeat split`; 1-3: `assumption`, gdyż kończy się to błędem składni (nie wspominając o tym, że jest to bez sensu, gdyż dla uzyskania pożądanego efektu wystarczy napisać `repeat split`; `assumption`.

```
Goal ∀ P Q R : Prop, P ∧ Q ∧ R → R ∧ Q ∧ P.
Proof.
  destruct 1 as [H [H' H'']].
  repeat split; only 1-3: assumption.
Qed.
```

Nie wszystko jednak stracone! Żeby móc używać wyrażeń zawierających selektory jako argumenty taktyk, możemy posłużyć się słowem *only*. Mimo tego, i tak nie możemy napisać `repeat split`; *only all*: ..., gdyż kończy się to błędem skadni.

```
Goal ∀ P Q R S : Prop, P → P ∧ Q ∧ R ∧ S.
Proof.
  repeat split.
  revgoals. all: revgoals. all: revgoals.
  swap 1 3. all: swap 1 3. all: swap 1 3.
  cycle 42. all: cycle 3. all: cycle -3.
Abort.
```

Jest jeszcze kilka innych taktyk do żonglowania celami. Pamiętaj, że wszystkie z nich działają na liście celów wybranych selektorami — domyślnie wybrany jest tylko cel numer 1 i wtedy taktyki te nie mają żadnego skutku.

revgoals odwraca kolejność celów, na których działa. W naszym przypadku *revgoals* nie robi nic (odwraca kolejność celu P na P), natomiast *all*: *revgoals* zamienia kolejność celów z P — Q — R — S na S — R — Q — P.

swap n m zamienia miejscami cele n-ty i m-ty. W przykładzie *swap* 1 3 nic nie robi, gdyś domyślnie wybrany jest tylko cel numer 1, a zatem nie można zamienić go miejscami z celem nr 3, którego nie ma. *all*: *swap* 1 3 zamienia kolejność celów z P — Q — R — S na R — Q — P — S.

cycle n przesuwa cele cyklicznie o n do przodu (lub do tyłu, jeżeli argument jest liczbą ujemną). W naszym przykładzie *cycle* 42 nic nie robi (przesuwa cyklicznie cel P o 42 miejsca, co daje w wyniku P), zaś *all*: *cycle* 3 zamienia kolejność celów z P — Q — R — S na S — P — Q — R.

Taktyki te nie są zbyt użyteczne, a przynajmniej ja nigdy ich nie użyłem, ale dla kompletności wypadało o nich wspomnieć. Jeżeli wątpisz w użyteczność selektorów... cóż, nie dziwię ci się. Selektory przydają się głównie gdy chcemy napisać taktykę rozwiązującą wszystkie cele i sprawdzamy jej działanie na każdym celu z osobna. W pozostałych przypadkach są tylko zbędnym balastem.

5.2 Podstawy języka Ltac

Ltac jest funkcyjnym językiem programowania, podobnie jak język termów Coqa (zwany Gallina), lecz te dwa języki są diametralnie różne:

- Ltac jest kompletny w sensie Turinga, a Gallina nie. W szczególności, taktyki mogą się zapętlać i nie rodzi to żadnych problemów natury logicznej.
- Ltac jest bardzo słabo typowany, podczas gdy Gallina dysponuje potężnym systemem typów.
- W Ltacu nie możemy definiować typów danych, a jedynie taktyki działające na kontekstach i celu, podczas gdy Gallina pozwala na definiowanie bardzo szerokiej klasy typów i działających na nich funkcji.
- Ltac, jako metajęzyk jezyka Gallina, posiada dostęp do różnych rzeczy, do których Gallina nie ma dostępu, takich jak dopasowanie termów dowolnego typu. Dla przykładu, w Ltacu możemy odróżnić termy 4 oraz 2 + 2 pomimo tego, że są konwertowalne.

W Ltacu możemy manipulować trzema rodzajami bytów: taktykami, termami Coqa oraz liczbami całkowitymi — te ostatnie nie są tym samym, co liczby całkowite Coqa i będziemy ich używać sporadycznie. Zanim zobaczymy przykład, przyjrzyjmy się taktyce `pose` oraz konstruktowi `let`.

```
Goal True.
Proof.
  pose true.
  pose (nazwa := 123).
Abort.
```

`pose` t dodaje do kontekstu term o domyślnej nazwie, którego ciałem jest t. Możemy też napisać `pose` $x := t$, dzięki czemu zyskujemy kontrolę nad nazwą termu.

```
Goal True.
Proof.
  Fail let x := 42 in pose x.
  let x := constr:(42) in pose x.
  let x := split in idtac x.
Abort.
```

W Ltacu, podobnie jak w języku Gallina, mamy do dyspozycji konstrukt `let`. Za jego pomocą możemy nadać nazwę dowolnemu wyrażeniu języka Ltac. Jego działanie jest podobne jak w języku Gallina, a więc nie ma co się nad nim rozwodzić. Jest też konstrukt `let rec`, który odpowiada `fix`owi Galliny.

Spróbujmy dodać do kontekstu liczbę 42, nazwaną dowolnie. Komendą `let` x := 42 `in pose` x nie udaje nam się tego osiągnąć. O przyczynie niepowodzenia Coq informuje nas wprost: zmienna x nie jest termem. Czym zatem jest? Jak już się rzekło, Ltac posiada wbudowany typ liczb całkowitych, które nie są tym samym, co induktywnie zdefiniowane liczby całkowite Coqa. W tym kontekście 42 jest więc liczbą całkowitą Ltaca, a zatem nie jest termem.

Aby wymusić na Ltacu zinterpretowanie 42 jako termu Coqa, musimy posłużyć się zapisem `constr`:(). Dzięki niemu argument znajdujący się w nawiasach zostanie zinterpretowany jako term. Efektem działania drugiej taktyki jest więc dodanie termu 42 : *nat* do kontekstu, nazwanego domyślnie n (co jest, o dziwo, dość rozsądną nazwą).

Wyrażenie `let` x := `split in idtac` x pokazuje nam, że taktyki również są wyrażeniami Ltaca i mogą być przypisywane do zmiennych (a także wyświetlane za pomocą taktyki `idtac`) tak jak każde inne wyrażenie.

```
Ltac garbage n :=
  pose n; idtac "Here's some garbage: " n.

Goal True.
Proof.
  garbage 0.
Abort.

Ltac garbage' :=
  fun n ⇒ pose n; idtac "Here's some garbage: " n.

Goal True.
Proof.
  garbage' 0.
Abort.
```

Dowolną taktykę, której możemy użyć w dowodzie, możemy też nazwać za pomocą komendy `Ltac` i odwoływać się do niej w dowodach za pomocą tej nazwy. Komenda `Ltac` jest więc taktykowym odpowiednikiem komendy `Fixpoint`.

Podobnie jak `Fixpointy` i inne definicje, tak i taktyki zdefiniowane za pomocą komendy `Ltac` mogą brać argumenty, którymi mogą być liczby, termy, nazwy hipotez albo inne taktyki.

Zapis `Ltac` *name arg_1 ... arg_n* := *body* jest jedynie skrótem, który oznacza `Ltac` *name* := `fun` *arg_1 ... arg_n* ⇒ *body*. Jest to uwaga mocno techniczna, gdyż pierwszy zapis jest prawie zawsze preferowany wobec drugiego.

5.3 Backtracking

Poznałeś już kombinator alternatywy ||. Nie jest to jednak jedyny kombinator służący do wyrażania tej idei — są jeszcze kombinatory + oraz *tryif t1* `then` *t2* `else` *t3*. Różnią się one działaniem — || jest left-biased, podczas gdy + nie jest biased i może powodować backtracking.

Nie przestrasz się tych dziwnych słów. Stojące za nimi idee są z grubsza bardzo proste. Wcześniej dowiedziałeś się, że taktyka może zawieść lub zakończyć się sukcesem. W rzeczywistości sprawa jest nieco bardziej ogólna: każda taktyka może zakończyć się dowolną ilością sukcesów. Zero sukcesów oznacza, że taktyka zawodzi. Większość taktyk, które dotychczas poznaliśmy, mogła zakończyć się co najwyżej jednym sukcesem. Są jednak i takie, które mogą zakończyć się dwoma lub więcej sukcesami.

Proces dowodzenia za pomocą taktyk można zobrazować za pomocą procesu przeszukiwania drzewa, którego wierzchołkami są częściowo skonstruowane prooftermy, zaś krawędziami — sukcesy pochodzące od wywoływania taktyk. Liśćmi są prooftermy (dowód się udał) lub ślepe zaułki (dowód się nie udał).

W takiej wizualizacji taktyka może wyzwalać backtracking, jeżeli jej użycie prowadzi do powstania rozgałęzienia w drzewie. Samo drzewo przeszukiwane jest w głąb, a backtracking polega na tym, że jeżeli trafimy na ślepy zaułek (dowód się nie powiódł), to cofamy się (ang. "to backtrack" — cofać się) do ostatniego punktu rozgałęzienia i próbujemy pójść inną gałęzią.

Tę intuicję dobrze widać na poniższym przykładzie.

```
Ltac existsNatFrom n :=
  ∃ n || existsNatFrom (S n).

Ltac existsNat := existsNatFrom O.

Goal ∃ n : nat, n = 42.
Proof.
  Fail (existsNat; reflexivity).
Abort.

Ltac existsNatFrom' n :=
  ∃ n + existsNatFrom' (S n).

Ltac existsNat' := existsNatFrom' O.

Goal ∃ n : nat, n = 42.
Proof.
  existsNat'; reflexivity.
Qed.
```

Próba użycia taktyki *existsNat*, która używa kombinatora ||, do udowodnienia, że $\exists\ n$: *nat*, $n = 42$ kończy się niepowodzeniem. Jest tak, gdyż || nie może powodować backtrackingu — jeżeli taktyka *t1* dokona postępu, to wtedy *t1* || *t2* ma taki sam efekt, jak *t1*, a w przeciwnym wypadku taki sam jak *t2*. Nawet jeżeli zarówno *t1* jak i *t2* zakończą się sukcesami, to sukcesy *t1* || *t2* będą sukcesami tylko *t1*.

Na mocy powyższych rozważań możemy skonkludować, że taktyka *existsNat* ma co najwyżej jeden sukces i działa jak $\exists\ n$ dla pewnej liczby naturalnej n. Ponieważ użycie $\exists$ 0 na celu $\exists\ n$: *nat*, $n = 42$ dokonuje postępu, to taktyka *existsNat* ma taki sam efekt, jak $\exists$ 0. Próba użycia `reflexivity` zawodzi, a ponieważ nie ma już więcej sukcesów pochodzących od *existsNat* do wypróbowania, nie wyzwala backtrackingu. Wobec tego cała taktyka *existsNat*; `reflexivity` kończy się porażką.

Inaczej sytuacja wygląda w przypadku *existsNat'*, która bazuje na kombinatorze +. Sukcesy *t1* + *t2* to wszystkie sukcesy *t1*, po których następują wszystkie sukcesy *t2*. Wobec tego zbiór sukcesów *existsNat'* jest nieskończony i wygląda tak: $\exists$ 0, $\exists$ 1, $\exists$ 2... Użycie taktyki `reflexivity`, które kończy się porażką wyzwala backtracking, więc całe wykonanie taktyki można zobrazować tak:

- $\exists$ 0; `reflexivity` — porażka
- $\exists$ 1; `reflexivity` — porażka
- ...
- $\exists$ 42; `reflexivity` — sukces

Na koniec zaznaczyć należy, że backtracking nie jest za darmo — im go więcej, tym więcej rozgałęzień w naszym drzewie poszukiwań, a zatem tym więcej czasu zajmie wykonanie taktyki. W przypadku użycia taktyk takich jak *existsNat*, które mają nieskończony zbiór sukcesów, dowód może nie zostać znaleziony nigdy, nawet jeżeli istnieje.

Jednym ze sposobów radzenia sobie z tym problemem może być kombinator *once*, który ogranicza liczbę sukcesów taktyki do co najwyżej jednego, zapobiegając w ten sposób potencjalnemu wyzwoleniu backtrackingu. Innymi słowy, *once t* zawsze ma 0 lub 1 sukcesów.

`Goal` $\exists\ n : nat,\ n = 42.$
`Proof.`
  Fail once existsNat'; `reflexivity.`
`Abort.`

Powyżej byliśmy w stanie udowodnić to twierdzenie za pomocą taktyki *existsNat'*, gdyż jej 42 sukces pozwalał taktyce `reflexivity` uporać się z celem. Jednak jeżeli użyjemy na tej taktyce kombinatora *once*, to zbiór jej sukcesów zostanie obcięty do co najwyżej jednego

Skoro *existsNat'* było równoważne któremuś z $\exists$ 0, $\exists$ 1, $\exists$ 2, ..., to *once existsNat'* jest równoważne $\exists$ 0, a zatem zawodzi.

Innym sposobem okiełznywania backtrackingu jest kombinator *exactly_once*, który pozwala upewnić się, że dana taktyka ma dokładnie jeden sukces. Jeżeli t zawodzi, to *exactly_once t* zawodzi tak jak t. Jeżeli t ma jeden sukces, *exactly_once t* działa tak jak t. Jeżeli t ma dwa lub więcej sukcesów, *exactly_once t* zawodzi.

```
Goal ∃ n : nat, n = 42.
Proof.
  exactly_once existsNat.
Restart.
  Fail exactly_once existsNat'.
Abort.
```

Taktyka *existsNat*, zrobiona kombinatorem alternatywy ||, ma dokładnie jeden sukces, a więc *exactly_once existsNat* działa tak jak *existsNat*. Z drugiej strony taktyka *existsNat'*, zrobiona mogącym dokonywać nawrotów kombinatorem alternatywy +, ma wiele sukcesów i wobec tego *exactly_once existsNat'* zawodzi.

Ćwiczenie (existsNat'') Przepisz taktykę *existsNat'* za pomocą konstruktu `let rec` — całość ma wyglądać tak: `Ltac` *existsNat''* := `let rec` ...

```
Goal ∃ n : nat, n = 42.
Proof.
  existsNat''; reflexivity.
Qed.
```

Ćwiczenie (exists_length_3_letrec) Udowodnij poniższe twierdzenie za pomocą pojedynczej taktyki, która generuje wszystkie możliwe listy wartości boolowskich. Całość zrób za pomocą konstruktu `let rec` w miejscu, tj. bez użycia komendy `Ltac`.

```
Require Import List.
Import ListNotations.

Theorem exists_length_3_letrec :
  ∃ l : list bool, length l = 3.
```

Ćwiczenie (trivial_goal) Znajdź taki trywialnie prawdziwy cel i taką taktykę, która wywołuje *existsNat'*, że taktyka ta nie skończy pracy i nigdy nie znajdzie dowodu, mimo że dla człowieka znalezienie dowodu jest banalne.

Ćwiczenie (search) Napisz taktykę *search*, która potrafi udowodnić cel będący dowolnie złożoną dysjunkcją pod warunkiem, że jeden z jej członów zachodzi na mocy założenia. Użyj rekursji, ale nie używaj konstruktu `let rec`.

Wskazówka: jeżeli masz problem, udowodnij połowę poniższych twierdzeń ręcznie i spróbuj dostrzec powtarzający si wzorzec.

```
Section search.

Hypotheses A B C D E F G H I J : Prop.

Theorem search_0 :
  I → A ∨ B ∨ C ∨ D ∨ E ∨ F ∨ G ∨ I ∨ J.
```

`Proof.` *search*. `Qed.`

`Theorem` *search_1* :
  $I \to (((((((A \lor B) \lor C) \lor D) \lor E) \lor F) \lor G) \lor I) \lor J.$
`Proof.` *search*. `Qed.`

`Theorem` *search_2* :
  $F \to (A \lor B) \lor (C \lor ((D \lor E \lor (F \lor G)) \lor H) \lor I) \lor J.$
`Proof.` *search*. `Qed.`

`Theorem` *search_3* :
  $C \to (J \lor J \lor ((A \lor A \lor (C \lor D \lor (E \lor E))))).$
`Proof.` *search*. `Qed.`

`Theorem` *search_4* :
  $A \to A \lor B \lor C \lor D \lor E \lor F \lor G \lor I \lor J.$
`Proof.` *search*. `Qed.`

`Theorem` *search_5* :
  $D \to \neg A \lor ((\tilde{}B \lor (I \to I) \lor (J \to J)) \lor (D \lor (\tilde{}D \to \tilde{}\tilde{}D) \lor B \lor B)).$
`Proof.` *search*. `Qed.`

`Theorem` *search_6* :
  $C \to (\tilde{}\tilde{}C \land \tilde{}\tilde{}\tilde{}C) \lor ((C \land \neg C) \lor (\tilde{}C \land C) \lor (C \to C) \lor (C \lor \neg C)).$
`Proof.` *search*. `Qed.`

`End` *search*.

Ćwiczenie (inne_kombinatory_dla_alternatywy) Zbadaj samodzielnie na podstawie dokumentacji, jak działają następujące kombinatory:

- *tryif t1* `then` *t2* `else` *t3*
- `first` [*t_1* | ... | *t_N*]
- `solve` [*t_1* | ... | *t_N*]

Precyzyjniej pisząc: sprawdź kiedy odnoszą sukces i zawodzą, czy mogą wyzwalać backtracking oraz wymyśl jakieś mądre przykłady, który dobrze ukazują ichdziałanie w kontraście do || i +.

5.4 Dopasowanie kontekstu i celu

Chyba najważniejszym konstruktem Ltaca jest `match goal`, który próbuje dopasować kontekst oraz cel do podanych wzorców. Mają one postać | *kontekst* ⊢ *cel* ⇒ *taktyka*.

Wyrażenie *kontekst* jest listą hipotez, których szukamy w kontekście, tzn. jest postaci *x_1* : *A_1*, *x_2* : *A_2*..., gdzie *x_i* jest nazwą hipotezy, zaś *A_1* jest wzorcem dopasowującym jej typ. Wyrażenie *cel* jest wzorcem dopasowującym typ, który reprezentuje nasz cel. Po strzałce ⇒ następuje taktyka, której chcemy użyć, jeżeli dany wzorzec zostanie dopasowany.

Zamiast wzorców postaci | *kontekst* ⊢ *cel* ⇒ *taktyka* możemy też używać wzorców postaci | ⊢ *cel* ⇒ *taktyka*, które dopasowują jedynie cel, zaś kontekst ignorują; wzorców postaci | *kontekst* ⊢ _ ⇒ *taktyka*, które dopasowują jedynie kontekst, a cel ignorują; oraz wzorca _, który oznacza "dopasuj cokolwiek".

Zobaczmy, jak to wygląda na przykładach.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
  match goal with
      | x : Prop ⊢ _ ⇒ idtac x
  end.
Abort.
```

W powyższym przykładzie szukamy w celu zdań logicznych, czyli termów typu `Prop` i wypisujemy je. Nazwy szukanych obiektów są lokalne dla każdej gałęzi dopasowania i nie muszą pokrywać się z rzeczywistymi nazwami termów w kontekście. W naszym przypadku nazywamy szukane przez nas zdanie x, choć zdania obecne w naszym kontekście tak naprawdę nazywają się P, Q, R oraz S.

Przeszukiwanie obiektów w kontekście odbywa się w kolejności od najnowszego do najstarszego. Do wzorca x : `Prop` najpierw próbujemy dopasować *H1* : R, ale R to nie `Prop`, więc dopasowanie zawodzi. Podobnie dla *H0* : Q oraz H : P. Następnie natrafiamy na S : `Prop`, które pasuje do wzorca. Dzięki temu na prawo od strzałki ⇒ nazwa x odnosi się do dopasowanego zdania S. Za pomocą taktyki `idtac` x wyświetlamy x i faktycznie odnosi się on do S. Po skutecznym dopasowaniu i wykonaniu taktyki `idtac` x cały `match` kończy się sukcesem.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
  Fail match goal with
      | x : Prop ⊢ _ ⇒ idtac x; fail
  end.
Abort.
```

W tym przykładzie podobnie jak wyżej szukamy w kontekście zdań logicznych, ale taktyka po prawej od ⇒ zawodzi. `match` działa tutaj następująco:

- próbujemy do wzorca x : `Prop` dopasować *H1* : R, ale bez powodzenia i podobnie dla *H0* : Q oraz H : P.
- znajdujemy dopasowanie S : `Prop`. Taktyka `idtac` x wypisuje do okna Messages wiadomość "S" i kończy się sukcesem, ale `fail` zawodzi.

- Wobec powyższego próbujemy kolejnego dopasowania, tym razem R : `Prop`, które pasuje. `idtac` x wypisuje na ekran "R", ale `fail` znów zawodzi.

- Próbujemy kolejno dopasowań Q : `Prop` i P : `Prop`, w wyniku których wypisane zostaje "Q" oraz "P", ale również w tych dwóch przypadkach taktyka `fail` zawodzi.

- Nie ma więcej potencjalnych dopasowań, więc cały `match` zawodzi.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
  Fail match reverse goal with
      | x : Prop ⊢ _ ⇒ idtac x; fail
  end.
Abort.
```

Ten przykład jest podobny do ostatniego, ale `match` *reverse* przeszukuje kontekst w kolejności od najstarszego do najnowszego. Dzięki temu od razu natrafiamy na dopasowanie P : `Prop`, potem na Q : `Prop` etc. Na samym końcu próbujemy do x : `Prop` dopasować H : P, *H0* : Q i *H1* : R, co kończy się niepowodzeniem.

Zauważmy, że w dwóch ostatnich przykładach nie wystąpił backtracking — `match` nigdy nie wyzwala backtrackingu. Obserwowane działanie `match`a wynika stąd, że jeżeli taktyka po prawej od ⇒ zawiedzie, to następuje próba znalezienia jakiegoś innego dopasowania wzorca x : `Prop`. Dopiero gdy taktyka na prawo od ⇒ zawiedzie dla wszystkich możliwych takich dopasowań, cały `match` zawodzi.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
  Fail
  match goal with
      | x : Prop ⊢ _ ⇒ idtac x
  end; fail.
Abort.
```

Ten przykład potwierdza naszą powyższą obserwację dotyczącą backtrackingu. Mamy tutaj identyczne dopasowanie jak w pierwszym przykładzie — wypisuje ono S i kończy się sukcesem, ale tuż po nim następuje taktyka `fail`, przez co cała taktyka `match ...; fail` zawodzi. Jak widać, nie następuje próba ponownego dopasownia wzorca x : `Prop`.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
```

```
  Fail
  lazymatch goal with
      | x : Prop ⊢ _ ⇒ idtac x; fail
  end.
Abort.
```

Konstrukt `lazymatch` różni się od `match`a tym, że jeżeli taktyka na prawo od ⇒ zawiedzie, to alternatywne dopasowania wzorca po lewej nie będą rozważane i nastąpi przejście do kolejnej gałęzi dopasowania. W naszym przypadku nie ma kolejnych gałęzi, więc po pierwszym dopasowaniu x : `Prop` do S : `Prop` i wypisaniu "S" cały `lazymatch` zawodzi.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
  Fail
  multimatch goal with
      | x : Prop ⊢ _ ⇒ idtac x
  end; fail.
Abort.
```

multimatch to wariant `match`a, który wyzwala backtracking. W powyższym przykładzie działa on następująco:

- do wzorca x : `Prop` dopasowujemy *H1* : R, a następnie *H0* : Q i H : P, co się rzecz jasna nie udaje.
- Znajdujemy dopasowanie S : `Prop` i cały *multimatch* kończy się sukcesem.
- Taktyka `fail` zawodzi i wobec tego cała taktyka *multimatch* ...; `fail` taże zawodzi.
- Następuje nawrót i znów próbujemy znaleźć dopasowanie wzorca x : `Prop`. Znajdujemy R : `Prop`, *multimatch* kończy się sukcesem, ale `fail` zawodzi.
- Następują kolejne nawroty i dopasowania do wzorca. Ostatecznie po wyczerpaniu się wszystkich możliwość cała taktyka zawodzi.

```
Goal
  ∀ P Q R S : Prop, P → Q → R → S.
Proof.
  intros.
  match goal with
      | x : Prop ⊢ _ ⇒ idtac x
  end.
  multimatch goal with
      | x : Prop ⊢ _ ⇒ idtac x
```

```
  end.
  repeat match goal with
    | x : Prop ⊢ _ ⇒ idtac x
  end.
  repeat multimatch goal with
    | x : Prop ⊢ _ ⇒ idtac x
  end.
Abort.
```

Przyjrzyjmy się jeszcze różnicy w zachowaniach `match`a i *multimatch*a w połączeniu z kombinatorem `repeat`. Bez `repeat` oba dopasowania zachowują się identycznie. Użycie `repeat` przed `match` nie zmienia w tym konkretnym wypadku jego działania, ale w przypadku *multimatch*a użycie `repeat` ujawnia wszystkie jego sukcesy.

Źródło różnego zachowania `match`a i *multimatch*a, jeżeli chodzi o backtracking, jest bardzo proste: tak naprawdę `match` jest jedynie skrótem dla *once multimatch*. `lazymatch`, choć nie pokazano tego na powyższym przykładzie, w obu wypadkach (z `repeat` i bez) zachowuję się tak jak `match`.

Przyjrzyjmy się teraz dopasowaniom celu.

```
Goal
  ∀ (P Q R S : Prop) (a b c : nat),
    42 = 43 ∧ (P → Q).
Proof.
  intros. split;
  match goal with
    | X : Prop ⊢ P → Q ⇒ idtac X
    | n : nat ⊢ 42 = 43 ⇒ idtac n
  end.
Abort.
```

Dopasowanie celu jest jeszcze prostsze niż dopasowanie hipotezy, bo cel jest tylko jeden i wobec tego nie trzeba dawać mu żadnej nazwy. Powyższa taktyka `split`; `match` ... działa następująco:

- `split` generuje dwa podcele i wobec tego `match` działa na każdym z nich z osobna
- pierwszy wzorzec głosi, że jeżeli w kontekście jest jakieś zdanie logiczne, które nazywamy X, a cel jest postaci $P \to Q$, to wypisujemy X
- drugi wzorzec głosi, że jeżeli w kontekście jest jakaś liczba naturalna, którą nazywamy n, a cel jest postaci $42 = 43$, to wypisujemy n
- następuje próba dopasowania pierwszego wzorca do pierwszego podcelu. Mimo, że w kontekście są zdania logiczne, to cel nie jest postaci $P \to Q$, a zatem dopasowanie zawodzi.

- następuje próba dopasowania drugiego wzorca do pierwszego podcelu. W kontekście jest liczba naturalna i cel jest postaci $42 = 43$, a zatem dopasowanie udaje się. Do okna Messages zostaje wypisane "c", które zostało dopasowane jako pierwsze, gdyż kontekst jest przeglądany w kolejności od najstarszej hipotezy do najświeższej.

- pierwszy wzorzec zostaje z powodzeniem dopasowany do drugiego podcelu i do okna Messages zostaje wypisane "S".

```
Goal
  ∀ (P Q R S : Prop) (a b c : nat), P.
Proof.
  intros.
  match goal with
      | _ ⇒ idtac "_"
  end.
  match goal with
      | _ ⇒ fail
      | X : Prop ⊢ _ ⇒ idtac X
  end.
Abort.
```

Pozostało nam jedynie zademonstrować działanie wzorca _. Pierwsza z powyższych taktyk z sukcesem dopasowuje wzorzec _ (gdyż pasuje on do każdego kontekstu i celu) i wobec tego do okna Messages zostaje wypisany napis "_".

W drugim `match`u również zostaje dopasowany wzorzec _, ale taktyka `fail` zawodzi i następuje przejście do kolejnego wzorca, który także pasuje. Wobec tego wypisane zostaje "S". Przypomina to nam o tym, że kolejność wzorców ma znaczenie i to nawet w przypadku, gdy któryś z nich (tak jak _) pasuje do wszystkiego.

Ćwiczenie (destr_and) Napisz taktykę *destr_and*, która rozbija wszystkie koniunkcje, które znajdzie w kontekście, a następnie udowodni cel, jeżeli zachodzi on na mocy założenia.

Dla przykładu, kontekst $H : P \land Q \land R \vdash \_$ powinien zostać przekształcony w $H : P$, $H0 : Q$, $H1 : R$.

Jeżeli to możliwe, nie używaj kombinatora ;

```
Section destr_and.

Hypotheses A B C D E F G H I J : Prop.

Theorem destruct_0 :
    A ∧ B ∧ C ∧ D ∧ E ∧ F ∧ G ∧ H ∧ I ∧ J → D.
Proof. destr_and. Qed.

Theorem destruct_1 :
    ((((((((A ∧ B) ∧ C) ∧ D) ∧ E) ∧ F) ∧ G) ∧ H) ∧ I) ∧ J → F.
Proof. destr_and. Qed.
```

Theorem *destruct_2* :
$A \land \neg B \land (C \lor C \lor C \lor C) \land ((((D \land I) \land I) \land I) \land J) \to I.$
Proof. *destr_and*. Qed.

End *destr_and*.

Ćwiczenie (solve_and_perm) Napisz taktykę *solve_and_perm*, która będzie potrafiła rozwiązywać cele postaci *P_1* $\land$ *P_2* $\land$ ... $\land$ *P_n* $\to$ *P_i1* $\land$ *P_i2* $\land$ ... $\land$ *P_iN*, gdzie prawa strona implikacji jest permutacją lewej strony, tzn. są w niej te same zdania, ale występujące w innej kolejności.

Section *solve_and_perm*.

Hypotheses $A\ B\ C\ D\ E\ F\ G\ H\ I\ J$: Prop.

Theorem *and_perm_0* :
$A \land B \land C \land D \land E \land F \land G \land H \land I \land J \to$
$J \land I \land H \land G \land F \land E \land D \land C \land B \land A.$
Proof. *solve_and_perm*. Qed.

Theorem *and_perm_1* :
$A \land B \land C \land D \land E \land F \land G \land H \land I \land J \to$
$(((((((((A \land B) \land C) \land D) \land E) \land F) \land G) \land H) \land I) \land J).$
Proof. *solve_and_perm*. Qed.

Theorem *and_perm_2* :
$(A \land B) \land (C \land (D \land E)) \land (((F \land G) \land H) \land I) \land J \to$
$(I \land I \land J) \land ((A \land B \land (A \land B)) \land J) \land (C \land (E \land (D \land F \land F))).$
Proof. *solve_and_perm*. Qed.

End *solve_and_perm*.

Ćwiczenie (solve_or_perm) Napisz taktykę *solve_or_perm*, która będzie potrafiła rozwiązywać cele postaci *P_1* $\lor$ *P_2* $\lor$ ... $\lor$ *P_n* $\to$ *P_i1* $\lor$ *P_i2* $\lor$ ... $\lor$ *P_iN*, gdzie prawa strona implikacji jest permutacją lewej strony, tzn. są w niej te same zdania, ale występujące w innej kolejności.

Wskazówka: wykorzystaj taktykę *search* z jednego z poprzednich ćwiczeń.

Section *solve_or_perm*.

Hypotheses $A\ B\ C\ D\ E\ F\ G\ H\ I\ J$: Prop.

Theorem *or_perm_0* :
$A \lor B \lor C \lor D \lor E \lor F \lor G \lor H \lor I \lor J \to$
$J \lor I \lor H \lor G \lor F \lor E \lor D \lor C \lor B \lor A.$
Proof. *solve_or_perm*. Qed.

Theorem *or_perm_1* :
$A \lor B \lor C \lor D \lor E \lor F \lor G \lor H \lor I \lor J \to$

$(((((((((A \lor B) \lor C) \lor D) \lor E) \lor F) \lor G) \lor H) \lor I) \lor J)$.
Proof. *solve_or_perm*. Qed.

Theorem *or_perm_2* :
$(A \lor B) \lor (C \lor (D \lor E)) \lor (((F \lor G) \lor H) \lor I) \lor J \to$
$(I \lor H \lor J) \lor ((A \lor B \lor (G \lor B)) \lor J) \lor (C \lor (E \lor (D \lor F \lor F)))$.
Proof. *solve_or_perm*. Qed.

Theorem *or_perm_3* :
$A \lor B \lor C \lor D \lor E \lor F \lor G \lor H \lor I \lor J \to$
$(((((((((A \lor B) \lor C) \lor D) \lor E) \lor F) \lor G) \lor H) \lor I) \lor J)$.
Proof. *solve_or_perm*. Qed.

End *solve_or_perm*.

Ćwiczenie (negn) Section *negn*.

Require Import *Arith*.

Napisz funkcję *negn* : *nat* → Prop → Prop, gdzie *negn n P* zwraca zdanie *P* zanegowane *n* razy.

Eval *cbn* in *negn* 10 *True*.
```
(*  ===> = ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ True : Prop *)
```

Udowodnij poniższe lematy.

Lemma *dbl_neg* :
∀ *P* : Prop, $P \to \neg \neg P$.

Lemma *double_n* :
∀ *n* : *nat*, $2 \times n = n + n$.

Przydadzą ci się one do pokazania dwóch właściwości fukncji *negn*. Zanim przystąpisz do dowodzenia drugiego z nich, spróbuj zgadnąć, po którym argumencie najprościej będzie przeprowadzić indukcję.

Theorem *even_neg* :
∀ (*n* : *nat*) (*P* : Prop), *P* → *negn* $(2 \times n)$ *P*.

Theorem *even_neg'* :
∀ (*n k* : *nat*) (*P* : Prop),
negn $(2 \times n)$ *P* → *negn* $(2 \times (n + k))$ *P*.

Napisz taktykę *negtac*, która będzie potrafiła udowadniać cele postaci ∀ *P* : Prop, *negn* $(2 \times n)$ *P* → *negn* $(2 \times (n + k))$ *P*, gdzie *n* oraz *k* są stałymi. Nie używaj twierdzeń, które udowodniłeś wyżej.

Wskazówka: przydatny może byc konstrukt match *reverse* goal.

Theorem *neg_2_14* :
∀ *P* : Prop, *negn* 2 *P* → *negn* 14 *P*.

```
Proof. negtac. Qed.

Theorem neg_100_200 :
  ∀ P : Prop, negn 100 P → negn 200 P.
Proof. negtac. Qed.

Theorem neg_42_1000 :
  ∀ P : Prop, negn 42 P → negn 200 P.
Proof. negtac. Qed.

End negn.
```

5.5 Wzorce i unifikacja

Skoro wiemy już jak działa dopasowywanie kontekstu do wzorca, czas nauczyć się jak dokładnie działają wzorce oraz czym są zmienne unifikacyjne i sama unifikacja.

Przede wszystkim, jak przekonaliśmy się wyżej, termy są wzorcami. Termy nie zawierają zmiennych unifikacyjnych, a wzorce będące termami dopasowują się tylko do identycznych termów. Dopasowanie takie nie wiąże żadnych nowych zmiennych. Zobaczmy to na przykładzie.

```
Goal
  ∀ P Q : Prop, P → P ∨ Q.
Proof.
  intros.
  match goal with
      | p : P ⊢ P ∨ Q ⇒ left; assumption
  end.
Qed.
```

Powyższy `match` nie zawiera zmiennych unifikacyjnych i działa w następujący sposób:

- szukamy w kontekście obiektu p, którego typ pasuje do wzorca P. Obiekt, który nazywamy p w rzeczywistości nie musi nazywać się p, ale jego typem rzeczywiście musi być P. W szczególności, wzorzec P nie pasuje do Q, gdyż P i Q nie są konwertowalne.
- jednocześnie żądamy, by cel był postaci $P \lor Q$, gdzie zarówno P jak i Q odnoszą się do obiektów z kontekstu, które rzeczywiście tak się nazywają.
- jeżeli powyższe wzorce zostaną dopasowane, to używamy taktyki `left; assumption`, która rozwiązuje cel.

Zobaczmy, co się stanie, jeżeli w powyższym przykładzie zmienimy nazwy hipotez.

```
Goal
  ∀ A B : Prop, A → A ∨ B.
Proof.
```

```
  intros.
  Fail match goal with
      | p : P ⊢ P ∨ Q ⇒ left; assumption
  end.
  match goal with
      | p : A ⊢ A ∨ B ⇒ left; assumption
  end.
Qed.
```

Tutaj zamiast P mamy A, zaś zamiast Q jest B. match identyczny jak poprzednio tym razem zawodzi. Dzieje się tak, gdyż P odnosi się tu do obiektu z kontekstu, który nazywa się P. Niestety, w kontekście nie ma obiektu o takiej nazwie, o czym Coq skrzętnie nas informuje.

W matchu w celu oraz po prawej stronie od : w hipotezie nie możemy za pomocą nazwy P dopasować obiektu, który nazywa się A. Dopasować A możemy jednak używając wzorca A. Ale co, gdybyśmy nie wiedzieli, jak dokładnie nazywa się poszukiwany obiekt?

```
Goal
  ∀ A B : Prop, A → A ∨ B.
Proof.
  intros.
  match goal with
      | p : ?P ⊢ ?P ∨ ?Q ⇒ idtac P; idtac Q; left; assumption
  end.
Qed.
```

Jeżeli chcemy dopasować term o nieznanej nam nazwie (lub term, którego podtermy mają nieznane nazwy) musimy użyć zmiennych unifikacyjnych. Wizualnie można rozpoznać je po tym, że ich nazwy zaczynają się od znaku ?. Zmienna unifkacyjna $?x$ pasuje do dowolnego termu, a udane dopasowanie sprawia, że po prawej stronie strzałki ⇒ możemy do dopasowanego termu odnosić się za pomocą nazwy x.

Powyższe dopasowanie działa w następujący sposób:

- próbujemy dopasować wzorzec p : $?P$ do najświeższej hipotezy w kontekście, czyli H : A. p jest nazwą tymczasową i wobec tego pasuje do H, zaś zmienna unifikacyjna $?P$ pasuje do dowolnego termu, a zatem pasuje także do A.

- dopasowanie hipotezy kończy się sukcesem i wskutek tego zmienna unifikacyjna $?P$ zostaje związana z termem A. Od teraz w dalszych wzorcach będzie ona pasować jedynie do termu A.

- następuje próba dopasowania celu do wzorca $?P \lor ?Q$. Ponieważ $?P$ zostało związane z A, to wzorzec $?P \lor ?Q$ oznacza tak naprawdę $A \lor ?Q$. Zmienna unifikacyjna $?Q$ nie została wcześniej związana i wobec tego pasuje do wszystkiego.

- wobec tego $?Q$ w szczególności pasuje do B, a zatem wzorzec $?P \lor ?Q$ pasuje do $A \lor B$ i całe dopasowanie kończy się sukcesem. W jego wyniku $?Q$ zostaje związane z B.

- zostaje wykonana taktyka `idtac` P; `idtac` Q, która potwierdza, że zmienna unifikacyjna $?P$ została związana z A, a $?Q$ z B, wobec czego na prawo od $\Rightarrow$ faktycznie możemy do A i B odwoływać się odpowiednio jako P i Q.

- taktyka `left`; `assumption` rozwiązuje cel.

Podkreślmy raz jeszcze, że zmienne unifikacyjne mogą występać tylko we wzorcach, a więc w hipotezach po prawej stronie dwukropka : oraz w celu. Błędem byłoby napisanie w hipotezie $?p : ?P$. Podobnie błędem byłoby użycie nazwy $?P$ na prawo od strzałki $\Rightarrow$.

Zauważmy też, że w danej gałęzi `match`a każda zmienna unifikacyjna może wystąpić więcej niż jeden raz. Wzorce, w których zmienne unifikacyjne występują więcej niż raz to wzorce nieliniowe. Możemy skontrastować je ze wzorcami liniowymi, w których każda zmienna może wystąpić co najwyżej raz.

Wzorcami liniowymi są wzorce, których używamy podczas definiowania zwykłych funkcji przez dopasowanie do wzorca (zauważmy jednak, że tamtejsze zmienne unifikacyjne nie zaczynają się od ?). Ograniczenie do wzorców liniowych jest spowodowane faktem, że nie zawsze możliwe jest stwierdzenie, czy dwa dowolne termy do siebie pasują.

Język termów Coqa w celu uniknięcia sprzeczności musi być zupełnie nieskazitelny i musi zakazywać używania wzorców nieliniowych. Język Ltac, który nie może sam z siebie wyczarować sprzeczności, może sobie pozwolić na więcej i wobec tego wzorce nieliniowe są legalne.

```
Goal
  [2] = [].
Proof.
  match goal with
      | ⊢ ?x = _ ⇒ idtac x
  end.
  match goal with
      | ⊢ cons ?h _ = nil ⇒ idtac h
  end.
  match goal with
      | ⊢ 2 :: _ = ?l ⇒ idtac l
  end.
  match goal with
      | ⊢ [?x] = [] ⇒ idtac x
  end.
Abort.
```

Zauważmy, że nie musimy używać zmiennych unifikacyjnych do dopasowywania całych termów — w pierwszym z powyższych przykładów używamy zmiennej $?x$, aby dopasować jedynie lewą stronę równania, które jest celem.

Ze zmiennych unifikacyjnych oraz stałych, zmiennych i funkcji (a więc także konstruktorów) możemy budować wzorce dopasowujące termy o różnych fikuśnych kształtach.

W drugim przykładzie wzorzec $cons\ ?h\ \_ = nil$ dopasowuje równanie, którego lewa strona jest listą niepustą o dowolnej głowie, do której możemy się odnosić jako h, oraz dowolnym ogonie, do którego nie chcemy móc się odnosić. Prawa strona tego równania jest listą pustą.

Wzorce radzą sobie bez problemu także z notacjami. Wzorzec $2 :: \_ = ?l$ dopasowuje równanie, którego lewa strona jest listą, której głowa to 2, zaś ogon jest dowolny, a prawa strona jest dowolną listą, do której będziemy się mogli odwoływać po prawej stronie $\Rightarrow$ jako l.

Ostatni wzorzec pasuje do równania, którego lewa strona jest singletonem (listą jednoelementową) zawierającym wartość, do której będziemy mogli odnosić się za pomocą nazwy x, zaś prawą stroną jest lista pusta.

Ćwiczenie (my_assumption) Napisz taktykę *my_assumption*, która działa tak samo, jak `assumption`. Nie używaj `assumption` — użyj `match`a.

```
Goal
  ∀ P : Prop, P → P.
Proof.
  intros. my_assumption.
Qed.
```

Ćwiczenie (forward) Napisz taktykę *forward*, która wyspecjalizuje wszystkie znalezione w kontekście implikacje, o ile oczywiście ich przesłanki również będą znajdowały się w kontekście, a następnie rozwiąże cel, jeżeli jest on prawdziwy na mocy założenia.

Dla przykładu, kontekst $H : P \to Q,\ H0 : Q \to R,\ H1 : P \vdash \_$ powinien zostać przekształcony w $H : Q,\ H0 : R,\ H1 : P \vdash \_$.

Wskazówka: przydatna będzie taktyka `specialize`.

```
Example forward_1 :
  ∀ P Q R : Prop, (P → Q) → (Q → R) → P → R.
Proof. forward. Qed.

Example forward_2 :
  ∀ P Q R S : Prop, (P → Q → R) → (S → Q → P → R).
Proof. forward. Qed.
```

5.6 Narzędzia przydatne przy dopasowywaniu

Poznawszy już konstrukt `match` i jego warianty oraz sposób dopasowywania wzorców i rolę unifikacji oraz zmiennych unifikacyjnych w tym procesie, czas rzucić okiem na kilka niezwykle przydatnych narzędzi, które uczynią nasze życie dopasowywacza łatwiejszym.

5.6.1 Dopasowanie podtermu

Pierwszym z nich jest wyrażenie `context` *ident* [*term*], dzięki któremu możemy tworzyć wzorce dopasowujące podtermy danego termu. Zobaczmy jego działanie na przykładzie.

```
Goal
  ∀ a b c : nat, a = b → b = c → a = c.
Proof.
  intros a b c.
  match goal with
      | ⊢ context G [?x = ?y] ⇒ idtac G x y
  end.
  repeat multimatch goal with
      | ⊢ context G [?x = ?y] ⇒ idtac G x y
  end.
Abort.
```

W powyższym przykładzie naszym celem jest znalezienie wszystkich równań, które są podtermami naszego celu. Dopasowanie wzorca `context` G $[?x = ?y]$ przebiega w następujący sposób:

- w celu wyszukiwae są wszystkie podtermy postaci $?x = ?y$. Są trzy takie: $a = b$, $b = c$ oraz $a = c$
- wzorzec $?x = ?y$ zostaje zunifikowany z pierwszym pasującym podtermem, czyli $a = b$. W wyniku dopasowania zmienna unifikacyjna $?x$ zostaje związana z a, zaś $?y$ z b
- cały term, którego podterm został dopasowany do wzorca, zostaje związany ze zmienną G, przy czym jego dopasowany podterm zostaje specjalnie zaznaczony (po wypisaniu w jego miejscu widać napis "?M-1")
- zostaje wykonana taktyka `idtac` G x y

Druga z powyższych taktyk działa podobnie, ale dzięki zastosowaniu `repeat` *multimatch* ujawnia nam ona wszystkie podtermy pasujące do wzorca $?x = ?y$.

Ćwiczenie (podtermy) Oblicz ile podtermów ma term 42. Następnie napisz taktykę *nat_subterm*, która potrafi wypisać wszystkie podtermy dowolnej liczby naturalnej, która znajduje się w celu. Wymyśl odpowiedni cel i przetestuj na nim swoje obliczenia.

5.6.2 Generowanie nieużywanych nazw

Drugim przydatnym narzędziem jest konstrukt `fresh`, który pozwala nam wygenerować nazwę, której nie nosi jeszcze żadna zmienna. Dzięki temu możemy uniknąć konfliktów nazw, gdy używamy taktyk takich jak `intros` czy `destruct`, które pozwalają nam nazywać obiekty. Przyjrzyjmy się następującemu przykładowi.

```
Goal ∀ x y : nat, {x = y} + {x ≠ y}.
Proof.
  intro x. Fail intro x.
  let x := fresh in intro x.
Restart.
  intro x. let x := fresh "y"in intro x.
Restart.
  intro x. let x := fresh x in intro x.
Restart.
  intro x. let x := fresh y in intro x.
Abort.
```

Mamy w kontekście liczbę naturalną x : *nat* i chcielibyśmy wprowadzić do niego kolejną. Cóż, nie jest to żaden problem — wystarczy nazwać go dowolną nazwą różną od "x". Ale co, jeżeli nie wiemy, jak nazywają się obiekty znajdujące się w kontekście?

Przy intensywnym posługiwaniu się taktykami i automatyzacją jest to nader częsta możliwość: gdy dopasujemy kontekst za pomocą `match`a, nie znamy oryginalnych nazw dopasowanych termów — możemy odwoływać się do nich tylko za pomocą nazw lokalnych, wprowadzonych na potrzeby danego wzorca.

Z odsieczą przychodzi nam generator świeżych nazw o wdzięcznej nazwie `fresh`. Zazwyczaj będziemy się nim posługiwać w następujący sposób: `let` *var* := `fresh` *arg_1 ... arg_N* `in` *t*. Tutaj *var* jest zmienną języka `Ltac`, której wartością jest świeżo wygenerowana nazwa, a *t* to jakaś taktyka, która w dowolny sposób korzysta z *var*.

Powyższe cztery taktyki działają tak:

- `let` x := `fresh in intro` x — `fresh` generuje świeżą nazwę, domyślnie jest nią "H". Nazwa ta staje się wartością Ltacowej zmiennej x. Owa zmienna jest argumentem taktyki `intro`, dzięki czemu wprowadzony do kontekstu obiekt typu *nat* zostaje nazwany "H".

- `let` x := `fresh "`y`"in intro` x — jeżeli `fresh` dostanie jako argument ciąg znaków, to wygeneruje nazwę zaczynającą się od tego ciągu, która nie jest jeszcze zajęta. Ponieważ nazwa "y" jest wolna, właśnie tak zostaje nazwany wprowadzany obiekt.

- `let` x := `fresh` x `in intro` x — tutaj mamy mały zamęt. Pierwszy i trzeci x jest zmienną Ltaca, zaś drugi odnosi się do obiektu z kontekstu. Jeżeli *arg* jest obiektem z kontekstu, to `fresh` *arg* tworzy świeżą nazwę zaczynającą się od nazwy, jaką *arg* nosi w kontekście. Tutaj nie ma to znaczenia, gdyż x nazywa się po prostu "x" i wobec tego `fresh` generuje nazwę "x0", ale mechanizm ten działa tak samo w przypadku zmiennych unifikacyjnych.

- `let` x := `fresh` y `in intro` x — jak widać, argumentem `fresh` może też być nazwa zmiennej nie odnosząca się zupełnie do niczego. W naszym przypadku nie ma w kontekście zmiennej y, a `fresh` generuje na jej podstawie świeżą nazwę "y".

5.6.3 `fail` (znowu)

Taktykę `fail` już poznaliśmy, ale nie w jej pełnej krasie. Czas więc odkryć resztę jej możliwości.

```
Goal False.
Proof.
  Fail fail "Hoho, czego się spodziewałeś?"1.
Abort.
```

Pierwsza z nich nie jest zbyt spektakularna — możemy do `fail` przekazać jako argumenty ciągi znaków lub termy, co spowoduje wyświetlenie ich w oknie wiadomości.

Drugą, znacznie ważniejszą możliwością, jaką daje nam taktyka `fail`, jest kontrola "poziomu porażki". Dzięki niemu zyskujemy władzę nad tym, jak "mocno" taktyka `fail` zawodzi. Domyśnie wynosi on 0. Użycie taktyki `fail` (która wobec tego oznacza to samo, co `fail 0`) powouje przerwanie wykonywania obecnej gałęzi `match`a i przejście do następnej. Użycie taktyki `fail` *n*, gdzie n nie jest równe 0, powoduje opuszczenie całego obecnego `match`a (tj. wszystkich gałęzi) lub bloku `do`/`repeat` i wywołanie `fail` (*n* - 1).

Przyjrzyjmy się temu zachowaniu na przykładzie.

```
Goal False.
Proof.
  match goal with
      | _ ⇒ idtac "first branch"; fail
      | _ ⇒ idtac śecond branch"
  end.
  Fail match goal with
      | _ ⇒ idtac "first branch"; fail 1
      | _ ⇒ idtac śecond branch"
  end.
  try match goal with
      | _ ⇒ idtac "first branch"; fail 1
      | _ ⇒ idtac śecond branch"
  end.
  Fail try match goal with
      | _ ⇒ idtac "first branch"; fail 2
      | _ ⇒ idtac śecond branch"
  end.
Abort.
```

Cztery powyższe dopasowania działają następująco:

- W pierwszym dopasowana jest pierwsza gałąź. Wyświetlona zostaje wiadomość, po czym taktyka `fail` zawodzi i następuje przejście do kolejnej gałęzi. Tutaj też wypisana zostaje wiadomość i cała taktyka `match` ... kończy się sukcesem.

- W drugim przypadku dopasowana jest pierwsza gałąź, która wypisuje wiadomość, ale taktyka `fail` 1 powoduje, że cały `match` zawodzi i druga gałąź nie jest w ogóle dopasowywana.

- Trzeci przypadek jest podobny do drugiego. `fail` 1 powoduje, że cały `match` zawodzi, ale dzięki kombinatorowi `try` cała taktyka `try match` ... kończy się sukcesem.

- Czwarta taktyka jest podobna do trzeciej, ale tym razem po udanym dopasowaniu pierwszej gałęzi taktyka `fail` 2 powoduje, że cały `match` zawodzi. Następnie ma miejsce wywołanie taktyki `fail` 1, które powoduje, że nawet mimo użycia kombinatora `try` cała taktyka `try match` ... zawodzi.

5.7 Inne (mało) wesołe rzeczy

Ten podrozdział będzie wesołą zbieraninką różnych niezbyt przydatnych (przynajmniej dla mnie) konstruktów języka Ltac, które nie zostały dotychczas omówione.

```
Goal False ∧ False ∧ False.
Proof.
  repeat split.
  let n := numgoals in idtac n.
  all: let n := numgoals in idtac n.
Abort.
```

Ilość celów możemy policzyć za pomocą taktyki *numgoals*. Liczy ona wszystkie cele, na które działa, więc jeżeli nie użyjemy żadnego selektora, zwróci ona 1. Nie jest ona zbyt użyteczna (poza bardzo skomplikowanymi taktykami, które z jakichś powodów nie operują tylko na jednym celu, lecz na wszystkich).

```
Goal False ∧ False ∧ False.
Proof.
  repeat split.
  all: let n := numgoals in guard n > 2.
  Fail all: let n := numgoals in guard n < 2.
Abort.
```

Taktyka *guard cond* pozwala nam dokonywać prostych testów na liczbach całkowitych Ltaca. Jeżeli warunek zachodzi, taktyka ta zachowuje się jak `idtac`, czyli kończy się sukcesem i nie robi nic więcej. Jeżeli warunek nie zachodzi, taktyka zawodzi.

W powyższym przykładzie taktyka *guard* $n > 2$ kończy się sukcesem, gdyż są 3 cele, a $3 > 2$, zaś taktyka *guard* $n < 2$ zawodzi, bo są 3 cele, a nie jest prawdą, że $3 < 2$.

```
Inductive even : nat → Prop :=
    | even0 : even 0
    | evenSS : ∀ n : nat, even n → even (S (S n)).

Goal even 42.
```

```
Proof.
  try timeout 1 repeat constructor.
Abort.

Goal even 1338.
Proof.
  try timeout 1 repeat constructor.
Abort.
```

Kombinator *timeout n t* pozwala nam sprawić, żeby taktyka t zawiodła, jeżeli jej wykonanie będzie zajmowało dłużej, niż n sekund. Nie jest on zbyt przydatny, gdyż szybkość wykonania danej taktyki jest kwestią mocno zależną on sprzętu. Jak można przeczytać w manualu, kombinator ten bywa przydatny głównie przy debugowaniu i nie zaleca się, żeby występował w finalnych dowodach, gdyż może powodować problemy z przenośnością.

W powyższym przykładzie taktyka *timeout* 1 `repeat constructor` kończy się sukcesem, gdyż udowodnienie *even* 42 zajmuje jej mniej, niż 1 sekundę (przynajmniej na moim komputerze; na twoim taktyka ta może zawieść), ale już udowodnienie *even* 1338 trwa więcej niż jedną sekundę i wobec tego taktyka *timeout* 1 `repeat constructor` dla tego celu zawodzi (przynajmniej u mnie; jeżeli masz mocny komputer, u ciebie może zadziałać).

Co więcej, kombinator *timeout* może zachowywać się różnie dla tego samego celu nawet na tym samym komputerze. Na przykład przed chwilą taktyka ta zakończyłą się na moim komputerze sukcesem, mimo że dotychczas zawsze zawodziła).

```
Goal even 666.
Proof.
  time repeat constructor.
Restart.
  Time repeat constructor.
Abort.
```

Kolejnym kombinatorem jest *time t*, który odpala taktykę *t*, a następnie wyświetla informację o czasie, jaki zajęło jej wykonanie. Czas ten jest czasem rzeczywistym, tzn. zależy od mocy twojego komputera. Nie jest zbyt stały — zazwyczaj różni się od jednego mierzenia do drugiego, czasem nawet dość znacznie.

Alternatywą dla taktyki *time* jest komenda `Time`, która robi dokładnie to samo. Jeżeli stoisz przed wyborem między tymi dwoma — wybierz komendę `Time`, gdyż komendy zachowują się zazwyczaj w sposób znacznie bardziej przewidywalny od taktyk.

5.8 Konkluzja

W niniejszym rozdziale zapoznaliśmy się z potężną maszynerią, dzięki której możemy zjeść ciastko i mieć ciastko: dzięki własnym taktykom jesteśmy w stanie połączyć Coqową pełnię formalnej poprawności oraz typowy dla matematyki uprawianej nieformalnie luźny styl dowodzenia, w którym mało interesujące szczegóły zostają pominięte. A wszystko to okraszone (wystarczającą, mam nadzieję) szczyptą zadań.

Ale to jeszcze nie wszystko, gdyż póki co pominięte zostały konstrukty Ltaca pozwalające dopasowywać termy, dzięki którym jesteśmy w stanie np. napisać taktykę, która odróżni $2 + 2$ od 4. Jeżeli odczuwasz niedosyt po przeczytaniu tego rozdziału, to uszy do góry — zapoznamy się z nimi już niedługo, przy omawianiu dowodu przez reflekcję. Zanim to jednak nastąpi, zrobimy przegląd taktyk wbudowanych.

Rozdział 6

R4: Spis przydatnych taktyk

Stare powiedzenie głosi: nie wymyślaj koła na nowo. Aby uczynić zadość duchom przodków, którzy je wymyślili (zarówno koło, jak i powiedzenie), w niniejszym rozdziale zapoznamy się z różnymi przydatnymi taktykami, które prędzej czy później i tak sami byśmy wymyślili, gdyby zaszła taka potrzeba.

Aby jednak nie popaść w inny grzech i nie posługiwać się czarami, których nie rozumiemy, część z poniżej omówionych taktyk zaimplementujemy jako ćwiczenie.

Omówimy kolejno:

- taktykę `refine`
- drobne taktyki służące głównie do kontrolowania tego, co dzieje się w kontekście
- "średnie" taktyki, wcielające w życie pewien konkretny sposób rozumowania
- taktyki służące do rozumowania na temat relacji równoważności
- taktyki służące do przeprowadzania obliczeń
- procedury decyzyjne
- ogólne taktyki służące do automatyzacji

Uwaga: przykłady użycia taktyk, których reimplementacja będzie ćwiczeniem, zostały połączone z testami w ćwiczeniach żeby nie pisać dwa razy tego samego.

6.1 `refine` — matka wszystkich taktyk

Fama głosi, że w zamierzchłych czasach, gdy nie było jeszcze taktyk, a światem Coqa rządził Chaos (objawiający się dowodzeniem przez ręczne wpisywanie termów), jeden z Coqowych bogów imieniem He-fait-le-stos, w przebłysku kreatywnego geniuszu wymyślił dedukcję naturalną i stworzył pierwszą taktykę, której nadał imię `refine`. Pomysł przyjął się i od tej pory Coqowi bogowie poczęli używać jej do tworzenia coraz to innych taktyk. Tak `refine` stała się matką wszystkich taktyk.

Oczywiście legenda ta jest nieprawdziwa — deduckcję naturalną wymyślił Gerhard Gentzen, a podstawowe taktyki są zaimplementowane w Ocamlu. Nie umniejsza to jednak mocy taktyki `refine`. Jej działanie podobne jest do taktyki `exact`, z tym że term będący jej argumentem może też zawierać dziury _. Jeżeli naszym celem jest G, to taktyka `refine` g rozwiązuje cel, jeżeli g jest termem typu G, i generuje taką ilość podcelów, ile g zawiera dziur, albo zawodzi, jeżeli g nie jest typu G.

Zobaczmy działanie taktyki `refine` na przykładach.

```
Example refine_0 : 42 = 42.
Proof.
  refine eq_refl.
Qed.
```

W powyższym przykładzie używamy `refine` tak jak użylibyśmy `exact`a. *eq_refl* jest typu $42 = 42$, gdyż Coq domyśla się, że tak naprawdę chodzi nam o @*eq_refl nat* 42. Ponieważ *eq_refl* zawiera 0 dziur, `refine` *eq_refl* rozwiązuje cel i nie generuje podcelów.

```
Example refine_1 :
  ∀ P Q : Prop, P ∧ Q → Q ∧ P.
Proof.
  refine (fun P Q : Prop ⇒ _).
  refine (fun H ⇒ match H with | conj p q ⇒ _ end).
  refine (conj _ _ ).
    refine q.
    refine p.
Restart.
  intros P Q. intro H. destruct H as [p q]. split.
    exact q.
    exact p.
Qed.
```

W tym przykładzie chcemy pokazać przemienność konunkcji. Ponieważ nasz cel jest kwantyfikacją uniwersalną, jego dowodem musi być jakaś funkcja zależna. Funkcję tę konstruujemy taktyką `refine (fun` P Q : `Prop` ⇒ _). Nie podajemy jednak ciała funkcji, zastępując je dzurą _, bo chcemy podać je później. W związku z tym nasz obecny cel zostaje rozwiązany, a w zamian dostajemy nowy cel postaci $P \land Q \to Q \land P$, gdyż takiego typu jest ciało naszej funkcji. To jednak nie wszystko: w kontekście pojawiają się P Q : `Prop`. Wynika to z tego, że P i Q mogą zostać użyte w definicji ciała naszej funkcji.

Jako, że naszym celem jest implikacja, jej dowodem musi być funkcja. Taktyka `refine` (`fun` H ⇒ `match` H `with` | *conj p q* ⇒ _ `end`) pozwala nam tę funkcję skonstruować. Ciałem naszej funkcji jest dopasowanie zawierające dziurę. Wypełnienie jej będzie naszym kolejnym celem. Przy jego rozwiązywaniu będziemy mogli skorzystać z H, p i q. Pierwsza z tych hipotez pochodzi o wiązania `fun` H ⇒ ..., zaś p i q znajdą się w kontekście dzięki temu, że zostały związane podczas dopasowania $conj\ p\ q$.

Teraz naszym celem jest $Q \land P$. Ponieważ dowody koniunkcji są postaci $conj\ l\ r$, gdzie l jest dowodem pierwszego członu, a r drugiego, używamy taktyki `refine` ($conj$ _ _), by osobno skonstruować oba człony. Tym razem nasz proofterm zawiera dwie dziury, więc wygenerowane zostaną dwa podcele. Obydwa zachodzą na mocy założenia, a rozwiązujemy je także za pomocą `refine`.

Powyższy przykład pokazuje, że `refine` potrafi zastąpić cała gamę przeróżnych taktyk, które dotychczas uważaliśmy za podstawowe: `intros`, `intro`, `destruct`, `split` oraz `exact`. Określenie "matka wszystkich taktyk" wydaje się całkiem uzasadnione.

Ćwiczenie (my_exact) Napisz taktykę *my_exact*, która działa tak, jak `exact`. Użyj taktyki `refine`.

```
Example my_exact_0 :
  ∀ P : Prop, P → P.
Proof.
  intros. my_exact H.
Qed.
```

Ćwiczenie (my_intro) Zaimplementuj taktykę *my_intro1*, która działa tak, jak `intro`, czyli próbuje wprowadzić do kontekstu zmienną o domyślnej nazwie. Zaimplementuj też taktykę *my_intro2 x*, która działa tak jak `intro` x, czyli próbuje wprowadzić do kontekstu zmienną o nazwie x. Użyj taktyki `refine`.

Bonus: przeczytaj dokumentację na temat notacji dla taktyk (komenda `Tactic Notation`) i napisz taktykę *my_intro*, która działa tak jak *my_intro1*, gdy nie dostanie argumentu, a tak jak *my_intro2*, gdy dostanie argument.

```
Example my_intro_0 :
  ∀ P : Prop, P → P.
Proof.
  my_intro1. my_intro2 H. my_exact H.
Restart.
  my_intro. my_intro H. my_exact H.
Qed.
```

Ćwiczenie (my_apply) Napisz taktykę *my_apply H*, która działa tak jak `apply` H. Użyj taktyki `refine`.

```
Example my_apply_0 :
  ∀ P Q : Prop, P → (P → Q) → Q.
Proof.
  my_intro P. my_intro Q. my_intro p. my_intro H.
  my_apply H. my_exact p.
Qed.
```

Ćwiczenie (taktyki dla konstruktorów 1) Napisz taktyki:

- *my_split*, która działa tak samo jak `split`
- *my_left* i *my_right*, które działają tak jak `left` i `right`
- *my_exists*, która działa tak samo jak ∃

Użyj taktyki `refine`.

```
Example my_split_0 :
  ∀ P Q : Prop, P → Q → P ∧ Q.
Proof.
  my_intro P; my_intro Q; my_intro p; my_intro q.
  my_split.
    my_exact p.
    my_exact q.
Qed.
```

```
Example my_left_right_0 :
  ∀ P : Prop, P → P ∨ P.
Proof.
  my_intro P; my_intro p. my_left. my_exact p.
Restart.
  my_intro P; my_intro p. my_right. my_exact p.
Qed.
```

```
Example my_exists_0 :
  ∃ n : nat, n = 42.
Proof.
  my_exists 42. reflexivity.
Qed.
```

6.2 Drobne taktyki

6.2.1 `clear`

```
Goal
  ∀ x y : nat, x = y → y = x → False.
Proof.
  intros. clear H H0.
Restart.
  intros. Fail clear x. Fail clear wut.
Restart.
  intros. clear dependent x.
```

```
Restart.
  intros. clear.
Restart.
  intros.
  pose (z := 42).
  clearbody z.
Abort.
```

clear to niesamowicie użyteczna taktyka, dzięki której możemy zrobić porządek w kontekście. Można używać jej na nastepujące sposoby:

- clear x usuwa x z kontekstu. Jeżeli x nie ma w kontekście lub są w nim jakieś rzeczy zależne od x, taktyka zawodzi. Można usunąć wiele rzeczy na raz: clear *x_1* ... *x_N*.
- clear -x usuwa z kontekstu wszystko poza x.
- clear dependent x usuwa z kontekstu x i wszystkie rzeczy, które od niego zależą. Taktyka ta zawodzi jedynie gdy x nie ma w kontekście.
- clear usuwa z kontekstu absolutnie wszystko. Serdecznie nie polecam.
- *clearbody* x usuwa definicję x (jeżeli x jakąś posiada).

Ćwiczenie (tru) Napisz taktykę *tru*, która czyści kontekst z dowodów na *True* oraz potrafi udowodnić cel *True*.

Dla przykładu, taktyka ta powinna przekształcać kontekst $a, b, c : True, p : P \vdash \_$ w $p : P \vdash \_$.

```
Section tru.

Example tru_0 :
  ∀ P : Prop, True → True → True → P.
Proof.
  tru. (* Kontekst: P : Prop ⊢ P *)
Abort.

Example tru_1 : True.
Proof. tru. Qed.

End tru.
```

Ćwiczenie (satans_neighbour_not_even) Inductive *even* : *nat* → Prop :=
  | *even0* : *even* 0
  | *evenSS* : ∀ *n* : *nat*, *even* *n* → *even* (*S* (*S* *n*)).

Napisz taktykę *even*, która potrafi udowodnić poniższy cel.

```
Theorem satans_neighbour_not_even : ¬ even 667.
```

Ćwiczenie (my_destruct_and) Napisz taktykę *my_destruct H p q*, która działa jak `destruct` H `as` $[p\ q]$, gdzie H jest dowodem koniunkcji. Użyj taktyk `refine` i `clear`.

Bonus 1: zaimplementuj taktykę *my_destruct_and H*, która działa tak jak `destruct` H, gdy H jest dowodem koniunkcji.

Bonus 2: zastanów się, jak (albo czy) można zaimplementować taktykę `destruct` x, gdzie x jest dowolnego typu induktywnego.

```
Example my_destruct_and_0 :
  ∀ P Q : Prop, P ∧ Q → P.
Proof.
  my_intro P; my_intro Q; my_intro H.
  my_destruct_and H p q. my_exact p.
Restart.
  my_intro P; my_intro Q; my_intro H.
  my_destruct_and H. my_exact H0.
Qed.
```

6.2.2 `fold`

`fold` to taktyka służąca do zwijania definicji. Jej działanie jest odwrotne do działania taktyki `unfold`. Niestety, z nieznanych mi bliżej powodów bardzo często jest ona nieskuteczna.

Ćwiczenie (my_fold) Napisz taktykę *my_fold x*, która działa tak jak `fold` x, tj. zastępuje we wszystkich miejscach w celu term powstały po rozwinięciu x przez x.

Wskazówka: zapoznaj się z konstruktem `eval` — zajrzyj do 9 rozdziału manuala.

```
Example fold_0 :
  ∀ n m : nat, n + m = m + n.
Proof.
  intros. unfold plus. fold plus.
Restart.
  intros. unfold plus. my_fold plus.
Abort.
```

6.2.3 `move`

```
Example move_0 :
  ∀ P Q R S T : Prop, P ∧ Q ∧ R ∧ S ∧ T → T.
Proof.
  destruct 1 as [p [q [r [s t]]]].
  move p after t.
  move p before s.
  move p at top.
```

move p at *bottom*.
Abort.

move to taktyka służąca do zmieniania kolejności obiektów w kontekście. Jej działanie jest tak ewidentnie oczywiste, ż nie ma zbytniego sensu, aby je opisywać.

Ćwiczenie Przeczytaj dokładny opis działania taktyki move w manualu.

6.2.4 pose i *remember*

```
Goal 2 + 2 = 4.
Proof.
  intros.
  pose (a := 2 + 2).
  remember (2 + 2) as b.
Abort.
```

Taktyka pose ($x := t$) dodaje do kontekstu zmienną x (pod warunkiem, że nazwa ta nie jest zajęta), która zostaje zdefiniowana za pomocą termu t.

Taktyka *remember* t as x zastępuje wszystkie wystąpienia termu t w kontekście zmienną x (pod warunkiem, że nazwa ta nie jest zajęta) i dodaje do kontekstu równanie postaci $x = t$.

W powyższym przykładzie działają one następująco: pose ($a := 2 + 2$) dodaje do kontekstu wiązanie $a := 2 + 2$, zaś *remember* $(2 + 2)$ as b dodaje do kontekstu równanie *Heqb* : $b = 2 + 2$ i zastępuje przez b wszystkie wystąpienia $2 + 2$ — także to w definicji a.

Taktyki te przydają się w tak wielu różnych sytuacjach, że nie ma co próbować ich tu wymieniać. Użyjesz ich jeszcze nie raz.

Ćwiczenie (set) Taktyki te są jedynie wariantami bardziej ogólnej taktyki set. Przeczytaj jej dokumentację w manualu.

6.2.5 rename

```
Goal ∀ P : Prop, P → P.
Proof.
  intros. rename H into wut.
Abort.
```

rename x *into* y zmienia nazwę x na y lub zawodzi, gdy x nie ma w kontekście albo nazwa y jest już zajęta

Ćwiczenie (satans_neighbour_not_even') Napisz taktykę *even'*, która potrafi udowodnić poniższy cel. Nie używaj matcha, a jedynie kombinatora repeat.

```
Theorem satans_neighbour_not_even' : ¬ even 667.
```

6.2.6 *admit*

```
Module admit.

Lemma forgery :
  ∀ P Q : Prop, P → Q ∧ P.
Proof.
  intros. split.
    admit.
    assumption.
Admitted.

Print forgery.
(* ===> *** [ forgery : ∀ P : Prop, P → ¬ P ∧ P ] *)

End admit.
```

admit to taktyka-oszustwo, która rozwiązuje dowolny cel. Nie jest ona rzecz jasna wszechwiedząca i przez to rozwiązanego za jej pomocą celu nie można zapisać za pomocą komend `Qed` ani `Defined`, a jedynie za pomocą komendy *Admitted*, która oszukańczo udowodnione twierdzenie przekształca w aksjomat.

W CoqIDE oszustwo jest dobrze widoczne, gdyż zarówno taktyka *admit* jak i komenda *Admitted* podświetlają się na żółto, a nie na zielono, tak jak prawdziwe dowody. Wyświetlenie `Print`em dowodu zakończonego komendą *Admitted* również pokazuje, że ma on status aksjomatu.

Na koniec zauważmy, że komendy *Admitted* możemy użyć również bez wcześniejszego użycia taktyki *admit*. Różnica między tymi dwoma bytami jest taka, że taktyka *admit* służy do "udowodnienia" pojedynczego celu, a komenda *Admitted* — całego twierdzenia.

6.3 Średnie taktyki

6.3.1 *case_eq*

case_eq to taktyka podobna do taktyki `destruct`, ale nieco mądrzejsza, gdyż nie zdarza jej się "zapominać", jaka była struktura rozbitego przez nią termu.

```
Goal
  ∀ n : nat, n + n = 42.
Proof.
  intros. destruct (n + n).
Restart.
  intros. case_eq (n + n); intro.
Abort.
```

Różnice między `destruct` i *case_eq* dobrze ilustruje powyższy przykład. `destruct` nadaje się jedynie do rozbijania termów, które są zmiennymi. Jeżeli rozbijemy coś, co nie jest zmienną (np. term $n + n$), to utracimy część informacji na jego temat. *case_eq* potrafi rozbijać dowolne termy, gdyż poza samym rozbiciem dodaje też do celu dodatkową hipotezę, która zawiera równanie "pamiętające" informacje o rozbitym termie, o których zwykły `destruct` zapomina.

Ćwiczenie (my_case_eq) Napisz taktykę *my_case_eq t Heq*, która działa tak jak *case_eq t*, ale nie dodaje równania jako hipotezę na początku celu, tylko bezpośrednio do kontekstu i nazywa je *Heq*. Użyj taktyk *remember* oraz `destruct`.

```
Goal
  ∀ n : nat, n + n = 42.
Proof.
  intros. destruct (n + n).
Restart.
  intros. case_eq (n + n); intro.
Restart.
  intros. my_case_eq (n + n) H.
Abort.
```

6.3.2 *contradiction*

contradiction to taktyka, która wprowadza do kontekstu wszystko co się da, a potem próbuje znaleźć sprzeczność. Potrafi rozpoznawać hipotezy takie jak *False*, $x \neq x$, $\neg$ *True*. Potrafi też znaleźć dwie hipotezy, które są ze sobą ewidentnie sprzeczne, np. P oraz $\neg P$. Nie potrafi jednak wykrywać lepiej ukrytych sprzeczności, np. nie jest w stanie odróżnić *true* od *false*.

Ćwiczenie (my_contradiction) Napisz taktykę *my_contradiction*, która działa tak jak standardowa taktyka *contradiction*, a do tego jest w stanie udowodnić dowolny cel, jeżeli w kontekście jest hipoteza postaci *true* = *false* lub *false* = *true*.

```
Section my_contradiction.

Example my_contradiction_0 :
  ∀ P : Prop, False → P.
Proof.
  contradiction.
Restart.
  my_contradiction.
Qed.

Example my_contradiction_1 :
  ∀ P : Prop, ¬ True → P.
Proof.
  contradiction.
Restart.
  my_contradiction.
```

```
Qed.

Example my_contradiction_2 :
  ∀ (P : Prop) (n : nat), n ≠ n → P.
Proof.
  contradiction.
Restart.
  my_contradiction.
Qed.

Example my_contradiction_3 :
  ∀ P Q : Prop, P → ¬ P → Q.
Proof.
  contradiction.
Restart.
  my_contradiction.
Qed.

Example my_contradiction_4 :
  ∀ P : Prop, true = false → P.
Proof.
  try contradiction.
Restart.
  my_contradiction.
Qed.

Example my_contradiction_5 :
  ∀ P : Prop, false = true → P.
Proof.
  try contradiction.
Restart.
  my_contradiction.
Qed.

End my_contradiction.
```

Ćwiczenie (taktyki dla sprzeczności) Innymi taktykami, które mogą przydać się przy rozumowaniach przez sprowadzenie do sprzeczności, są *absurd*, *contradict* i *exfalso*. Przeczytaj ich opisy w manualu i zbadaj ich działanie.

6.3.3 constructor

```
Example constructor_0 :
  ∀ P Q : Prop, P → Q ∨ P.
Proof.
  intros. constructor 2. assumption.
```

```
Restart.
  intros. constructor.
Restart.
  intros. constructor; assumption.
Qed.
```

`constructor` to taktyka ułatwiająca aplikowanie konstruktorów typów induktywnych. Jeżeli aktualnym celem jest *T*, to taktyka `constructor` *i* jest równoważna wywołaniu jego i-tego konstruktora, gdzie porządek konstruktorów jest taki jak w definicji typu.

```
Print or.
(*  ===> Inductive or (A B : Prop) : Prop :=
           or_introl : A -> A \/ B | or_intror : B -> A \/ B *)
```

W powyższym przykładzie `constructor` 2 działa tak jak `apply` *or_intror* (czyli tak samo jak taktyka `right`), gdyż w definicji spójnika *or* konstruktor *or_intror* występuje jako drugi (licząc od góry).

Użycie taktyki `constructor` bez liczby oznacza zaaplikowanie pierwszego konstruktora, który pasuje do celu, przy czym taktyka ta może wyzwalać backtracking. W drugim przykładzie powyżej `constructor` działa jak `apply` *or_intro* (czyli jak taktyka `left`), gdyż zaaplikowanie tego konstruktora nie zawodzi.

W trzecim przykładzie `constructor`; `assumption` działa tak: najpierw aplikowany jest konstruktor *or_introl*, ale wtedy `assumption` zawodzi, więc następuje nawrót i aplikowany jest konstruktor *or_intror*, a wtedy `assumption` rozwiązuje cel.

Ćwiczenie (taktyki dla konstruktorów 2) Jaki jest związek taktyki `constructor` z taktykami `split`, `left`, `right` i ∃?

6.3.4 *decompose*

```
Example decompose_0 :
  ∀ P Q R S : nat → Prop,
    (∃ n : nat, P n) ∧ (∃ n : nat, Q n) ∧
    (∃ n : nat, R n) ∧ (∃ n : nat, S n) →
      ∃ n : nat, P n ∨ Q n ∨ R n ∨ S n.
Proof.
  intros. decompose [and ex] H. clear H. ∃ x. left. assumption.
Qed.
```

decompose to bardzo użyteczna taktyka, która potrafi za jednym zamachem rozbić bardzo skomplikowane hipotezy. *decompose* [*t_1* ... *t_n*] *H* rozbija rekurencyjnie hipotezę *H* tak długo, jak jej typem jest jeden z typów *t_i*. W powyższym przykładzie *decompose* [*and ex*] *H* najpierw rozbija *H*, gdyż jest ona koniunkcją, a następnie rozbija powstałe z niej hipotezy, gdyż są one kwantyfikacjami egzystencjalnymi ("exists" jest notacją dla *ex*). *decompose* nie usuwa z kontekstu hipotezy, na której działa, więc często następuje po niej taktyka `clear`.

6.3.5 intros

Dotychczas używałeś taktyk `intro` i `intros` jedynie z nazwami lub wzorcami do rozbijania elementów typów induktywnych. Taktyki te potrafią jednak dużo więcej.

```
Example intros_0 :
  ∀ P Q R S : Prop, P ∧ Q ∧ R → S.
Proof.
  intros P Q R S [p [q r]].
Restart.
  intros ? ?P Q R. intros (p, (p0, q)).
Restart.
  intros ×.
Restart.
  intros A B **.
Restart.
  intros × _.
Restart.
  Fail intros _.
Abort.
```

Pierwszy przykład to standardowe użycie `intros` — wprowadzamy cztery zmienne, która nazywamy kolejno P, Q, R i S, po czym wprowadzamy bezimienną hipotezę typu $P \land Q \land R$, która natychmiast rozbijamy za pomocą wzorca p $[q\ r]$.

W kolejnym przykładzie mamy już nowości: wzorzec ? służy do nadania zmiennej domyślnej nazwy. W naszym przypadku wprowadzone do kontekstu zdanie zostaje nazwane P, gdyż taką nazwę nosi w kwantyfikatorze, gdy jest jeszcze w celu.

Wzorzec $?P$ służy do nadania zmiennej domyślnej nazwy zaczynając się od tego, co następuje po znaku ?. W naszym przypadku do konteksu wprowadzona zostaje zmienna $P0$, gdyż żądamy nazwy zaczynającej się od "P", ale samo "P" jest już zajęte. Widzimy też wzorzec $(p,\ (p0,\ q))$, który służy do rozbicia hipotezy. Wzorce tego rodzaju działają tak samo jak wzorce w kwadratowych nawiasach, ale możemy używać ich tylko na elementach typu induktywnego z jednym konstruktorem.

Wzorzec × wprowadza do kontekstu wszystkie zmienne kwantyfikowane uniwersalnie i zatrzymuje sie na pierwszej nie-zależnej hipotezie. W naszym przykładzie uniwersalnie kwantyfikowane są P, Q, R i S, więc zostają wprowadzane, ale $P \land Q \land R$ nie jest już kwantyfikowane uniwersalnie — jest przesłanką implikacji — więc nie zostaje wprowadzone.

Wzorzec ** wprowadza do kontekstu wszystko. Wobec tego `intros **` jest synonimem `intros`. Mimo tego nie jest on bezużyteczny — możemy użyć go po innych wzorcach, kiedy nie chcemy już więcej nazywać/rozbijać naszych zmiennych. Wtedy dużo szybciej napisać ** niż ; `intros`. W naszym przypadku chcemy nazwać jedynie pierwsze dwie zmienne, a resztę wrzucamy do kontekstu jak leci.

Wzorzec _ pozwala pozbyć się zmiennej lub hipotezy. Taktyka `intros _` jest wobec tego równoważna `intro` H; `clear` H (przy założeniu, że H jest wolne), ale dużo bardziej zwięzła w zapisie. Nie możemy jednak usunąć zmiennych lub hipotez, od których zależą inne zmienne lub hipotezy. W naszym przedostatnim przykładzie bez problemu usuwamy hipotezę $P \land Q \land R$, gdyż żaden term od niej nie zależy. Jednak w ostatnim przykładzie nie możemy usunąć P, gdyż zależy od niego hipoteza $P \land Q \land R$.

```
Example intros_1 :
  ∀ P0 P1 P2 P3 P4 P5 : Prop,
    P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4 ∧ P5 → P3.
Proof.
  intros × [p0 [p1 [p2 [p3 [p4 p5]]]]].
Restart.
  intros × (p0 & p1 & p2 & p3 & p4 & p5).
Abort.
```

Wzorce postaci (*p_1* & ... & *p_n*) pozwalają rozbijać termy zagnieżdżonych typów induktywnych. Jak widać na przykładzie, im bardziej zagnieżdżony jest typ, tym bardziej opłaca się użyć tego rodzaju wzorca.

```
Example intros_2 :
  ∀ x y : nat, x = y → y = x.
Proof.
  intros × →.
Restart.
  intros × ←.
Abort.
```

Wzorców → oraz ← możemy użyć, gdy chcemy wprowadzić do kontekstu równanie, przepisać je i natychmiast się go pozbyć. Wobec tego taktyka `intros` → jest równoważna czemuś w stylu `intro` *H*; `rewrite` *H* `in` *; `clear` *H* (oczywiście pod warunkiem, że nazwa *H* nie jest zajęta).

```
Example intros_3 :
  ∀ a b c d : nat, (a, b) = (c, d) → a = c.
Proof.
  Fail intros × [p1 p2].
Restart.
  intros × [= p1 p2].
Abort.
```

Wzorzec postaci = *p_1* ... *p_n* pozwala rozbić równanie między parami (i nie tylko) na składowe. W naszym przypadu mamy równanie $(a, b) = (c, d)$ — zauważmy, że nie jest ono koniunkcją dwóch równości $a = c$ oraz $b = d$, co jasno widać na przykładzie, ale można z niego ową koniunkcję wywnioskować. Taki właśnie efekt ma wzorzec = *p1* *p2* — dodaje on nam do kontekstu hipotezy *p1* : $a = c$ oraz *p2* : $b = d$.

```
Example intros_4 :
  ∀ P Q R : Prop, (P → Q) → (Q → R) → P → R.
Proof.
```

```
  intros until 2. intro p. apply H in p. apply H0 in p.
Restart.
  intros until 2. intros p %H %H0.
Abort.
```

Taktyka `intros until` x wprowadza do kontekstu wszystkie zmienne jak leci dopóki nie natknie się na taką, która nazywa się "x". Taktyka `intros until` n, gdzie n jest liczbą, wprowadza do kontekstu wszyskto jak leci aż do n-tej nie-zależnej hipotezy (tj. przesłanki implikacji). W naszym przykładzie mamy 3 przesłanki implikacji: $(P \to Q)$, $(Q \to R)$ i P, więc taktyka `intros until` 2 wprowadza do kontekstu dwie pierwsze z nich oraz wszystko, co jest poprzedza.

Wzorzec x %*H_1* ... %*H_n* wprowadza do kontekstu zmienną x, a następnie aplikuje do niej po kolei hipotezy *H_1*, ..., *H_n*. Taki sam efekt można osiągnąć ręcznie za pomocą taktyki `intro` x; `apply` *H_1* `in` x; ... `apply` *H_n* `in` x.

Ćwiczenie (intros) Taktyka `intros` ma jeszcze trochę różnych wariantów. Poczytaj o nich w manualu.

6.3.6 `fix`

`fix` to taktyka służąca do dowodzenia bezpośrednio przez rekursję. W związku z tym nadeszła dobra pora, żeby pokazać wszystkie możliwe sposoby na użycie rekursji w Coqu. Żeby dużo nie pisać, przyjrzyjmy się przykładom: zdefiniujemy/udowodnimy regułę indukcyjną dla liczb naturalnych, którą powinieneś znać jak własną kieszeń (a jeżeli nie, to marsz robić zadania z liczb naturalnych!).

```
Definition nat_ind_fix_term
  (P : nat → Prop) (H0 : P 0)
  (HS : ∀ n : nat, P n → P (S n))
    : ∀ n : nat, P n :=
      fix f (n : nat) : P n :=
      match n with
          | 0 ⇒ H0
          | S n' ⇒ HS n' (f n')
      end.
```

Pierwszy, najbardziej prymitywny sposób to użycie konstruktu `fix`. `fix` to podstawowy budulec Coqowej rekursji, ale ma tę wadę, że trzeba się trochę napisać: w powyższym przykładzie najpierw piszemy ∀ n : *nat*, P n, a następnie powtarzamy niemal to samo, pisząc `fix` f (n : *nat*) : P n.

```
Fixpoint nat_ind_Fixpoint_term
  (P : nat → Prop) (H0 : P 0)
  (HS : ∀ n : nat, P n → P (S n))
  (n : nat) : P n :=
```

```
match n with
    | 0 ⇒ H0
    | S n' ⇒ HS n' (nat_ind_Fixpoint_term P H0 HS n')
end.
```

Rozwiązaniem powyższej robnej niedogodności jest komenda `Fixpoint`, która jest skrótem dla `fix`. Oszczędza nam ona pisania dwa razy tego samego, dzięki czemu definicja jest o linijkę krótsza.

```
Fixpoint nat_ind_Fixpoint_tac
  (P : nat → Prop) (H0 : P 0)
  (HS : ∀ n : nat, P n → P (S n))
  (n : nat) : P n.
Proof.
  apply nat_ind_Fixpoint_tac; assumption.
  Fail Guarded.
  (*  ===> Długi komunikat o błędzie. *)
   Show Proof.
  (*  ===> (fix nat_ind_Fixpoint_tac
                 (P : nat -> Prop) (H0 : P 0)
                 (HS : forall n : nat, P n -> P (S n))
                 (n : nat) {struct n} : P n :=
                   nat_ind_Fixpoint_tac P H0 HS n) *)
Restart.
  destruct n as [| n'].
    apply H0.
    apply HS. apply nat_ind_Fixpoint_tac; assumption.
  Guarded.
  (*  ===> The condition holds up to here *)
Defined.
```

W trzecim podejściu również używamy komendy `Fixpoint`, ale tym razem, zamiast ręcznie wpisywać term, definiujemy naszą regułę za pomocą taktyk. Sposób ten jest prawie zawsze (dużo) dłuższy niż poprzedni, ale jego zaletą jest to, że przy skomplikowanych celach jest dużo ławiejszy do ogarnięcia dla człowieka.

Korzystając z okazji rzućmy okiem na komendę `Guarded`. Jest ona przydatna gdy, tak jak wyżej, dowodzimy lub definiujemy bezpośrednio przez rekursję. Sprawdza ona, czy wszystkie dotychczasowe wywołania rekurencyjne odbyły się na strukturalnie mniejszych podtermach. Jeżeli nie, wyświetla ona wiadomość, która informuje nas, gdzie jest błąd. Niestety wiadomości te nie zawsze są czytelne.

Tak właśnie jest, gdy w powyższym przykładzie używamy jej po raz pierwszy. Na szczęście ratuje nas komenda `Show Proof`, która pokazuje, jak wygląda term, która póki co wygenerowały taktyki. Pokazuje on nam term postaci *nat_ind_Fixpoint_tac P H0 HS n* := *nat_ind_Fixpoint_tac P H0 HS n*, który próbuje wywołać się rekurencyjnie na tym samym argumencie, na którym sam został wywołany. Nie jest więc legalny.

Jeżeli z wywołaniami rekurencyjnymi jest wszystko ok, to komenda `Guarded` wyświetla przyjazny komunikat. Tak właśnie jest, gdy używamy jej po raz drugi — tym razem wywołanie rekurencyjne odbywa się na *n'*, które jest podtermem *n*.

```
Definition nat_ind_fix_tac :
  ∀ (P : nat → Prop) (H0 : P 0)
    (HS : ∀ n : nat, P n → P (S n)) (n : nat), P n.
Proof.
  Show Proof.
  (*  ===> ?Goal *)
  fix IH 4.
  Show Proof.
  (*  ===> (fix nat_ind_fix_tac
               (P : nat -> Prop) (H0 : P 0)
               (HS : forall n : nat, P n -> P (S n))
               (n : nat) {struct n} : P n := ... *)
  destruct n as [| n'].
    apply H0.
    apply HS. apply IH; assumption.
Defined.
```

Taktyki `fix` możemy użyć w dowolnym momencie, aby rozpocząć dowodzenie/ definiowanie bezpośrednio przez rekursję. Jej argumentami są nazwa, którą chcemy nadać hipotezie indukcyjnej oraz numer argument głównego. W powyższym przykładzie chcemy robić rekursję po *n*, który jest czwarty z kolei (po *P*, *H0* i *HS*).

Komenda `Show Proof` pozwala nam odkryć, że użycie taktyki `fix` w trybie dowodzenia odpowiada po prostu użyciu konstruktu `fix` lub komendy `Fixpoint`.

Taktyka `fix` jest bardzo prymitywna i prawie nigdy nie jest używana, tak samo jak konstrukt `fix` (najbardziej poręczne są sposoby, które widzieliśmy w przykladach 2 i 3), ale była dobrym pretekstem, żeby omówić wszystkie sposoby użycia rekursji w jednym miejscu.

6.3.7 *functional* `induction` i *functional* `inversion`

Taktyki *functional* `induction` i *functional* `inversion` są związane z pojęciem indukcji funkcyjnej. Dość szczegółowy opis tej pierwszej jest w notatkach na seminarium: https://zeimer.github.io/Semin

Drugą z nich póki co pominiemy. Kiedyś z pewnością napiszę coś więcej o indukcji funkcyjnej lub chociaż przetłumaczę zalinkowane notatki na polski.

6.3.8 `generalize dependent`

`generalize dependent` to taktyka będąca przeciwieństwem `intro` — dzięki niej możemy przerzucić rzeczy znajdujące się w kontekście z powrotem do kontekstu. Nieformalnie odpowiada ona sposobowi rozumowania: aby pokazać, że cel zachodzi dla pewnego konkretnego x, wystarczy czy pokazać, że zachodzi dla dowolnego x.

W rozumowaniu tym z twierdzenia bardziej ogólnego wyciągamy wniosek, że zachodzi twierdzenie bardziej szczegółowe. Nazwa `generalize` bierze się stąd, że w dedukcji naturalnej nasze rozumowania przeprowadzamy "od tyłu". Człon "dependent" bierze się stąd, że żeby zgeneralizować x, musimy najpierw zgeneralizować wszystkie obiekty, które są od niego zależne. Na szczęście taktyka `generalize dependent` robi to za nas.

```
Example generalize_dependent_0 :
  ∀ n m : nat, n = m → m = n.
Proof.
  intros. generalize dependent n.
Abort.
```

Użycie `intros` wprowadza do kontekstu n, m i H. `generalize dependent` n przenosi n z powrotem do celu, ale wymaga to, aby do celu przenieść również H, gdyż typ H, czyli $n = m$, zależy od n.

Ćwiczenie (generalize i revert) `generalize dependent` jest wariantem taktyki `generalize`. Taktyką o niemal identycznym działaniu jest *revert* `dependent`, wariant taktyki *revert*. Przeczytaj dokumentację `generalize` i *revert* w manualu i sprawdź, jak działają.

Ćwiczenie (my_rec) Zaimplementuj taktykę `rec` x, która będzie pomagała przy dowodzeniu bezpośrednio przez rekursję po x. Taktyka `rec` x ma działać jak `fix` *IH* n; `destruct` x, gdzie n to pozycja argumentu x w celu. Twoja taktyka powinna działać tak, żeby poniższy dowód zadziałał bez potrzeby wprowadzania modyfikacji.

Wskazówka: połącz taktyki `fix`, `intros`, `generalize dependent` i `destruct`.

```
Lemma plus_comm_rec :
  ∀ n : nat, n + 1 = S n.
Proof.
  rec n.
    reflexivity.
    cbn. f_equal. rewrite IH. reflexivity.
Qed.
```

6.4 Taktyki dla równości i równoważności

6.4.1 reflexivity, symmetry i transitivity

```
Require Import Arith.

Example reflexivity_0 :
  ∀ n : nat, n ≤ n.
Proof. reflexivity. Qed.
```

Znasz już taktykę `reflexivity`. Mogłoby się wydawać, że służy ona do udowadniania celów postaci $x = x$ i jest w zasadzie równoważna taktyce `apply` *eq_refl*, ale nie jest tak. Taktyka `reflexivity` potrafi rozwiązać każdy cel postaci $R\ x\ y$, gdzie R jest relacją zwrotną, a x i y są konwertowalne (oczywiście pod warunkiem, że udowodnimy wcześniej, że R faktycznie jest zwrotna; w powyższym przykładzie odpowiedni fakt został zaimportowany z modułu *Arith*).

Żeby zilustrować ten fakt, zdefiniujmy nową relację zwrotną i zobaczmy, jak użyć taktyki `reflexivity` do radzenia sobie z nią.

```
Definition eq_ext {A B : Type} (f g : A → B) : Prop :=
  ∀ x : A, f x = g x.
```

W tym celu definiujemy relację *eq_ext*, która głosi, że funkcja $f : A \to B$ jest w relacji z funkcją $g : A \to B$, jeżeli $f\ x$ jest równe $g\ x$ dla dowolnego $x : A$.

```
Require Import RelationClasses.
```

Moduł *RelationClasses* zawiera definicję zwrotności *Reflexive*, z której korzysta taktyka `reflexivity`. Jeżeli udowodnimy odpowiednie twierdzenie, będziemy mogli używać taktyki `reflexivity` z relacją *eq_ext*.

```
Instance Reflexive_eq_ext :
  ∀ A B : Type, Reflexive (@eq_ext A B).
Proof.
  unfold Reflexive, eq_ext. intros A B f x. reflexivity.
Defined.
```

A oto i rzeczone twierdzenie oraz jego dowód. Zauważmy, że taktyki `reflexivity` nie używamy tutaj z relacją *eq_ext*, a z relacją $=$, gdyż używamy jej na celu postaci $f\ x = f\ x$.

Uwaga: żeby taktyka `reflexivity` "widziała" ten dowód, musimy skorzystać ze słowa kluczowego `Instance` zamiast z `Theorem` lub `Lemma`.

```
Example reflexivity_1 :
  eq_ext (fun _ : nat ⇒ 42) (fun _ : nat ⇒ 21 + 21).
Proof. reflexivity. Defined.
```

Voilà! Od teraz możemy używać taktyki `reflexivity` z relacją *eq_ext*.

Są jeszcze dwie taktyki, które czasem przydają się przy dowodzeniu równości (oraz równoważności).

```
Example symmetry_transitivity_0 :
  ∀ (A : Type) (x y z : nat), x = y → y = z → z = x.
Proof.
  intros. symmetry. transitivity y.
    assumption.
    assumption.
Qed.
```

Mogłoby się wydawać, że taktyka `symmetry` zamienia cel postaci $x = y$ na $y = x$, zaś taktyka `transitivity` y rozwiązuje cel postaci $x = z$ i generuje w zamian dwa cele postaci $x = y$ i $y = z$. Rzeczywistość jest jednak bardziej hojna: podobnie jak w przypadku `reflexivity`, taktyki te działają z dowolnymi relacjami symetrycznymi i przechodnimi.

```
Instance Symmetric_eq_ext :
  ∀ A B : Type, Symmetric (@eq_ext A B).
Proof.
  unfold Symmetric, eq_ext. intros A B f g H x. symmetry. apply H.
Defined.

Instance Transitive_eq_ext :
  ∀ A B : Type, Transitive (@eq_ext A B).
Proof.
  unfold Transitive, eq_ext. intros A B f g h H H' x.
  transitivity (g x); [apply H | apply H'].
Defined.
```

Użycie w dowodach taktyk `symmetry` i `transitivity` jest legalne, gdyż nie używamy ich z relacją *eq_ext*, a z relacją $=$.

```
Example symmetry_transitivity_1 :
  ∀ (A B : Type) (f g h : A → B),
    eq_ext f g → eq_ext g h → eq_ext h f.
Proof.
  intros. symmetry. transitivity g.
    assumption.
    assumption.
Qed.
```

Dzięki powyższym twierdzeniom możemy teraz posługiwać się taktykami `symmetry` i `transitivity` dowodząc faktów na temat relacji *eq_ext*. To jednak wciąż nie wyczerpuje naszego arsenału taktyk do radzenia sobie z relacjami równoważności.

6.4.2 f_equal

```
Check f_equal.
(*  ===> f_equal : forall (A B : Type) (f : A -> B) (x y : A),
                       x = y -> f x = f y *)
```

`f_equal` to jedna z podstawowych właściwości relacji *eq*, która głosi, że wszystkie funkcje zachowują równość. Innymi słowy: aby pokazać, że wartości zwracane przez funkcję są równe, wystarczy pokazać, że argumenty są równe. Ten sposób rozumowania, choć nie jest ani jedyny, ani skuteczny na wszystkie cele postaci $f\ x = f\ y$, jest wystarczająco częsty, aby mieć swoją własną taktykę, którą zresztą powinieneś już dobrze znać — jest nią `f_equal`.

Taktyka ta sprowadza się w zasadzie do jak najsprytniejszego aplikowania faktu `f_equal`. Nie potrafi ona wprowadzać zmiennych do kontekstu, a z wygenerowanych przez siebie podcelów rozwiązuje jedynie te postaci $x = x$, ale nie potrafi rozwiązać tych, które zachodzą na mocy założenia.

Ćwiczenie (my_f_equal) Napisz taktykę *my_f_equal*, która działa jak `f_equal` na sterydach, tj. poza standardową funkcjonalnością `f_equal` potrafi też wprowadzać zmienne do kontekstu oraz rozwiązywać cele prawdziwe na mocy założenia.

Użyj tylko jednej klauzuli `match`a. Nie używaj taktyki `subst`. Bonus: wykorzystaj kombinator `first`, ale nie wciskaj go na siłę. Z czego łatwiej jest skorzystać: rekursji czy iteracji?

```
Example f_equal_0 :
  ∀ (A : Type) (x : A), x = x.
Proof.
  intros. f_equal.
  (*  Nie działa, bo x = x nie jest podcelem
     wygenerowanym przez f_equal. *)
Restart.
  my_f_equal.
Qed.

Example f_equal_1 :
  ∀ (A : Type) (x y : A), x = y → x = y.
Proof.
  intros. f_equal.
Restart.
  my_f_equal.
Qed.

Example f_equal_2 :
  ∀ (A B C D E : Type) (f f' : A → B → C → D → E)
     (a a' : A) (b b' : B) (c c' : C) (d d' : D),
       f = f' → a = a' → b = b' → c = c' → d = d' →
          f a b c d = f' a' b' c' d'.
Proof.
  intros. f_equal. all: assumption.
Restart.
  my_f_equal.
Qed.
```

Ćwiczenie (właściwości `f_equal`) Przyjrzyj się definicjom `f_equal`, *id*, *compose*, *eq_sym*, *eq_trans*, a następnie udowodnij poniższe lematy. Ich sens na razie niech pozostanie ukryty — kiedyś być może napiszę coś na ten temat. Jeżeli intrygują cię one, przyjrzyj się książce https://homotopytypetheory.org/book/

```
Require Import Coq.Program.Basics.

Print f_equal.
Print eq_sym.
Print eq_trans.
```

```
Print compose.

Section f_equal_properties.

Variables
  (A B C : Type)
  (f : A -> B) (g : B -> C)
  (x y z : A)
  (p : x = y) (q : y = z).

Lemma f_equal_refl :
  f_equal f (eq_refl x) = eq_refl (f x).

Lemma f_equal_id :
  f_equal id p = p.

Lemma f_equal_compose :
  f_equal g (f_equal f p) = f_equal (compose g f) p.

Lemma eq_sym_map_distr :
  f_equal f (eq_sym p) = eq_sym (f_equal f p).

Lemma eq_trans_map_distr :
  f_equal f (eq_trans p q) = eq_trans (f_equal f p) (f_equal f q).

End f_equal_properties.
```

Ostatnią taktyką, którą poznamy w tym podrozdziale, jest *f_equiv*, czyli pewne uogólnienie taktyki `f_equal`. Niech nie zmyli cię nazwa tej taktyki — bynajmniej nie przydaje się ona jedynie do rozumowań dotyczących relacji równoważności.

```
Require Import Classes.Morphisms.
```

Aby móc używać tej taktyki, musimy najpierw zaimportować moduł *Classes.Morphisms*.

```
Definition len_eq {A : Type} (l1 l2 : list A) : Prop :=
  length l1 = length l2.
```

W naszym przykładzie posłużymy się relacją *len_eq*, która głosi, że dwie listy są w relacji gdy mają taką samą długość.

```
Instance Proper_len_eq_map {A : Type} :
  Proper (@len_eq A ==> @len_eq A ==> @len_eq A) (@app A).
Proof.
  Locate "==>".
  unfold Proper, respectful, len_eq.
  induction x as [| x xs]; destruct y; inversion 1; cbn; intros.
    assumption.
    f_equal. apply IHxs; assumption.
Qed.
```

Taktyka `f_equal` działa na celach postaci $f\ x = f\ y$, gdzie f jest dowolne, albowiem wszystkie funkcje zachowują równość. Analogicznie taktyka *f_equiv* działa na celach postaci

$R\ (f\ x)\ (f\ y)$, gdzie R jest dowolną relacją, ale tylko pod warunkiem, że funkcja f zachowuje relację R.

Musi tak być, bo gdyby f nie zachowywała R, to mogłoby jednocześnie zachodzić $R\ x\ y$ oraz $\neg\ R\ (f\ x)\ (f\ y)$, a wtedy sposób rozumowania analogiczny do tego z twierdzenia `f_equal` byłby niepoprawny.

Aby taktyka *f_equiv* "widziała", że f zachowuje R, musimy znów posłużyć się komendą `Instance` i użyć *Proper*, które służy do zwięzłego wyrażania, które konkretnie relacje i w jaki sposób zachowuje dana funkcja.

W naszym przypadku będziemy chcieli pokazać, że jeżeli listy *l1* oraz *l1'* są w relacji *len_eq* (czyli mają taką samą długość) i podobnie dla *l2* oraz *l2'*, to wtedy konkatenacja *l1* i *l2* jest w relacji *len_eq* z konkatenacją *l1'* i *l2'*. Ten właśnie fakt jest wyrażany przez zapis *Proper* (@*len_eq* *A* ==> @*len_eq* *A* ==> @*len_eq* *A*) (@*app* *A*).

Należy też zauważyć, że strzałka ==> jest jedynie notacją dla tworu zwanego *respectful*, co możemy łatwo sprawdzić komendą `Locate`.

```
Example f_equiv_0 :
  ∀ (A B : Type) (f : A → B) (l1 l1' l2 l2' : list A),
    len_eq l1 l1' → len_eq l2 l2' →
      len_eq (l1 ++ l2) (l1' ++ l2').
Proof.
  intros. f_equiv.
    assumption.
    assumption.
Qed.
```

Voilà! Teraz możemy używać taktyki *f_equiv* z relacją *len_eq* oraz funkcją *app* dokładnie tak, jak taktyki `f_equal` z równością oraz dowolną funkcją.

Trzeba przyznać, że próba użycia *f_equiv* z różnymi kombinacjami relacji i funkcji może zakończyć się nagłym i niekontrolowanym rozmnożeniem lematów mówiących o tym, że funkcje zachowują relacje. Niestety, nie ma na to żadnego sposobu — jak przekonaliśmy się wyżej, udowodnienie takiego lematu to jedyny sposób, aby upewnić się, że nasz sposób rozumowania jest poprawny.

Ćwiczenie (f_equiv_filter)

```
Require Import List.
Import ListNotations.

Definition stupid_id {A : Type} (l : list A) : list A :=
  filter (fun _ ⇒ true) l.
```

Oto niezbyt mądry sposób na zapisanie funkcji identycznościowej na listach typu A. Pokaż, że *stupid_id* zachowuje relację *len_eq*, tak aby poniższy dowód zadziałał bez wprowadzania zmian.

```
Example f_equiv_1 :
  ∀ (A : Type) (l l' : list A),
```

```
    len_eq l l' → len_eq (stupid_id l) (stupid_id l').
Proof.
  intros. f_equiv. assumption.
Qed.
```

6.4.3 rewrite

Powinieneś być już nieźle wprawiony w używaniu taktyki `rewrite`. Czas najwyższy więc opisać wszystkie jej możliwości.

Podstawowe wywołanie tej taktyki ma postać `rewrite` *H*, gdzie *H* jest typu $\forall$ (*x_1* : *A_1*) ... (*x_n* : *A_n*), *R t_1 t_2*, zaś *R* to *eq* lub dowolna relacja równoważności. Przypomnijmy, że relacja równoważności to relacja, która jest zwrotna, symetryczna i przechodnia.

`rewrite` *H* znajduje pierwszy podterm celu, który pasuje do *t_1* i zamienia go na *t_2*, generując podcele *A_1*, ..., *A_n*, z których część (a często całość) jest rozwiązywana automatycznie.

```
Check plus_n_Sm.
(*  ===> plus_n_Sm :
          forall n m : nat, S (n + m) = n + S m *)
Goal 2 + 3 = 6 → 4 + 4 = 42.
Proof.
  intro.
  rewrite ← plus_n_Sm.
  rewrite plus_n_Sm.
  rewrite ← plus_n_Sm.
  rewrite → plus_n_Sm.
  rewrite ← !plus_n_Sm.
  Fail rewrite ← !plus_n_Sm.
  rewrite ← ?plus_n_Sm.
  rewrite 4!plus_n_Sm.
  rewrite ← 3?plus_n_Sm.
  rewrite 2 plus_n_Sm.
Abort.
```

Powyższy skrajnie bezsensowny przykład ilustruje fakt, że działanie taktyki `rewrite` możemy zmieniać, poprzedzając hipotezę *H* następującymi modyfikatorami:

- `rewrite` → *H* oznacza to samo, co `rewrite` *H*
- `rewrite` ← *H* zamienia pierwsze wystąpienie *t_2* na *t_1*, czyli przepisuje z prawa na lewo
- `rewrite` ?*H* przepisuje *H* 0 lub więcej razy
- `rewrite` *n*?*H* przepisuje *H* co najwyżej n razy

- `rewrite` $!H$ przepisuje H 1 lub więcej razy
- `rewrite` $n!H$ lub `rewrite` n H przepisuje H dokładnie n razy

Zauważmy, że modyfikator ← można łączyć z modyfikatorami określającymi ilość przepisań.

```
Lemma rewrite_ex_1 :
  ∀ n m : nat, 42 = 42 → S (n + m) = n + S m.
Proof.
  intros. apply plus_n_Sm.
Qed.
```

```
Goal 2 + 3 = 6 → 5 + 5 = 12 → (4 + 4) + ((5 + 5) + (6 + 6)) = 42.
Proof.
  intros.
  rewrite ← plus_n_Sm, ← plus_n_Sm.
  rewrite ← plus_n_Sm in H.
  rewrite ← plus_n_Sm in * ⊢.
  rewrite !plus_n_Sm in *.
  rewrite ← rewrite_ex_1. 2: reflexivity.
  rewrite ← rewrite_ex_1 by reflexivity.
Abort.
```

Pozostałe warianty taktyki `rewrite` przedstawiają się następująco:

- `rewrite` *H_1*, ..., *H_n* przepisuje kolejno hipotezy *H_1*, ..., *H_n*. Każdą z hipotez możemy poprzedzić osobnym zestawem modyfikatorów.
- `rewrite` H `in` *H'* przepisuje H nie w celu, ale w hipotezie *H'*
- `rewrite` H `in` * ⊢ przepisuje H we wszystkich hipotezach różnych od H
- `rewrite` H `in` * przepisuje H we wszystkich hipotezach różnych od H oraz w celu
- `rewrite` H `by` *tac* działa jak `rewrite` H, ale używa taktyki *tac* do rozwiązania tych podcelów, które nie mogły zostać rozwiązane automatycznie

Jest jeszcze wariant `rewrite` H `at` n (wymagający zaimportowania modułu *Setoid*), który zamienia n-te (licząc od lewej) wystąpienie *t_1* na *t_2*. Zauważmy, że `rewrite` H znaczy to samo, co `rewrite` H `at` 1.

6.5 Taktyki dla redukcji i obliczeń (TODO)

6.6 Procedury decyzyjne

Procedury decyzyjne to taktyki, które potrafią zupełnie same rozwiązywać cele należące do pewnej konkretnej klasy, np. cele dotyczące funkcji boolowskich albo nierówności liniowych na liczbach całkowitych. W tym podrozdziale omówimy najprzydatniejsze z nich.

6.6.1 *btauto*

btauto to taktyka, która potrafi rozwiązywać równania boolowskie, czyli cele postaci $x = y$, gdzie x i y są wyrażeniami mogącymi zawierać boolowskie koniunkcje, dysjunkcje, negacje i inne rzeczy (patrz manual).

Taktykę można zaimportować komendą `Require Import` *Btauto*. Uwaga: nie potrafi ona wprowadzać zmiennych do kontekstu.

Ćwiczenie (my_btauto) Napisz następujące taktyki:

- *my_btauto* — taktyka podobna do *btauto*. Potrafi rozwiązywać cele, które są kwantyfikowanymi równaniami na wyrażeniach boolowskich, składającymi się z dowolnych funkcji boolowskich (np. *andb*, *orb*). W przeciwieństwie do *btauto* powinna umieć wprowadzać zmienne do kontekstu.
- *my_btauto_rec* — tak samo jak *my_btauto*, ale bez używana kombinatora `repeat`. Możesz używać jedynie rekurencji.
- *my_btauto_iter* — tak samo jak *my_btauto*, ale bez używania rekurencji. Możesz używać jedynie kombinatora `repeat`.
- *my_btauto_no_intros* — tak samo jak *my_btauto*, ale bez używania taktyk `intro` oraz `intros`.

Uwaga: twoja implementacja taktyki *my_btauto* będzie diametralnie różnić się od implementacji taktyki *btauto* z biblioteki standardowej. *btauto* jest zaimplementowana za pomocą reflekcji. Dowód przez reflekcję omówimy później.

```
Section my_btauto.

Require Import Bool.
Require Import Btauto.

Theorem andb_dist_orb :
  ∀ b1 b2 b3 : bool,
    b1 && (b2 || b3) = (b1 && b2) || (b1 && b3).
Proof.
```

```
  intros. btauto.
Restart.
  my_btauto.
Restart.
  my_btauto_rec.
Restart.
  my_btauto_iter.
Restart.
  my_btauto_no_intros.
Qed.

Theorem negb_if :
  ∀ b1 b2 b3 : bool,
    negb (if b1 then b2 else b3) = if negb b1 then negb b3 else negb b2.
Proof.
  intros. btauto.
Restart.
  my_btauto.
Restart.
  my_btauto_rec.
Restart.
  my_btauto_iter.
Restart.
  my_btauto_no_intros.
Qed.
```

Przetestuj działanie swoich taktyk na reszcie twierdzeń z rozdziału o logice boolowskiej.

```
End my_btauto.
```

6.6.2 congruence

```
Example congruence_0 :
  ∀ P : Prop, true ≠ false.
Proof. congruence. Qed.

Example congruence_1 :
  ∀ (A : Type) (f : A → A) (g : A → A → A) (a b : A),
    a = f a → g b (f a) = f (f a) → g a b = f (g b a) →
      g a b = a.
Proof.
  congruence.
Qed.

Example congruence_2 :
  ∀ (A : Type) (f : A → A × A) (a c d : A),
```

$f = pair\ a \to Some\ (f\ c) = Some\ (f\ d) \to c = d.$
```
Proof.
  congruence.
Qed.
```

congruece to taktyka, która potrafi rozwiązywać cele dotyczące nieinterpretowanych równości, czyli takie, których prawdziwość zależy jedynie od hipotez postaci $x = y$ i które można udowodnić ręcznie za pomocą mniejszej lub większej ilości `rewrite`'ów. `congruence` potrafi też rozwiązywać cele dotyczące konstruktorów. W szczególności wie ona, że konstruktory są injektywne i potrafi odróżnić *true* od *false*.

Ćwiczenie (congruence) Udowodnij przykłady *congruence_1* i *congruence_2* ręcznie.

Ćwiczenie (discriminate) Inną taktyką, która potrafi rozróżniać konstruktory, jest `discriminate`. Zbadaj, jak działa ta taktyka. Znajdź przykład celu, który `discriminate` rozwiązuje, a na którym `congruence` zawodzi. Wskazówka: `congruence` niebardzo potrafi odwijać definicje.

Ćwiczenie (injection i simplify_eq) Kolejne dwie taktyki do walki z konstruktorami typów induktywnych to `injection` i *simplify_eq*. Przeczytaj ich opisy w manualu. Zbadaj, czy są one w jakikolwiek sposób przydatne (wskazówka: porównaj je z taktykami `inversion` i `congruence`.

6.6.3 *decide equality*

`Inductive` C : `Type` :=
  $|\ c0 : C$
  $|\ c1 : C \to C$
  $|\ c2 : C \to C \to C$
  $|\ c3 : C \to C \to C \to C.$

Przyjrzyjmy się powyższemu, dosć enigmatycznemu typowi. Czy posiada on rozstrzygalną równość? Odpowiedź jest twierdząca: rozstrzygalną równość posiada każdy typ induktywny, którego konstruktory nie biorą argumentów będących dowodami, funkcjami ani termami typów zależnych.

`Theorem` *C_eq_dec* :
  $\forall\ x\ y : C, \{x = y\} + \{x \neq y\}.$

Zanim przejdziesz dalej, udowodnij ręcznie powyższe twierdzenie. Przyznasz, że dowód nie jest zbyt przyjemny, prawda? Na szczęście nie musimy robić go ręcznie. Na ratunek przychodzi nam taktyka *decide equality*, która umie udowadniać cele postaci $\forall\ x\ y : T, \{x = y\} + \{x \neq y\}$, gdzie T spełnia warunki wymienione powyżej.

`Theorem` *C_eq_dec'* :
  $\forall\ x\ y : C, \{x = y\} + \{x \neq y\}.$
`Proof.` *decide equality*. `Defined.`

Ćwiczenie Pokrewną taktyce *decide equality* jest taktyka *compare*. Przeczytaj w manualu, co robi i jak działa.

6.6.4 `omega` i `abstract`

`omega` to taktyka, która potrafi rozwiązywać cele dotyczące arytmetyki Presburgera. Jej szerszy opis można znaleźć w manualu. Na nasze potrzeby przez arytmetykę Presburgera możemy rozumieć równania ($=$), nie-równania ($\neq$) oraz nierówności ($<$, $\leq$, $>$, $\geq$) na typie *nat*, które mogą zawierać zmienne, 0, S, dodawanie i mnożenie przez stałą. Dodatkowo zdania tej postaci mogą być połączone spójnikami $\land$, $\lor$, $\rightarrow$ oraz $\neg$, ale nie mogą być kwantyfikowane — `omega` nie umie wprowadzać zmiennych do kontekstu.

```
Require Import Arith Omega.

Example omega_0 :
  ∀ n : nat, n + n = 2 × n.
Proof. intro. omega. Qed.

Example omega_1 :
  ∀ n m : nat, 2 × n + 1 ≠ 2 × m.
Proof. intros. omega. Qed.

Example omega_2 :
  ∀ n m : nat, n × m = m × n.
Proof. intros. try omega. Abort.
```

Jedną z wad taktyki `omega` jest rozmiar generowanych przez nią termów. Są tak wielkie, że wszelkie próby rozwinięcia definicji czy dowodów, które zostały przy jej pomocy skonstruowane, muszą z konieczności źle się skończyć. Zobaczmy to na przykładzie.

```
Lemma filter_length :
  ∀ (A : Type) (f : A → bool) (l : list A),
    length (filter f l) ≤ length l.
Proof.
  induction l; cbn; try destruct (f a); cbn; omega.
Qed.

Print filter_length.
(*  ===> Proofterm o długości 317 linijek. *)
```

Oto jedna ze standardowych właściwości list: filtrowanie nie zwiększa jej rozmiaru. Term skonstruowany powyższym dowodem, będący świadkiem tego faktu, liczy sobie 317 linijek długości (po wypisaniu, wklejeniu do CoqIDE i usunięciu tego, co do termu nie należy).

```
Lemma filter_length' :
  ∀ (A : Type) (f : A → bool) (l : list A),
    length (filter f l) ≤ length l.
Proof.
  induction l; cbn; try destruct (f a); cbn.
```

```
    trivial.
    apply le_n_S. assumption.
    apply le_trans with (length l).
      assumption.
      apply le_S. apply le_n.
Qed.
```

```
Print filter_length'.
(* ===> Proofterm o długości 14 linijek. *)
```

Jak widać, ręczny dowód tego faktu daje w wyniku proofterm, który jest o ponad 300 linijk krótszy niż ten wyprodukowany przez taktykę `omega`. Mogłoby się zdawać, że jesteśmy w sytuacji bez wyjścia: albo dowodzimy ręcznie, albo prooftermy będą tak wielkie, że nie będziemy mogli ich odwijać. Możemy jednak zjeść ciastko i mieć ciastko, a wszystko to za sprawą taktyki `abstract` i towarzyszącej jej komendy `Qed` *exporting*.

```
(* TODO: w najnowszym Coqu nie działa *)
```

```
Lemma filter_length'' :
  ∀ (A : Type) (f : A → bool) (l : list A),
    length (filter f l) ≤ length l.
Proof.
  induction l; cbn; try destruct (f a); cbn;
  abstract omega using lemma_name.
Qed.
```

```
Print filter_length''.
(* ===> Proofterm o długości 13 linijek. *)
```

Taktyka `abstract` t działa tak jak t, ale z tą różnicą, że ukrywa term wygenerowany przez t w zewnętrznym lemacie. Po zakończeniu dowodu możemy zakończyć go komendą `Qed` *exporting*, co spowoduje zapisanie go w takiej skróconej postaci z odwołaniami do zewnętrznych lematów, albo standardowym `Qed`, przez co term będzie wyglądał tak, jakbyśmy wcale nie użyli taktyki `abstract`.

6.6.5 Procedury decyzyjne dla logiki

```
Example tauto_0 :
  ∀ A B C D : Prop,
    ¬ A ∨ ¬ B ∨ ¬ C ∨ ¬ D → ¬ (A ∧ B ∧ C ∧ D).
Proof. tauto. Qed.
```

```
Example tauto_1 :
  ∀ (P : nat → Prop) (n : nat),
    n = 0 ∨ P n → n ≠ 0 → P n.
Proof. auto. tauto. Qed.
```

`tauto` to taktyka, która potrafi udowodnić każdą tautologię konstruktywnego rachunku zdań. Taktyka ta radzi sobie także z niektórymi nieco bardziej skomplikowanymi celami, w tym takimi, których nie potrafi udowodnić `auto`. `tauto` zawodzi, gdy nie potrafi udowodnić celu.

```
Example intuition_0 :
  ∀ (A : Prop) (P : nat → Prop),
    A ∨ (∀ n : nat, ¬ A → P n) → ∀ n : nat, ¬ ¬ (A ∨ P n).
Proof.
  Fail tauto. intuition.
Qed.
```

`intuition` to `tauto` na sterydach — potrafi rozwiązać nieco więcej celów, a poza tym nigdy nie zawodzi. Jeżeli nie potrafi rozwiązać celu, upraszcza go.

Może też przyjmować argument: `intuition` t najpierw upraszcza cel, a później próbuje go rozwiązać taktyką t. Tak naprawdę `tauto` jest jedynie synonimem dla `intuition fail`, zaś samo `intuition` to synonim `intuition auto with` ×, co też tłumaczy, dlaczego `intuition` potrafi więcej niż `tauto`.

```
Record and3 (P Q R : Prop) : Prop :=
{
    left : P;
    mid : Q;
    right : R;
}.

Example firstorder_0 :
  ∀ (B : Prop) (P : nat → Prop),
    and3 (∀ x : nat, P x) B B →
      and3 (∀ y : nat, P y) (P 0) (P 0) ∨ B ∧ P 0.
Proof.
  Fail tauto.
  intuition.
Restart.
  firstorder.
Qed.

Example firstorder_1 :
  ∀ (A : Type) (P : A → Prop),
    (∃ x : A, ¬ P x) → ¬ ∀ x : A, P x.
Proof.
  Fail tauto. intuition.
Restart.
  firstorder.
Qed.
```

Jednak nawet `intuition` nie jest w stanie sprostać niektórym prostym dla człowieka celom — powyższy przykład pokazuje, że nie potrafi ona posługiwać się niestandardowymi spójnikami logicznymi, takimi jak potrójna koniunkcja *and3*.

Najpotężniejszą taktyką potrafiącą dowodzić tautologii jest `firstorder`. Nie tylko rozumie ona niestandardowe spójniki (co i tak nie ma większego praktycznego znaczenia), ale też świetnie radzi sobie z kwantyfikatorami. Drugi z powyższych przykładów pokazuje, że potrafi ona dowodzić tautologii konstruktywnego rachunku predykatów, z którymi problem ma `intuition`.

Ćwiczenie (my_tauto) Napisz taktykę *my_tauto*, która będzie potrafiła rozwiązać jak najwięcej tautologii konstruktywnego rachunku zdań.

Wskazówka: połącz taktyki z poprzednich ćwiczeń. Przetestuj swoją taktykę na ćwiczeniach z rozdziału pierwszego — być może ujawni to problemy, o których nie pomyślałeś.

Nie używaj żadnej zaawansowanej automatyzacji. Użyj jedynie `unfold`, `intro`, `repeat`, `match`, `destruct`, `clear`, `exact`, `split`, `specialize` i `apply`.

6.7 Ogólne taktyki automatyzacyjne

W tym podrozdziale omówimy pozostałe taktyki przydające się przy automatyzacji. Ich cechą wspólną jest rozszerzalność — za pomocą specjalnych baz podpowiedzi będziemy mogli nauczyć je radzić sobie z każdym celem.

6.7.1 `auto` i `trivial`

`auto` jest najbardziej ogólną taktyką służącą do automatyzacji.

Example *auto_ex0* :
  $\forall$ (P : Prop), $P \to P$.
Proof. auto. Qed.

Example *auto_ex1* :
  $\forall$ A B C D E : Prop,
    $(A \to B) \to (B \to C) \to (C \to D) \to (D \to E) \to A \to E$.
Proof. auto. Qed.

Example *auto_ex2* :
  $\forall$ (A : Type) (x : A), $x = x$.
Proof. auto. Qed.

Example *auto_ex3* :
  $\forall$ (A : Type) (x y : A), $x = y \to y = x$.
Proof. auto. Qed.

`auto` potrafi używać założeń, aplikować hipotezy i zna podstawowe własności równości — całkiem nieźle. Wprawdzie nie wystarczy to do udowodnienia żadnego nietrywialnego twierdzenia, ale przyda się z pewnością do rozwiązywania prostych podcelów generowanych przez inne taktyki. Często spotykanym idiomem jest *t*; `auto` — "użyj taktyki *t* i pozbądź się prostych podcelów za pomocą `auto`".

```
Section auto_ex4.

Parameter P : Prop.
Parameter p : P.

Example auto_ex4 : P.
Proof.
  auto.
Restart.
  auto using p.
Qed.
```

Jak widać na powyższym przykładzie, `auto` nie widzi aksjomatów (ani definicji/lematów/twierdzeń etc.), nawet jeżeli zostały zadeklarowane dwie linijki wyżej. Tej przykrej sytuacji możemy jednak łatwo zaradzić, pisząc `auto using` *t_1*, ..., *t_n*. Ten wariant taktyki `auto` widzi definicje termów *t_1*, ..., *t_n*.

Co jednak w sytuacji, gdy będziemy wielokrotnie chcieli, żeby `auto` widziało pewne definicje? Nietrudno wyobrazić sobie ogrom pisaniny, którą mogłoby spowodować użycie do tego celu klauzuli `using`. Na szczęście możemy temu zaradzić za pomocą podpowiedzi, które bytują w specjalnych bazach.

```
Hint Resolve p : my_hint_db.

Example auto_ex4' : P.
Proof. auto with my_hint_db. Qed.
```

Komenda `Hint Resolve` *ident* : *db_name* dodaje lemat o nazwie *ident* do bazy podpowiedzi o nazwie *db_name*. Dzięki temu taktyka `auto with` *db_1* ... *db_n* widzi wszystkie lematy dodane do baz *db_1*, ..., *db_n*. Jeżeli to dla ciebie wciąż zbyt wiele pisania, uszy do góry!

```
Example auto_ex4'' : P.
Proof. auto with ×. Qed.
```

Taktyka `auto with` × widzi wszystkie możliwe bazy podpowiedzi.

```
Hint Resolve p.

Example auto_ex4''' : P.
Proof. auto. Qed.
```

Komenda `Hint Resolve` *ident* dodaje lemat o nazwie *ident* do bazy podpowiedzi o nazwie *core*. Taktyka `auto` jest zaś równoważna taktyce `auto with` *core*. Dzięki temu nie musimy pisać już nic ponad zwykłe `auto`.

```
End auto_ex4.
```

Tym oto sposobem, używając komendy `Hint Resolve`, jesteśmy w stanie zaznajomić `auto` z różnej maści lematami i twierdzeniami, które udowodniliśmy. Komendy tej możemy używać po każdym lemacie, dzięki czemu taktyka `auto` rośnie w siłę w miarę rozwoju naszej teorii.

```
Example auto_ex5 : even 8.
Proof.
  auto.
Restart.
  auto using even0, evenSS.
Qed.
```

Kolejną słabością `auto` jest fakt, że taktyka ta nie potrafi budować wartości typów induktywnych. Na szczęście możemy temu zaradzić używając klauzuli `using` *c_1 ... c_n*, gdzie *c_1*, ..., *c_n* są konstruktorami naszego typu, lub dodając je jako podpowiedzi za pomocą komendy `Hint Resolve` *c_1 ... c_n : db_name*.

```
Hint Constructors even.
```

```
Example auto_ex5' : even 8.
Proof. auto. Qed.
```

Żeby jednak za dużo nie pisać (wypisanie nazw wszystkich konstruktorów mogłoby być bolesne), możemy posłużyć się komendą `Hint Constructors` *I : db_name*, która dodaje konstruktory typu induktywnego *I* do bazy podpowiedzi *db_name*.

```
Example auto_ex6 : even 10.
Proof.
  auto.
Restart.
  auto 6.
Qed.
```

Kolejnym celem, wobec którego `auto` jest bezsilne, jest *even* 10. Jak widać, nie wystarczy dodać konstruktorów typu induktywnego jako podpowiedzi, żeby wszystko było cacy. Niemoc `auto` wynika ze sposobu działania tej taktyki. Wykonuje ona przeszukiwanie w głąb z nawrotami, które działa mniej więcej tak:

- zrób pierwszy lepszy możliwy krok dowodu
- jeżeli nie da się nic więcej zrobić, a cel nie został udowodniony, wykonaj nawrót i spróbuj czegoś innego
- w przeciwnym wypadku wykonaj następny krok dowodu i powtarzaj całą procedurę

Żeby ograniczyć czas poświęcony na szukanie dowodu, który może być potencjalnie bardzo długi, `auto` ogranicza się do wykonania jedynie kilku kroków w głąb (domyślnie jest to 5).

```
Print auto_ex5'.
(*  ===> evenSS 6 (evenSS 4 (evenSS 2 (evenSS 0 even0)))
        : even 8 *)
```

```
Print auto_ex6.
(*  ===> evenSS 8 (evenSS 6 (evenSS 4 (evenSS 2 (evenSS 0 even0))))
       : even 10 *)
```

`auto` jest w stanie udowodnić *even* 8, gdyż dowód tego faktu wymaga jedynie 5 kroków, mianowicie czeterokrotnego zaaplikowania konstruktora *evenSS* oraz jednokrotnego zaaplikowania *even0*. Jednak 5 kroków nie wystarcza już, by udowodnić *even* 10, gdyż tutaj dowód liczy sobie 6 kroków: 5 użyć *evenSS* oraz 1 użycie *even0*.

Nie wszystko jednak stracone — możemy kontrolować głębokość, na jaką `auto` zapuszcza się, poszukując dowodu, piząc `auto` n. Zauważmy, że `auto` jest równoważne taktyce `auto` 5.

```
Example auto_ex7 :
  ∀ (A : Type) (x y z : A), x = y → y = z → x = z.
Proof.
  auto.
Restart.
  Fail auto using eq_trans.
Abort.
```

Kolejnym problemem taktyki `auto` jest udowodnienie, że równość jest relacją przechodnią. Tym razem jednak problem jest poważniejszy, gdyż nie pomaga nawet próba użycia klauzuli `using` *eq_trans*, czyli wskazanie `auto` dokładnie tego samego twierdzenia, którego próbujemy dowieść!

Powód znów jest dość prozaiczny i wynika ze sposobu działania taktyki `auto` oraz postaci naszego celu. Otóż konkluzja celu jest postaci $x = z$, czyli występują w niej zmienne x i z, zaś kwantyfikujemy nie tylko po x i z, ale także po A i y.

Wywnioskowanie, co wstawić za A nie stanowi problemu, gdyż musi to być typ x i z. Problemem jest jednak zgadnięcie, co wstawić za y, gdyż w ogólności możliwości może być wiele (nawet nieskończenie wiele). Taktyka `auto` działa w ten sposób, że nawet nie próbuje tego zgadywać.

```
Hint Extern 0 ⇒
match goal with
    | H : ?x = ?y, H' : ?y = ?z ⊢ ?x = ?z ⇒ apply (@eq_trans _ x y z)
end : extern_db.

Example auto_ex7 :
  ∀ (A : Type) (x y z : A), x = y → y = z → x = z.
Proof. auto with extern_db. Qed.
```

Jest jednak sposób, żeby uporać się i z tym problemem: jest nim komenda `Hint Extern`. Jej ogólna postać to `Hint Extern` n `pattern` ⇒ *tactic* : *db*. W jej wyniku do bazy podpowiedzi *db* zostanie dodana podpowiedź, która sprawi, że w dowolnym momencie dowodu taktyka `auto`, jeżeli wypróbowała już wszystkie podpowiedzi o koszcie mniejszym niż n i cel pasuje do wzorca `pattern`, to spróbuje użyć taktyki *tac*.

W naszym przypadku koszt podpowiedzi wynosi 0, a więc podpowiedź będzie odpalana niemal na samym początku dowodu. Wzorzec `pattern` został pominięty, a więc `auto` użyje naszej podpowiedzi niezależnie od tego, jak wygląda cel. Ostatecznie jeżeli w konktekście będą odpowiednie równania, to zaaplikowany zostanie lemat @*eq_trans* _ x y z, wobec czego wygenerowane zostaną dwa podcele, $x = y$ oraz $y = z$, które `auto` będzie potrafiło rozwiązać już bez naszej pomocy.

```
Hint Extern 0 (?x = ?z) ⇒
match goal with
    | H : ?x = ?y, H' : ?y = ?z ⊢ _ ⇒ apply (@eq_trans _ x y z)
end.
```

```
Example auto_ex7' :
  ∀ (A : Type) (x y z : A), x = y → y = z → x = z.
Proof. auto. Qed.
```

A tak wygląda wersja `Hint Extern`, w której nie pominięto wzorca `pattern`. Jest ona rzecz jasna równoważna z poprzednią.

Jest to dobry moment, by opisać dokładniej działanie taktyki `auto`. `auto` najpierw próbuje rozwiązać cel za pomocą taktyki `assumption`. Jeżeli się to nie powiedzie, to `auto` używa taktyki `intros`, a następnie dodaje do tymczasowej bazy podpowiedzi wszystkie hipotezy. Następnie przeszukuje ona bazę podpowiedzi dopasowując cel do wzorca stowarzyszonego z każdą podpowiedzią, zaczynając od podpowiedzi o najmniejszym koszcie (podpowiedzi pochodzące od komend `Hint Resolve` oraz `Hint Constructors` są skojarzone z pewnymi domyślnymi kosztami i wzorcami). Następnie `auto` rekurencyjnie wywołuje się na podcelach (chyba, że przekroczona została maksymalna głębokość przeszukiwania — wtedy następuje nawrót).

```
Example trivial_ex0 :
  ∀ (P : Prop), P → P.
Proof. trivial. Qed.
```

```
Example trivial_ex1 :
  ∀ A B C D E : Prop,
    (A → B) → (B → C) → (C → D) → (D → E) → A → E.
Proof. trivial. Abort.
```

```
Example trivial_ex2 :
  ∀ (A : Type) (x : A), x = x.
Proof. trivial. Qed.
```

```
Example trivial_ex3 :
  ∀ (A : Type) (x y : A), x = y → y = x.
Proof. trivial. Abort.
```

```
Example trivial_ex5 : even 0.
Proof. trivial. Qed.
```

```
Example trivial_ex5' : even 8.
Proof. trivial. Abort.
```

Taktyka `trivial`, którą już znasz, działa dokładnie tak samo jak `auto`, ale jest nierekurencyjna. To tłumaczy, dlaczego potrafi ona posługiwać się założeniami i zna właciwości równości, ale nie umie używać implikacji i nie radzi sobie z celami pokroju *even* 8, mimo że potrafi udowodnić *even* 0.

Ćwiczenie (auto i trivial) Przeczytaj w manualu dokładny opis działania taktyk `auto` oraz `trivial`: https://coq.inria.fr/refman/tactics.html#hevea_tactic161

6.7.2 `autorewrite` i *autounfold*

`autorewrite` to bardzo pożyteczna taktyka umożliwiająca zautomatyzowanie części dowodów opierających się na przepisywaniu.

Dlaczego tylko części? Zastanówmy się, jak zazwyczaj przebiegają dowody przez przepisywanie. W moim odczuciu są dwa rodzaje takich dowodów:

- dowody pierwszego rodzaju to te, w których wszystkie przepisania mają charakter upraszczający i dzięki temu możemy przepisywać zupełnie bezmyślnie

- dowody drugiego rodzaju to te, w których niektóre przepisania nie mają charakteru upraszczającego albo muszą zostać wykonane bardzo precyzyjnie. W takich przypadkach nie możemy przepisywać bezmyślnie, bo grozi to zapętleniem taktyki `rewrite` lub po prostu porażką

Dowody pierwszego rodzaju ze względu na swoją bezmyślność są dobrymi kandydatami do automatyzacji. Właśnie tutaj do gry wkracza taktyka `autorewrite`.

```
Section autorewrite_ex.

Variable A : Type.
Variable l1 l2 l3 l4 l5 : list A.
```

Zacznijmy od przykładu (a raczej ćwiczenia): udowodnij poniższe twierdzenie. Następnie udowodnij je w jednej linijce.

```
Example autorewrite_intro :
  rev (rev (l1 ++ rev (rev l2 ++ rev l3) ++ rev l4) ++ rev (rev l5)) =
  (rev (rev (rev l5 ++ l1)) ++ (l3 ++ rev (rev l2))) ++ rev l4.
```

Ten dowód nie był zbyt twórczy ani przyjemny, prawda? Wyobraź sobie teraz, co by było, gdybyś musiał udowodnić 100 takich twierdzeń (i to w czasach, gdy jeszcze nie można było pisać `rewrite` ?*t_0*, ..., ?*t_n*). Jest to dość ponura wizja.

```
Hint Rewrite rev_app_distr rev_involutive : list_rw.
Hint Rewrite ← app_assoc : list_rw.

Example autorewrite_ex :
  rev (rev (l1 ++ rev (rev l2 ++ rev l3) ++ rev l4) ++ rev (rev l5)) =
  (rev (rev (rev l5 ++ l1)) ++ (l3 ++ rev (rev l2))) ++ rev l4.
```

```
Proof.
  autorewrite with list_rw. reflexivity.
Qed.

End autorewrite_ex.
```

Komenda `Hint Rewrite` [<-] *ident_0* ... *ident_n* : *db_name* dodaje podpowiedzi *ident_0*, ..., *ident_n* do bazy podpowidzi *db_nam*. Domyślnie będą one przepisywane z lewa na prawo, chyba że dodamy przełącznik ← — wtedy wszystkie będą przepisywane z prawa na lewo. W szczególności znaczy to, że jeżeli chcemy niektóre lematy przepisywać w jedną stronę, a inne w drugą, to musimy komendy `Hint Rewrite` użyć dwukrotnie.

Sama taktyka `autorewrite with` *db_0* ... *db_n* przepisuje lematy ze wszystkich baz podpowiedzi *db_0*, ..., *db_n* tak długo, jak to tylko możliwe (czyli tak długo, jak przepisywanie skutkuje dokonaniem postępu).

Jest kilka ważnych cech, które powinna posiadać baza podpowiedzi:

- przede wszystkim nie może zawierać tego samego twierdzenia do przepisywania w obydwie strony. Jeżeli tak się stanie, taktyka `autorewrite` się zapętli, gdyż przepisanie tego twierdzenia w jedną lub drugą stronę zawsze będzie możliwe
- w ogólności, nie może zawierać żadnego zbioru twierdzeń, których przepisywanie powoduje zapętlenie
- baza powinna być deterministyczna, tzn. jedne przepisania nie powinny blokować kolejnych
- wszystkie przepisywania powinny być upraszczające

Oczywiście dwa ostatnie kryteria nie są zbyt ścisłe — ciężko sprawdzić determinizm systemu przepisywania, zaś samo pojęcie "uproszczenia" jest bardzo zwodnicze i niejasne.

Ćwiczenie (autorewrite) Przeczytaj opis taktyki `autorewrite` w manualu: https://coq.inria.fr/refman

```
Section autounfold_ex.

Definition wut : nat := 1.
Definition wut' : nat := 1.

Hint Unfold wut wut' : wut_db.

Example autounfold_ex : wut = wut'.
Proof.
  autounfold.
  autounfold with wut_db.
Restart.
  auto.
Qed.
```

Na koniec omówimy taktykę *autounfold*. Działa ona na podobnej zasadzie jak `autorewrite`. Za pomocą komendy `Hint Unfold` dodajemy definicje do do bazy podpowiedzi, dzięki czemu taktyka *autounfold* `with` *db_0*, ..., *db_n* potrafi odwinąć wszystkie definicje z baz *db_0*, ..., *db_n*.

Jak pokazuje nasz głupi przykład, jest ona średnio użyteczna, gdyż taktyka `auto` potrafi (przynajmniej do pewnego stopnia) odwijać definicje. Moim zdaniem najlepiej sprawdza się ona w zestawieniu z taktyką `autorewrite` i kombinatorem `repeat`, gdy potrzebujemy na przemian przepisywać lematy i odwijać definicje.

`End` *autounfold_ex*.

Ćwiczenie (autounfold) Przeczytaj w manualu opis taktyki *autounfold*: https://coq.inria.fr/refman/ta

Ćwiczenie (bazy podpowiedzi) Przeczytaj w manualu dokładny opis działania systemu baz podpowiedzi oraz komend pozwalających go kontrolować: https://coq.inria.fr/refman/tactics.html#sec

6.8 Pierścienie, ciała i arytmetyka

Pierścień (ang. ring) to struktura algebraiczna składająca się z pewnego typu A oraz działań + i *, które zachowują się mniej więcej tak, jak dodawanie i mnożenie liczb całkowitych. Przykładów jest sporo: liczby wymierne i rzeczywiste z dodawaniem i mnożeniem, wartości boolowskie z dysjunkcją i koniunkcją oraz wiele innych, których na razie nie wymienię.

Kiedyś z pewnością napiszę coś na temat algebry oraz pierścieni, ale z taktykami do radzenia sobie z nimi możemy zapoznać się już teraz. W Coqu dostępne są dwie taktyki do radzenia sobie z pierścieniami: taktyka *ring_simplify* potrafi upraszczać wyrażenia w pierścieniach, zaś taktyka `ring` potrafi rozwiązywać równania wielomianowe w pierścieniach.

Ciało (ang. field) to pierścień na sterydach, w którym poza dodawaniem, odejmowaniem i mnożeniem jest także dzielenie. Przykładami ciał są liczby wymierne oraz liczby rzeczywiste, ale nie liczby naturalne ani całkowite (bo dzielenie naturalne/całkowitoliczbowe nie jest odwrotnością mnożenia). Je też kiedyś pewnie opiszę.

W Coqu są 3 taktyki pomagające w walce z ciałami: *field_simplify* upraszcza wyrażenia w ciałach, *field_simplify_eq* upraszcza cele, które są równaniami w ciałach, zaś `field` rozwiązuje równania w ciałach.

Ćwiczenie (pierścienie i ciała) Przyczytaj w manualu opis 5 wymienionych wyżej taktyk: https://coq.inria.fr/refman/ring.html

6.9 Zmienne egzystencjalne i ich taktyki (TODO)

Napisać o co chodzi ze zmiennymi egzystencjalnymi. Opisać taktykę `evar` i wspomnieć o taktykach takich jak `eauto`, `econstructor`, `eexists`, *edestruct*, *erewrite* etc., a także taktykę *shelve* i komendę *Unshelve*.

6.10 Taktyki do radzenia sobie z typami zależnymi (TODO)

Opisać taktyki `dependent induction`, `dependent inversion`, `dependent destruction`, `dependent rewrite` etc.

6.11 Dodatkowe ćwiczenia

Ćwiczenie (assert) Znasz już taktyki `assert`, `cut` i `specialize`. Okazuje się, że dwie ostatnie są jedynie wariantami taktyki `assert`. Przeczytaj w manualu opis taktyki `assert` i wszystkich jej wariantów.

Ćwiczenie (easy i now) Taktykami, których nie miałem nigdy okazji użyć, są *easy* i jej wariant *now*. Przeczytaj ich opisy w manualu. Zbadaj, czy są do czegokolwiek przydatne oraz czy są wygodne w porównaniu z innymi taktykami służącymi do podobnych celów.

Ćwiczenie (inversion_sigma) Przeczytaj w manualu o wariantach taktyki `inversion`. Szczególnie interesująca wydaje się taktyka *inversion_sigma*, która pojawiła się w wersji 8.7 Coqa. Zbadaj ją. Wymyśl jakiś przykład jej użycia.

Ćwiczenie (pattern) Przypomnijmy, że podstawą wszelkich obliczeń w Coqu jest redkucja beta. Redukuje ona aplikację funkcji, np. ($\texttt{fun}\ n : nat \Rightarrow 2 \times n$) 42 betaredukuje się do 2×42. Jej wykonywanie jest jednym z głównych zadań taktyk obliczeniowych.

Przeciwieństwem redukcji beta jest ekspansja beta. Pozwala ona zamienić dowolny term na aplikację jakiejś funkcji do jakiegoś argumentu, np. term 2×42 można betaekspandować do ($\texttt{fun}\ n : nat \Rightarrow 2 \times n$) 42.

O ile redukcja beta jest trywialna do automatycznego wykonania, o tyle ekspansja beta już nie, gdyż występuje tu duża dowolność. Dla przykładu, term 2×42 można też betaekspandować do ($\texttt{fun}\ n : nat \Rightarrow n \times 42$) 2.

Ekspansję beta implementuje taktyka `pattern`. Rozumowanie za jej pomocą nie jest zbyt częstne, ale niemniej jednak kilka razy mi się przydało. Przeczytaj opis taktyki `pattern` w manuaulu.

TODO: być może ćwiczenie to warto byłoby rozszerzyć do pełnoprawnego podrozdziału.

Ćwiczenie (arytmetyka) Poza taktykami radzącymi sobie z pierścieniami i ciałami jest też wiele taktyk do walki z arytmetyką. Poza omówioną już taktyką `omega` są to *lia*, *nia*, *lra*, *nra*. Nazwy taktyk można zdekodować w następujący sposób:

- l — linear
- n — nonlinar
- i — integer

- r — real/rational
- a — arithmetic

Spróbuj ogarnąć, co one robią: https://coq.inria.fr/refman/micromega.html

Ćwiczenie (wyższa magia) Spróbuj ogarnąć, co robią taktyki *nsatz*, *psatz* i *fourier*.

6.12 Inne języki taktyk

Ltac w pewnym sensie nie jest jedynym językiem taktyk, jakiego możemy użyć do dowodzenia w Coqu — są inne. Głównymi konkurentami Ltaca są:

- Rtac: gmalecha.github.io/reflections/2016/rtac-technical-overview
- Mtac: plv.mpi-sws.org/mtac/
- ssreflect: coq.inria.fr/refman/ssreflect.html oraz math-comp.github.io/math-comp/

Pierwsze dwa, *Rtac* i *Mtac*, faktycznie są osobnymi językami taktyk, znacznie różniącymi się od Ltaca. Nie będziemy się nimi zajmować, gdyż ich droga do praktycznej użyteczności jest jeszcze dość długa.

ssreflect to nieco inna bajka. Nie jest on w zasadzie osobnym językiem taktyk, lecz jest oparty na Ltacu. Różni się on od niego filozofią, podstawowym zestawem taktyk i stylem dowodzenia. Od wersji 8.7 Coqa język ten jet dostępny w bibliotece standardowej, mimo że nie jest z nią w pełni kompatybilny.

Ćwiczenie (ssreflect) Najbardziej wartościowym moim zdaniem elementem języka ssreflect jest taktyka `rewrite`, dużo potężniejsza od tej opisanej w tym rozdziale. Jest ona warta uwagi, gdyż:

- daje jeszcze większą kontrolę nad przepisywaniem, niż standardowa taktyka `rewrite`
- pozwala łączyć kroki przepisywania z odwijaniem definicji i wykonywaniem obliczeń, a więc zastępuje taktyki `unfold`, `fold`, `change`, `replace`, *cbn*, `simpl` etc.
- daje większe możliwości radzenia sobie z generowanymi przez siebie podcelami

Przeczytaj rozdział manuala opisujący język ssreflect. Jeżeli nie chce ci się tego robić, zapoznaj się chociaż z jego taktyką `rewrite`: coq.inria.fr/refman/ssreflect.html#hevea_tactic265

6.13 Konkluzja

W niniejszym rozdziale przyjrzeliśmy się bliżej znacznej części Coqowych taktyk. Moje ich opisanie nie jest aż tak kompletne i szczegółowe jak to z manuala, ale nadrabia (mam nadzieję) wplecionymi w tekst przykładami i zadaniami. Jeżeli jednak uważasz je za upośledzone, nie jesteś jeszcze stracony! Alternatywne opisy niektórych taktyk dostępne są też tu:

- pjreddie.com/coq-tactics/
- cs.cornell.edu/courses/cs3110/2017fa/a5/coq-tactics-cheatsheet.html
- typesofnote.com/posts/coq-cheat-sheet.html

Poznawszy podstawy Ltaca oraz całe zoo przeróżnych taktyk, do zostania pełnoprawnym inżynierem dowodu (ang. proof engineer, ukute przez analogię do software engineer) brakuje ci jeszcze tylko umiejętności dowodzenia przez reflekcję, którą zajmiemy się już niedługo.

Rozdział 7

Seminar: Induction

This chapter is written in English because its main purpose is to serve as notes for my seminar talk whose topic was "Inductive predicates". It is a bit broader than that and covers all forms of structural induction, including functional induction, and even more. It does not touch the topic of well-founded induction, though.

The grading policy is at the end.

7.1 Inductive propositions and types with a grain of axioms

Let's start with inductive propositions:

```
Print False.
```

```
(*  ===> Inductive False : Prop := *)
```

```
Print True.
```

```
(*  ===> Inductive True : Prop :=
            | I : True *)
```

False and *True* have very simple definitions. *False* doesn't have any constructors and *True* does have one, called *I*. This name is arbitrary — it has to be there, because constructors can't be nameless, but doesn't really matter, because we're only interested in its existence, not name.

The definitions are not that interesting if you have seen them already, so let's have a look at something weirder:

```
Inductive VeryTrue : Prop :=
    | proof : VeryTrue
    | other_proof : VeryTrue.
```

How is this one different from *True*? Well, *False*, *True* and *VeryTrue* can be likened to *Empty_set*, *unit* and *bool*:

```
Print Empty_set.
```

```
(*  ===> Inductive Empty_set : Set := .*)
```

```
Print unit.
(*  ===> Inductive unit : Set :=
            | tt : unit *)
Print bool.
(*  ===> Inductive bool : Set :=
            | true : bool
            | false : bool *)
```

The similarities are striking — these two series of types from a high-level point of view look (nearly) the same. The dissimilarities are more subtle.

Besides names the only difference lies in the sorts — `Prop` and `Set` — but it's a colossal one.

Note: if a proof isn't there, it's an exercise.

```
Theorem bool_inhabited : bool.
Theorem VeryTrue_inhabited : VeryTrue.
```

First off, both *bool* and *VeryTrue*, which correspond in the series, are inhabited. This means that *bool* has an element and *VeryTrue* is indeed true. No surprise here.

```
Theorem unit_contractible :
  ∀ x y : unit, x = y.
Theorem bool_not_contractible :
  ∃ x y : bool, x ≠ y.
```

Secondly, *unit* is contractible and *bool* is not. This means roughly that *unit* has one element and *bool* has more than one (in fact it has two). We could believe that this is also the case for *True* and *VeryTrue*, but it's not that simple.

```
Theorem True_contractible :
  ∀ x y : True, x = y.
Theorem VeryTrue_not_contractible :
  ∃ x y : VeryTrue, x ≠ y.
Proof.
  ∃ proof, other_proof. inversion 1.
Abort.
```

Even though *True* has the same property that holds for *unit*, it looks like the trickery of `inversion` is not enough to prove that *VeryTrue* is contractible.

```
Require Import ProofIrrelevance.
Theorem VeryTrue_contractible :
  ∀ x y : VeryTrue, x = y.
```

No, it's not about lack of trickery — it may simply be not true. In fact, it's axiom-dependent. In vanilla Coq we can't do much, but if we assume the Axiom of Proof Irrelevance, we can prove that both constructors of *VeryTrue* construct the very same proof.

Check *proof_irrelevance*.

```
(*  ===> proof_irrelevance :
       forall (P : Prop) (p1 p2 : P), p1 = p2 *)
```

The Axiom of Proof Irrelevance states that all proofs of any proposition are equal. This is exactly the statement we wanted to prove. You may wonder if we're allowed to assume such an axiom, but it was proved that this axiom is consistent with the Calculus of Inductive Constructions (so, it won't introduce any contradictions to Coq).

But before we go on we have to clarify what was meant by "proof" in the above statement in order to avoid falling into an equivocation lurking in the darkness out there.

In the context of Coq the word "proof" can mean two things:

- "proofterm". Proofterm is just a term. To prove P we have to construct $p : P$ and this p is the proofterm that proves P.

- "proofscript". A proofscript is the text that appears between the theorem statement and the keyword Qed.

What I meant by "all proofs of any proposition are equal" could be better rephrased as "all proofterms of any proposition are equal".

Require Import *Logic.FinFun*.

Note: module FinFun contains definitions of injections, surjections and bijections.

These are not the only differences.

Theorem *no_bij_unit_bool* :
  $\neg \exists f : unit \rightarrow bool$, *Bijective f*.

Even though there are functions going from *unit* to *bool* and the other way around too, they are in no sense equivalences. Particularly, *unit* is not in bijection with *bool*.

Theorem *unit_not_bool* : $unit \neq bool$.

Since Leibniz it is known that two equal things must also have the same properties. Using this fact we can prove that *unit* is not equal to *bool* (since they differ in the property of being in bijection with *unit*).

Theorem *VeryTrue_True* : $VeryTrue \leftrightarrow True$.

VeryTrue and *True* are, however, logically equivalent. This really shouldn't surprise us, because both can be proved without any assumptions. But this is not all that's true about their relationship.

Theorem *bij_VeryTrue_True* :
  $\exists f : VeryTrue \rightarrow True$, *Bijective f*.

From the Axiom of Proof Irrelevance it follows that *VeryTrue* and *True* are in bijection with each other. This too isn't that much of a surprise because we have shown before that they both are contractible, i.e. both have only a single element.

Module *Classical*.

```
Require Import ClassicalFacts.
```

There's another axiom we can assume: the Axiom of Propositional Extensionality, which lives in the module *ClassicalFacts*.

```
Print prop_extensionality.
(*  ===> prop_extensionality =
        forall A B : Prop, A <-> B -> A = B : Prop *)
```

This axiom states that if two propositions are logically equivalent, then they are equal. It also has been proved that adding it to Coq is consistent (even if we already have proof irrelevance).

```
Axiom prop_ext : prop_extensionality.
```

Note: assuming this axiom is technically realized differently than assuming *proof_irrelevance*. To assume *proof_irrelevance*, we only had to import the relevant module and the axiom was sitting there waiting for us. However, to assume *prop_extensionality* we have to write it out explicitly after importing the module.

```
Theorem VeryTrue_is_True : VeryTrue = True.
```

Using this axiom, we can prove that *VeryTrue* and *True* are not only logically equivalent and in bijection with each other, but actually equal. Propositions are very different from ordinary types.

```
End Classical.
```

Let's try to sum up the above lesson using slogans and metaphors. An often repeated phrase is "propositions as types". By this it is most often meant that proposition can be represented by types and logic can be reduced to operations on types.

This is mostly accurate, but as we have seen, proposition and types are not exactly the same thing in Coq. We can imagine that a type is just a bag of dots. The bag is the type proper and the dots are just repeated applications of the type's constructors.

We can then in search of dots put our hand into the bag and if there are some, we can pull them out. If there are many of them, we can distinguish them by looking at them or doing more complicated operations.

Propositions can be thought of as bags of dots in the same way, but if we put our hand into the bag, only two things can possibly happen:

- We pull out a big blob of glue with some dots glued to it. We can't unglue them, but we know there are some dots
- There's nothing

The moral of this story is as follows: in case of types (those terms whose type is `Set` or `Type`, to be more precise) we are interested in what the terms look like, how many are there and so on. In the case of propositions (terms whose type is `Prop`) we can possibly only be interested in whether there are any terms (proofs) or not.

Exercise (easy) Prove that $bool \neq nat$.

7.2 On the number of constructors

Let's consider how the truth of a proposition depends on the number of its constructors.

```
Inductive TheTruest : Prop :=
    | ShortProof : TheTruest
    | LongProof : TheTruest → TheTruest.
```

This proposition dangerously resembles the type *nat*. It appears to have an infinite amount of proofs constructred by two constructors, one of which corresponds to 0 and the other to *S*.

```
Theorem TheTruest_True : TheTruest ↔ True.
```

```
Theorem TheTruest_contractible :
  ∀ x y : TheTruest, x = y.
```

This is not really the case. Since all the proofs are equal to each other, there in fact is only a single proof. So even though this proposition has two constructors one of which is recursive, it resembles *unit* more than *nat*, to which it would be equivalent if it lived in `Set` or `Type`.

This leads us to ask a somewhat contrived question: for propositions, are all constructors besides one useless? Or rather, does a single one suffice to make a proposition true?

```
Inductive BePatient : Prop :=
    | the_longest_proof : BePatient → BePatient.
```

The above proposition has a single constructor, so we could be led to believe it's true.

```
Theorem BePatient_false : ¬ BePatient.
```

```
Theorem BePatient_False : BePatient ↔ False.
```

However it happens that this proposition is not at all true. How is this possible? It has a constructor, so it must be true... not quite. Recall that we can build terms of inductive types by applying their constructors only a *finite* number of times.

The "word" finite is crucial here. If we try to apply the constructor *the_longest_proof* repeatedly, our only hope at finishing is if we do it *ad infinitum*. This is exactly the thing that is forbidden by the definition of inductive types.

So, a proposition with one constructor can be false. We could add more constructors like *the_longest_proof* and they wouldn't help if all of them were recursive. Thus we have proven (but only at the metatheoretical level) that a false proposition can have any number of constructors: 0 (*False*), 1 (*BePatient*), 2 or more (*BePatient* on steroids with more constructors).

But what about nonrecursive constructors? Can a proposition with a nonrecursive constructor be false?

```
Inductive LatentFalsity : Prop :=
```

| *I_am_true* : *False* → *LatentFalsity*.

Well, this one has one nonrecursive constructor, so again we could be led to believe it's true... but beware.

`Theorem` *LatentFalsity_false* : ¬ *LatentFalsity*.

`Theorem` *LatentFalsity_False* : *LatentFalsity* ↔ *False*.

This proposition is false because its only constructor takes a proof of *False* as an argument. Since there isn't one, this constructor can never be used to build a proof of *LatentFalsity*.

Our voyage into the land of constructors looks rather grim. We have met false propositions with any number of constructors, recursive or not. We have also seen that true propositions can have one or more constructors.

The only certainty we discovered is that a proposition without any constructors must necessarily be false. As soon as it has at least one, it can be provable or not depending on how they look like.

Exercise (easy) Does *TheTruest* = *VeryTrue* hold? If yes, under what assumptions?

Exercise (medium) Find a simple heuristic for deciding (at the metatheoretical level) whether an inductive proposition can be proven or not. Assume that this inductive proposition's constructors can only refer to other inductive propositions.

7.3 Induction and induction principles for types

Let's say we want to prove that all elements of a type T have some property. How can we go about it? There are three possibilites that depend on the particular form of T.

First of all, if T is finite, then we can prove our desired statement just by considering each element of T separately. This reasoning can be captured by a case analysis principle. Such a principle is a theorem of the form $\forall\ P : T \to$ `Prop`, $P\ x1 \to ... \to P\ xN \to \forall\ x : T$, $P\ x$, where $x1$, ..., xN are all the elements of T.

Let's see an example case analysis principle for *bool*:

```
Check bool_ind.
(*  ===> bool_ind :
        forall P : bool -> Prop,
          P true -> P false -> forall b : bool, P b *)
```

Note: *bool_ind* is the case analysis principle for *bool* that Coq automatically generated for us. The suffix *_ind* stands for induction, since case analysis principles are a special case of induction principles. Because induction principles can be used not only to prove that all terms have some property, but also to define a dependent function, Coq actually generates us three principles for each definition we make.

For type T these are called *T_rect*, *T_rec* and *T_ind*. They differ only in the sort of the type that depends on T: for *T_rect* this is `Type`, for *T_rec* it's Set and for *T_ind* it's `Prop`. The two latter are implemented by simply calling *T_rect*.

We see that in order to prove that all terms of type *bool* have some property, we only have to prove that *true* has it and that *false* also has it. This is because these two are the only terms of type *bool*.

```
Print bool_rect.
(*  ===> bool_rect =
        fun (P : bool -> Type)
            (f : P true) (f0 : P false) (b : bool) =>
              if b as b0 return (P b0) then f else f0
        : forall P : bool -> Type, P true -> P false ->
            forall b : bool, P b *)
```

When we look at the definition we can see an `if` here. This is just a syntactic sugar for `match`, which shows us that the principle is not magical or built-in and can be derived manually using pattern matching.

But how to use this principle to prove something?

```
Theorem negb_involutive' :
  ∀ b : bool, negb (negb b) = b.
Proof.
  apply bool_ind; cbn; trivial.
Restart.
  destruct b; cbn; trivial.
Qed.
```

Since an induction principle is just a normal theorem, we can apply it like a theorem. This is precisely what the `destruct` tactic does — it applies for us the standard case analysis principle associated with the type of its argument.

Let's get back to our question of how to prove that all terms of some type have some property. The above was just the finite case for inductive types. But what if our type is infinite?

This is the second case. In general, if our type is infinite and we want to be done with our proving in finite time and using finite resources, then we're doomed. A representative example of this case are functions $nat \to nat$ — there are so many of them that we can't prove anything nontrivial about all of them.

Not all hope is lost, however. The third case is: our type is infinite, but defined inductively. In this case we can do as much as in the finite case. Even though we can't check each element of T separately, we know that it is "finitely generated", meaning that it can be constructed by applying one of finitely many constructors of T a finite number of times.

It turns out this is enough for us. Let's see how the induction principle for *nat* looks like.

```
Check nat_ind.
```

```
(*  ===> nat_ind :
        forall P : nat -> Prop,
          P 0 -> (forall n : nat, P n -> P (S n)) ->
            forall n : nat, P n *)
```

To prove that all natural numbers have the property P, we only have to prove that 0 has it and that $S\ n$ has it under the assumption that n does. This is because these two constructors are everything we need in order to generate all natural numbers.

The second argument is of particular interest to us: if it were $\forall\ n : nat,\ P\ (S\ n)$, we would only have a case analysis principle. However, it also has the premise $P\ n$, which makes it recursive.

```
Print nat_rect.
(*  ===> nat_rect =
        fun (P : nat -> Type) (f : P 0)
            (f0 : forall n : nat, P n -> P (S n)) =>
        fix F (n : nat) : P n :=
        match n as n0 return (P n0) with
            | 0 => f
            | S n0 => f0 n0 (F n0)
        end
        : forall P : nat -> Type,
            P 0 -> (forall n : nat, P n -> P (S n)) ->
              forall n : nat, P n *)
```

As we see, this principle too is neither magical nor built-in. It can be derived manually using just pattern matching and recursion. This is the case for all induction principles of inductive types: they can be derived manually using `match` and `fix`.

```
Theorem plus_assoc' :
  ∀ a b c : nat, a + (b + c) = (a + b) + c.
Proof.
  apply nat_ind.
Restart.
  apply (nat_ind
  (fun a : nat ⇒ ∀ b c : nat, a + (b + c) = (a + b) + c)).
    cbn. trivial.
    cbn. intros. rewrite H. trivial.
Restart.
  induction a; cbn; intros.
    trivial.
    rewrite IHa. trivial.
Defined.
```

As in the case of *bool_ind*, we can use our induction principle just by applying it. This time however it is a bit harder: simply doing `apply` is weird, because it produces three goals, first of which is equivalent to the original one, the second is a trivial implication and the third asks us for a natural number.

It works that way because Coq can't guess what is the $P : nat \to \texttt{Prop}$ that we want to use *nat_ind* with. We can solve this problem by giving it manually, just as in our second try.

We also see that the tactic `induction` is there to do just this for us automatically: it correctly identifies the P we want to use and then applies the induction principle.

Exercise (medium) Find a simple metatheoretic algorithm for generating standard induction principles for any inductive type T.

7.4 Parameters and indices

It's high time to see how to define inductive predicates, relations and, more generally, families of types. There are two ways: using parameters and indices. After we study both in separation, we will compare them.

A parametric family of inductive types looks like this:

```
Print option.
(*  ===> Inductive option (A : Type) : Type :=
            | Some : A -> option A
            | None : option A *)
```

Because (A : `Type`) appears before the final colon, *option* is not a type, but a family of types, parametrized by a type A. We can see this with the `Check` command.

```
Check option.
(*  ===> option : Type -> Type *)
```

Parametric definitions basically tell Coq to define a separate inductive type for every possible value of the parameter. In the constructors of the type family, the parameter is fixed: every time it appears, it has to be the same as in the header of the inductive definition.

This is sometimes a limitation, but often it's just fine. The most common use of parameters is the one presented above: defining polymorphic types, containers, relations and predicates. We can easily find more examples of such a usage:

```
Print prod.
(*  ===> Inductive prod (A B : Type) : Type :=
            | pair : A -> B -> A * B *)

Print sum.
(*  ===> Inductive sum (A B : Type) : Type :=
            | inl : A -> A + B
            | inr : B -> A + B *)

Print sigT.
(*  ===> Inductive sigT (A : Type) (P : A -> Type) : Type :=
            | existT : forall x : A, P x -> {x : A & P x} *)
```

```
Print and.
(*  ===> Inductive and (A B : Prop) : Prop :=
            | conj : A -> B -> A /\ B *)

Print or.
(*  ===> Inductive or (A B : Prop) : Prop :=
            | or_introl : A -> A \/ B
            | or_intror : B -> A \/ B *)

Print ex.
(*  ===> Inductive ex (A : Type) (P : A -> Prop) : Prop :=
            | ex_intro : forall x : A, P x -> exists y, P y *)
```

We see that parameters are engouh to implement a whole bunch of most common polymorphic types like products, sums and dependent pairs.

On the other hand, we can define fully general indexed families of types like this:

```
Inductive even : nat → Prop :=
   | even0 : even 0
   | evenSS : ∀ n : nat, even n → even (S (S n)).
```

Here we have nothing before the final colon (the one after *even*), so there aren't any parameters. However, after the colon we see *nat* → `Prop`, which means *even* is a family of propositions indexed by a natural number, i.e. a predicate on *nat*. We can verify this with the command `Check`:

```
Check even.
(*  ===> even : nat -> Prop *)
```

In the constructors of *even*, the index is not fixed: in *even0* it is 0, whereas in *evenSS* it is $S\ (S\ n)$, which is different from 0. Because of this, indexed families are more general than parametric families.

Exercise (easy) These exercises are stolen from the first chapter of Essentials of Programming Languages (exercises 1.1, 1.2). Each one is in a separate module in order to avoid name clashes. Do them.

You can use the tactic `omega`, which can solve arithmetical goals with 0, S, + and multiplication by a constant.

```
Require Import Omega.

Module Ex1_1.

Inductive P : nat → Prop :=
   | c0 : P 2
   | c1 : ∀ n : nat, P n → P (S (S (S n))).

Theorem P_char :
  ∀ n : nat, P n ↔ ∃ k : nat, n = 2 + 3 × k.

End Ex1_1.
```

```
Module Ex1_2.

Inductive P : nat → Prop :=
    | c0 : P 1
    | c1 : ∀ n : nat, P n → P (2 + n)
    | c2 : ∀ n : nat, P n → P (3 + n).

Theorem P_char :
  ∀ n : nat, P n ↔ ∃ k l : nat, n = 1 + 2 × k + 3 × l.

End Ex1_2.

Module Ex1_3.

Inductive P : nat → nat → Prop :=
    | c0 : P 0 1
    | c1 : ∀ n m : nat, P n m → P (1 + n) (2 + m).

Theorem P_char :
  ∀ n m : nat, P n m ↔ m = 1 + 2 × n.

End Ex1_3.

Module Ex1_4.

Inductive P : nat → nat → Prop :=
    | c0 : P 0 0
    | c1 : ∀ n m : nat, P n m → P (S n) (m + 2 × n + 1).

Theorem P_char :
  ∀ n m : nat, P n m ↔ m = n × n.

End Ex1_4.

Module Ex2_1.

Inductive P : nat → nat → Prop :=
    | c0 : P 0 1
    | c1 : ∀ n m : nat, P n m → P (1 + n) (7 + m).

Theorem P_char :
  ∀ n m : nat, P n m ↔ m = 1 + 7 × n.

End Ex2_1.

Module Ex2_2.

Inductive P : nat → nat → Prop :=
    | c0 : P 0 1
    | c1 : ∀ n m : nat, P n m → P (S n) (2 × m).

Require Import FunInd.

Function pow2 (n : nat) : nat :=
match n with
    | 0 ⇒ 1
```

```
    | S n' ⇒ 2 × pow2 n'
end.

Theorem P_char :
  ∀ n m : nat, P n m ↔ m = pow2 n.

End Ex2_2.

Module Ex2_3.

Inductive P : nat → nat → nat → Prop :=
    | c0 : P 0 0 1
    | c1 : ∀ a b c : nat, P a b c → P (S a) c (b + c).

Function fib (n : nat) : nat :=
match n with
    | 0 ⇒ 0
    | 1 ⇒ 1
    | S (S n'' as n') ⇒ fib n' + fib n''
end.

Theorem P_char :
  ∀ a b c : nat, P a b c ↔ b = fib a ∧ c = fib (S a).

End Ex2_3.

Module Ex2_4.

Inductive P : nat → nat → nat → Prop :=
    | c0 : P 0 1 0
    | c1 : ∀ a b c : nat, P a b c → P (1 + a) (2 + b) (b + c).

Theorem P_char :
  ∀ a b c : nat, P a b c ↔ b = 1 + 2 × a ∧ c = a × a.

End Ex2_4.
```

Exercise (easy) This exercise is also stolen from Essentials of Programming Languages (besides the last part of it, which is mine).

Define two inductives predicates T and T', such that:

- they both satisfy the property P
- they are not equal
- T is contained in T'

```
Module Ex3.

Definition P (R : nat → Prop) : Prop :=
  R 0 ∧ ∀ n : nat, R n → R (3 + n).
```

Theorem $P\_T : P\ T$.

Theorem $P\_T' : P\ T'$.

Theorem $T\_not\_T' : T \neq T'$.

Theorem T_sub_T' :
  $\forall\ n : nat,\ T\ n \to T'\ n$.

End *Ex3*.

Parameters and indices are not mutually exclusive. They can be combined freely. A typical example of such usage is the less or equal order relation on the natural numbers.

Print *le*.

```
(*  ===> Inductive le (n : nat) : nat -> Prop :=
            | le_n : n <= n
            | le_S : forall m : nat, n <= m -> n <= S m *)
```

In the definition of *le*, the parameter n is fixed in all constructors, but the index varies: in *le_n* it is n, but in *le_S* it is first m and then $S\ m$. Because of this it can't be rewritten using parameters only, but it can be rewritten using only indices.

Inductive $le' : nat \to nat \to$ Prop :=
  | $le'\_n : \forall\ n : nat,\ le'\ n\ n$
  | $le'\_S : \forall\ n\ m : nat,\ le'\ n\ m \to le'\ n\ (S\ m)$.

We see that in the definition of *le'* we have to quantify over n in each constructor separately in order to use it. This is a disadvantage of indexed families: if the index doesn't vary, we have to write more code when compared to a definition using a parameter instead.

Theorem le_le' :
  $\forall\ n\ m : nat,\ n \leq m \leftrightarrow le'\ n\ m$.

Theorem le_plus :
  $\forall\ n\ m\ k : nat,\ le\ n\ m \to le\ (n + k)\ (m + k)$.

Theorem le'_plus :
  $\forall\ n\ m\ k : nat,\ le'\ n\ m \to le'\ (n + k)\ (m + k)$.

We can easily prove that both definitions yield equivalent relations. Moreover, it is equally easy to prove both *le_plus* and *le'_plus*. So, which of these two definitions is better? The answer is that *le* is better. To see it, let's take a look at the induction principles.

Check *le_ind*.

```
(*  ===> le_ind :
        forall (n : nat) (P : nat -> Prop),
          P n -> (forall m : nat, n <= m -> P m -> P (S m)) ->
            forall n0 : nat, n <= n0 -> P n0 *)
```

Check *le'_ind*.

```
(*  ===> le'_ind :
        forall P : nat -> nat -> Prop,
```

```
(forall n : nat, P n n) ->
(forall n m : nat, le' n m -> P n m -> P n (S m)) ->
  forall n n0 : nat, le' n n0 -> P n n0 *)
```

Don't wory if you don't understand these principles yet. We will come back to them later. The only thing to you need to notice now is that *le'_ind* is more complex than *le_ind*.

Because using parameters means writing less code while having simpler induction principles, we should prefer them wherever possible. There isn't much more to be said: knowing whether to use parameters or indices comes with practice. The only simple heuristic is that often you will want types (and propositions) to be parameters.

If you don't know which one to use, you may use indices first and then check whether any one of them is fixed throughout the definition. If there is one, you can refactor it into a parameter.

Let's now proceed to learn about induction principles for families of types.

7.5 Induction principles for type families

Now that we know how to define parametric and indexed families of types and how to derive standard recursion principles for ordinary inductive types, the next natural question to consider is how to derive induction principles for families of types. Let's take the next step on our path to enlightenment.

Let's start by looking at parametric families.

`Module` *MonomorphicList*.

We will make a new module in order not pollute the global namespace. The name *MonomorphicList* means that we're going to define lists that can hold elements of only a single type.

`Parameter` A : `Type`.

We will call this type A. The command `Parameter` is a synonym of `Axiom`, `Hypothesis` and a few more.

`Inductive` *listA* : `Type` :=
  | *nilA* : *listA*
  | *consA* : $A \to$ *listA* $\to$ *listA*.

listA is just like *list* A (note the lack of space in the first name), but it's not parametric. Rather, the A is fixed. Let's compare the induction principle for *listA* with that of *list*.

`Check` *listA_ind*.

```
(*  ===> listA_ind :
        forall P : listA -> Prop,
          P nilA ->
          (forall (a : A) (l : listA), P l -> P (consA a l)) ->
            forall l : listA, P l *)
```

```
Check list_ind.
(*  ===> list_ind :
        forall (A : Type) (P : list A -> Prop),
          P nil ->
          (forall (a : A) (l : list A), P l -> P (a :: l)list) -> forall l :  list
A, P l *)
```

The principle *listA_ind* says that in order to prove that all *listA*s have some property P : *listA* → Prop, it is necessary to prove that P holds for *nilA* and that if it holds for l, then it holds for *consA* a l where $a : A$ is arbitrary. Not a big surprise.

If we look at the principle for *list*, it says nearly the same thing. The only difference is that the type A is not fixed. It is a parameter and this is reflected in the principle: the first quantifier says ∀ (A : Type).

When it comes to induction principles for parametric families, this is it: quantifiers have to first quantify over the parameters of the type family. The rest of the principle is as usual.

```
End MonomorphicList.
```

We have described only maximal principles for parametric families, but don't worry: the minimal ones work exactly the same way, so let's move on to indexed families.

```
Module IndexedList.

Inductive ilist : Type → Type :=
  | inil : ∀ A : Type, ilist A
  | icons : ∀ A : Type, A → ilist A → ilist A.
```

We will continue the above comparison by looking at the induction principle of a polymorphic list type, but written using indices instead of parameters.

```
Check icons _ 5 (icons _ 42 (inil _)).
(*  ===> icons nat 5 (icons nat 42 (inil nat)) : ilist nat *)

Fail Check icons _ 42 (icons _ true (inil _)).
(*  ===> The term "icons bool true (inil bool)" has type "ilist bool"
        while it is expected to have type "ilist nat". *)
```

This type is just like the ordinary *list*: we can have lists of elements of any type we like, but we can't put elements of different types in the same list.

```
Check ilist_ind.
(*  ===> ilist_ind :
        forall P : forall T : Type, ilist T -> Prop,
          (forall A : Type, P A (inil A)) ->
          (forall (A : Type) (a : A) (i : ilist A),
            P A i -> P A (icons A a i)) ->
              forall (T : Type) (i : ilist T), P T i *)
```

The principle is much more complex than that for ordinary *list*s. First thing to notice is that there is no quantification over the type T at the beginning of the principle: T in theory can be different each time it appears, so we can't quantify it once and for all (even if in reality it is always the same).

Because of this, we havy to quantify over it each time separately. This can be seen in the type of the predicate P. It is a predicate on *ilist* T, but in order to be well-typed it has to quantify over T : `Type`.

Then the principle looks very similar to that of *list*, but as we said before, the case for each constructor has to quantify over T (here called A) separately. The end of the principle is also similar, but once again a quantification over T appears.

This is it: induction principles for indexed families are about having to quantify over the indices separately each time they appear somewhere.

`End` *IndexedList*.

The last thing are definitions having both parameters and indices. They adhere to the above rules: in their principles parameters are quantified over right at the start and the indices are quantified over separately each time they appear.

7.6 Maximal and minimal principles

If you look carefully at the induction principle for *le*, it seems not to follow the rules we saw above. This is because propositions (and all families of propositions) are treated a bit differently from ordinary types (and type families). Our goal in this subchapter will be to learn about this difference.

Let's compare the induction principles for *prod* and *sum* (which live in `Type`) with *and* and *or*, their counterparts living in `Prop`. Before we start, I have to admit that I lied to you once more.

I told you that for every inductive type we define, Coq generates us three induction principles. This is actually the case only for types living in `Set` and `Type`. For these in `Prop`, only one induction principles is generated.

Therefore, in our comparison we will compare the principles *prod_rect* and *sum_rect* to *and_ind* and *or_ind*.

```
Check prod_rect.
(*  ===> prod_rect :
        forall (A B : Type) (P : A * B -> Type),
          (forall (a : A) (b : B), P (a, b)) ->
            forall p : A * B, P p *)
```

```
Check sum_rect.
(*  ===> sum_rect :
        forall (A B : Type) (P : A + B -> Type),
          (forall a : A, P (inl a)) ->
          (forall b : B, P (inr b)) ->
            forall s : A + B, P s *)
```

`Check` *and_ind*.

```
(*  ===> and_ind :
        forall A B P : Prop,
          (A -> B -> P) -> A /\ B -> P *)
```

Check *or_ind*.

```
(*  ===> or_ind :
        forall A B P : Prop,
          (A -> P) -> (B -> P) -> A \/ B -> P *)
```

We could have expected them to be very alike, since the only differences between *prod*/*and* and *sum*/*or* (and the associated principles) are in the sorts. Yet they are very different. Why is this?

In order to answer, we have to introduce the distinction between maximal and minimal induction principles. All principles you have seen until now (not including *and_ind*/*or_ind*) were maximal principles. When we define an inductive type or family living in `Set` or `Type`, a maximal principle is generated for it by default. But when its sort is `Prop`, a minimal principle is generated.

It is not the case however that maximal principles are unique to `Set`s and `Type`s and minimal principles are unique to `Prop`s. We can have both for an inductive type of any sort. To understand the distinction, let's generate minimal principles for *prod* and *sum*.

We can do this with the command `Scheme`. By the way, here's a joke: Scheme is not a programming language, `Scheme` is just a Coq command for generating induction principles.

`Scheme` *prod_rect_min* := *Minimality* `for` *prod* `Sort` `Type`.
`Scheme` *sum_rect_min* := *Minimality* `for` *sum* `Sort` `Type`.

Check *prod_rect_min*.

```
(*  ===> prod_ind_min :
        forall (A B : Type) (P : Type),
          (A -> B -> P) -> A * B -> P *)
```

Check *sum_rect_min*.

```
(*  ===> sum_ind_min :
        forall (A B : Type) (P : Type),
          (A -> P) -> (B -> P) -> A + B -> P *)
```

These are much more like the principles for *and*/*or* than the maximal principles for *prod*/*sum*.

prod_rect tells us that in order to define a dependent function $\forall\ p :\ A \times B,\ P\ p$, we have to provide a dependent function $\forall\ (a :\ A)\ (b :\ B),\ P\ (a,\ b)$. On the other hand, *prod_rect_min* tells us that in order to define a nondependent function $A \times B \to P$ we have to provide another nondependent function $A \to B \to P$.

Likewise *sum_rect* tells us that in order to define a dependent function $\forall\ s :\ A + B,\ P\ s$ we have to provide two dependent functions $\forall\ a :\ A,\ P\ (inl\ a)$ and $\forall\ b :\ B,\ P\ (inr\ b)$. On the other hand, *sum_rect_min* tells us that in order to define a nondependent function $A + B \to P$ we have to provide two nondependent functions $A \to P$ and $B \to P$.

This is it! The distinction between maximal and minimal principles boils down to the fact that maximal principles are for defining dependent functions (and proving predicates), whereas the minimal ones are for nondependent functions (and proving propositions).

So, maximal principles are more general than minimal principles. The last question that remains is: why are the less general principles generated by Coq, if it can generate the more general ones?

To answer this, let's first have a look at the maximal principles for *and* and *or*. These can be generated with a different variant of the command `Scheme`.

```
Scheme and_ind_max := Induction for and Sort Prop.
Scheme or_ind_max := Induction for or Sort Prop.
```

```
Check and_ind_max.
(*  ===> and_ind_max :
        forall (A B : Prop) (P : A /\ B -> Prop),
          (forall (a : A) (b : B), P (conj a b)) ->
            forall a : A /\ B, P a *)
```

```
Check or_ind_max.
(*  ===> or_ind_max :
        forall (A B : Prop) (P : A \/ B -> Prop),
          (forall a : A, P (or_introl a)) ->
          (forall b : B, P (or_intror b)) ->
            forall o : A \/ B, P o *)
```

We now see another side of the coin: the maximal principles are not only more general, but also more complex. So, the question boils down to why should we prefer simplicity over generality.

The last thing we have to do is notice that the generality is really just the ability to prove properties of proofs of propositions. They are uninteresting because when we assume proof irrelevance, for any property P, either all proofs have it or none has.

However, what if we don't assume proof irrelevance? Even though Coq is not proof irrelevant by default — we saw we have to explicitly assume irrelevance if we want it — it is proof irrelevant at the philosophical level.

This is because we can't use proofs to construct programs. We can only use proofs to construct other proofs and, as we saw, when it comes to proofs the only thing that matters is whether they exist or not. In other words, Coq's philosophical proof irrelevance comes from the fact that it was designed to allow the actual proof irrelevance.

To sum it up: the mere ability to assume irrelevance makes investigating properties of proofs uninteresting and therefore we are not interested in having maximal induction principles generated for propositions by default. We instead prefer minimal principles.

Exercise (medium) Come up with a (metatheoretical) algorithm that can derive a minimal principle from a maximal one. Start by comparing maximal and minimal principles for some types other than *prod* and *sum*. Make sure it also works for indexed families.

Do you now understand why the induction principle for *le* looks the way it does?

7.7 Mutual induction

There's one more kind of induction we haven't covered yet: mutual induction. It is a very straightforward generalization of ordinary induction.

When we define and inductive type (or an inductive family), we can make references to everything we have defined before and also to the thing we are currently defining. But we are defining only one thing at time. Recall the definition of *even*:

```
Print even.
(*  ===> Inductive even : nat -> Prop :=
            | even0 : even 0
            | evenSS : forall n : nat, even n -> even (S (S n)) *)
```

We have a predicate for even numbers, but what about odd numbers?

```
Inductive odd : nat → Prop :=
  | odd1 : odd 1
  | oddSS : ∀ n : nat, odd n → odd (S (S n)).
```

Looks fine. These definitions say that an even number is either zero or 2 + n, where n is even and that an odd number is either one or 2 + n, where n is odd.

In English we can give a different, but equivalent definition of both even and odd numbers at once:

- 0 is even
- if n is even, then n + 1 is odd
- if n is odd, then n + 1 is even

In the above paragraph, we defined evenness and oddness not separately, but together. This kind of definition is not specific to English (or any other natural language for that matter) — it is also possible to use it in Coq and it's called mutual induction.

```
Inductive even' : nat → Prop :=
  | even'_0 : even' 0
  | even'_S : ∀ n : nat, odd' n → even' (S n)

with odd' : nat → Prop :=
  | odd'_S : ∀ n : nat, even' n → odd' (S n).
```

In definitions by mutual induction the first definition is introduced by the keyword `Inductive` and each subsequent one by the keyword `with`. The dot we have to put only after the last definition finishes. It's just like the ordinary `Inductive` definitions, but in the constructors of a type T we can reference all the other types that we are defining simultaneously.

The inductive definition of *even'* and *odd'* mimics what we have seen above (we just have to call our predicates *even'* and *odd'* because the names *even* and *odd* are already taken).

Let's see how good this definition is when compared to the previous non-inductive definitions.

`Theorem` *even_2n* :
  $\forall$ n : *nat*, *even* $n \to \exists$ k : *nat*, $n = 2 \times k$.

`Theorem` *odd_2n_1* :
  $\forall$ n : *nat*, *odd* $n \to \exists$ k : *nat*, $n = 2 \times k + 1$.

The ordinary ones are easy.

`Theorem` *even'_2n* :
  $\forall$ n : *nat*, *even'* $n \to \exists$ k : *nat*, $n = 2 \times k$.
`Proof`.
  `induction` 1.
    $\exists$ 0. `trivial`.
    `induction` *H*.
      `trivial`.
`Abort`.

This proof attempt fails because we lack the necessary inductive hypothesis saying something alone the lines of $\forall$ n : *nat*, *odd'* $n \to \exists$ k : *nat*, $n = 2 \times k + 1$. Let's try to prove this as a theorem.

`Theorem` *odd'_2n_1* :
  $\forall$ n : *nat*, *odd'* $n \to \exists$ k : *nat*, $n = 2 \times k + 1$.
`Proof`.
  `induction` 1.
    $\exists$ 0. `trivial`.
    `induction` *H*.
      `trivial`.
`Abort`.

We have the same problem here. An induction hypothesis is missing that would look just like the theorem *even'_2n* we wanted to prove above. So, if the induction doesn't work as well as we would like it to, let's check the induction principles of *even'* and *odd'*.

`Check` *even'_ind*.
```
(*  ===> even'_ind :
        forall P : nat -> Prop,
          P 0 ->
          (forall n : nat, odd' n -> P (S n)) ->
            forall n : nat, even' n -> P n *)
```

A careful glance reveals that there's no induction in this induction principle! If you check *odd'_ind*, you will see the same thing: no induction there.

This doesn't have any deep philosophical reasons. It's just that for some mystical reason Coq by default decides to generate nonmutual induction principles even for mutually inductive types. We can easily fix it.

```
Scheme even'_odd'_ind := Induction for even' Sort Prop
with odd'_even'_ind := Induction for odd' Sort Prop.
```

```
Check even'_odd'_ind.
(*  ===> even'_odd'_ind :
        forall
        (P : forall n : nat, even' n -> Prop)
        (P0 : forall n : nat, odd' n -> Prop),
          P 0 even'_0 ->
          (forall (n : nat) (o : odd' n),
            P0 n o -> P (S n) (even'_S n o)) ->
          (forall (n : nat) (e : even' n),
            P n e -> P0 (S n) (odd'_S n e)) ->
              forall (n : nat) (e : even' n), P n e *)
```

This is exactly what we need. First, we have two predicates, *P* and *P0*. *P* is the one we will be proving. It will be automatically inferred by Coq. *P0* is the induction hypothesis that was missing in our last attempt There are three cases in the principle. This is because our definition had three constructors: two for *even'* and one for *odd'*.

Notice that this principle is maximal. There's also the other principle — *odd'_even'_ind*. It's there because when we define stuff by mutual induction, we have to mutually generate the induction principles too.

Let's see how to use this principle.

```
Theorem even'_2n :
  ∀ n : nat, even' n → ∃ k : nat, n = 2 × k.
Proof.
  induction 1 using even'_odd'_ind
  with (P0 := fun (n : nat) (H : odd' n) ⇒ ∃ k : nat, n = 2 × k + 1).
    ∃ 0. trivial.
    destruct IHeven' as [k IH]. ∃ (S k). omega.
    destruct IHeven' as [k IH]. ∃ k. omega.
Qed.
```

It was much easier this time. To prove the theorem we use `induction` 1 as before, but this time we add the clause `using` *even'_odd'_ind* which tells Coq which induction principle to use. Then we use the `with` clause to tell Coq how *P0* should look like. We don't have to tell it about *P* because it can infer it from the goal.

Note that the `with` keyword here has nothing to do with the `with` we used in the inductive definition. It's just a coincidence of names. We have three cases that we solve easily. This shape of induction was what we expected at the very beginnig.

```
Theorem odd'_2n_1 :
```

$\forall$ *n* : *nat*, *odd'* *n* $\rightarrow$ $\exists$ *k* : *nat*, *n* = 2 $\times$ *k* + 1.

The second proof is quite similar. But how does the two proofs compare? The ones for *even* and *odd* were 14 lines long in total (at least for me) and weren't too hard. Quite the opposite. The ones for *even'* and *odd'* were 18 lines long in total and we're harder to carry out: we had to manually specify the predicates Coq couldn't infer from context.

So, is mutual induction useless? Not really.

```
Theorem even'_2n' :
  ∀ n : nat, even' n → ∃ k : nat, n = 2 × k
with odd'_2n_1' :
  ∀ n : nat, odd' n → ∃ k : nat, n = 2 × k + 1.
Proof.
  destruct 1.
    ∃ 0. trivial.
    destruct (odd'_2n_1' _ H) as [k IH]. ∃ (S k). omega.
  destruct 1.
    destruct (even'_2n' _ H) as [k IH]. ∃ k. omega.
Qed.
```

The above alternative proof is only 11 lines long, which is better than the proofs for *even* and *odd* (but don't take it too seriously — proof length measured in lines of code need not be a good measure of easiness).

These theorems are similar to the previous ones, but this time we stated and proved both of them at once. We also didn't need to use `induction` (and thus write any predicates explicitly in any place other than the theorem statement). This is because using the command `Theorem` in the above way adds the inductive hypotheses we need to our context.

We then proceed as above, but with `destruct` inside of induction. We also have to write a bit more in order to destruct our inductive hypotheses, but that's not a problem.

So, is mutual induction useful, if it can give us shorter proofs? I don't know a good answer to this question. I can only tell you that I haven't seen it used too often.

7.8 Custom induction principles

We saw in the previous sections how induction principles look like and that Coq generates us one (or rather, three) for every inductive definition we make. We also saw that these principles can be derived by hand.

Another fact is that these principles are not unique — quite the opposite. Usually there can be many different principles for each type: for dependent and non-dependent functions, for true induction or just for case analysis, they can even differ by their order of taking arguments.

But even more is possible. Induction principles don't need, in general, to be associated with any particular type. We can thus have principles for proving facts about relations of type P : *nat* $\rightarrow$ *list bool* $\rightarrow$ `Prop` or proving properties of the function *plus*.

We will call the principles Coq generates us "standard" and the ones we write ourselves "non-standard" or "custom". In this subchapter we will learn to devise and implement non-standard principles and to use them with standard tactics like `destruct` and `induction`.

We will also learn about the basics of functional induction, a powerful tool for proving properties of recursive functions.

```
Definition bool_ind'
  (P : bool → Prop) (f : P false) (t : P true) (b : bool) : P b :=
match b with
    | true ⇒ t
    | false ⇒ f
end.
```

This is a nonstandard case analysis principle for *bool*. The only thing that makes it different from the principle *bool_ind* is the order of arguments: it takes that of type *P false* first, whereas *bool_ind*'s first argument has type *P true*. It is obvious that it is correct, because we can prove *P* for *true* and *false* in any order we like.

```
Definition bool_ind'' :
  ∀ P : bool → Prop,
    P false → P true →
      ∀ b : bool, P b.
Proof.
  destruct b; assumption.
Qed.
```

We can also use the tactic language to simply prove the principle as a theorem. This doesn't make much difference now, but for more complex principles it is often easier to establish them that way.

But how do we use such nonstndard principles? Standard case analysis on the term *t* is usually performed by using the tactic `destruct` *t*. Our nonstandard one can be likewise performed with the tactic *destrct t* `using` *our_principle*. Of course it can also be used directly with `apply`, but we already saw that it isn't the best possibility.

Let's see both principles in action.

```
Theorem negb_involutive'' :
  ∀ b : bool, negb (negb b) = b.
Proof.
  destruct b; cbn. all: trivial.
Restart.
  destruct b using bool_ind; cbn. all: trivial.
Restart.
  destruct b using bool_ind'; cbn. all: trivial.
Restart.
  destruct b using bool_ind''; cbn. all: trivial.
Qed.
```

Here we see that the tactic `destruct` *b* is in fact equivalent to `destruct` *b* `using` *bool_ind*, in which the standard induction principle is named explicitly. In these two equivalent cases, our first subgoal is *true* = *true* and the second is *false* = *false*. On the contrary, when we use one of our custom principles, our first goal is of the form *false* = *false* and the second is *true* = *true*.

Let's see a custom principle for *nat*.

```
Fixpoint nat_ind_2
  (P : nat → Prop) (H0 : P 0) (H1 : P 1)
  (HSS : ∀ n : nat, P n → P (S (S n)))
  (n : nat) : P n :=
match n with
    | 0 ⇒ H0
    | 1 ⇒ H1
    | S (S n') ⇒ HSS n' (nat_ind_2 P H0 H1 HSS n')
end.
```

```
Fixpoint nat_ind_2'
  (P : nat → Prop) (H0 : P 0) (H1 : P 1)
  (HSS : ∀ n : nat, P n → P (S (S n)))
  (n : nat) : P n.
Proof.
  destruct n as [| [| n']].
    apply H0.
    apply H1.
    apply HSS. apply nat_ind_2'; assumption.
Defined.
```

In case of *nat*, we can customize our induction principle more than by just changing the order of the arguments. For example, we can do "induction by 2's". This means that rather than starting at 0 and applying *S* once in each step, we have two base cases, 0 and 1, and we apply *S* twice in each step.

This principle is correct because we can generate all natural numbers this way: starting from 0 and applying *S* twice in each step we can generate all even numbers, whereas starting from 1 and applying *S* twice in each step we can generate all odd numbers. Because each natural number is either even or odd, we see that our custom principle covers all cases.

We also see that when our principles have recursion, defining them with tactics is easier than without tactics.

```
Require Import Div2.
```

Div2 is a module that contains the function *div2*, which performs integer division by two. Here is its definition:

```
Eval compute in div2.
(*  ===> = fix div2 (n : nat) : nat :=
          match n with
```

```
        | 0 => 0
        | 1 => 0
        | S (S n') => S (div2 n')
      end
      : nat -> nat *)
```

Note: we use `Eval compute` instead of `Print` because *div2* is only a notation.

We see that *div2* is defined differently than *plus* or *mult*. These two have just two cases in their respective `match`es: 0 and *S n'*, whereas *div2* has 0, 1 and *S* (*S n'*). Let's try proving some theorem.

```
Theorem lt_div2' :
  ∀ n : nat, 0 < n → div2 n < n.
Proof.
  induction n as [| n']; cbn; intros; auto.
    destruct n'; cbn in *; auto.
Restart.
  induction n as [| | n'] using nat_ind_2; cbn; intros; auto.
    destruct n' as [| [| n'']]; cbn in *; auto.
      unfold lt in *. apply le_n_S. apply le_S. apply IHn'. apply le_n_S.
        apply le_0_n.
Qed.
```

Trying induction on n using the standard induction principle doesn't help us much. A kind of a "mismatch" occurs: when we have *n'* in the hypothesis, we have `match` *n'* `with` ... in the goal and vice versa. This is because the shape of induction in the standard principle is different from the shape of recursion that was used to define *div2*. If we use the custom principle, we get rid of that problem and we are able to prove the goal easily.

So, our nonstandard principles are no different from the standard ones, we just have to create them manually and type a bit more in order to use them... or do we?

```
Require Import List.
Import ListNotations.
```

```
Fixpoint swap_blocks {A : Type} (l : list A) : list A :=
match l with
    | [] ⇒ []
    | [x] ⇒ [x]
    | x :: y :: t ⇒ y :: x :: swap_blocks t
end.
```

```
Compute swap_blocks [1; 2; 3; 4; 5].
(*  ===> = [2; 1; 4; 3; 5] : list nat *)
```

Note: `Compute` is equivalent to `Eval compute in`, but shorter.

This is a function that swaps places of adjacent elements in a list. It is easy to see that it is an involution, which means that applying it twice gives us the original list. Let's try to prove that.

```
Theorem swap_blocks_involutive :
  ∀ (A : Type) (l : list A),
    swap_blocks (swap_blocks l) = l.
Proof.
  induction l as [| h t]; cbn; intros.
    trivial.
    destruct t; cbn in *.
      trivial.
Abort.
```

We see that we have the same problem we had with *div2*. If we have t in the induction hypothesis, then it's going to appear inside a `match` in the goal. We could prove this theorem by writing a custom induction principle for lists, but what if we are too lazy to do that?

Don't worry, there's a solution called *functional* `induction`. It is a tactic which performs induction that fits the recursive structure of our function perfectly. We can use it like this:

```
Functional Scheme swap_blocks_ind :=
  Induction for swap_blocks Sort Prop.

Check swap_blocks_ind.
(*  ===> swap_blocks_ind :
        forall (A : Type) (P : list A -> list A -> Prop),
          (forall l : list A, l =  -> P [] []) ->
          (forall (l : list A) (x : A) (l0 : list A),
            l = x :: l0 -> l0 =  -> P [x] [x]) ->
          (forall (l : list A) (x : A) (l0 : list A),
            l = x :: l0 -> forall (y : A) (t : list A),
            l0 = y :: t -> P t (swap_blocks t) ->
              P (x :: y :: t) (y :: x :: swap_blocks t)) ->
          forall l : list A, P l (swap_blocks l) *)
```

The command *Functional* `Scheme` generates an induction principle that fits the shape of our function's recursion. The principle may look intimidating, but if you take a closer look, it just repeats the cases found in the `match` in *swap_blocks*' definition. We can use it like this (but first we have to import the *Recdef* module):

```
Require Import Recdef.

Theorem swap_blocks_involutive :
  ∀ (A : Type) (l : list A),
    swap_blocks (swap_blocks l) = l.
Proof.
  intros. functional induction @swap_blocks A l; cbn.
    trivial.
    trivial.
    rewrite IHl0. trivial.
Qed.
```

A piece of cake, wasn't it? But we can be even more lazy:

```
Function swap_blocks' {A : Type} (l : list A) : list A :=
match l with
    | [] => []
    | [x] => [x]
    | x :: y :: t => y :: x :: swap_blocks' t
end.
```

```
Theorem swap_blocks'_involutive :
  forall (A : Type) (l : list A),
    swap_blocks' (swap_blocks' l) = l.
Proof.
  intros. functional induction @swap_blocks' A l; cbn.
    trivial.
    trivial.
    rewrite IHl0. trivial.
Qed.
```

We can replace the command `Fixpoint` with `Function` to make Coq generate us the same principle that the *Functional* `Scheme` command (and many other things, too).

Exercise (easy) Prove the following nonstandard induction principles for lists. What do they mean? Give an informal description.

```
Theorem list_ind_rev :
  forall (A : Type) (P : list A -> Prop) (Hnil : P [])
  (Hsnoc : forall (x : A) (l : list A), P l -> P (l ++ [x]))
    (l : list A), P l.
```

```
Theorem list_ind_app :
  forall (A : Type) (P : list A -> Prop)
  (Hnil : P []) (Hsingl : forall x : A, P [x])
  (IH : forall l l' : list A, P l -> P l' -> P (l ++ l'))
    (l : list A), P l.
```

Exercise (easy) `Function take {A : Type} (n : nat) (l : list A) : list A :=`

```
match n, l with
    | 0, _ => []
    | _, [] => []
    | S n', h :: t => h :: take n' t
end.
```

take n l takes at most *n* initial elements from the list *l*. Prove some of its properties using standard induction principles and then reprove them using functional induction. Which method is easier?

If you are not lazy, write a custom induction principle that would fit *take*'s recursion structure. Which method requires least work to get things done?

Theorem *take_length'* :
  $\forall$ (A : Type) (n : *nat*) (l : *list* A),
    length $l \leq n \rightarrow$ *take* n $l = l$.

Theorem *length_take* :
  $\forall$ (A : Type) (n : *nat*) (l : *list* A),
    $n \leq$ *length* $l \rightarrow$ *length* (*take* n l) $= n$.

Theorem *length_take'* :
  $\forall$ (A : Type) (n : *nat*) (l : *list* A),
    length (*take* n l) $\leq n$.

7.9 Case analysis on non-inductive types

Two subchapters ago we have said the we can prove that all elements of a finite type have some property by using case analysis, but actually we only did this for an inductive type.

In this subchapter, we will see that we can do case analysis on a finite type even when it is not defined inductively (we won't define "finite" formally here). But before we do that, we have to discuss the equality of functions.

Theorem *S_plus_1_l_eq* :
  $S =$ fun n : *nat* $\Rightarrow 1 + n$.
Proof.
  cbn. reflexivity.
Qed.

By default, $x = y$ holds only when x and y are convertible, meaning both reduce to the same thing when computed. This is a very weak notion of equality that is often not satisfying.

Theorem *S_plus_1_r_eq* :
  $S =$ fun n : *nat* $\Rightarrow n + 1$.
Proof.
  cbn. *Fail* reflexivity.
Abort.

Theorem *S_plus_1_r_ext_eq* :
  $\forall$ n : *nat*, S $n = n + 1$.
Proof.
  intros. rewrite $\leftarrow$ *plus_n_Sm*, $\leftarrow$ *plus_n_O*. trivial.
Qed.

When trying to prove that S equals fun n : *nat* $\Rightarrow n + 1$, Coq tells us it is 'Unable to unify "n + 1" with "S n"'. These terms are not not convertible terms and thus the two functions can't be proven equal. This is very disappointing because we can easily show that they compute they very same thing.

Here comes our saviour: the Axiom of Functional Extensionality.

```
Require Import FunctionalExtensionality.

Check @functional_extensionality.
(*  ===> @functional_extensionality :
        forall (A B : Type) (f g : A -> B),
          (forall x : A, f x = g x) -> f = g *)
```

This axiom asserts that if two functions compute the same value for each argument, then they are equal. This axiom has been proven consistent with Coq's logic, so we can use it safely. Note: there also is a version of this axiom for dependent functions.

```
Theorem S_plus_1_r_eq :
  S = fun n : nat ⇒ n + 1.
Proof.
  extensionality n. apply S_plus_1_r_ext_eq.
Qed.
```

If we need to prove two functions equal, we can use the `extensionality` tactic, which applies the axiom for us. We are then left to prove that both functions compute the same thing.

One last thing we need is to define the constant function:

```
Definition const {A B : Type} (b : B) (_ : A) : B := b.
```

Armed with the axiom and this definition, we can proceed to the clou of this subchapter: we will establish a case analysis principle for boolean functions.

How many ways are there to assign two values to two arguments?

```
Lemma bool_fun_char :
  ∀ f : bool → bool,
    f = @id bool ∨ f = negb ∨ f = const true ∨ f = const false.
Proof.
  intros. case_eq (f true); case_eq (f false); intros.
    right; right; left. extensionality x. destruct x; auto.
    left. extensionality x. destruct x; auto.
    right; left. extensionality x. destruct x; auto.
    do 3 right. extensionality x. destruct x; auto.
Qed.
```

Well, you guessed it right — there are only four: identity, negation and two constant ones. We can prove this easily by looking at the values of *f true* and *f false*.

```
Theorem bool_fun_ind :
  ∀ P : (bool → bool) → Prop,
    P (@id bool) → P negb → P (const true) → P (const false) →
      ∀ f : bool → bool, P f.
```

```
Proof.
  intros. destruct (bool_fun_char f) as [H' | [H' | [H' | H']]];
  subst; assumption.
Qed.
```

The last thing left is to pack the above lemma into a case analysis principle. We will call it *bool_fun_ind*, even though it has nothing to do with induction.

This is it! We can now do case analysis when proving properties of boolean functions. But there's a problem: I can't think of any useful property of boolean functions that can be proved by case analysis but not with some other method.

Exercise (medium) Try to come up with such a property.

7.10 Functions and functional relations

After you have learned about the commands `Function` and *Functional* `Scheme`, you may be wondering how they work. We will try to answer this question now, but it will be easier if we first see how we can use inductive families to represent functions.

First we have to explicitly spell out what a function is. There are two widely used definitions:

- the computer scientist's one, which equates functions with algorithms. They are always computable, which means we can implement and run them on a computer to get the result. Coq functions are of this kind.
- the mathematician's one, which equates functions with functional relations, which are relations with additional properties. These need not be computable.

The second meaning of "function" stems from set theory and is widely used in mathematics. Here a function is synonyomous with its graph. A graph of a function is a relation that relates the function's inputs to its output.

Let's see an example underlining the difference between these two.

```
Function div2 (n : nat) : nat :=
match n with
    | 0 ⇒ 0
    | 1 ⇒ 0
    | S (S n') ⇒ S (div2 n')
end.
```

Note: we use `Function` only to get the functional induction principle so that we will later be able to compare it to our own.

div2 is an ordinary structurally recursive function. Nothing fancy.

```
Inductive div2_rel : nat → nat → Prop :=
```

| $div2\_rel\_even : \forall\ n : nat,\ div2\_rel\ (2 \times n)\ n$
| $div2\_rel\_odd : \forall\ n : nat,\ div2\_rel\ (2 \times n + 1)\ n.$

div2_rel is an inductive relation defined like this:

- the result of dividing $2 \times n$ by 2 is n
- the result of dividing $2 \times n + 1$ by 2 is n

The difference is quite stark: the recursive one (*div2*) tells us how to divide a number by two (how to compute the result). The inductive one (*div2_rel*) tells us what is the relation between inputs and outputs of a function that divides its argument by two. Nonetheless these two formulations of division are somehow equivalent.

`Lemma` $div2\_rel\_SS$:
$\forall\ n\ m : nat,\ div2\_rel\ n\ m \rightarrow div2\_rel\ (S\ (S\ n))\ (S\ m).$

This is just a lemma for the proper proof.

`Theorem` $div2\_rel\_correct$:
$\forall\ n\ m : nat,\ div2\ n = m \rightarrow div2\_rel\ n\ m.$

This theorem states that *div2_rel* contains the graph of *div2*.

`Theorem` $div2\_rel\_complete$:
$\forall\ n\ m : nat,\ div2\_rel\ n\ m \rightarrow div2\ n = m.$

And this one says that the graph of *div2* contains *div2_rel*. Thus *div2_rel* really is the graph of *div2*. *div2_rel* may also be thought of as a specification for *div2*, but specifying functions isn't everything we can achieve using inductive relations. Another thing is specifying partial functions.

`Module` *Zfact*.

`Require Import` *ZArith*.

`Open Scope` *Z*.

`Inductive` $Zfact : Z \rightarrow Z \rightarrow$ `Prop` $:=$
| $Zfact\_0 : Zfact\ 0\ 1$
| $Zfact\_S : \forall\ (k\ r : Z),$
$Zfact\ k\ r \rightarrow Zfact\ (k + 1)\ ((k + 1) \times r).$

This looks like a specification for the factorial function.

`Theorem` $Zfact\_nonneg$:
$\forall\ n : nat,\ Zfact\ (Z.of\_nat\ n)\ (Z.of\_nat\ (fact\ n)).$

And indeed, for nonnegative integers it is exactly the factorial function. But what about the negative ones?

`Theorem` $Zfact\_neg$:
$\forall\ n\ k : Z,\ n < 0 \rightarrow \neg\ Zfact\ n\ k.$

It turns out that if $n < 0$, then it isn't related to any result at all and therefore this relation can't be a graph of any function that is definable in Coq. We may call this the "programming newbie's factorial".

This example clearly shows that definitions by recursion and induction differ. The latter can describe more things than the former (another example would be nonterminating functions — something that's not there in Coq, but is very frequent in other programming languages), but this comes at the cost of computability — we can't compute with *Zfact*.

`End` *Zfact*.

Exercise (easy) Consider the following function on natural numbers:

- f(0) = 1
- f(1) = 1
- f(n) = f(n/2) if n is even
- f(n) = f(3n + 1) if n is odd

Is this function definable in Coq? Implement it using recursion or induction, whichever is easier. Then show that f(42) = 1. Can you prove that f(n) = 1 for all n?

Now that we know inductive relations can represent graphs of functions, we can ask ourselves: what's the point of it? And what does it have to do with functional induction? Well, it turns out that quite a lot...

Consider this alternative definition of *div2*'s graph.

```
Inductive R_div2' : nat → nat → Prop :=
  | R_div2'_0 : R_div2' 0 0
  | R_div2'_1 : R_div2' 1 0
  | R_div2'_2 :
      ∀ n r : nat,
        R_div2' n r → R_div2' (S (S n)) (S r).
```

Note: in the names of the following functions we utilize the naming conventions of `Function` and *Functional* `Scheme`, but add a ' at the end.

No doubt this is the graph of *div2*, because this definition follows precisely *div2*'s definition: there's one constructor for each case in its pattern match. Therefore, unlike the original *div2_rel*, it is not well suited for a specification of *div2*.

So, what's the point?

```
Check R_div2'_ind.
(*  ===> R_div2'_ind :
        forall P : nat -> nat -> Prop,
          P 0 0 ->
          P 1 0 ->
```

```
      (forall n r : nat, R_div2' n r ->
        P n r -> P (S (S n)) (S r)) ->
          forall n n0 : nat, R_div2' n n0 -> P n n0 *)
```

```
Check div2_ind.
(*  ===> div2_ind :
        forall P : nat -> nat -> Prop,
          (forall n : nat, n = 0 -> P 0 0) ->
          (forall n n0 : nat, n = S n0 -> n0 = 0 -> P 1 0) ->
          (forall n n0 : nat,
            n = S n0 -> forall n' : nat, n0 = S n' ->
              P n' (Init.Nat.div2 n') ->
              P (S (S n')) (S (Init.Nat.div2 n'))) ->
               forall n : nat, P n (Init.Nat.div2 n) *)
```

The point is that *R_div2'_ind*, the induction principle for the graph of *div2* is very much like *div2_ind*, the functional induction principle for *div2*. This fact is overshadowed by weirdness of *div2_ind* (it was, after all, generated automagically). There's something very deep going on here.

Let's ask ourselves a related question: how to prove a property of a given function f by induction?

The simple answer is: we have to apply the right induction principle and then finish the proof. But which induction principle is the right one? A philosophical answer would be: the one that best matches the function's recursion shape. But which one is it?

Recall that every induction principle tells us how to use a value of some inductively defined type (or type family) to define dependent functions. Therefore, every induction principle is associated to some type or type family. So the question boils down to this: which inductive type has the most similar shape to that of f's recursion shape?

The ultimate answer is: the graph of f (using the definition that follows exactly f's recursion's shape). Now that you know this illuminating fact, let's see how to derive the functional induction principle for *div2* by hand.

```
Print R_div2.
(*  ===> Inductive R_div2 : nat -> nat -> Set :=
            | R_div2_0 : forall n : nat,
                n = 0 -> R_div2 0 0
            | R_div2_1 : forall n n0 : nat,
                n = S n0 -> n0 = 0 -> R_div2 1 0
            | R_div2_2 : forall n n0 : nat,
                n = S n0 -> forall n' : nat,
                n0 = S n' -> forall _res : nat,
                R_div2 n' _res -> R_div2 (S (S n')) (S _res) *)
```

The first thing the commands `Function` and *Functional* `Scheme` do is define the graph of the function in question. *R_div2* is the graph of *div2* as generated by *Functional* `Scheme`.

It's a bit clumsier than our definition (note that, for example, the quantified variable n : *nat* in the constructor *R_div2_0* is not used), but also more general (note that its sort is `Set`).

```
Theorem R_div2'_correct :
  ∀ n m : nat, div2 n = m → R_div2' n m.
Proof.
  induction n as [| | n'] using nat_ind_2; cbn; intros; subst;
  constructor; auto.
Qed.
```

```
Theorem R_div2'_complete :
  ∀ n m : nat, R_div2' n m → div2 n = m.
Proof.
  induction 1; cbn; auto.
Qed.
```

Then we prove that *R_div2'* really is a graph of *div2*.

```
Check R_div2_correct.
(*  ===> R_div2_correct :
        forall n _res : nat, _res = div2 n -> R_div2 n _res *)
```

```
Check R_div2_complete.
(*  ===> R_div2_complete :
        forall n _res : nat, R_div2 n _res -> _res = div2 n *)
```

Functional `Scheme` doesn't do it, but `Function` does.

```
Theorem div2_ind' :
  ∀ P : nat → nat → Prop,
    P 0 0 →
    P 1 0 →
    (∀ n : nat, P n (div2 n) → P (S (S n)) (S (div2 n))) →
      ∀ n : nat, P n (div2 n).
Proof.
  intros. apply R_div2'_ind.
    assumption.
    assumption.
    intros. apply R_div2'_complete in H2. subst. apply H1.
      assumption.
    apply R_div2'_correct. reflexivity.
Qed.
```

We can prove the induction principle for *div2* by using the induction principle for *R_div2'* and the fact that *R_div2'* is the graph of *div2*.

```
Theorem div2_equation' :
  ∀ n : nat,
    div2 n = match n with
```

```
              | 0 ⇒ 0
              | 1 ⇒ 0
              | S (S n') ⇒ S (div2 n')
          end.
Proof.
  destruct n; reflexivity.
Qed.
```

In general to prove *div2_ind'* we could need *div2_equation'*, which is called a fixed point equation for *div2*. Here it wasn't needed, because it can be obtained by just unfolding *div2*, but in general, when defining stuff by well-founded recursion, we wouldn't get that equation for free. We would have to prove it.

```
Theorem lt_div2'' :
  ∀ n : nat, 0 < n → div2 n < n.
Proof.
  intro. apply div2_ind'.
    omega.
    omega.
    intros. destruct n0 as [| [| n']]; cbn in *.
      omega.
      omega.
      apply lt_n_S. apply lt_trans with (S (S n')); omega.
Restart.
  intro. functional induction @div2 n.
    omega.
    omega.
    intros. destruct n' as [| [| n]]; cbn in *.
      omega.
      omega.
      apply lt_n_S. apply lt_trans with (S (S n)); omega.
Qed.
```

Here is an example use of the induction principle we crafted for ourselves. Applying it works exactly like using the tactic *functional* `induction`.

Exercise (easy) `Fixpoint div2l {A : Type} (l : list A) : list A :=`

```
match l with
    | [] ⇒ []
    | [_] ⇒ []
    | h :: _ :: t ⇒ h :: div2l t
end.
```

Here's a function *div2l*, that divides a list by 2 (whatever that means). Define its graph by induction, prove it really is its graph. Derive the fixed point equation and functional induction principle for *div2l* by hand. Then prove the theorem.

```
Theorem div2l_length :
  ∀ (A : Type) (l : list A),
    l ≠ [] → length (div2l l) < length l.
```

7.11 Generalizing the induction hypothesis

Proof by induction is not always straightforward. An often encountered obstacle is the induction hypothesis' lack of generality. This lack of generality can actually arise in two independent ways:

- bad use of tactics
- too weak theorem statement

This subchapter will be about dealing with them. Let's start with the first possibility.

```
Fixpoint plus' (n m : nat) : nat :=
match m with
    | 0 ⇒ n
    | S m' ⇒ plus' (S n) m'
end.
```

This is a tail recursive version of +. Let's try to prove something about it.

```
Theorem plus'_S :
  ∀ n m : nat, plus' (S n) m = S (plus' n m).
Proof.
  induction m as [| m']; cbn; intros.
    trivial.
    rewrite IHm'.
Abort.
```

We failed, but what happened? We tried to prove the theorem by induction on m, because it is the main argument of *plus'*. The base case was trivial, but we got stuck in the inductive step case. We can in no way help ourselves by applying or rewriting the induction hypothesis.

This is because n is fixed in the induction hypothesis and it is so because when we did `induction` m, all the things being quantified over before m got introduced into the context. We can solve this problem easily by using the tactic `generalize dependent`.

```
Theorem plus'_S :
  ∀ n m : nat, plus' (S n) m = S (plus' n m).
Proof.
  intros. generalize dependent n. generalize dependent m.
  induction m as [| m']; cbn; intros.
```

```
    trivial.
    specialize (IHm' (S n)). assumption.
Qed.
```

The first line of the proof script has the effect of swapping n and m in the theorem statement. Because of this, when we do `induction` m, the induction hypothesis quantifies over n, so we can instantiate it with $S\ n$ and finish the proof. Last time we weren't able to do that, because n was fixed.

```
Theorem plus'_S_v2 :
  ∀ m n : nat, plus' (S n) m = S (plus' n m).
Proof.
  induction m as [| m']; cbn; auto.
Qed.
```

A different way of solving the same problem is to simply change the theorem statement. It is easier and quicker than trying to achieve the same effect through `intros` and `generalize dependent`, but I don't consider it to be a good one. I recommend that the order of quantification over the arguments follow the order of the arguments in the function's definition.

Exercise (easy) Prove *plus'_assoc_1* without generalizing the induction hypothesis. Prove *plus'_assoc_2* by generalizing the induction hypothesis. Prove *plus'_assoc_3*, which quantifies over c first.

Which proof was the easiest one?

```
Theorem plus'_assoc_1 :
  ∀ a b c : nat, plus' a (plus' b c) = plus' (plus' a b) c.
Theorem plus'_assoc_2 :
  ∀ a b c : nat, plus' a (plus' b c) = plus' (plus' a b) c.
Theorem plus'_assoc_3 :
  ∀ c a b : nat, plus' a (plus' b c) = plus' (plus' a b) c.
```

Exercise (hard) Write a tactic *gen_ind* that performs induction on its argument after having generalized the induction hypothesis. Make sure you can prove the theorem *plus'_S_v3* with it.

```
Theorem plus'_S_v3 :
  ∀ n m : nat, plus' (S n) m = S (plus' n m).
Proof.
  gen_ind m.
Qed.
```

The first kind of situation in which our induction hypothesis was not general enough arised from careless use of `intros` and `induction`. It was neither deep nor fundamental and we easily mitigated it by simple tactic manipulation.

The second kind of lack of generality is more fundamental and arises when we are trying to prove too weak a theorem. It may appear paradoxical at first, but sometimes proving a stronger theorem is easier than proving a weaker one. Let's see this in an example.

```
Print rev.
(*  ===> rev = fun A : Type =>
        fix rev (l : list A) : list A :=
        match l with
            | [] => []
            | x :: l' => rev l' ++ [x]
        end
        : forall A : Type, list A -> list A *)
```

Recall that *rev* is a function that reverses its argument. It is very inefficient: it's running time is O(n^2). Let's try to improve on that by writing a tail recursive version.

```
Fixpoint rev_acc {A : Type} (l acc : list A) : list A :=
match l with
    | [] => acc
    | h :: t => rev_acc t (h :: acc)
end.
```

rev_acc works by putting its argument's head to the accumulator until its argument is empty, in which case it returns the accumulator. It's clear that if we call it with an empty accumulator, then in result we will get its argument reversed. Let's try to prove that.

```
Theorem rev_acc_spec_weak :
  ∀ (A : Type) (l : list A), rev_acc l [] = rev l.
Proof.
  induction l as [| h t]; cbn.
    trivial.
Abort.
```

The proof failed miserably. Our goal mentions *rev_acc t* [*h*], but the induction hypothesis is about *rev_acc t* []. There is no way to solve this problem the way we did before: by rearranging the order of quantifications or by using `generalize dependent`.

This is a problem of fundamentally different nature — the induction hypothesis is too weak because the theorem itself is too weak. To go around this, we have to devise a stronger version of the theorem, but how to do this?

Let's take a glance at the arguments of *rev_acc*: the second one is []. Let's try to generalize the theorem to an arbitrary *acc* : *list A*. If we run *rev_acc* with an arbitrary accumulator, it will prepend its reversed argument to the accumulator and then return it. Let's try this as our theorem.

```
Theorem rev_acc_spec_strong :
  ∀ (A : Type) (l acc : list A),
    rev_acc l acc = rev l ++ acc.
Proof.
```

```
  induction l as [| h t]; cbn; intros.
    trivial.
    rewrite IHt. rewrite ← app_assoc. cbn. trivial.
Qed.
```

This time our induction hypothesis talks about a general *acc* and not just [], so we can easily use it with `rewrite`. Now we can derive the weaker version of the theorem we first tried to prove.

```
Theorem rev_acc_spec_weak :
  ∀ (A : Type) (l : list A), rev_acc l [] = rev l.
Proof.
  intros. rewrite rev_acc_spec_strong. rewrite app_nil_r. trivial.
Qed.
```

The technique presented above allows us to explain the paradoxical fact that proving a stronger theorem may be easier than proving a weaker one, at least in the case of proofs by induction.

The explanation is: the stronger the theorem, the stronger the induction hypothesis. Sometimes the increase in the induction hypothesis' strength will be greater than the increase in the theorem's difficulty and thus the proof will be easier. Other times, the stronger theorem will actually be harder to prove.

Exercise (easy) *app'* is a tail recursive version of *app*. Prove that they are equal.

```
Fixpoint revapp {A : Type} (l1 l2 : list A) : list A :=
match l1 with
    | [] ⇒ l2
    | h :: t ⇒ revapp t (h :: l2)
end.

Definition app' {A : Type} (l1 l2 : list A) : list A :=
  revapp (revapp l1 []) l2.

Theorem app'_spec :
  ∀ (A : Type) (l1 l2 : list A),
    app' l1 l2 = app l1 l2.
```

7.12 Technical shortcomings of `induction`

`induction` is a very smart tactic: it can introduce quantified variables into the context, infer the predicate we're trying to prove and many more. But there is one situation in which its behaviour is really stupid and unexpected.

Let's see a somewhat complicated example, taken from Coq'Art. We will define a simple imperative programming language and then try to prove a fact by induction.

```
Section little_semantics.
```

```
Axioms Var aExp bExp : Set.
```

Note: there's a reason for using `Axioms` instead of, say, `Variables`, that will be explained later.

We will use *Var* to represent variables, *aExp* for arithmetic expressions and *bExp* for boolean expressions.

```
Inductive instr : Set :=
  | Skip : instr
  | Assign : Var → aExp → instr
  | Seq : instr → instr → instr
  | While : bExp → instr → instr.
```

There are three kinds of instructions: the empty instruction *Skip*, assignment *Assign* and the while loop *While*. They can be sequenced using *Seq*.

```
Axiom
  (state : Set)
  (update : state → Var → Z → option state)
  (evalA : state → aExp → option Z)
  (evalB : state → bExp → option bool).
```

We also assume *state* for representing the state of programs. This state will be updated with *update*. We can "evaluate" arithmetic expressions using *evalA* and boolean expressions using *evalB*.

```
Inductive exec : state → instr → state → Prop :=
  | execSkip :
      ∀ s : state, exec s Skip s
  | execAssign :
      ∀ (s1 s2 : state) (v : Var) (a : aExp) (k : Z),
        evalA s1 a = Some k → update s1 v k = Some s2 →
          exec s1 (Assign v a) s2
  | execSeq :
      ∀ (s1 s2 s3 : state) (i1 i2 : instr),
        exec s1 i1 s2 → exec s2 i2 s3 →
          exec s1 (Seq i1 i2) s3
  | execWhileTrue :
      ∀ (s1 s2 s3 : state) (i : instr) (be : bExp),
        evalB s1 be = Some true →
        exec s1 i s2 →
        exec s2 (While be i) s3 →
          exec s1 (While be i) s3
  | execWhileFalse :
      ∀ (s : state) (i : instr) (be : bExp),
        evalB s be = Some false → exec s (While be i) s.
```

This is the relation that describes how to execute programs. *Skip* does nothing, *Assign* and *While* work as expected. *Seq i1 i2* means "execute i1 and then i2".

```
Hint Constructors instr exec.
```

We want to prove a rule for while from Hoare logic. It says, roughly, that if we execute a while loop and if a loop invariant P holds before the loop and after each iteration, then after the last iteration P also holds and moreover the condition of the while loop is false (for otherwise the loop wouldn't have terminated).

```
Theorem HoareWhileRule :
  ∀ (P : state → Prop) (b : bExp) (i : instr) (s s' : state),
    (∀ s1 s2 : state, P s1 → evalB s1 b = Some true →
      exec s1 i s2 → P s2) → P s → exec s (While b i) s' →
        P s' ∧ evalB s' b = Some false.
Proof.
  intros. induction H1.
Restart.
  intros. destruct H1.
Restart.
  intros. inversion H1; subst.
Restart.
  intros. remember (While b i) as w.
  induction H1; intros; inversion Heqw; subst; eauto.
Qed.
```

We attempt to prove the theorem by induction on the hypothesis *H1* : *exec s* (*While b i*) *s'* as this seems to be the only reasonable option. However, we fail miserably: `induction` *H1* gives us 5 subgoals that we can't solve.

Clearly *H1* could have been created only with the constructors *execWhileTrue* and *execWhileFalse*, but `induction` wants us to consider all 5 constructors. If we try `destruct`, the same happens, but when we use `inversion`, we get only 2 subgoals as expected. Why is this so?

It's all very simple: if we do induction on a term whose type contains non-variables, `induction` and `destruct` both mess up and "forget" the form of this particular term, leaving us with unprovable subgoals.

In our case, the type of *H1* is *exec s* (*While b i*) *s'*. The arguments of *exec* are *s*, *While b i* and *s'*, of which *s* and *s'* are variables and *While b i* is not. So, both `induction` and `destruct` forget all information regarding the term *While b i*.

The situation is easy to deal with: we have to replace the problematic term *While b i* with a fresh variable *w*, keeping an equality saying that *w* = *While b i*. This little bookkeeping saves us from losing the desired information.

Two questions remain: why `inversion` succeeds, when `induction` and `destruct` fail? The answer is: `inversion` was crafted specially for not losing information in such cases. So, why doesn't `destruct` and `induction` work that way?

As for `destruct`, if you want it not to lose information, you can use `inversion` anyway. As for `induction`: it is supposed to be a basic and simple tactic, whereas `inversion` tends to generate very large proofterms. For example, in the above proofscript, after `intros` the proofterm is only 4 lines long, but after `intros`. `inversion` *H1* it is 214 lines long (you can check this yourself with the command `Show Proof`). This is the price of `inversion`.

Exercise (hard) Write a tactic *replace_nonvars H* that replaces all nonvariables with variables in the hypothesis *H* and adds appropriate equalities to the context. Combine this tactic with the tactic *generalize_IH* that you implemented in one of the earlier exercises and some more tinkering to get a tactic *ind H*, which:

- generalizes the inductive hypothesis
- replaces nonvariables with variables in the inductive hypothesis
- does induction
- inverses and substitutes all necessary equalities
- cleans the mess it may have created in the context
- finishes off easy subgoals with some basic automation

Your tactic should be able to solve *HoareWhilRule'* as shown below. How big is the proofterm it generates? Measure it in lines (mine is 314 lines).

```
Theorem HoareWhileRule' :
  ∀ (P : state → Prop) (b : bExp) (i : instr) (s s' : state),
    (∀ s1 s2 : state, P s1 → evalB s1 b = Some true →
      exec s1 i s2 → P s2) → P s → exec s (While b i) s' →
        P s' ∧ evalB s' b = Some false.
Proof.
  intros. ind H1.
Qed.

End little_semantics.
```

7.13 Grading

To get a 3 you need to prove all theorems that are stated, but whose proofs are omitted, and solve all exercises which are marked as easy.

To get a 4 you need to get a 3 and additionally solve all exercises which are marked as medium.

To get a 5 you need to get a 4 and additionally solve all exercises which are marked as hard.

Rozdział 8

X1: Logika boolowska

Zadania z funkcji boolowskich, sprawdzające radzenie sobie w pracy z enumeracjami, definiowaniem funkcji przez pattern matching i dowodzeniem przez rozbicie na przypadki.

Chciałem, żeby nazwy twierdzeń odpowiadały tym z biblioteki standardowej, ale na razie nie mam siły tego ujednolicić.

`Section` *boolean_functions*.
`Variables` *b b1 b2 b3* : *bool*.

8.1 Definicje

Zdefiniuj następujące funkcje boolowskie:

- *negb* (negacja)
- *andb* (koniunkcja)
- *orb* (alternatywa)
- *implb* (implikacja)
- *eqb* (równoważność)
- *xorb* (alternatywa wykluczająca)
- *nor*
- *nand*

`Notation` "b1 && b2":= (*andb b1 b2*).
`Notation` "b1 || b2":= (*orb b1 b2*).

8.2 Twierdzenia

Udowodnij, że zdefiniowane przez ciebie funkcje mają spodziewane właściwości.

Właściwości negacji

`Theorem` $negb\_inv : negb\ (negb\ b) = b.$

`Theorem` $negb\_ineq : negb\ b \neq b.$

Eliminacja

`Theorem` $andb\_elim\_l : b1\ \&\&\ b2 = true \to b1 = true.$

`Theorem` $andb\_elim\_r : b1\ \&\&\ b2 = true \to b2 = true.$

`Theorem` $andb\_elim : b1\ \&\&\ b2 = true \to b1 = true \land b2 = true.$

`Theorem` $orb\_elim : b1\ ||\ b2 = true \to b1 = true \lor b2 = true.$

Elementy neutralne

`Theorem` $andb\_true\_neutral\_l : true\ \&\&\ b = b.$

`Theorem` $andb\_true\_neutral\_r : b\ \&\&\ true = b.$

`Theorem` $orb\_false\_neutral\_l : false\ ||\ b = b.$

`Theorem` $orb\_false\_neutral\_r : b\ ||\ false = b.$

Anihilacja

`Theorem` $andb\_false\_annihilation\_l : false\ \&\&\ b = false.$

`Theorem` $andb\_false\_annihilation\_r : b\ \&\&\ false = false.$

`Theorem` $orb\_true\_annihilation\_l : true\ ||\ b = true.$

`Theorem` $orb\_true\_annihilation\_r : b\ ||\ true = true.$

Łączność

`Theorem` $andb\_assoc : b1\ \&\&\ (b2\ \&\&\ b3) = (b1\ \&\&\ b2)\ \&\&\ b3.$

`Theorem` $orb\_assoc : b1\ ||\ (b2\ ||\ b3) = (b1\ ||\ b2)\ ||\ b3.$

Przemienność

`Theorem` $andb\_comm : b1\ \&\&\ b2 = b2\ \&\&\ b1.$

`Theorem` $orb\_comm : b1\ ||\ b2 = b2\ ||\ b1.$

Rozdzielność

Theorem $andb\_dist\_orb$:
$b1$ && $(b2 \mid\mid b3) = (b1$ && $b2) \mid\mid (b1$ && $b3)$.

Theorem orb_dist_andb :
$b1 \mid\mid (b2$ && $b3) = (b1 \mid\mid b2)$ && $(b1 \mid\mid b3)$.

Wyłączony środek i niesprzeczność

Theorem $andb\_negb$: b && $negb\ b = false$.

Theorem orb_negb : $b \mid\mid negb\ b = true$.

Prawa de Morgana

Theorem $negb\_andb$: $negb\ (b1$ && $b2) = negb\ b1 \mid\mid negb\ b2$.

Theorem $negb\_orb$: $negb\ (b1 \mid\mid b2) = negb\ b1$ && $negb\ b2$.

eqb, *xorb*, *norb*, *nandb*

Theorem eqb_spec : $eqb\ b1\ b2 = true \rightarrow b1 = b2$.

Theorem eqb_spec' : $eqb\ b1\ b2 = false \rightarrow b1 \neq b2$.

Theorem $xorb\_spec$:
$xorb\ b1\ b2 = negb\ (eqb\ b1\ b2)$.

Theorem $xorb\_spec'$:
$xorb\ b1\ b2 = true \rightarrow b1 \neq b2$.

Theorem $norb\_spec$:
$norb\ b1\ b2 = negb\ (b1 \mid\mid b2)$.

Theorem $nandb\_spec$:
$nandb\ b1\ b2 = negb\ (b1$ && $b2)$.

Różne

Theorem $andb\_eq\_orb$:
$b1$ && $b2 = b1 \mid\mid b2 \rightarrow b1 = b2$.

Theorem $all3\_spec$:
$(b1$ && $b2) \mid\mid (negb\ b1 \mid\mid negb\ b2) = true$.

Theorem $noncontradiction\_bool$:
$negb\ (eqb\ b\ (negb\ b)) = true$.

Theorem $excluded\_middle\_bool$:

$b \mid\mid negb\ \ b = true.$

`End` *boolean_functions*.

Rozdział 9

X2: Arytmetyka Peano

Poniższe zadania mają służyć utrwaleniu zdobytej dotychczas wiedzy na temat prostej rekursji i indukcji. Większość powinna być robialna po przeczytaniu rozdziału o konstruktorach rekurencyjnych, ale niczego nie gwarantuję.

Celem zadań jest rozwinięcie arytmetyki do takiego poziomu, żeby można było tego używać gdzie indziej w jakotakim stopniu. Niektóre zadania mogą pokrywać się z zadaniami obecnymi w tekście, a niektóre być może nawet z przykładami. Staraj się nie podglądać.

Nazwy twierdzeń nie muszą pokrywać się z tymi z biblioteki standardowej, choć starałem się, żeby tak było.

`Module` *MyNat*.

9.1 Podstawy

9.1.1 Definicja i notacje

Zdefiniuj liczby naturalne.

`Notation` "0":= O.
`Notation` "1":= $(S\ 0)$.

9.1.2 0 i S

Udowodnij właściwości zera i następnika.

`Lemma` *neq_0_Sn* :
  $\forall\ n : nat, 0 \neq S\ n.$

`Lemma` *neq_n_Sn* :
  $\forall\ n : nat, n \neq S\ n.$

`Lemma` *not_eq_S* :
  $\forall\ n\ m : nat, n \neq m \to S\ n \neq S\ m.$

Lemma $S\_injective$:
$\forall\ n\ m : nat,\ S\ n = S\ m \to n = m.$

9.1.3 Poprzednik

Zdefiniuj funkcję zwracającą poprzednik danej liczby naturalnej. Poprzednikiem 0 jest 0.

Lemma $pred\_0 : pred\ 0 = 0.$

Lemma $pred\_Sn$:
$\forall\ n : nat,\ pred\ (S\ n) = n.$

9.2 Proste działania

9.2.1 Dodawanie

Zdefiniuj dodawanie (rekurencyjnie po pierwszym argumencie) i udowodnij jego właściwości.

Lemma $plus\_0\_l$:
$\forall\ n : nat,\ plus\ 0\ n = n.$

Lemma $plus\_0\_r$:
$\forall\ n : nat,\ plus\ n\ 0 = n.$

Lemma $plus\_n\_Sm$:
$\forall\ n\ m : nat,\ S\ (plus\ n\ m) = plus\ n\ (S\ m).$

Lemma $plus\_Sn\_m$:
$\forall\ n\ m : nat,\ plus\ (S\ n)\ m = S\ (plus\ n\ m).$

Lemma $plus\_assoc$:
$\forall\ a\ b\ c : nat,$
$plus\ a\ (plus\ b\ c) = plus\ (plus\ a\ b)\ c.$

Lemma $plus\_comm$:
$\forall\ n\ m : nat,\ plus\ n\ m = plus\ m\ n.$

Lemma $plus\_no\_annihilation\_l$:
$\neg\ \exists\ a : nat,\ \forall\ n : nat,\ plus\ a\ n = a.$

Lemma $plus\_no\_annihilation\_r$:
$\neg\ \exists\ a : nat,\ \forall\ n : nat,\ plus\ n\ a = a.$

Lemma $plus\_no\_inverse\_l$:
$\neg\ \forall\ n : nat,\ \exists\ i : nat,\ plus\ i\ n = 0.$

Lemma $plus\_no\_inverse\_r$:
$\neg\ \forall\ n : nat,\ \exists\ i : nat,\ plus\ n\ i = 0.$

Lemma $plus\_no\_inverse\_l\_strong$:
$\forall\ n\ i : nat,\ n \neq 0 \to plus\ i\ n \neq 0.$

```
Lemma plus_no_inverse_r_strong :
  ∀ n i : nat, n ≠ 0 → plus n i ≠ 0.
```

9.2.2 Alternatywne definicje dodawania

Udowodnij, że poniższe alternatywne metody zdefiniowania dodawania rzeczywiście definiują dodawanie.

```
Fixpoint plus' (n m : nat) : nat :=
match m with
    | 0 ⇒ n
    | S m' ⇒ S (plus' n m')
end.

Lemma plus'_is_plus :
  ∀ n m : nat, plus' n m = plus n m.

Fixpoint plus'' (n m : nat) : nat :=
match n with
    | 0 ⇒ m
    | S n' ⇒ plus'' n' (S m)
end.

Lemma plus''_is_plus :
  ∀ n m : nat, plus'' n m = plus n m.

Fixpoint plus''' (n m : nat) : nat :=
match m with
    | 0 ⇒ n
    | S m' ⇒ plus''' (S n) m'
end.

Lemma plus'''_is_plus :
  ∀ n m : nat, plus''' n m = plus n m.
```

9.2.3 Odejmowanie

Zdefiniuj odejmowanie i udowodnij jego właściwości.

```
Lemma minus_pred :
  ∀ n : nat, minus n 1 = pred n.

Lemma minus_0_l :
  ∀ n : nat, minus 0 n = 0.

Lemma minus_0_r :
  ∀ n : nat, minus n 0 = n.

Lemma minus_S :
```

$\forall$ n m : nat,
  $minus$ $(S\ n)$ $(S\ m)$ $=$ $minus$ n m.

`Lemma` $minus\_n$:
 $\forall$ n : nat, $minus$ n n $= 0$.

`Lemma` $minus\_plus\_l$:
 $\forall$ n m : nat,
  $minus$ $(plus\ n\ m)$ n $=$ m.

`Lemma` $minus\_plus\_r$:
 $\forall$ n m : nat,
  $minus$ $(plus\ n\ m)$ m $=$ n.

`Lemma` $minus\_plus\_distr$:
 $\forall$ a b c : nat,
  $minus$ a $(plus\ b\ c)$ $=$ $minus$ $(minus\ a\ b)$ c.

`Lemma` $minus\_exchange$:
 $\forall$ a b c : nat,
  $minus$ $(minus\ a\ b)$ c $=$ $minus$ $(minus\ a\ c)$ b.

`Lemma` $minus\_not\_comm$:
 $\neg$ $\forall$ n m : nat,
   $minus$ n m $=$ $minus$ m n.

9.2.4 Mnożenie

Zdefiniuj mnożenie i udowodnij jego właściwości.

`Lemma` $mult\_0\_l$:
 $\forall$ n : nat, $mult$ 0 n $= 0$.

`Lemma` $mult\_0\_r$:
 $\forall$ n : nat, $mult$ n 0 $= 0$.

`Lemma` $mult\_1\_l$:
 $\forall$ n : nat, $mult$ 1 n $=$ n.

`Lemma` $mult\_1\_r$:
 $\forall$ n : nat, $mult$ n 1 $=$ n.

`Lemma` $mult\_comm$:
 $\forall$ n m : nat,
  $mult$ n m $=$ $mult$ m n.

`Lemma` $mult\_plus\_distr\_r$:
 $\forall$ a b c : nat,
  $mult$ $(plus\ a\ b)$ c $=$ $plus$ $(mult\ a\ c)$ $(mult\ b\ c)$.

`Lemma` $mult\_minus\_distr\_l$:

$\forall$ a b c : nat,
  $mult$ a ($minus$ b c) = $minus$ ($mult$ a b) ($mult$ a c).

`Lemma` $mult\_minus\_distr\_r$:
  $\forall$ a b c : nat,
    $mult$ ($minus$ a b) c = $minus$ ($mult$ a c) ($mult$ b c).

`Lemma` $mult\_assoc$:
  $\forall$ a b c : nat,
    $mult$ a ($mult$ b c) = $mult$ ($mult$ a b) c.

`Lemma` $mult\_no\_inverse\_l$:
  $\neg$ $\forall$ n : nat, $\exists$ i : nat, $mult$ i n = 1.

`Lemma` $mult\_no\_inverse\_r$:
  $\neg$ $\forall$ n : nat, $\exists$ i : nat, $mult$ n i = 1.

`Lemma` $mult\_no\_inverse\_l\_strong$:
  $\forall$ n i : nat, $n \neq 1 \rightarrow$ $mult$ i $n \neq 1$.

`Lemma` $mult\_no\_inverse\_r\_strong$:
  $\forall$ n i : nat, $n \neq 1 \rightarrow$ $mult$ n $i \neq 1$.

`Lemma` $mult\_2\_plus$:
  $\forall$ n : nat, $mult$ (S (S 0)) n = $plus$ n n.

9.2.5 Potęgowanie

Zdefiniuj potęgowanie i udowodnij jego właściwości.

`Lemma` pow_0_r :
  $\forall$ n : nat, pow n 0 = 1.

`Lemma` pow_0_l :
  $\forall$ n : nat, pow 0 (S n) = 0.

`Lemma` pow_1_l :
  $\forall$ n : nat, pow 1 n = 1.

`Lemma` pow_1_r :
  $\forall$ n : nat, pow n 1 = n.

`Lemma` $pow\_no\_neutr\_l$:
  $\neg$ $\exists$ e : nat, $\forall$ n : nat, pow e n = n.

`Lemma` $pow\_no\_annihilator\_r$:
  $\neg$ $\exists$ a : nat, $\forall$ n : nat, pow n a = a.

`Lemma` pow_plus :
  $\forall$ a b c : nat,
    pow a ($plus$ b c) = $mult$ (pow a b) (pow a c).

`Lemma` pow_mult :

$\forall$ a b c : nat,
  pow ($mult$ a b) c = $mult$ (pow a c) (pow b c).

`Lemma` *pow_pow* :
  $\forall$ a b c : nat,
    pow (pow a b) c = pow a ($mult$ b c).

9.3 Porządek

9.3.1 Porządek $\leq$

Zdefiniuj relację "mniejszy lub równy" i udowodnij jej właściwości.

`Notation` ń <= m":= (le n m).

`Lemma` *le_0_n* :
  $\forall$ n : nat, $0 \leq n$.

`Lemma` *le_n_Sm* :
  $\forall$ n m : nat, $n \leq m \to n \leq S\ m$.

`Lemma` *le_Sn_m* :
  $\forall$ n m : nat, $S\ n \leq m \to n \leq m$.

`Lemma` *le_n_S* :
  $\forall$ n m : nat, $n \leq m \to S\ n \leq S\ m$.

`Lemma` *le_S_n* :
  $\forall$ n m : nat, $S\ n \leq S\ m \to n \leq m$.

`Lemma` *le_Sn_n* :
  $\forall$ n : nat, $\neg$ $S\ n \leq n$.

`Lemma` *le_refl* :
  $\forall$ n : nat, $n \leq n$.

`Lemma` *le_trans* :
  $\forall$ a b c : nat,
    $a \leq b \to b \leq c \to a \leq c$.

`Lemma` *le_antisym* :
  $\forall$ n m : nat,
    $n \leq m \to m \leq n \to n = m$.

`Lemma` *le_pred* :
  $\forall$ n : nat, $pred\ n \leq n$.

`Lemma` *le_n_pred* :
  $\forall$ n m : nat,
    $n \leq m \to pred\ n \leq pred\ m$.

`Lemma` *no_le_pred_n* :

$\neg\ \forall\ n\ m\ :\ nat,$
$\quad pred\ n \le pred\ m \to n \le m.$

Lemma le_plus_l :
$\forall\ a\ b\ c\ :\ nat,$
$\quad b \le c \to plus\ a\ b \le plus\ a\ c.$

Lemma le_plus_r :
$\forall\ a\ b\ c\ :\ nat,$
$\quad a \le b \to plus\ a\ c \le plus\ b\ c.$

Lemma le_plus :
$\forall\ a\ b\ c\ d\ :\ nat,$
$\quad a \le b \to c \le d \to plus\ a\ c \le plus\ b\ d.$

Lemma le_minus_S :
$\forall\ n\ m\ :\ nat,$
$\quad minus\ n\ (S\ m) \le minus\ n\ m.$

Lemma le_minus_l :
$\forall\ a\ b\ c\ :\ nat,$
$\quad b \le c \to minus\ a\ c \le minus\ a\ b.$

Lemma le_minus_r :
$\forall\ a\ b\ c\ :\ nat,$
$\quad a \le b \to minus\ a\ c \le minus\ b\ c.$

Lemma le_mult_l :
$\forall\ a\ b\ c\ :\ nat,$
$\quad b \le c \to mult\ a\ b \le mult\ a\ c.$

Lemma le_mult_r :
$\forall\ a\ b\ c\ :\ nat,$
$\quad a \le b \to mult\ a\ c \le mult\ b\ c.$

Lemma le_mult :
$\forall\ a\ b\ c\ d\ :\ nat,$
$\quad a \le b \to c \le d \to mult\ a\ c \le mult\ b\ d.$

Lemma le_plus_exists :
$\forall\ n\ m\ :\ nat,$
$\quad n \le m \to \exists\ k\ :\ nat,\ plus\ n\ k = m.$

Lemma le_pow_l :
$\forall\ a\ b\ c\ :\ nat,$
$\quad a \ne 0 \to b \le c \to pow\ a\ b \le pow\ a\ c.$

Lemma le_pow_r :
$\forall\ a\ b\ c\ :\ nat,$
$\quad a \le b \to pow\ a\ c \le pow\ b\ c.$

9.3.2 Porządek <

Definition *lt* $(n\ m : nat)$: Prop := $S\ n \leq m$.

Notation ń < m":= $(lt\ n\ m)$.

Lemma *lt_irrefl* :
  $\forall\ n : nat, \neg\ n < n$.

Lemma *lt_trans* :
  $\forall\ a\ b\ c : nat,\ a < b \rightarrow b < c \rightarrow a < c$.

Lemma *lt_asym* :
  $\forall\ n\ m : nat,\ n < m \rightarrow \neg\ m < n$.

9.3.3 Minimum i maksimum

Zdefiniuj operacje brania minimum i maksimum z dwóch liczb naturalnych oraz udowodnij ich właściwości.

Lemma *min_0_l* :
  $\forall\ n : nat,\ min\ 0\ n = 0$.

Lemma *min_0_r* :
  $\forall\ n : nat,\ min\ n\ 0 = 0$.

Lemma *max_0_l* :
  $\forall\ n : nat,\ max\ 0\ n = n$.

Lemma *max_0_r* :
  $\forall\ n : nat,\ max\ n\ 0 = n$.

Lemma *min_le* :
  $\forall\ n\ m : nat,\ n \leq m \rightarrow min\ n\ m = n$.

Lemma *max_le* :
  $\forall\ n\ m : nat,\ n \leq m \rightarrow max\ n\ m = m$.

Lemma *min_assoc* :
  $\forall\ a\ b\ c : nat,$
    $min\ a\ (min\ b\ c) = min\ (min\ a\ b)\ c$.

Lemma *max_assoc* :
  $\forall\ a\ b\ c : nat,$
    $max\ a\ (max\ b\ c) = max\ (max\ a\ b)\ c$.

Lemma *min_comm* :
  $\forall\ n\ m : nat,\ min\ n\ m = min\ m\ n$.

Lemma *max_comm* :
  $\forall\ n\ m : nat,\ max\ n\ m = max\ m\ n$.

Lemma *min_refl* :

$\forall\ n : nat,\ min\ n\ n = n.$

Lemma max_refl :
$\forall\ n : nat,\ max\ n\ n = n.$

Lemma $min\_no\_neutr\_l$:
$\neg\ \exists\ e : nat, \forall\ n : nat,\ min\ e\ n = n.$

Lemma $min\_no\_neutr\_r$:
$\neg\ \exists\ e : nat, \forall\ n : nat,\ min\ n\ e = n.$

Lemma $max\_no\_annihilator\_l$:
$\neg\ \exists\ a : nat, \forall\ n : nat,\ max\ a\ n = a.$

Lemma $max\_no\_annihilator\_r$:
$\neg\ \exists\ a : nat, \forall\ n : nat,\ max\ n\ a = a.$

Lemma is_it_true :
$(\forall\ n\ m : nat,\ min\ (S\ n)\ m = S\ (min\ n\ m)) \lor$
$(\tilde{}\ \forall\ n\ m : nat,\ min\ (S\ n)\ m = S\ (min\ n\ m)).$

9.4 Rozstrzygalność

9.4.1 Rozstrzygalność porządku

Zdefiniuj funkcję *leb*, która sprawdza, czy $n \leq m$.

Lemma leb_n :
$\forall\ n : nat,$
$leb\ n\ n = true.$

Lemma leb_spec :
$\forall\ n\ m : nat,$
$n \leq m \leftrightarrow leb\ n\ m = true.$

9.4.2 Rozstrzygalność równości

Zdefiniuj funkcję *eqb*, która sprawdza, czy $n = m$.

Lemma eqb_spec :
$\forall\ n\ m : nat,$
$n = m \leftrightarrow eqb\ n\ m = true.$

9.5 Dzielenie i podzielność

9.5.1 Dzielenie przez 2

Pokaż, że indukcję na liczbach naturalnych można robić "co 2". Wskazówka: taktyk można używać nie tylko do dowodzenia. Przypomnij sobie, że taktyki to programy, które generują dowody, zaś dowody są programami. Dzięki temu nic nie stoi na przeszkodzie, aby taktyki interpretować jako programy, które piszą inne programy. I rzeczywiście — w Coqu możemy używać taktyk do definiowania dowolnych termów. W niektórych przypadkach jest to bardzo częsta praktyka.

```
Fixpoint nat_ind_2
  (P : nat → Prop) (H0 : P 0) (H1 : P 1)
  (HSS : ∀ n : nat, P n → P (S (S n))) (n : nat) : P n.
```

Zdefiniuj dzielenie całkowitoliczbowe przez 2 oraz funkcję obliczającą resztę z dzielenia przez 2.

```
Notation "2":= (S (S 0)).

Lemma div2_even :
  ∀ n : nat, div2 (mult 2 n) = n.

Lemma div2_odd :
  ∀ n : nat, div2 (S (mult 2 n)) = n.

Lemma mod2_even :
  ∀ n : nat, mod2 (mult 2 n) = 0.

Lemma mod2_odd :
  ∀ n : nat, mod2 (S (mult 2 n)) = 1.

Lemma div2_mod2_spec :
  ∀ n : nat, plus (mult 2 (div2 n)) (mod2 n) = n.

Lemma div2_le :
  ∀ n : nat, div2 n ≤ n.

Lemma div2_pres_le :
  ∀ n m : nat, n ≤ m → div2 n ≤ div2 m.

Lemma mod2_le :
  ∀ n : nat, mod2 n ≤ n.

Lemma mod2_not_pres_e :
  ∃ n m : nat, n ≤ m ∧ mod2 m ≤ mod2 n.

Lemma div2_lt :
  ∀ n : nat,
    0 ≠ n → div2 n < n.
```

9.5.2 Podzielność

Definition $divides$ $(k\ n : nat)$: Prop :=
  $\exists\ m : nat,\ mult\ k\ m = n.$

Notation "k | n":= $(divides\ k\ n)$ (at level 40).

k dzieli n jeżeli n jest wielokrotnością k. Udowodnij podstawowe właściwości tej relacji.

Lemma $divides\_0$:
  $\forall\ n : nat,\ n \mid 0.$

Lemma $not\_divides\_0$:
  $\forall\ n : nat,\ n \neq 0 \rightarrow \neg\ 0 \mid n.$

Lemma $divides\_1$:
  $\forall\ n : nat,\ 1 \mid n.$

Lemma $divides\_refl$:
  $\forall\ n : nat,\ n \mid n.$

Lemma $divides\_trans$:
  $\forall\ k\ n\ m : nat,\ k \mid n \rightarrow n \mid m \rightarrow k \mid m.$

Lemma $divides\_plus$:
  $\forall\ k\ n\ m : nat,\ k \mid n \rightarrow k \mid m \rightarrow k \mid plus\ n\ m.$

Lemma $divides\_mult\_l$:
  $\forall\ k\ n\ m : nat,\ k \mid n \rightarrow k \mid mult\ n\ m.$

Lemma $divides\_mult\_r$:
  $\forall\ k\ n\ m : nat,\ k \mid m \rightarrow k \mid mult\ n\ m.$

Lemma $divides\_le$:
  $\neg\ \forall\ k\ n : nat,\ k \mid n \rightarrow k \leq n.$

End $MyNat$.

Rozdział 10

X3: Listy

Lista to najprostsza i najczęściej używana w programowaniu funkcyjnym struktura danych. Czas więc przeżyć na własnej skórze ich implementację.

UWAGA: ten rozdział wyewoluował do stanu dość mocno odbiegającego od tego, co jest w bibliotece standardowej — moim zdanem na korzyść.

```
Require Export Bool.
Require Export Nat.
```

W części dowodów przydadzą nam się fakty dotyczące arytmetyki liczb naturalnych, które możemy znaleźć w module *Arith*.

Zdefiniuj *list* (bez podglądania).

```
Arguments nil [A].
Arguments cons [A] _ _.
```

```
(* Notation  := nil.*)
Notation "[ ]":= nil (format "[ ]").
Notation "x :: y":= (cons x y) (at level 60, right associativity).
Notation "[ x ; .. ; y ]":= (cons x .. (cons y nil) ..).
```

10.1 Proste funkcje

10.1.1 *isEmpty*

Zdefiniuj funkcję *isEmpty*, która sprawdza, czy lista jest pusta.

10.1.2 *length*

Zdefiniuj funkcję *length*, która oblicza długość listy.

Przykład: *length* [1; 2; 3] = 3

```
Lemma length_nil :
```

$\forall\ A$: Type, *length* (@*nil* A) = 0.

Lemma *length_cons* :
  $\forall$ (A : Type) (h : A) (t : *list* A),
    $\exists\ n$: *nat*, *length* (h :: t) = S n.

Lemma *length_0* :
  $\forall$ (A : Type) (l : *list* A),
    length $l = 0 \rightarrow l = []$.

10.1.3 *snoc*

Zdefiniuj funkcję *snoc*, która dokleja element x na koniec listy l.
Przykład: *snoc* 42 [1; 2; 3] = [1; 2; 3; 42]

Lemma *snoc_isEmpty* :
  $\forall$ (A : Type) (x : A) (l : *list* A),
    isEmpty l = *true* $\rightarrow$ *snoc* x l = [x].

Lemma *isEmpty_snoc* :
  $\forall$ (A : Type) (x : A) (l : *list* A),
    isEmpty (*snoc* x l) = *false*.

Lemma *length_snoc* :
  $\forall$ (A : Type) (x : A) (l : *list* A),
    length (*snoc* x l) = S (*length* l).

Lemma *snoc_inv* :
  $\forall$ (A : Type) (*l1* *l2* : *list* A) (x y : A),
    snoc x *l1* = *snoc* y *l2* $\rightarrow x = y \land$ *l1* = *l2*.

10.1.4 *app*

Zdefiniuj funkcję *app*, która skleja dwie listy.
Przykład: *app* [1; 2; 3] [4; 5; 6] = [1; 2; 3; 4; 5; 6]

Notation ł1 ++ l2":= (*app* *l1* *l2*).

Lemma *app_nil_l* :
  $\forall$ (A : Type) (l : *list* A),
    [] ++ l = l.

Lemma *app_nil_r* :
  $\forall$ (A : Type) (l : *list* A),
    l ++ [] = l.

Lemma *app_assoc* :
  $\forall$ (A : Type) (*l1* *l2* *l3* : *list* A),
    l1 ++ (*l2* ++ *l3*) = (*l1* ++ *l2*) ++ *l3*.

Lemma $isEmpty\_app$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $isEmpty$ ($l1$ ++ $l2$) = $andb$ ($isEmpty$ $l1$) ($isEmpty$ $l2$).

Lemma $length\_app$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $length$ ($l1$ ++ $l2$) = $length$ $l1$ + $length$ $l2$.

Lemma $snoc\_app$:
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $snoc$ x ($l1$ ++ $l2$) = $l1$ ++ $snoc$ x $l2$.

Lemma app_snoc_l :
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $snoc$ x $l1$ ++ $l2$ = $l1$ ++ x :: $l2$.

Lemma app_snoc_r :
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $l1$ ++ $snoc$ x $l2$ = $snoc$ x ($l1$ ++ $l2$).

Lemma $snoc\_app\_singl$:
  $\forall$ (A : Type) (x : A) (l : $list$ A),
    $snoc$ x l = l ++ [x].

Lemma app_cons_l :
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    (x :: $l1$) ++ $l2$ = x :: ($l1$ ++ $l2$).

Lemma app_cons_r :
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $l1$ ++ x :: $l2$ = ($l1$ ++ [x]) ++ $l2$.

Lemma $no\_infinite\_cons$:
  $\forall$ (A : Type) (x : A) (l : $list$ A),
    l = x :: l $\rightarrow$ $False$.

Lemma $no\_infinite\_app$:
  $\forall$ (A : Type) (l l' : $list$ A),
    l' $\neq$ [] $\rightarrow$ l = l' ++ l $\rightarrow$ $False$.

Lemma app_inv_l :
  $\forall$ (A : Type) (l $l1$ $l2$: $list$ A),
    l ++ $l1$ = l ++ $l2$ $\rightarrow$ $l1$ = $l2$.

Lemma app_inv_r :
  $\forall$ (A : Type) (l $l1$ $l2$: $list$ A),
    $l1$ ++ l = $l2$ ++ l $\rightarrow$ $l1$ = $l2$.

Lemma app_eq_nil :
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $l1$ ++ $l2$ = [] $\rightarrow$ $l1$ = [] $\wedge$ $l2$ = [].

10.1.5 *rev*

Zdefiniuj funkcję *rev*, która odwraca listę.

Przykład: *rev* [1; 2; 3] = [3; 2; 1]

`Lemma` *isEmpty_rev* :
 ∀ (A : `Type`) (l : *list* A),
  isEmpty (*rev* l) = *isEmpty* l.

`Lemma` *length_rev* :
 ∀ (A : `Type`) (l : *list* A),
  length (*rev* l) = *length* l.

`Lemma` *snoc_rev* :
 ∀ (A : `Type`) (x : A) (l : *list* A),
  snoc x (*rev* l) = *rev* (x :: l).

`Lemma` *rev_snoc* :
 ∀ (A : `Type`) (x : A) (l : *list* A),
  rev (*snoc* x l) = x :: *rev* l.

`Lemma` *rev_app* :
 ∀ (A : `Type`) (*l1 l2* : *list* A),
  rev (*l1* ++ *l2*) = *rev* *l2* ++ *rev* *l1*.

`Lemma` *rev_inv* :
 ∀ (A : `Type`) (l : *list* A),
  rev (*rev* l) = l.

10.1.6 *map*

Zdefiniuj funkcję *map*, która aplikuje funkcję f do każdego elementu listy.

Przykład:
map *isEmpty* [[]; [1]; [2; 3]; []] = [*true*; *false*; *false*; *true*]

`Lemma` *map_id* :
 ∀ (A : `Type`) (l : *list* A),
  map *id* l = l.

`Lemma` *map_map* :
 ∀ (A B C : `Type`) (f : $A \to B$) (g : $B \to C$) (l : *list* A),
  map g (*map* f l) = *map* (`fun` x : A ⇒ g (f x)) l.

`Lemma` *isEmpty_map* :
 ∀ (A B : `Type`) (f : $A \to B$) (l : *list* A),
  isEmpty (*map* f l) = *isEmpty* l.

`Lemma` *length_map* :
 ∀ (A B : `Type`) (f : $A \to B$) (l : *list* A),

$length\ (map\ f\ l) = length\ l.$

`Lemma` *map_snoc* :
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (x : A)\ (l : list\ A),$
$map\ f\ (snoc\ x\ l) = snoc\ (f\ x)\ (map\ f\ l).$

`Lemma` *map_app* :
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (l1\ l2 : list\ A),$
$map\ f\ (l1\ {+}{+}\ l2) = map\ f\ l1\ {+}{+}\ map\ f\ l2.$

`Lemma` *map_rev* :
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (l : list\ A),$
$map\ f\ (rev\ l) = rev\ (map\ f\ l).$

`Lemma` *map_ext* :
$\forall\ (A\ B : \texttt{Type})\ (f\ g : A \to B)\ (l : list\ A),$
$(\forall\ x : A, f\ x = g\ x) \to map\ f\ l = map\ g\ l.$

10.1.7 *join*

Napisz funkcję *join*, która spłaszcza listę list.

Przykład: *join* [[1; 2; 3]; [4; 5; 6]; [7]] = [1; 2; 3; 4; 5; 6; 7]

`Lemma` *join_snoc* :
$\forall\ (A : \texttt{Type})\ (x : list\ A)\ (l : list\ (list\ A)),$
$join\ (snoc\ x\ l) = join\ l\ {+}{+}\ x.$

`Lemma` *join_app* :
$\forall\ (A : \texttt{Type})\ (l1\ l2 : list\ (list\ A)),$
$join\ (l1\ {+}{+}\ l2) = join\ l1\ {+}{+}\ join\ l2.$

`Lemma` *rev_join* :
$\forall\ (A : \texttt{Type})\ (l : list\ (list\ A)),$
$rev\ (join\ l) = join\ (rev\ (map\ rev\ l)).$

`Lemma` *map_join* :
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (l : list\ (list\ A)),$
$map\ f\ (join\ l) = join\ (map\ (map\ f)\ l).$

10.1.8 *bind*

Napisz funkcję *bind*, która spełnia specyfikację *bind_spec*. Użyj rekursji, ale nie używaj funkcji *join* ani *map*.

`Lemma` *bind_spec* :
$\forall\ (A\ B : \texttt{Type})\ (f : A \to list\ B)\ (l : list\ A),$
$bind\ f\ l = join\ (map\ f\ l).$

`Lemma` *bind_snoc* :

$\forall$ (A B : Type) (f : $A \to$ *list* B) (x : A) (l : *list* A),
bind f (*snoc* x l) = *bind* f l ++ f x.

10.1.9 *replicate*

Napisz funkcję *replicate*, która powiela dany element n razy, tworząc listę.

Przykład: *replicate* 5 0 = [0; 0; 0; 0; 0]

```
Definition isZero (n : nat) : bool :=
match n with
    | 0 ⇒ true
    | _ ⇒ false
end.
```

Lemma *isEmpty_replicate* :
$\forall$ (A : Type) (n : *nat*) (x : A),
isEmpty (*replicate* n x) = if *isZero* n then *true* else *false*.

Lemma *length_replicate* :
$\forall$ (A : Type) (n : *nat*) (x : A),
length (*replicate* n x) = n.

Lemma *snoc_replicate* :
$\forall$ (A : Type) (x : A) (n : *nat*),
snoc x (*replicate* n x) = *replicate* (S n) x.

Lemma *replicate_plus* :
$\forall$ (A : Type) (n m : *nat*) (x : A),
replicate (n + m) x = *replicate* n x ++ *replicate* m x.

Lemma *replicate_plus_comm* :
$\forall$ (A : Type) (n m : *nat*) (x : A),
replicate (n + m) x = *replicate* (m + n) x.

Lemma *rev_replicate* :
$\forall$ (A : Type) (n : *nat*) (x : A),
rev (*replicate* n x) = *replicate* n x.

Lemma *map_replicate* :
$\forall$ (A B : Type) (f : $A \to B$) (n : *nat*) (x : A),
map f (*replicate* n x) = *replicate* n (f x).

10.1.10 *iterate* i *iter*

Napisz funkcję *iterate*. *iterate* f n x to lista postaci [x, f x, f (f x), ..., f (... (f x) ...)] o długości n.

Przykład:
iterate S 5 0 = [0; 1; 2; 3; 4]

Napisz też funkcję *iter*, która przyda się do podania charakteryzacji funkcji *iterate*. Zgadnij, czym ma ona być.

Lemma $isEmpty\_iterate$:
  $\forall$ (A : Type) (f : $A \to A$) (n : nat) (x : A),
    $isEmpty$ ($iterate$ f n x) =
    match n with
      | $0 \Rightarrow true$
      | $\_ \Rightarrow false$
    end.

Lemma $length\_iterate$:
  $\forall$ (A : Type) (f : $A \to A$) (n : nat) (x : A),
    $length$ ($iterate$ f n x) = n.

Lemma $snoc\_iterate$:
  $\forall$ (A : Type) (f : $A \to A$) (n : nat) (x : A),
    $snoc$ ($iter$ f n x) ($iterate$ f n x) =
    $iterate$ f (S n) x.

Lemma $iterate\_plus$:
  $\forall$ (A : Type) (f : $A \to A$) (n m : nat) (x : A),
    $iterate$ f ($n + m$) x =
    $iterate$ f n x ++ $iterate$ f m ($iter$ f n x).

Lemma $snoc\_iterate\_iter$:
  $\forall$ (A : Type) (f : $A \to A$) (n : nat) (x : A),
    $iterate$ f n x ++ [$iter$ f n x] = $iterate$ f (S n) x.

Lemma $map\_iterate$:
  $\forall$ (A : Type) (f : $A \to A$) (n : nat) (x : A),
    map f ($iterate$ f n x) = $iterate$ f n (f x).

Lemma $map\_iter\_iterate$:
  $\forall$ (A : Type) (f : $A \to A$) (n m : nat) (x : A),
    map ($iter$ f m) ($iterate$ f n x) =
    $iterate$ f n ($iter$ f m x).

Lemma $iterate\_replicate$:
  $\forall$ (A : Type) (n : nat) (x : A),
    $iterate$ id n x = $replicate$ n x.

10.1.11 *head*, *tail* i *uncons*

head

Zdefiniuj funkcję *head*, która zwraca głowę (pierwszy element) listy lub *None*, gdy lista jest pusta.

Przykład: $head\ [1; 2; 3] = Some\ 1$

```
Lemma head_nil :
  ∀ (A : Type), head [] = (@None A).

Lemma head_cons :
  ∀ (A : Type) (h : A) (t : list A),
    head (h :: t) = Some h.

Lemma head_isEmpty_true :
  ∀ (A : Type) (l : list A),
    isEmpty l = true → head l = None.

Lemma isEmpty_head_not_None :
  ∀ (A : Type) (l : list A),
    head l ≠ None → isEmpty l = false.

Lemma head_snoc :
  ∀ (A : Type) (x : A) (l : list A),
    head (snoc x l) =
    if isEmpty l then Some x else head l.

Lemma head_app :
  ∀ (A : Type) (l1 l2 : list A),
    head (l1 ++ l2) =
    match l1 with
        | [] ⇒ head l2
        | h :: _ ⇒ Some h
    end.

Lemma head_map :
  ∀ (A B : Type) (f : A → B) (l : list A),
    head (map f l) =
    match l with
        | [] ⇒ None
        | h :: _ ⇒ Some (f h)
    end.

Lemma head_replicate_S :
  ∀ (A : Type) (n : nat) (x : A),
    head (replicate (S n) x) = Some x.

Lemma head_replicate :
  ∀ (A : Type) (n : nat) (x : A),
    head (replicate n x) =
    match n with
        | 0 ⇒ None
        | _ ⇒ Some x
    end.
```

Lemma *head_iterate* :
  ∀ (*A* : Type) (*f* : *A* → *A*) (*n* : *nat*) (*x* : *A*),
    head (*iterate f n x*) =
    match *n* with
        | 0 ⇒ *None*
        | *S n'* ⇒ *Some x*
    end.

tail

Zdefiniuj funkcję *tail*, która zwraca ogon listy (czyli wszystkie jej elementy poza głową) lub *None*, gdy lista jest pusta.

Przykład: *tail* [1; 2; 3] = *Some* [2; 3]

Lemma *tail_nil* :
  ∀ *A* : Type, *tail* (@*nil A*) = *None*.

Lemma *tail_cons* :
  ∀ (*A* : Type) (*h* : *A*) (*t* : *list A*),
    tail (*h* :: *t*) = *Some t*.

Lemma *tail_isEmpty_true* :
  ∀ (*A* : Type) (*l* : *list A*),
    isEmpty l = *true* → *tail l* = *None*.

Lemma *isEmpty_tail_not_None* :
  ∀ (*A* : Type) (*l* : *list A*),
    tail l ≠ *None* → *isEmpty l* = *false*.

Lemma *tail_snoc* :
  ∀ (*A* : Type) (*x* : *A*) (*l* : *list A*),
    tail (*snoc x l*) =
    match *tail l* with
        | *None* ⇒ *Some* []
        | *Some t* ⇒ *Some* (*snoc x t*)
    end.

Lemma *tail_app* :
  ∀ (*A* : Type) (*l1 l2* : *list A*),
    tail (*l1* ++ *l2*) =
    match *l1* with
        | [] ⇒ *tail l2*
        | *h* :: *t* ⇒ *Some* (*t* ++ *l2*)
    end.

Lemma *tail_map* :
  ∀ (*A B* : Type) (*f* : *A* → *B*) (*l* : *list A*),
    tail (*map f l*) =

```
    match l with
        | [] ⇒ None
        | _ :: t ⇒ Some (map f t)
    end.
```

```
Lemma tail_replicate :
  ∀ (A : Type) (n : nat) (x : A),
    tail (replicate n x) =
    match n with
        | 0 ⇒ None
        | S n' ⇒ Some (replicate n' x)
    end.
```

```
Lemma tail_iterate :
  ∀ (A : Type) (f : A → A) (n : nat) (x : A),
    tail (iterate f n x) =
    match n with
        | 0 ⇒ None
        | S n' ⇒ Some (iterate f n' (f x))
    end.
```

uncons

Napisz funkcję *uncons*, która zwraca parę złożoną z głowy i ogona listy lub *None*, gdy lista jest pusta. Nie używaj funkcji *head* ani *tail*. Udowodnij poniższą specyfikację.

Przykład: *uncons* [1; 2; 3] = *Some* (1, [2; 3])

```
Lemma uncons_spec :
  ∀ (A : Type) (l : list A),
    uncons l =
    match head l, tail l with
        | Some h, Some t ⇒ Some (h, t)
        | _, _ ⇒ None
    end.
```

10.1.12 *last*, *init* i *unsnoc*

last

Zdefiniuj funkcję *last*, która zwraca ostatni element listy lub *None*, gdy lista jest pusta.

Przykład: *last* [1; 2; 3] = *Some* 3

```
Lemma last_nil :
  ∀ (A : Type), last [] = (@None A).
```

```
Lemma last_isEmpty_true :
```

$\forall$ (A : Type) (l : *list* A),
  isEmpty l = *true* $\rightarrow$ *last* l = *None*.

Lemma *isEmpty_last_not_None* :
  $\forall$ (A : Type) (l : *list* A),
  last $l \neq$ *None* $\rightarrow$ *isEmpty* l = *false*.

Lemma *last_snoc* :
  $\forall$ (A : Type) (x : A) (l : *list* A),
  last (*snoc* x l) = *Some* x.

Lemma *last_spec* :
  $\forall$ (A : Type) (l : *list* A) (x : A),
  last (l ++ [x]) = *Some* x.

Lemma *last_app* :
  $\forall$ (A : Type) ($l1$ $l2$: *list* A),
  last ($l1$ ++ $l2$) =
  match $l2$ with
    | [] $\Rightarrow$ *last* $l1$
    | _ $\Rightarrow$ *last* $l2$
  end.

Lemma *last_replicate_S* :
  $\forall$ (A : Type) (n : *nat*) (x : A),
  last (*replicate* (S n) x) = *Some* x.

Lemma *last_replicate* :
  $\forall$ (A : Type) (n : *nat*) (x : A),
  last (*replicate* n x) =
  match n with
    | 0 $\Rightarrow$ *None*
    | _ $\Rightarrow$ *Some* x
  end.

Lemma *last_iterate* :
  $\forall$ (A : Type) (f : $A \rightarrow A$) (n : *nat*) (x : A),
  last (*iterate* f n x) =
  match n with
    | 0 $\Rightarrow$ *None*
    | S n' $\Rightarrow$ *Some* (*iter* f n' x)
  end.

init

Zdefiniuj funkcję *init*, która zwraca wszystkie elementy listy poza ostatnim lub *None*, gdy lista jest pusta.

Przykład: *init* [1; 2; 3] = *Some* [1; 2]

```
Lemma init_None :
  ∀ (A : Type) (l : list A),
    init l = None → l = [].

Lemma init_snoc :
  ∀ (A : Type) (x : A) (l : list A),
    init (snoc x l) = Some l.

Lemma init_app :
  ∀ (A : Type) (l1 l2 : list A),
    init (l1 ++ l2) =
    match init l2 with
        | None ⇒ init l1
        | Some i ⇒ Some (l1 ++ i)
    end.

Lemma init_spec :
  ∀ (A : Type) (l : list A) (x : A),
    init (l ++ [x]) = Some l.

Lemma init_map :
  ∀ (A B : Type) (f : A → B) (l : list A),
    init (map f l) =
    match l with
        | [] ⇒ None
        | h :: t ⇒
            match init t with
                | None ⇒ Some []
                | Some i ⇒ Some (map f (h :: i))
            end
    end.

Lemma init_replicate :
  ∀ (A : Type) (n : nat) (x : A),
    init (replicate n x) =
    match n with
        | 0 ⇒ None
        | S n' ⇒ Some (replicate n' x)
    end.

Lemma init_iterate :
  ∀ (A : Type) (f : A → A) (n : nat) (x : A),
    init (iterate f n x) =
    match n with
        | 0 ⇒ None
        | S n' ⇒ Some (iterate f n' x)
    end.
```

```
Lemma init_last :
  ∀ (A : Type) (l l' : list A) (x : A),
    init l = Some l' → last l = Some x → l = l' ++ [x].
```

unsnoc

Zdefiniuj funkcję *unsnoc*, która rozbija listę na parę złożoną z ostatniego elementu oraz całej reszty lub zwraca *None* gdy lista jest pusta. Nie używaj funkcji *last* ani *init*. Udowodnij poniższą specyfikację.

Przykład: *unsnoc* [1; 2; 3] = *Some* (3, [1; 2])

```
Lemma unsnoc_None :
  ∀ (A : Type) (l : list A),
    unsnoc l = None → l = [].

Lemma unsnoc_spec :
  ∀ (A : Type) (l : list A),
    unsnoc l =
    match last l, init l with
        | Some x, Some l' ⇒ Some (x, l')
        | _, _ ⇒ None
    end.
```

Dualności *head* i *last*, *tail* i *init* oraz ciekawostki

```
Lemma last_rev :
  ∀ (A : Type) (l : list A),
    last (rev l) = head l.

Lemma head_rev :
  ∀ (A : Type) (l : list A),
    head (rev l) = last l.

Lemma tail_rev :
  ∀ (A : Type) (l : list A),
    tail (rev l) =
    match init l with
        | None ⇒ None
        | Some t ⇒ Some (rev t)
    end.

Lemma init_rev :
  ∀ (A : Type) (l : list A),
    init (rev l) =
    match tail l with
        | None ⇒ None
```

```
    | Some t ⇒ Some (rev t)
  end.
```

```
Lemma init_decomposition :
  ∀ (A : Type) (l : list A),
    l = [] ∨
    ∃ (h : A) (t : list A),
      init l = Some t ∧ last l = Some h ∧ l = t ++ [h].
```

```
(*  end hide *)
```

```
Lemma bilateral_decomposition :
  ∀ (A : Type) (l : list A),
    l = [] ∨
    (∃ x : A, l = [x]) ∨
    ∃ (x y : A) (l' : list A), l = x :: l' ++ [y].
```

10.1.13 *nth*

Zdefiniuj funkcję *nth*, która zwraca n-ty element listy lub *None*, gdy nie ma n-tego elementu.

Przykład:

nth 1 [1; 2; 3] = *Some* 2

nth 42 [1; 2; 3] = *None*

```
Lemma nth_nil :
  ∀ (A : Type) (n : nat),
    nth n (@nil A) = None.
```

```
Lemma nth_isEmpty_true :
  ∀ (A : Type) (l : list A) (n : nat),
    isEmpty l = true → nth n l = None.
```

```
Lemma isEmpty_nth_not_None :
  ∀ (A : Type) (l : list A) (n : nat),
    nth n l ≠ None → isEmpty l = false.
```

```
Lemma nth_length_lt :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l → ∃ x : A, nth n l = Some x.
```

```
Lemma nth_length_ge :
  ∀ (A : Type) (l : list A) (n : nat),
    length l ≤ n → nth n l = None.
```

```
Lemma nth_snoc_length_lt :
  ∀ (A : Type) (x : A) (l : list A) (n : nat),
    n < length l → nth n (snoc x l) = nth n l.
```

```
Lemma nth_snoc_length_eq :
```

```
  ∀ (A : Type) (x : A) (l : list A),
    nth (length l) (snoc x l) = Some x.

Lemma nth_snoc :
  ∀ (A : Type) (x : A) (l : list A) (n : nat),
    nth n (snoc x l) =
    if n <? length l then nth n l
    else if n =? length l then Some x
    else None.

Lemma nth_app :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    nth n (l1 ++ l2) =
    match nth n l1 with
        | None ⇒ nth (n - length l1) l2
        | Some x ⇒ Some x
    end.

Lemma nth_app_l :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    n < length l1 → nth n (l1 ++ l2) = nth n l1.

Lemma nth_app_r :
  ∀ (A : Type) (n : nat) (l1 l2 : list A),
    length l1 ≤ n → nth n (l1 ++ l2) = nth (n - length l1) l2.

Lemma nth_rev :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l → nth n (rev l) = nth (length l - S n) l.

Lemma nth_None :
  ∀ (A : Type) (l : list A) (n : nat),
    nth n l = None → length l ≤ n.

Lemma nth_Some :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    nth n l = Some x → n < length l.

Lemma nth_spec' :
  ∀ (A : Type) (l : list A) (n : nat),
    match nth n l with
        | None ⇒ length l ≤ n
        | Some x ⇒ ∃ l1 l2 : list A,
                        l = l1 ++ x :: l2 ∧ length l1 = n
    end.

Lemma nth_map_Some :
  ∀ (A B : Type) (f : A → B) (l : list A) (n : nat) (x : A),
    nth n l = Some x → nth n (map f l) = Some (f x).
```

```
Lemma nth_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (n : nat),
    nth n (map f l) =
    match nth n l with
        | None ⇒ None
        | Some x ⇒ Some (f x)
    end.

Lemma nth_replicate :
  ∀ (A : Type) (n i : nat) (x : A),
    i < n → nth i (replicate n x) = Some x.

Lemma nth_iterate :
  ∀ (A : Type) (f : A → A) (n m : nat) (x : A),
    nth m (iterate f n x) =
    if leb n m then None else Some (iter f m x).

Lemma head_nth :
  ∀ (A : Type) (l : list A),
    nth 0 l = head l.

Lemma last_nth :
  ∀ (A : Type) (l : list A),
    last l = nth (length l - 1) l.
```

10.1.14 *take*

Zdefiniuj funkcję *take*, która bierze z listy n początkowych elementów.

Przykład:

take 2 [1; 2; 3] = [1; 2]

```
Lemma take_0 :
  ∀ (A : Type) (l : list A),
    take 0 l = [].

Lemma take_nil :
  ∀ (A : Type) (n : nat),
    take n [] = @nil A.

Lemma take_S_cons :
  ∀ (A : Type) (n : nat) (h : A) (t : list A),
    take (S n) (h :: t) = h :: take n t.

Lemma isEmpty_take :
  ∀ (A : Type) (l : list A) (n : nat),
    isEmpty (take n l) = orb (beq_nat 0 n) (isEmpty l).

Lemma take_length :
  ∀ (A : Type) (l : list A),
```

$take\ (length\ l)\ l = l.$

Lemma $take\_length'$:
$\forall\ (A : \texttt{Type})\ (l : list\ A)\ (n : nat),$
$length\ l \leq n \to take\ n\ l = l.$

Lemma $length\_take$:
$\forall\ (A : \texttt{Type})\ (l : list\ A)\ (n : nat),$
$length\ (take\ n\ l) = min\ (length\ l)\ n.$

(* TODO: zabij *) Lemma $take\_snoc\_lt$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n : nat),$
$n < length\ l \to take\ n\ (snoc\ x\ l) = take\ n\ l.$

Lemma $take\_snoc\_le$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n : nat),$
$n \leq length\ l \to take\ n\ (snoc\ x\ l) = take\ n\ l.$

Lemma $take\_app$:
$\forall\ (A : \texttt{Type})\ (l1\ l2 : list\ A)\ (n : nat),$
$take\ n\ (l1\ {+}{+}\ l2) = take\ n\ l1\ {+}{+}\ take\ (n\ \text{-}\ length\ l1)\ l2.$

Lemma $take\_app\_l$:
$\forall\ (A : \texttt{Type})\ (l1\ l2 : list\ A)\ (n : nat),$
$n \leq length\ l1 \to take\ n\ (l1\ {+}{+}\ l2) = take\ n\ l1.$

Lemma $take\_app\_r$:
$\forall\ (A : \texttt{Type})\ (n : nat)\ (l1\ l2 : list\ A),$
$length\ l1 < n \to$
$take\ n\ (l1\ {+}{+}\ l2) = l1\ {+}{+}\ take\ (n\ \text{-}\ length\ l1)\ l2.$

Lemma $take\_map$:
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (l : list\ A)\ (n : nat),$
$take\ n\ (map\ f\ l) = map\ f\ (take\ n\ l).$

Lemma $take\_replicate$:
$\forall\ (A : \texttt{Type})\ (n\ m : nat)\ (x : A),$
$take\ m\ (replicate\ n\ x) = replicate\ (min\ n\ m)\ x.$

Lemma $take\_iterate$:
$\forall\ (A : \texttt{Type})\ (f : A \to A)\ (n\ m : nat)\ (x : A),$
$take\ m\ (iterate\ f\ n\ x) = iterate\ f\ (min\ n\ m)\ x.$

Lemma $head\_take$:
$\forall\ (A : \texttt{Type})\ (l : list\ A)\ (n : nat),$
$head\ (take\ n\ l) =$
$\texttt{if}\ beq\_nat\ 0\ n\ \texttt{then}\ None\ \texttt{else}\ head\ l.$

Lemma $last\_take$:
$\forall\ (A : \texttt{Type})\ (l : list\ A)\ (n : nat),$
$last\ (take\ (S\ n)\ l) = nth\ (min\ (length\ l\ \text{-}\ 1)\ n)\ l.$

```
Lemma tail_take :
  ∀ (A : Type) (l : list A) (n : nat),
    tail (take n l) =
    match n, l with
        | 0, _ ⇒ None
        | _, [] ⇒ None
        | S n', h :: t ⇒ Some (take n' t)
    end.

Lemma init_take :
  ∀ (A : Type) (l : list A) (n : nat),
    init (take n l) =
    match n, l with
        | 0, _ ⇒ None
        | _, [] ⇒ None
        | S n', h :: t ⇒ Some (take (min n' (length l - 1)) l)
    end.

Lemma nth_take :
  ∀ (A : Type) (l : list A) (n m : nat),
    nth m (take n l) =
    if leb (S m) n then nth m l else None.

Lemma take_take :
  ∀ (A : Type) (l : list A) (n m : nat),
    take m (take n l) = take (min n m) l.

Lemma take_interesting :
  ∀ (A : Type) (l1 l2 : list A),
    (∀ n : nat, take n l1 = take n l2) → l1 = l2.
```

10.1.15 *drop*

Zdefiniuj funkcję *drop*, która wyrzuca z listy n początkowych elementów i zwraca to, co zostało.

Przykład:
drop 2 [1; 2; 3] = [3]

```
Lemma drop_0 :
  ∀ (A : Type) (l : list A),
    drop 0 l = l.

Lemma drop_nil :
  ∀ (A : Type) (n : nat),
    drop n [] = @nil A.

Lemma drop_S_cons :
```

$\forall$ (A : Type) (n : *nat*) (h : A) (t : *list* A),
drop (S n) (h :: t) = *drop* n t.

Lemma *isEmpty_drop* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*),
isEmpty (*drop* n l) = *leb* (*length* l) n.

Lemma *drop_length* :
$\forall$ (A : Type) (l : *list* A),
drop (*length* l) l = [].

Lemma *drop_length'* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*),
length $l \leq n \rightarrow$ *drop* n l = [].

Lemma *length_drop* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*),
length (*drop* n l) = *length* l - n.

Lemma *drop_snoc_le* :
$\forall$ (A : Type) (x : A) (l : *list* A) (n : *nat*),
$n \leq$ *length* $l \rightarrow$ *drop* n (*snoc* x l) = *snoc* x (*drop* n l).

Lemma *drop_app* :
$\forall$ (A : Type) (*l1* *l2* : *list* A) (n : *nat*),
drop n (*l1* ++ *l2*) = *drop* n *l1* ++ *drop* (n - *length* *l1*) *l2*.

Lemma *drop_app_l* :
$\forall$ (A : Type) (*l1* *l2* : *list* A) (n : *nat*),
$n \leq$ *length* *l1* $\rightarrow$ *drop* n (*l1* ++ *l2*) = *drop* n *l1* ++ *l2*.

Lemma *drop_app_r* :
$\forall$ (A : Type) (*l1* *l2* : *list* A) (n : *nat*),
length *l1* $< n \rightarrow$ *drop* n (*l1* ++ *l2*) = *drop* (n - *length* *l1*) *l2*.

Lemma *drop_map* :
$\forall$ (A B : Type) (f : $A \rightarrow B$) (l : *list* A) (n : *nat*),
drop n (*map* f l) = *map* f (*drop* n l).

Lemma *drop_replicate* :
$\forall$ (A : Type) (n m : *nat*) (x : A),
drop m (*replicate* n x) = *replicate* (n - m) x.

Lemma *drop_iterate* :
$\forall$ (A : Type) (f : $A \rightarrow A$) (n m : *nat*) (x : A),
drop m (*iterate* f n x) =
iterate f (n - m) (*iter* f (*min* n m) x).

Lemma *head_drop* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*),
head (*drop* n l) = *nth* n l.

```
Lemma last_drop :
  ∀ (A : Type) (l : list A) (n : nat),
    last (drop n l) = if leb (S n) (length l) then last l else None.
```

```
Lemma tail_drop :
  ∀ (A : Type) (l : list A) (n : nat),
    tail (drop n l) =
    if leb (S n) (length l) then Some (drop (S n) l) else None.
```

```
Lemma init_drop :
  ∀ (A : Type) (l : list A) (n : nat),
    init (drop n l) =
    if n <? length l
    then
      match init l with
          | None ⇒ None
          | Some l' ⇒ Some (drop n l')
      end
    else None.
```

```
Lemma nth_drop :
  ∀ (A : Type) (l : list A) (n m : nat),
    nth m (drop n l) = nth (n + m) l.
```

```
Lemma nth_spec_Some :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    nth n l = Some x → l = take n l ++ x :: drop (S n) l.
```

```
Lemma nth_spec :
  ∀ (A : Type) (l : list A) (n : nat),
    match nth n l with
        | None ⇒ length l ≤ n
        | Some x ⇒ l = take n l ++ x :: drop (S n) l
    end.
```

```
Lemma drop_drop :
  ∀ (A : Type) (l : list A) (n m : nat),
    drop m (drop n l) = drop (n + m) l.
```

```
Lemma drop_not_so_interesting :
  ∀ (A : Type) (l1 l2 : list A),
    (∀ n : nat, drop n l1 = drop n l2) → l1 = l2.
```

Dualność *take* i *drop*

```
Lemma take_rev :
  ∀ (A : Type) (l : list A) (n : nat),
```

take n (*rev l*) = *rev* (*drop* (*length l* - *n*) *l*).

```
Lemma rev_take :
  ∀ (A : Type) (l : list A) (n : nat),
    rev (take n l) = drop (length l - n) (rev l).

Lemma drop_rev :
  ∀ (A : Type) (n : nat) (l : list A),
    drop n (rev l) = rev (take (length l - n) l).

Lemma take_drop :
  ∀ (A : Type) (l : list A) (n m : nat),
    take m (drop n l) = drop n (take (n + m) l).

Lemma drop_take :
  ∀ (A : Type) (l : list A) (n m : nat),
    drop m (take n l) = take (n - m) (drop m l).

Lemma app_take_drop :
  ∀ (A : Type) (l : list A) (n : nat),
    take n l ++ drop n l = l.
```

10.1.16 *splitAt*

Zdefiniuj funkcję *splitAt*, która spełnia poniższą specyfikację. Nie używaj take ani drop - użyj rekursji.

Przykład:
splitAt 2 [1; 2; 3; 4; 5] = *Some* ([1; 2], 3, [4; 5])

```
Lemma splitAt_spec :
  ∀ (A : Type) (l : list A) (n : nat),
    match splitAt n l with
        | None ⇒ length l ≤ n
        | Some (l1, x, l2) ⇒ l = l1 ++ x :: l2
    end.

Lemma splitAt_spec' :
  ∀ (A : Type) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n l = Some (l1, x, l2) →
      l1 = take n l ∧ l2 = drop (S n) l.

Lemma splitAt_megaspec :
  ∀ (A : Type) (l : list A) (n : nat),
    match splitAt n l with
        | None ⇒ length l ≤ n
        | Some (l1, x, l2) ⇒
            nth n l = Some x ∧
            l1 = take n l ∧
```

```
        l2 = drop (S n) l ∧
        l = l1 ++ x :: l2
    end.

Lemma splitAt_isEmpty_true :
  ∀ (A : Type) (l : list A),
    isEmpty l = true → ∀ n : nat, splitAt n l = None.

Lemma isEmpty_splitAt_false :
  ∀ (A : Type) (l : list A) (n : nat),
    splitAt n l ≠ None → isEmpty l = false.

Lemma splitAt_length_inv :
  ∀ (A : Type) (l : list A) (n : nat),
    splitAt n l ≠ None ↔ n < length l.

Lemma splitAt_Some_length :
  ∀ (A : Type) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n l = Some (l1, x, l2) → n < length l.

Lemma splitAt_length_lt :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l → ∃ x : A,
      splitAt n l = Some (take n l, x, drop (S n) l).

Lemma splitAt_length_ge :
  ∀ (A : Type) (l : list A) (n : nat),
    length l ≤ n → splitAt n l = None.

Lemma splitAt_snoc :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    splitAt n (snoc x l) =
    if n <? length l
    then
      match splitAt n l with
          | None ⇒ None
          | Some (b, y, e) ⇒ Some (b, y, snoc x e)
      end
    else
      if beq_nat n (length l)
      then Some (l, x, [])
      else None.

Lemma splitAt_app :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    splitAt n (l1 ++ l2) =
    match splitAt n l1 with
        | Some (l11, x, l12) ⇒ Some (l11, x, l12 ++ l2)
```

```
        | None ⇒
            match splitAt (n - length l1) l2 with
                | Some (l21, x, l22) ⇒ Some (l1 ++ l21, x, l22)
                | None ⇒ None
            end
    end.

Lemma splitAt_app_lt :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    n < length l1 →
      splitAt n (l1 ++ l2) =
      match splitAt n l1 with
          | None ⇒ None
          | Some (x, l11, l12) ⇒ Some (x, l11, l12 ++ l2)
      end.

Lemma splitAt_app_ge :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    length l1 ≤ n →
      splitAt n (l1 ++ l2) =
      match splitAt (n - length l1) l2 with
          | None ⇒ None
          | Some (l21, x, l22) ⇒ Some (l1 ++ l21, x, l22)
      end.

Lemma splitAt_rev_aux :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l →
      splitAt n l =
      match splitAt (length l - S n) (rev l) with
          | None ⇒ None
          | Some (l1, x, l2) ⇒ Some (rev l2, x, rev l1)
      end.

Lemma splitAt_rev :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l →
      splitAt n (rev l) =
      match splitAt (length l - S n) l with
          | None ⇒ None
          | Some (l1, x, l2) ⇒ Some (rev l2, x, rev l1)
      end.

Lemma splitAt_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (n : nat),
    splitAt n (map f l) =
```

```
    match splitAt n l with
        | None ⇒ None
        | Some (l1, x, l2) ⇒ Some (map f l1, f x, map f l2)
    end.

Lemma splitAt_replicate :
  ∀ (A : Type) (n m : nat) (x : A),
    splitAt m (replicate n x) =
      if m <? n
      then Some (replicate m x, x, replicate (n - S m) x)
      else None.

Lemma splitAt_iterate :
  ∀ (A : Type) (f : A → A) (n m : nat) (x : A),
    splitAt m (iterate f n x) =
    if m <? n
    then Some (iterate f m x, iter f m x, iterate f (n - S m) (iter f (S m) x))
    else None.

Lemma splitAt_head_l :
  ∀ (A : Type) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n l = Some (l1, x, l2) →
      head l1 =
      match n with
          | 0 ⇒ None
          | _ ⇒ head l
      end.

Lemma splitAt_head_r :
  ∀ (A : Type) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n l = Some (l1, x, l2) →
      head l2 = nth (S n) l.

Lemma splitAt_last_l :
  ∀ (A : Type) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n l = Some (l1, x, l2) →
      last l1 =
      match n with
          | 0 ⇒ None
          | S n' ⇒ nth n' l
      end.

Lemma splitAt_last_r :
  ∀ (A : Type) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n l = Some (l1, x, l2) →
      last l2 =
      if length l <=? S n
```

```
    then None
    else last l2.
```

```
(*  TODO: init, unsnoc *)
```

```
Lemma take_splitAt :
  ∀ (A : Type) (l l1 l2 : list A) (n m : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      take m l1 = take (min n m) l.

Lemma take_splitAt' :
  ∀ (A : Type) (l l1 l2 : list A) (n m : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      take m l2 = take m (drop (S n) l).

Lemma drop_splitAt_l :
  ∀ (A : Type) (l l1 l2 : list A) (n m : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      drop m l1 = take (n - m) (drop m l).

Lemma drop_splitAt_r :
  ∀ (A : Type) (l l1 l2 : list A) (n m : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      drop m l2 = drop (S n + m) l.
```

10.1.17 *insert*

Napisz funkcję *insert*, która wstawia do listy l na n-tą pozycję element x.

Przykład:

insert [1; 2; 3; 4; 5] 2 42 = [1; 2; 42; 3; 4; 5]

```
Lemma insert_0 :
  ∀ (A : Type) (l : list A) (x : A),
    insert l 0 x = x :: l.

Lemma isEmpty_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    isEmpty (insert l n x) = false.

Lemma length_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    length (insert l n x) = S (length l).

Lemma insert_length :
  ∀ (A : Type) (l : list A) (x : A),
    insert l (length l) x = snoc x l.

Lemma insert_snoc :
  ∀ (A : Type) (l : list A) (n : nat) (x y : A),
```

```
    insert (snoc x l) n y =
    if n <=? length l then snoc x (insert l n y) else snoc y (snoc x l).

Lemma insert_app :
  ∀ (A : Type) (l1 l2 : list A) (n : nat) (x : A),
    insert (l1 ++ l2) n x =
    if leb n (length l1)
    then insert l1 n x ++ l2
    else l1 ++ insert l2 (n - length l1) x.

Lemma insert_rev :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    insert (rev l) n x = rev (insert l (length l - n) x).

Lemma rev_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    rev (insert l n x) = insert (rev l) (length l - n) x.

Lemma map_insert :
  ∀ (A B : Type) (f : A → B) (l : list A) (n : nat) (x : A),
    map f (insert l n x) = insert (map f l) n (f x).

Lemma insert_join :
  ∀ (A : Type) (ll : list (list A)) (n : nat) (x : A) (l : list A),
    join (insert ll n [x]) = l →
      ∃ m : nat, l = insert (join ll) m x.

Lemma insert_replicate :
  ∀ (A : Type) (n m : nat) (x : A),
    insert (replicate n x) m x = replicate (S n) x.

Lemma head_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    head (insert l n x) =
    match l, n with
        | [], _ ⇒ Some x
        | _, 0 ⇒ Some x
        | _, _ ⇒ head l
    end.

Lemma tail_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    tail (insert l n x) =
    match l, n with
        | [], _ ⇒ Some []
        | _, 0 ⇒ Some l
        | h :: t, S n' ⇒ Some (insert t n' x)
    end.
```

```
Lemma last_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    last (insert l n x) =
    if isEmpty l
    then Some x
    else if leb (S n) (length l) then last l else Some x.

Lemma nth_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    n ≤ length l → nth n (insert l n x) = Some x.

Lemma nth_insert' :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    nth n (insert l n x) =
    if leb n (length l) then Some x else None.

Lemma insert_spec :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    insert l n x = take n l ++ x :: drop n l.

Lemma insert_take :
  ∀ (A : Type) (l : list A) (n m : nat) (x : A),
    insert (take n l) m x =
    if leb m n
    then take (S n) (insert l m x)
    else snoc x (take n l).

Lemma take_S_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    take (S n) (insert l n x) = snoc x (take n l).

Lemma take_insert :
  ∀ (A : Type) (l : list A) (n m : nat) (x : A),
    take m (insert l n x) =
    if m <=? n then take m l else snoc x l.

Lemma drop_S_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    drop (S n) (insert l n x) = drop n l.

Lemma insert_drop :
  ∀ (A : Type) (l : list A) (n m : nat) (x : A),
    insert (drop n l) m x =
    drop (n - 1) (insert l (n + m) x).
```

10.1.18 replace

Napisz funkcję `replace`, która na liście l zastępuje element z pozycji n elementem x.

Przykład:
replace [1; 2; 3; 4; 5] 2 42 = [1; 2; 42; 4; 5]

```
Lemma isEmpty_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      isEmpty l' = isEmpty l.

Lemma length_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' → length l' = length l.

Lemma replace_length_lt :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    n < length l →
      ∃ l' : list A, replace l n x = Some l'.

Lemma replace_length_ge :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    length l ≤ n → replace l n x = None.

Lemma replace_snoc_eq :
  ∀ (A : Type) (l : list A) (n : nat) (x y : A),
    n = length l → replace (snoc x l) n y = Some (snoc y l).

Lemma replace_snoc_neq :
  ∀ (A : Type) (l : list A) (n : nat) (x y : A),
    n ≠ length l →
      replace (snoc x l) n y =
      match replace l n y with
          | None ⇒ None
          | Some l' ⇒ Some (snoc x l')
      end.

Lemma replace_snoc :
  ∀ (A : Type) (l : list A) (n : nat) (x y : A),
    replace (snoc x l) n y =
    if beq_nat n (length l)
    then Some (snoc y l)
    else
      match replace l n y with
          | None ⇒ None
          | Some l' ⇒ Some (snoc x l')
      end.

Lemma replace_app :
  ∀ (A : Type) (l1 l2 : list A) (n : nat) (x : A),
    replace (l1 ++ l2) n x =
```

```
    match replace l1 n x, replace l2 (n - length l1) x with
        | None, None ⇒ None
        | Some l', _ ⇒ Some (l' ++ l2)
        | _, Some l' ⇒ Some (l1 ++ l')
    end.
Lemma replace_spec :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    replace l n x =
    if n <? length l
    then Some (take n l ++ x :: drop (S n) l)
    else None.
Lemma replace_spec' :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    n < length l →
      replace l n x = Some (take n l ++ x :: drop (S n) l).
Lemma replace_spec'' :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' → l' = take n l ++ x :: drop (S n) l.
Lemma replace_rev_aux :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    n < length l →
      replace l n x =
      match replace (rev l) (length l - S n) x with
          | None ⇒ None
          | Some l' ⇒ Some (rev l')
      end.
Definition omap {A B: Type} (f : A → B) (oa : option A) : option B :=
match oa with
    | None ⇒ None
    | Some a ⇒ Some (f a)
end.
Lemma replace_rev :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    n < length l →
      replace (rev l) n x = omap rev (replace l (length l - S n) x).
Lemma map_replace :
  ∀ (A B : Type) (f : A → B) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      Some (map f l') = replace (map f l) n (f x).
Lemma replace_join :
```

```
  ∀ (A : Type) (ll : list (list A)) (n : nat) (x : A) (l : list A),
    replace (join ll) n x = Some l →
      ∃ n m : nat,
        match nth n ll with
            | None ⇒ False
            | Some l' ⇒
                match replace l' m x with
                    | None ⇒ False
                    | Some l'' ⇒
                        match replace ll n l'' with
                            | None ⇒ False
                            | Some ll' ⇒ join ll' = l
                        end
                end
        end.

Lemma replace_replicate :
  ∀ (A : Type) (l l' : list A) (n m : nat) (x y : A),
    replace (replicate n x) m y =
    if n <=? m
    then None
    else Some (replicate m x ++ y :: replicate (n - S m) x).

Lemma replace_iterate :
  ∀ (A : Type) (f : A → A) (l : list A) (n m : nat) (x y : A),
    replace (iterate f n x) m y =
    if n <=? m
    then None
    else Some (iterate f m x ++
                  y :: iterate f (n - S m) (iter f (S m) x)).

Lemma head_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x y : A),
    replace l n x = Some l' →
      head l' =
      match n with
          | 0 ⇒ Some x
          | _ ⇒ head l
      end.

Lemma tail_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      tail l' =
      match n with
```

```
        | 0 ⇒ tail l
        | S n' ⇒
            match tail l with
                | None ⇒ None
                | Some t ⇒ replace t n' x
            end
    end.

Lemma replace_length_aux :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' → length l = length l'.

Lemma nth_replace :
  ∀ (A : Type) (l l' : list A) (n m : nat) (x : A),
    replace l n x = Some l' →
      nth m l' = if n =? m then Some x else nth m l.

Lemma replace_nth_eq :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      l = l' ↔ nth n l = Some x.

Lemma last_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      last l' =
      if n =? length l - 1
      then Some x
      else last l.

Lemma init_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      init l' =
      match init l with
          | None ⇒ None
          | Some i ⇒ if length i <=? n then Some i else replace i n x
      end.

Lemma take_replace :
  ∀ (A : Type) (l l' : list A) (n m : nat) (x : A),
    replace l n x = Some l' →
      take m l' =
      if m <=? n
      then take m l
      else take n l ++ x :: take (m - S n) (drop (S n) l).

Lemma drop_replace :
```

```
∀ (A : Type) (l l' : list A) (n m : nat) (x : A),
  replace l n x = Some l' →
    drop m l' =
    if n <? m
    then drop m l
    else take (n - m) (drop m l) ++ x :: drop (S n) l.
```

```
Lemma replace_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x y : A),
    n ≤ length l →
      replace (insert l n x) n y = Some (insert l n y).
```

```
Lemma replace_plus :
  ∀ (A : Type) (l : list A) (n m : nat) (x : A),
    replace l (n + m) x =
    match replace (drop n l) m x with
        | None ⇒ None
        | Some l' ⇒ Some (take n l ++ l')
    end.
```

10.1.19 *remove*

Napisz funkcję *remove*, która bierze liczbę naturalną n oraz listę l i zwraca parę składającą się z n-tego elementu listy l oraz tego, co pozostanie na liście po jego usunięciu. Jeżeli lista jest za krótka, funkcja ma zwracać *None*.

Przykład:

remove 2 [1; 2; 3; 4; 5] = *Some* (3, [1; 2; 4; 5])

remove 42 [1; 2; 3; 4; 5] = *None*

Uwaga TODO: w ćwiczeniach jest burdel.

```
Lemma remove'_S_cons :
  ∀ (A : Type) (n : nat) (h : A) (t : list A),
    remove' (S n) (h :: t) = h :: remove' n t.
```

```
Lemma remove_isEmpty_true :
  ∀ (A : Type) (l : list A) (n : nat),
    isEmpty l = true → remove n l = None.
```

```
Lemma isEmpty_remove_not_None :
  ∀ (A : Type) (l : list A) (n : nat),
    remove n l ≠ None → isEmpty l = false.
```

```
Lemma isEmpty_remove :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    remove n l = Some (x, l') →
      isEmpty l' = isEmpty l || ((length l <=? 1) && isZero n).
```

```
Lemma length_remove :
  ∀ (A : Type) (h : A) (l t : list A) (n : nat),
    remove n l = Some (h, t) → length l = S (length t).

Lemma remove_length_lt :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l →
      nth n l =
      match remove n l with
          | None ⇒ None
          | Some (h, _) ⇒ Some h
      end.

Lemma remove_length_lt' :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l → remove n l ≠ None.

Lemma remove_length_ge :
  ∀ (A : Type) (l : list A) (n : nat),
    length l ≤ n → remove n l = None.

Lemma remove_length_snoc :
  ∀ (A : Type) (x : A) (l : list A),
    remove (length l) (snoc x l) = Some (x, l).

Lemma remove_snoc_lt :
  ∀ (A : Type) (x : A) (l : list A) (n : nat),
    n < length l →
      remove n (snoc x l) =
      match remove n l with
          | None ⇒ None
          | Some (h, t) ⇒ Some (h, snoc x t)
      end.

Lemma remove_app :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    remove n (l1 ++ l2) =
    match remove n l1 with
        | Some (h, t) ⇒ Some (h, t ++ l2)
        | None ⇒
            match remove (n - length l1) l2 with
                | Some (h, t) ⇒ Some (h, l1 ++ t)
                | None ⇒ None
            end
    end.

Lemma remove_app_lt :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
```

```
    n < length l1 →
      remove n (l1 ++ l2) =
      match remove n l1 with
          | None ⇒ None
          | Some (h, t) ⇒ Some (h, t ++ l2)
      end.

Lemma remove_app_ge :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    length l1 ≤ n →
      remove n (l1 ++ l2) =
      match remove (n - length l1) l2 with
          | None ⇒ None
          | Some (h, t) ⇒ Some (h, l1 ++ t)
      end.

Lemma remove'_app :
  ∀ (A : Type) (n : nat) (l1 l2 : list A),
    n < length l1 →
      remove' n (l1 ++ l2) = remove' n l1 ++ l2.

Lemma remove_app' :
  ∀ (A : Type) (n : nat) (l1 l2 : list A),
    length l1 ≤ n →
      remove' n (l1 ++ l2) = l1 ++ remove' (n - length l1) l2.

Lemma remove_rev_aux :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l →
      remove n l =
      match remove (length l - S n) (rev l) with
          | None ⇒ None
          | Some (h, t) ⇒ Some (h, rev t)
      end.

Lemma remove_rev :
  ∀ (A : Type) (l : list A) (n : nat),
    n < length l →
      remove n (rev l) =
      match remove (length l - S n) l with
          | None ⇒ None
          | Some (h, t) ⇒ Some (h, rev t)
      end.

Lemma remove_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (n : nat),
    remove n (map f l) =
```

```
    match remove n l with
        | None ⇒ None
        | Some (x, l') ⇒ Some (f x, map f l')
    end.
```

```
Lemma remove_replicate :
  ∀ (A : Type) (n m : nat) (x : A),
    m < n → remove m (replicate n x) = Some (x, replicate (n - 1) x).
```

```
Lemma remove_iterate :
  ∀ (A : Type) (f : A → A) (n m : nat) (x : A),
    m < n →
      remove m (iterate f n x) =
      Some (iter f m x,
            iterate f m x ++
            (iterate f (n - S m) (iter f (S m) x))).
```

```
Lemma remove_nth_take_drop :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    nth n l = Some x ↔
    remove n l = Some (x, take n l ++ drop (S n) l).
```

```
Lemma remove_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    n < length l →
      remove n (insert l n x) = Some (x, l).
```

```
Lemma remove'_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      remove' n l' = remove' n l.
```

10.1.20 *zip*

Napisz funkcję *zip*, która bierze dwie listy i skleja je w listę par. Wywnioskuj z poniższej specyfikacji, jak dokładnie ma się zachowywać ta funkcja.

Przykład:
zip [1; 3; 5; 7] [2; 4; 6] = [(1, 2); (3, 4); (5, 6)]

```
Lemma zip_nil_l :
  ∀ (A B : Type) (l : list B), zip (@nil A) l = [].
```

```
Lemma zip_nil_r :
  ∀ (A B : Type) (l : list A), zip l (@nil B) = [].
```

```
Lemma isEmpty_zip :
  ∀ (A B : Type) (la : list A) (lb : list B),
    isEmpty (zip la lb) = orb (isEmpty la) (isEmpty lb).
```

```
Lemma length_zip :
  ∀ (A B : Type) (la : list A) (lb : list B),
    length (zip la lb) = min (length la) (length lb).

Lemma zip_not_rev :
  ∃ (A B : Type) (la : list A) (lb : list B),
    zip (rev la) (rev lb) ≠ rev (zip la lb).

Lemma head_zip :
  ∀ (A B : Type) (la : list A) (lb : list B) (a : A) (b : B),
    head la = Some a → head lb = Some b →
      head (zip la lb) = Some (a, b).

Lemma tail_zip :
  ∀ (A B : Type) (la ta : list A) (lb tb : list B),
    tail la = Some ta → tail lb = Some tb →
      tail (zip la lb) = Some (zip ta tb).

Lemma zip_not_app :
  ∃ (A B : Type) (la la' : list A) (lb lb' : list B),
    zip (la ++ la') (lb ++ lb') ≠ zip la lb ++ zip la' lb'.

Lemma zip_map :
  ∀ (A B A' B' : Type) (f : A → A') (g : B → B')
  (la : list A) (lb : list B),
    zip (map f la) (map g lb) =
    map (fun x ⇒ (f (fst x), g (snd x))) (zip la lb).

Lemma zip_replicate :
  ∀ (A B : Type) (n m : nat) (a : A) (b : B),
    zip (replicate n a) (replicate m b) =
    replicate (min n m) (a, b).

Lemma zip_iterate :
  ∀
    (A B : Type) (fa : A → A) (fb : B → B) (na nb : nat) (a : A) (b : B),
      zip (iterate fa na a) (iterate fb nb b) =
      iterate (fun '(a, b) ⇒ (fa a, fb b)) (min na nb) (a, b).

Lemma nth_zip :
  ∀ (A B : Type) (la : list A) (lb : list B) (n : nat),
    nth n (zip la lb) =
    if n <=? min (length la) (length lb)
    then
      match nth n la, nth n lb with
          | Some a, Some b ⇒ Some (a, b)
          | _, _ ⇒ None
      end
```

```
    else None.
Lemma nth_zip' :
  ∀ (A B : Type) (la : list A) (lb : list B) (n : nat),
    nth n (zip la lb) =
    match nth n la, nth n lb with
        | Some a, Some b ⇒ Some (a, b)
        | _, _ ⇒ None
    end.
Lemma zip_take :
  ∀ (A B : Type) (la : list A) (lb : list B) (n : nat),
    zip (take n la) (take n lb) = take n (zip la lb).
Lemma zip_drop :
  ∀ (A B : Type) (la: list A) (lb : list B) (n : nat),
    zip (drop n la) (drop n lb) = drop n (zip la lb).
Lemma splitAt_zip :
  ∀ (A B : Type) (la : list A) (lb : list B) (n : nat),
    splitAt n (zip la lb) =
    match splitAt n la, splitAt n lb with
        | Some (la1, a, la2), Some (lb1, b, lb2) ⇒
            Some (zip la1 lb1, (a, b), zip la2 lb2)
        | _, _ ⇒ None
    end.
Lemma insert_zip :
  ∀ (A B : Type) (la : list A) (lb : list B) (a : A) (b : B) (n : nat),
    insert (zip la lb) n (a, b) =
    if n <=? min (length la) (length lb)
    then zip (insert la n a) (insert lb n b)
    else snoc (a, b) (zip la lb).
Lemma replace_zip :
  ∀
    (A B : Type) (la la' : list A) (lb lb' : list B)
    (n : nat) (a : A) (b : B),
      replace la n a = Some la' →
      replace lb n b = Some lb' →
        replace (zip la lb) n (a, b) = Some (zip la' lb').
Lemma replace_zip' :
  ∀
    (A B : Type) (la : list A) (lb : list B) (n : nat) (a : A) (b : B),
      replace (zip la lb) n (a, b) =
      match replace la n a, replace lb n b with
          | Some la', Some lb' ⇒ Some (zip la' lb')
```

```
    | _, _ ⇒ None
  end.
```

```
Lemma remove_zip :
  ∀ (A B : Type) (la : list A) (lb : list B) (n : nat),
    remove n (zip la lb) =
    match remove n la, remove n lb with
        | Some (a, la'), Some (b, lb') ⇒ Some ((a, b), zip la' lb')
        | _, _ ⇒ None
    end.
```

10.1.21 *unzip*

Zdefiniuj funkcję *unzip*, która jest w pewnym sensie "odwrotna" do *zip*.

Przykład:

unzip [(1, 2); (3, 4); (5, 6)] = ([1; 3; 5], [2; 4; 6])

```
Lemma zip_unzip :
  ∀ (A B : Type) (l : list (A × B)),
    zip (fst (unzip l)) (snd (unzip l)) = l.
```

```
Lemma unzip_zip :
  ∃ (A B : Type) (la : list A) (lb : list B),
    unzip (zip la lb) ≠ (la, lb).
```

```
Lemma isEmpty_unzip :
  ∀ (A B : Type) (l : list (A × B)) (la : list A) (lb : list B),
    unzip l = (la, lb) → isEmpty l = orb (isEmpty la) (isEmpty lb).
```

```
Lemma unzip_snoc :
  ∀ (A B : Type) (x : A × B) (l : list (A × B)),
    unzip (snoc x l) =
      let (la, lb) := unzip l in (snoc (fst x) la, snoc (snd x) lb).
```

10.1.22 *zipWith*

Zdefiniuj funkcję *zipWith*, która zachowuje się jak połączenie *zip* i *map*. Nie używaj *zip* ani *map* - użyj rekursji.

Przykład:

zipWith plus [1; 2; 3] [4; 5; 6] = [5; 7; 9]

```
Lemma zipWith_spec :
  ∀ (A B C : Type) (f : A → B → C)
  (la : list A) (lb : list B),
    zipWith f la lb =
    map (fun '(a, b) ⇒ f a b) (zip la lb).
```

```
Lemma zipWith_pair :
  ∀ (A B : Type) (la : list A) (lb : list B),
    zipWith pair la lb = zip la lb.

Lemma isEmpty_zipWith :
  ∀ (A B C : Type) (f : A → B → C) (la : list A) (lb : list B),
    isEmpty (zipWith f la lb) = orb (isEmpty la) (isEmpty lb).

Lemma zipWith_snoc :
  ∀
    (A B C : Type) (f : A → B → C)
    (a : A) (b : B) (la : list A) (lb : list B),
      length la = length lb →
        zipWith f (snoc a la) (snoc b lb) =
        snoc (f a b) (zipWith f la lb).

Lemma zipWith_iterate :
  ∀
    (A B C: Type) (fa : A → A) (fb : B → B) (g : A → B → C)
    (na nb : nat) (a : A) (b : B),
      zipWith g (iterate fa na a) (iterate fb nb b) =
      map (fun '(a, b) ⇒ g a b)
        (iterate (fun '(a, b) ⇒ (fa a, fb b)) (min na nb) (a, b)).

Lemma take_zipWith :
  ∀
    (A B C : Type) (f : A → B → C) (la : list A) (lb : list B) (n : nat),
      take n (zipWith f la lb) = zipWith f (take n la) (take n lb).

Lemma drop_zipWith :
  ∀
    (A B C : Type) (f : A → B → C) (la : list A) (lb : list B) (n : nat),
      drop n (zipWith f la lb) = zipWith f (drop n la) (drop n lb).

Lemma splitAt_zipWith :
  ∀ (A B C : Type) (f : A → B → C)
    (la : list A) (lb : list B) (n : nat),
      splitAt n (zipWith f la lb) =
      match splitAt n la, splitAt n lb with
          | Some (la1, a, la2), Some (lb1, b, lb2) ⇒
              Some (zipWith f la1 lb1, f a b, zipWith f la2 lb2)
          | _, _ ⇒ None
      end.

Lemma replace_zipWith :
  ∀
    (A B C : Type) (f : A → B → C) (la la' : list A) (lb lb' : list B)
```

```
    (n : nat) (a : A) (b : B),
      replace la n a = Some la' →
      replace lb n b = Some lb' →
        replace (zipWith f la lb) n (f a b) = Some (zipWith f la' lb').
```

```
Lemma remove_zipWith :
  ∀ (A B C : Type) (f : A → B → C)
    (la : list A) (lb : list B) (n : nat),
      remove n (zipWith f la lb) =
      match remove n la, remove n lb with
          | Some (a, la'), Some (b, lb') ⇒
              Some (f a b, zipWith f la' lb')
          | _, _ ⇒ None
      end.
```

10.1.23 *unzipWith*

Zdefiniuj funkcję *unzipWith*, która ma się tak do *zipWith*, jak *unzip* do *zip*. Oczywiście użyj rekursji i nie używaj żadnych funkcji pomocniczych.

```
Lemma isEmpty_unzipWith :
  ∀ (A B C : Type) (f : A → B × C) (l : list A)
  (lb : list B) (lc : list C),
    unzipWith f l = (lb, lc) →
      isEmpty l = orb (isEmpty lb) (isEmpty lc).
```

```
Lemma unzipWith_spec :
  ∀ (A B C : Type) (f : A → B × C) (l : list A),
    unzipWith f l = unzip (map f l).
```

```
Lemma unzipWith_id :
  ∀ (A B : Type) (l : list (A × B)),
    unzipWith id l = unzip l.
```

10.2 Funkcje z predykatem boolowskim

10.2.1 *any*

Napisz funkcję *any*, która sprawdza, czy lista l zawiera jakiś element, który spełnia predykat boolowski p.

Przykład: *any even* [3; 5; 7; 11] = *false*

```
Lemma any_isEmpty_true :
  ∀ (A : Type) (p : A → bool) (l : list A),
    isEmpty l = true → any p l = false.
```

Lemma $isEmpty\_any\_true$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $any\ p\ l = true \rightarrow isEmpty\ l = false$.

Lemma any_length :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $any\ p\ l = true \rightarrow 1 \leq length\ l$.

Lemma any_snoc :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (x : A) (l : $list\ A$),
    $any\ p\ (snoc\ x\ l) = orb\ (any\ p\ l)\ (p\ x)$.

Lemma any_app :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) ($l1\ l2$: $list\ A$),
    $any\ p\ (l1 ++ l2) = orb\ (any\ p\ l1)\ (any\ p\ l2)$.

Lemma any_rev :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $any\ p\ (rev\ l) = any\ p\ l$.

Lemma any_map :
  $\forall$ ($A\ B$: Type) (f : $A \rightarrow B$) (p : $B \rightarrow bool$) (l : $list\ A$),
    $any\ p\ (map\ f\ l) = any$ (fun x : $A \Rightarrow p\ (f\ x)$) l.

Lemma any_join :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ (list\ A)$),
    $any\ p\ (join\ l) = any\ (any\ p)\ l$.

Lemma $any\_replicate$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (n : nat) (x : A),
    $any\ p\ (replicate\ n\ x) = andb\ (leb\ 1\ n)\ (p\ x)$.

Lemma $any\_iterate$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (f : $A \rightarrow A$) (n : nat) (x : A),
    ($\forall\ x : A,\ p\ (f\ x) = p\ x$) $\rightarrow$
      $any\ p\ (iterate\ f\ n\ x) =$
      match n with
        | $0 \Rightarrow false$
        | $\_ \Rightarrow p\ x$
      end.

Lemma any_nth :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $any\ p\ l = true \leftrightarrow$
    $\exists\ (n : nat)\ (x : A),\ nth\ n\ l = Some\ x \land p\ x = true$.

Lemma any_take :
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$) (n : nat),
    $any\ p\ (take\ n\ l) = true \rightarrow any\ p\ l = true$.

Lemma any_take_conv :

$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
$any\ p\ l = false \to any\ p\ (take\ n\ l) = false$.

Lemma any_drop :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
$any\ p\ (drop\ n\ l) = true \to any\ p\ l = true$.

Lemma any_drop_conv :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
$any\ p\ l = false \to any\ p\ (drop\ n\ l) = false$.

Lemma any_insert :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat) (x : A),
$any\ p\ (insert\ l\ n\ x) = orb\ (p\ x)\ (any\ p\ l)$.

Lemma $any\_replace$:
$\forall$ (A : Type) (p : $A \to bool$) ($l\ l'$: $list\ A$) (n : nat) (x : A),
replace $l\ n\ x = Some\ l' \to$
$any\ p\ l' = any\ p\ (take\ n\ l)\ ||\ p\ x\ ||\ any\ p\ (drop\ (S\ n)\ l)$.

Lemma $any\_replace'$:
$\forall$ (A : Type) (p : $A \to bool$) ($l\ l'$: $list\ A$) (n : nat) (x : A),
replace $l\ n\ x = Some\ l' \to$
$any\ p\ l = true \to p\ x = true \to any\ p\ l' = true$.

Lemma any_true :
$\forall$ (A : Type) (l : $list\ A$),
any (fun _ $\Rightarrow true$) $l = negb\ (isEmpty\ l)$.

Lemma any_false :
$\forall$ (A : Type) (l : $list\ A$),
any (fun _ $\Rightarrow false$) $l = false$.

Lemma any_orb :
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
any (fun x : $A \Rightarrow orb\ (p\ x)\ (q\ x)$) l =
$orb\ (any\ p\ l)\ (any\ q\ l)$.

Lemma any_andb :
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
any (fun x : $A \Rightarrow andb\ (p\ x)\ (q\ x)$) $l = true \to$
$any\ p\ l = true \land any\ q\ l = true$.

10.2.2 *all*

Napisz funkcję *all*, która sprawdza, czy wszystkie wartości na liście l spełniają predykat boolowski p.

Przykład: $all\ even\ [2;\ 4;\ 6] = true$

Lemma $all\_isEmpty\_true$:
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $isEmpty\ l = true \to all\ p\ l = true.$

Lemma $isEmpty\_all\_false$:
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $all\ p\ l = false \to isEmpty\ l = false.$

Lemma all_length :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $all\ p\ l = false \to 1 \leq length\ l.$

Lemma all_snoc :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (x : A)\ (l : list\ A),$
    $all\ p\ (snoc\ x\ l) = andb\ (all\ p\ l)\ (p\ x).$

Lemma all_app :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l1\ l2 : list\ A),$
    $all\ p\ (l1 \mathrel{++} l2) = andb\ (all\ p\ l1)\ (all\ p\ l2).$

Lemma all_rev :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $all\ p\ (rev\ l) = all\ p\ l.$

Lemma all_map :
  $\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (p : B \to bool)\ (l : list\ A),$
    $all\ p\ (map\ f\ l) = all\ (\texttt{fun}\ x : A \Rightarrow p\ (f\ x))\ l.$

Lemma all_join :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ (list\ A)),$
    $all\ p\ (join\ l) = all\ (all\ p)\ l.$

Lemma $all\_replicate$:
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (n : nat)\ (x : A),$
    $all\ p\ (replicate\ n\ x) = orb\ (n <=?\ 0)\ (p\ x).$

Lemma $all\_iterate$:
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (f : A \to A)\ (n : nat)\ (x : A),$
    $(\forall\ x : A,\ p\ (f\ x) = p\ x) \to$
      $all\ p\ (iterate\ f\ n\ x) = orb\ (isZero\ n)\ (p\ x).$

Lemma all_nth :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $all\ p\ l = true \leftrightarrow$
    $\forall\ n : nat,\ n < length\ l \to \exists\ x : A,$
      $nth\ n\ l = Some\ x \land p\ x = true.$

Lemma all_take :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A)\ (n : nat),$
    $all\ p\ (take\ n\ l) = false \to all\ p\ l = false.$

Lemma all_take_conv :
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
    $all\ p\ l = true \to all\ p\ (take\ n\ l) = true$.

Lemma all_drop :
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
    $all\ p\ (drop\ n\ l) = false \to all\ p\ l = false$.

Lemma all_drop_conv :
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
    $all\ p\ l = true \to all\ p\ (drop\ n\ l) = true$.

Lemma all_insert :
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat) (x : A),
    $all\ p\ (insert\ l\ n\ x) = andb\ (p\ x)\ (all\ p\ l)$.

Lemma $all\_replace$:
  $\forall$ (A : Type) (p : $A \to bool$) ($l\ l'$: $list\ A$) (n : nat) (x : A),
    replace $l\ n\ x = Some\ l' \to$
      $all\ p\ l' = all\ p\ (take\ n\ l)$ && $p\ x$ && $all\ p\ (drop\ (S\ n)\ l)$.

Lemma $all\_replace'$:
  $\forall$ (A : Type) (p : $A \to bool$) ($l\ l'$: $list\ A$) (n : nat) (x : A),
    replace $l\ n\ x = Some\ l' \to$
      $all\ p\ l = true \to p\ x = true \to all\ p\ l' = true$.

Lemma all_true :
  $\forall$ (A : Type) (l : $list\ A$),
    all (fun _ $\Rightarrow true$) $l = true$.

Lemma all_false :
  $\forall$ (A : Type) (l : $list\ A$),
    all (fun _ $\Rightarrow false$) $l = isEmpty\ l$.

Lemma all_orb :
  $\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
    $orb\ (all\ p\ l)\ (all\ q\ l) = true \to$
    all (fun x : $A \Rightarrow orb\ (p\ x)\ (q\ x)$) $l = true$.

Lemma all_andb :
  $\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
    all (fun x : $A \Rightarrow andb\ (p\ x)\ (q\ x)$) $l =$
    $andb\ (all\ p\ l)\ (all\ q\ l)$.

Lemma any_all :
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
    $any\ p\ l = negb$ (all (fun x : $A \Rightarrow negb\ (p\ x)$) l).

Lemma all_any :
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),

$all\ p\ l = negb\ (any\ (\texttt{fun}\ x : A \Rightarrow negb\ (p\ x))\ l).$

```
Lemma isEmpty_join :
  ∀ (A : Type) (l : list (list A)),
    isEmpty (join l) = all isEmpty l.
```

10.2.3 *find* i *findLast*

Napisz funkcję *find*, która znajduje pierwszy element na liście, który spełnia podany predykat boolowski.

Przykład: *find even* [1; 2; 3; 4] = *Some* 2

Napisz też funkcję *findLast*, która znajduje ostatni element na liście, który spełnia podany predykat boolowski.

Przykład: *findLast even* [1; 2; 3; 4] = *Some* 4

```
Lemma find_false :
  ∀ (A : Type) (l : list A),
    find (fun _ ⇒ false) l = None.

Lemma find_true :
  ∀ (A : Type) (l : list A),
    find (fun _ ⇒ true) l = head l.

Lemma find_isEmpty_true :
  ∀ (A : Type) (p : A → bool) (l : list A),
    isEmpty l = true → find p l = None.

Lemma isEmpty_find_not_None :
  ∀ (A : Type) (p : A → bool) (l : list A),
    find p l ≠ None → isEmpty l = false.

Lemma find_length :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    find p l = Some x → 1 ≤ length l.

Lemma find_snoc :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    find p (snoc x l) =
    match find p l with
        | None ⇒ if p x then Some x else None
        | Some y ⇒ Some y
    end.

Lemma findLast_snoc :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    findLast p (snoc x l) =
    if p x then Some x else findLast p l.

Lemma find_app :
```

$\forall$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
  $find$ p ($l1$ ++ $l2$) =
  match $find$ p $l1$ with
    | $Some$ $x \Rightarrow Some$ x
    | $None \Rightarrow find$ p $l2$
  end.

Lemma $find\_rev$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list$ A),
  $find$ p (rev l) = $findLast$ p l.

Lemma $find\_findLast$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list$ A),
  $find$ p l = $findLast$ p (rev l).

Lemma $find\_map$:
$\forall$ (A B : Type) (f : $A \to B$) (p : $B \to bool$) (l : $list$ A),
  $find$ p (map f l) =
  match $find$ (fun x : $A \Rightarrow p$ (f x)) l with
    | $None \Rightarrow None$
    | $Some$ $a \Rightarrow Some$ (f a)
  end.

Lemma $find\_join$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list$ ($list$ A)),
  $find$ p ($join$ l) =
  (fix aux (l : $list$ ($list$ A)) : $option$ A :=
  match l with
    | [] $\Rightarrow None$
    | h :: $t \Rightarrow$
      match $find$ p h with
        | $None \Rightarrow aux$ t
        | $Some$ $x \Rightarrow Some$ x
      end
  end) l.

Lemma $find\_replicate$:
$\forall$ (A : Type) (p : $A \to bool$) (n : nat) (x : A),
  $find$ p ($replicate$ n x) =
  match n, p x with
    | 0, _ $\Rightarrow None$
    | _, $false \Rightarrow None$
    | _, $true \Rightarrow Some$ x
  end.

Lemma $find\_iterate$:
$\forall$ (A : Type) (p : $A \to bool$) (f : $A \to A$) (n : nat) (x : A),

```
    (∀ x : A, p (f x) = p x) →
      find p (iterate f n x) =
      if isZero n then None else if p x then Some x else None.
Lemma findLast_iterate :
  ∀ (A : Type) (p : A → bool) (f : A → A) (n : nat) (x : A),
    (∀ x : A, p (f x) = p x) →
      findLast p (iterate f n x) =
      match n with
          | 0 ⇒ None
          | S n' ⇒ if p x then Some (iter f n' x) else None
      end.
Lemma find_nth :
  ∀ (A : Type) (p : A → bool) (l : list A),
    (∃ (n : nat) (x : A), nth n l = Some x ∧ p x = true) ↔
    find p l ≠ None.
Lemma find_tail :
  ∀ (A : Type) (p : A → bool) (l t : list A),
    tail l = Some t → find p t ≠ None → find p l ≠ None.
Lemma find_init :
  ∀ (A : Type) (p : A → bool) (l t : list A),
    init l = Some t → find p t ≠ None → find p l ≠ None.
Lemma find_take_Some :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat) (x : A),
    find p (take n l) = Some x → find p l = Some x.
Lemma find_take_None :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat),
    find p l = None → find p (take n l) = None.
Lemma find_drop_not_None :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat),
    find p (drop n l) ≠ None → find p l ≠ None.
Lemma find_drop_None :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat),
    find p l = None → find p (drop n l) = None.
Lemma findLast_take :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat),
    findLast p (take n l) ≠ None → findLast p l ≠ None.
Lemma findLast_drop :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat) (x : A),
    findLast p (drop n l) = Some x → findLast p l = Some x.
Lemma find_replace :
```

$\forall$ (A : Type) (p : $A \to bool$) (l l' : $list$ A) (n : nat) (x : A),
replace l n x = $Some$ l' $\to$
find p l' =
match *find* p (*take* n l), p x with
| *Some* y, _ $\Rightarrow$ *Some* y
| _, *true* $\Rightarrow$ *Some* x
| _, _ $\Rightarrow$ *find* p (*drop* (S n) l)
end.

Lemma *replace_findLast* :
$\forall$ (A : Type) (p : $A \to bool$) (l l' : $list$ A) (n : nat) (x : A),
replace l n x = $Some$ l' $\to$
findLast p l' =
match *findLast* p (*drop* (S n) l), p x with
| *Some* y, _ $\Rightarrow$ *Some* y
| _, *true* $\Rightarrow$ *Some* x
| _, _ $\Rightarrow$ *findLast* p (*take* n l)
end.

10.2.4 *removeFirst* i *removeLast*

Napisz funkcje *removeFirst* i *removeLast* o sygnaturach, które zwracają pierwszy/ostatni element z listy spełniający predykat boolowski p oraz resztę listy bez tego elementu.

Przykład:
removeFirst even [1; 2; 3; 4] = *Some* (2, [1; 3; 4])
removeLast even [1; 2; 3; 4] = *Some* (4, [1; 2; 3])

Lemma *removeFirst_isEmpty_true* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list$ A),
isEmpty l = *true* $\to$ *removeFirst* p l = *None*.

Lemma *isEmpty_removeFirst_not_None* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list$ A),
removeFirst p l $\neq$ *None* $\to$ *isEmpty* l = *false*.

Lemma *removeFirst_satisfies* :
$\forall$ (A : Type) (p : $A \to bool$) (l l' : $list$ A) (x : A),
removeFirst p l = *Some* (x, l') $\to$ p x = *true*.

Lemma *removeFirst_length* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list$ A),
length l = 0 $\to$ *removeFirst* p l = *None*.

Lemma *removeFirst_snoc* :
$\forall$ (A : Type) (p : $A \to bool$) (x : A) (l : $list$ A),
removeFirst p (*snoc* x l) =
match *removeFirst* p l with

```
      | None ⇒ if p x then Some (x, l) else None
      | Some (h, t) ⇒ Some (h, snoc x t)
  end.

Lemma removeLast_snoc :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    removeLast p (snoc x l) =
    if p x
    then Some (x, l)
    else
      match removeLast p l with
          | None ⇒ None
          | Some (h, t) ⇒ Some (h, snoc x t)
      end.

Lemma removeFirst_app :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    removeFirst p (l1 ++ l2) =
    match removeFirst p l1, removeFirst p l2 with
        | Some (h, t), _ ⇒ Some (h, t ++ l2)
        | _, Some (h, t) ⇒ Some (h, l1 ++ t)
        | _, _ ⇒ None
    end.

Lemma removeLast_app :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    removeLast p (l1 ++ l2) =
    match removeLast p l2, removeLast p l1 with
        | Some (y, l'), _ ⇒ Some (y, l1 ++ l')
        | _, Some (y, l') ⇒ Some (y, l' ++ l2)
        | _, _ ⇒ None
    end.

Lemma removeFirst_rev :
  ∀ (A : Type) (p : A → bool) (l : list A),
    removeFirst p (rev l) =
    match removeLast p l with
        | Some (x, l) ⇒ Some (x, rev l)
        | None ⇒ None
    end.

Lemma removeLast_rev :
  ∀ (A : Type) (p : A → bool) (l : list A),
    removeLast p (rev l) =
    match removeFirst p l with
        | None ⇒ None
```

```
        | Some (x, l) ⇒ Some (x, rev l)
    end.

Lemma removeFirst_map :
  ∀ (A B : Type) (p : B → bool) (f : A → B) (l : list A),
    removeFirst p (map f l) =
    match removeFirst (fun x ⇒ p (f x)) l with
        | Some (x, l) ⇒ Some (f x, map f l)
        | None ⇒ None
    end.

Lemma removeFirst_join :
  ∀ (A : Type) (p : A → bool) (l : list (list A)),
    removeFirst p (join l) =
    (fix f (l : list (list A)) : option (A × list A) :=
    match l with
        | [] ⇒ None
        | hl :: tl ⇒
            match removeFirst p hl with
                | Some (x, l') ⇒ Some (x, join (l' :: tl))
                | None ⇒
                    match f tl with
                        | Some (x, l) ⇒ Some (x, hl ++ l)
                        | None ⇒ None
                    end
            end
    end) l.

Lemma removeFirst_replicate :
  ∀ (A : Type) (p : A → bool) (n : nat) (x : A),
    removeFirst p (replicate n x) =
    if p x
    then
        match n with
            | 0 ⇒ None
            | S n' ⇒ Some (x, replicate n' x)
        end
    else None.

Lemma removeFirst_nth_None :
  ∀ (A : Type) (p : A → bool) (l : list A),
    removeFirst p l = None ↔
      ∀ (n : nat) (x : A), nth n l = Some x → p x = false.

Lemma removeFirst_nth_Some :
  ∀ (A : Type) (p : A → bool) (x : A) (l l' : list A),
```

```
    removeFirst p l = Some (x, l') →
    ∃ n : nat, nth n l = Some x ∧ p x = true.
Lemma removeFirst_nth_Some' :
  ∃ (A : Type) (p : A → bool) (n : nat) (x y : A) (l l' : list A),
    removeFirst p l = Some (x, l') ∧
    nth n l = Some y ∧ p y = true.
Lemma head_removeFirst :
  ∀ (A : Type) (p : A → bool) (x : A) (l l' : list A),
    removeFirst p l = Some (x, l') →
    head l' =
    match l with
        | [] ⇒ None
        | h :: t ⇒ if p h then head t else Some h
   end.
Lemma removeFirst_take_None :
  ∀ (A : Type) (p : A → bool) (n : nat) (l : list A),
    removeFirst p l = None → removeFirst p (take n l) = None.
Lemma removeFirst_take :
  ∀ (A : Type) (p : A → bool) (n : nat) (x : A) (l l' : list A),
    removeFirst p (take n l) = Some (x, l') →
      removeFirst p l = Some (x, l' ++ drop n l).
Lemma removeLast_drop :
  ∀ (A : Type) (p : A → bool) (n : nat) (x : A) (l l' : list A),
    removeLast p (drop n l) = Some (x, l') →
      removeLast p l = Some (x, take n l ++ l').
Lemma removeFirst_replace :
  ∀ (A : Type) (p : A → bool) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      removeFirst p l' =
      match removeFirst p (take n l), p x, removeFirst p (drop (S n) l) with
          | Some (y, l''), _, _ ⇒ Some (y, l'' ++ x :: drop (S n) l)
          | _, true, _ ⇒ Some (x, take n l ++ drop (S n) l)
          | _, _, Some (y, l'') ⇒ Some (y, take n l ++ x :: l'')
          | _, _, _ ⇒ None
      end.
Lemma removeLast_replace :
  ∀ (A : Type) (p : A → bool) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
    removeLast p l' =
    match removeLast p (drop (S n) l), p x, removeLast p (take n l) with
        | Some (y, l''), _ , _ ⇒ Some (y, take n l ++ x :: l'')
```

| _, *true*, _ ⇒ *Some* (*x*, *take n l* ++ *drop* (*S n*) *l*)
| _, _, *Some* (*y*, *l''*) ⇒ *Some* (*y*, *l''* ++ *x* :: *drop* (*S n*) *l*)
| _, _, _ ⇒ *None*
`end`.

`Lemma` *removeFirst_any_None* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l* : *list A*),
removeFirst p l = *None* ↔ *any p l* = *false*.

`Lemma` *removeFirst_not_None_any* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l* : *list A*),
removeFirst p l ≠ *None* ↔ *any p l* = *true*.

`Lemma` *removeFirst_None_iff_all* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l* : *list A*),
removeFirst p l = *None* ↔
all (`fun` *x* : *A* ⇒ *negb* (*p x*)) *l* = *true*.

10.2.5 *findIndex*

Napisz funkcję *findIndex*, która znajduje indeks pierwszego elementu, który spełnia predykat boolowski p. Pamiętaj, że indeksy liczone są od 0.

Przykład:
findIndex even [1; 3; 4; 5; 7] = 2

`Lemma` *findIndex_false* :
∀ (*A* : `Type`) (*l* : *list A*),
findIndex (`fun` _ ⇒ *false*) *l* = *None*.

`Lemma` *findIndex_true* :
∀ (*A* : `Type`) (*l* : *list A*),
findIndex (`fun` _ ⇒ *true*) *l* =
`match` *l* `with`
| [] ⇒ *None*
| _ ⇒ *Some* 0
`end`.

`Lemma` *findIndex_orb* :
∀ (*A* : `Type`) (*p q* : *A* → *bool*) (*l* : *list A*),
findIndex (`fun` *x* : *A* ⇒ *orb* (*p x*) (*q x*)) *l* =
`match` *findIndex p l*, *findIndex q l* `with`
| *Some n*, *Some m* ⇒ *Some* (*min n m*)
| *Some n*, *None* ⇒ *Some n*
| *None*, *Some m* ⇒ *Some m*
| _, _ ⇒ *None*
`end`.

```
Lemma findIndex_isEmpty_true :
  ∀ (A : Type) (p : A → bool) (l : list A),
    isEmpty l = true → findIndex p l = None.

Lemma isEmpty_findIndex_not_None :
  ∀ (A : Type) (p : A → bool) (l : list A),
    findIndex p l ≠ None → isEmpty l = false.

Lemma findIndex_length :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat),
    findIndex p l = Some n → n < length l.

Lemma findIndex_snoc :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    findIndex p (snoc x l) =
    match findIndex p l with
        | None ⇒ if p x then Some (length l) else None
        | Some n ⇒ Some n
    end.

Lemma findIndex_app_l :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A) (n : nat),
    findIndex p l1 = Some n → findIndex p (l1 ++ l2) = Some n.

Lemma findIndex_app_r :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A) (n : nat),
    findIndex p l1 = None → findIndex p l2 = Some n →
      findIndex p (l1 ++ l2) = Some (length l1 + n).

Lemma findIndex_app_None :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    findIndex p l1 = None → findIndex p l2 = None →
      findIndex p (l1 ++ l2) = None.

Lemma findIndex_app :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    findIndex p (l1 ++ l2) =
    match findIndex p l1, findIndex p l2 with
        | Some n, _ ⇒ Some n
        | _, Some n ⇒ Some (length l1 + n)
        | _, _ ⇒ None
    end.

Lemma findIndex_map :
  ∀ (A B : Type) (p : B → bool) (f : A → B) (l : list A) (n : nat),
    findIndex (fun x : A ⇒ p (f x)) l = Some n →
      findIndex p (map f l) = Some n.

Lemma findIndex_map_conv :
```

$\forall$ (A B : `Type`) (p : $B \to bool$) (f : $A \to B$) (l : $list$ A) (n : nat),
  ($\forall$ x y : A, f x = f y $\to$ x = y) $\to$
  $findIndex$ p (map f l) = $Some$ n $\to$
    $findIndex$ (`fun` x : $A \Rightarrow p$ (f x)) l = $Some$ n.

`Lemma` $findIndex\_join$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (ll : $list$ ($list$ A)),
  $findIndex$ p ($join$ ll) =
  `match` ll `with`
    | [] $\Rightarrow$ $None$
    | h :: t $\Rightarrow$
      `match` $findIndex$ p h, $findIndex$ p ($join$ t) `with`
        | $Some$ n, _ $\Rightarrow$ $Some$ n
        | _, $Some$ n $\Rightarrow$ $Some$ ($length$ h + n)
        | _, _ $\Rightarrow$ $None$
      `end`
  `end`.

`Lemma` $findIndex\_replicate$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (n : nat) (x : A),
  $findIndex$ p ($replicate$ n x) =
  `match` n `with`
    | 0 $\Rightarrow$ $None$
    | _ $\Rightarrow$ `if` p x `then` $Some$ 0 `else` $None$
  `end`.

`Lemma` $findIndex\_nth$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (l : $list$ A) (n : nat),
  $findIndex$ p l = $Some$ n $\to$
    $\exists$ x : A, nth n l = $Some$ x $\land$ p x = $true$.

`Lemma` $findIndex\_nth\_conv$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (l : $list$ A) (n : nat) (x : A),
  nth n l = $Some$ x $\to$ p x = $true$ $\to$
    $\exists$ m : nat, $findIndex$ p l = $Some$ m $\land$ $m \le n$.

`Lemma` $findIndex\_nth'$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (l : $list$ A) (n : nat),
  $findIndex$ p l = $Some$ n $\to$ $find$ p l = nth n l.

`Lemma` $findIndex\_head$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (l : $list$ A),
  $findIndex$ p l = $Some$ 0 $\leftrightarrow$
  $\exists$ x : A, $head$ l = $Some$ x $\land$ p x = $true$.

`Lemma` $findIndex\_last$:
  $\forall$ (A : `Type`) (p : $A \to bool$) (l : $list$ A),
  $findIndex$ p l = $Some$ ($length$ l - 1) $\leftrightarrow$

$\exists\ x : A,$
$\quad last\ l = Some\ x \land$
$\quad p\ x = true \land$
$\quad \forall\ (n : nat)\ (y : A),$
$\quad\quad n < length\ l$ - $1 \to nth\ n\ l = Some\ y \to p\ y = false.$

`Lemma` *findIndex_spec* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A)\ (n : nat),$
$\quad findIndex\ p\ l = Some\ n \to$
$\quad\quad \forall\ m : nat,\ m < n \to$
$\quad\quad\quad \exists\ x : A,\ nth\ m\ l = Some\ x \land p\ x = false.$

`Lemma` *findIndex_take* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A)\ (n\ m : nat),$
$\quad findIndex\ p\ (take\ n\ l) = Some\ m \to$
$\quad\quad findIndex\ p\ l = Some\ m \land m \leq n.$

`Lemma` *findIndex_drop* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A)\ (n\ m : nat),$
$\quad findIndex\ p\ l = Some\ m \to n \leq m \to$
$\quad\quad findIndex\ p\ (drop\ n\ l) = Some\ (m$ - $n).$

`Lemma` *findIndex_zip* :
$\forall\ (A\ B : \texttt{Type})\ (pa : A \to bool)\ (pb : B \to bool)$
$(la : list\ A)\ (lb : list\ B)\ (n : nat),$
$\quad findIndex\ pa\ la = Some\ n \to findIndex\ pb\ lb = Some\ n \to$
$\quad\quad findIndex\ (\texttt{fun}\ \text{'}(a,\ b) \Rightarrow andb\ (pa\ a)\ (pb\ b))\ (zip\ la\ lb) = Some\ n.$

`Lemma` *findIndex_zip_conv* :
$\forall\ (A\ B : \texttt{Type})\ (pa : A \to bool)\ (pb : B \to bool)$
$(la : list\ A)\ (lb : list\ B)\ (n : nat),$
$\quad findIndex\ (\texttt{fun}\ \text{'}(a,\ b) \Rightarrow andb\ (pa\ a)\ (pb\ b))\ (zip\ la\ lb) = Some\ n \to$
$\quad \exists\ na\ nb : nat,$
$\quad\quad findIndex\ pa\ la = Some\ na \land$
$\quad\quad findIndex\ pb\ lb = Some\ nb \land$
$\quad\quad na \leq n \land$
$\quad\quad nb \leq n.$

10.2.6 *count*

Napisz funkcję *count*, która liczy, ile jest na liście l elementów spełniających predykat boolowski p.

Przykład:
count even [1; 2; 3; 4] = 2

`Lemma` *count_isEmpty* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$

$isEmpty\ l = true \to count\ p\ l = 0.$

Lemma $isEmpty\_count\_not\_0$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$count\ p\ l \neq 0 \to isEmpty\ l = false.$

Lemma $count\_length$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$count\ p\ l \leq length\ l.$

Lemma $count\_snoc$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (x : A)\ (l : list\ A),$
$count\ p\ (snoc\ x\ l) = count\ p\ l + \texttt{if}\ p\ x\ \texttt{then}\ 1\ \texttt{else}\ 0.$

Lemma $count\_app$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l1\ l2 : list\ A),$
$count\ p\ (l1\ {+}{+}\ l2) = count\ p\ l1 + count\ p\ l2.$

Lemma $count\_rev$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$count\ p\ (rev\ l) = count\ p\ l.$

Lemma $count\_map$:
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (p : B \to bool)\ (l : list\ A),$
$count\ p\ (map\ f\ l) = count\ (\texttt{fun}\ x : A \Rightarrow p\ (f\ x))\ l.$

(* Lemma count_join *)

Lemma $count\_replicate$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (n : nat)\ (x : A),$
$count\ p\ (replicate\ n\ x) =$
$\texttt{if}\ p\ x\ \texttt{then}\ n\ \texttt{else}\ 0.$

Lemma $count\_insert$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A)\ (n : nat)\ (x : A),$
$count\ p\ (insert\ l\ n\ x) =$
$(\texttt{if}\ p\ x\ \texttt{then}\ 1\ \texttt{else}\ 0) + count\ p\ l.$

Lemma $count\_replace$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l\ l' : list\ A)\ (n : nat)\ (x : A),$
$\texttt{replace}\ l\ n\ x = Some\ l' \to$
$count\ p\ l' = count\ p\ (take\ n\ l) + count\ p\ [x] + count\ p\ (drop\ (S\ n)\ l).$

Lemma $count\_remove$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l\ l' : list\ A)\ (n : nat)\ (x : A),$
$remove\ n\ l = Some\ (x, l') \to$
$S\ (count\ p\ l') = \texttt{if}\ p\ x\ \texttt{then}\ count\ p\ l\ \texttt{else}\ S\ (count\ p\ l).$

Lemma $count\_take$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A)\ (n : nat),$
$count\ p\ (take\ n\ l) \leq n.$

Lemma $count\_take'$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
$count\ p\ (take\ n\ l) \leq min\ n\ (count\ p\ l)$.

Lemma $count\_drop$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
$count\ p\ (drop\ n\ l) \leq length\ l$ - n.

Lemma $count\_splitAt$:
$\forall$ (A : Type) (p : $A \to bool$) ($l\ l1\ l2$: $list\ A$) (n : nat) (x : A),
$splitAt\ n\ l = Some\ (l1, x, l2) \to$
$count\ p\ l =$ (if $p\ x$ then 1 else 0) $+\ count\ p\ l1 + count\ p\ l2$.

Lemma $count\_false$:
$\forall$ (A : Type) (l : $list\ A$),
$count$ (fun _ $\Rightarrow$ $false$) $l = 0$.

Lemma $count\_true$:
$\forall$ (A : Type) (l : $list\ A$),
$count$ (fun _ $\Rightarrow$ $true$) $l = length\ l$.

Lemma $count\_negb$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
$count$ (fun x : $A \Rightarrow negb\ (p\ x)$) $l = length\ l$ - $count\ p\ l$.

Lemma $count\_andb\_le\_l$:
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
$count$ (fun x : $A \Rightarrow andb\ (p\ x)\ (q\ x)$) $l \leq count\ p\ l$.

Lemma $count\_andb\_le\_r$:
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
$count$ (fun x : $A \Rightarrow andb\ (p\ x)\ (q\ x)$) $l \leq count\ q\ l$.

Lemma $count\_orb$:
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
$count$ (fun x : $A \Rightarrow orb\ (p\ x)\ (q\ x)$) $l =$
$(count\ p\ l + count\ q\ l)$ - $count$ (fun x : $A \Rightarrow andb\ (p\ x)\ (q\ x)$) l.

Lemma $count\_orb\_le$:
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
$count$ (fun x : $A \Rightarrow orb\ (p\ x)\ (q\ x)$) $l \leq$
$count\ p\ l + count\ q\ l$.

Lemma $count\_andb$:
$\forall$ (A : Type) ($p\ q$: $A \to bool$) (l : $list\ A$),
$count$ (fun x : $A \Rightarrow andb\ (p\ x)\ (q\ x)$) $l =$
$count\ p\ l + count\ q\ l$ - $count$ (fun x : $A \Rightarrow orb\ (p\ x)\ (q\ x)$) l.

10.2.7 *filter*

Napisz funkcję *filter*, która zostawia na liście elementy, dla których funkcja p zwraca *true*, a usuwa te, dla których zwraca *false*.

Przykład:
filter even [1; 2; 3; 4] = [2; 4]

Lemma *filter_false* :
  ∀ (*A* : Type) (*l* : *list A*),
    filter (fun _ ⇒ *false*) *l* = [].

Lemma *filter_true* :
  ∀ (*A* : Type) (*l* : *list A*),
    filter (fun _ ⇒ *true*) *l* = *l*.

Lemma *filter_andb* :
  ∀ (*A* : Type) (*f g* : *A* → *bool*) (*l* : *list A*),
    filter (fun *x* : *A* ⇒ *andb* (*f x*) (*g x*)) *l* =
    filter f (*filter g l*).

Lemma *isEmpty_filter* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    isEmpty (*filter p l*) = *all* (fun *x* : *A* ⇒ *negb* (*p x*)) *l*.

Lemma *length_filter* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    length (*filter p l*) ≤ *length l*.

Lemma *filter_snoc* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*x* : *A*) (*l* : *list A*),
    filter p (*snoc x l*) =
    if *p x* then *snoc x* (*filter p l*) else *filter p l*.

Lemma *filter_app* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
    filter p (*l1* ++ *l2*) = *filter p l1* ++ *filter p l2*.

Lemma *filter_rev* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    filter p (*rev l*) = *rev* (*filter p l*).

Lemma *filter_map* :
  ∀ (*A B* : Type) (*f* : *A* → *B*) (*p* : *B* → *bool*) (*l* : *list A*),
    filter p (*map f l*) = *map f* (*filter* (fun *x* : *A* ⇒ *p* (*f x*)) *l*).

Lemma *filter_join* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*lla* : *list* (*list A*)),
    filter p (*join lla*) = *join* (*map* (*filter p*) *lla*).

Lemma *filter_replicate* :

```
  ∀ (A : Type) (p : A → bool) (n : nat) (x : A),
    filter p (replicate n x) =
    if p x then replicate n x else [].

Lemma filter_iterate :
  ∀ (A : Type) (p : A → bool) (f : A → A) (n : nat) (x : A),
    (∀ x : A, p (f x) = p x) →
      filter p (iterate f n x) =
      if p x then iterate f n x else [].

Lemma head_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    head (filter p l) = find p l.

Lemma last_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    last (filter p l) = findLast p l.

Lemma filter_nth :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat) (x : A),
    nth n l = Some x → p x = true →
      ∃ m : nat, m ≤ n ∧ nth m (filter p l) = Some x.

Lemma splitAt_filter :
  ∀ (A : Type) (p : A → bool) (l l1 l2 : list A) (x : A) (n : nat),
    splitAt n (filter p l) = Some (l1, x, l2) →
      ∃ m : nat,
      match splitAt m l with
          | None ⇒ False
          | Some (l1', y, l2') ⇒
              x = y ∧ l1 = filter p l1' ∧ l2 = filter p l2'
      end.

Lemma filter_insert :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat) (x : A),
    filter p (insert l n x) =
      filter p (take n l) ++
      (if p x then [x] else []) ++
      filter p (drop n l).

Lemma replace_filter :
  ∀ (A : Type) (p : A → bool) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      filter p l' =
      filter p (take n l) ++ filter p [x] ++ filter p (drop (S n) l).

Lemma remove_filter :
  ∀ (A : Type) (p : A → bool) (l l' : list A) (x : A) (n : nat),
```

```
    remove n (filter p l) = Some (x, l') →
      ∃ m : nat,
      match remove m l with
            | None ⇒ False
            | Some (y, l'') ⇒ x = y ∧ l' = filter p l''
      end.
Lemma filter_idempotent :
  ∀ (A : Type) (f : A → bool) (l : list A),
    filter f (filter f l) = filter f l.
Lemma filter_comm :
  ∀ (A : Type) (f g : A → bool) (l : list A),
    filter f (filter g l) = filter g (filter f l).
Lemma zip_not_filter :
  ∃ (A B : Type) (pa : A → bool) (pb : B → bool)
  (la : list A) (lb : list B),
    zip (filter pa la) (filter pb lb) ≠
    filter (fun x ⇒ andb (pa (fst x)) (pb (snd x))) (zip la lb).
Lemma any_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    any p l = negb (isEmpty (filter p l)).
Lemma all_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    all p (filter p l) = true.
Lemma all_filter' :
  ∀ (A : Type) (p : A → bool) (l : list A),
    all p l = isEmpty (filter (fun x : A ⇒ negb (p x)) l).
Lemma filter_all :
  ∀ (A : Type) (p : A → bool) (l : list A),
    all p l = true → filter p l = l.
Lemma removeFirst_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    removeFirst p (filter p l) =
    match filter p l with
          | [] ⇒ None
          | h :: t ⇒ Some (h, t)
    end.
Lemma removeFirst_negb_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    removeFirst (fun x : A ⇒ negb (p x)) (filter p l) = None.
Lemma findIndex_filter :
```

$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (l : *list* A),
findIndex p (*filter* p l) = *None* $\lor$
findIndex p (*filter* p l) = *Some* 0.

`Lemma` *count_filter* :
$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (l : *list* A),
count p (*filter* p l) = *length* (*filter* p l).

10.2.8 *partition*

Napisz funkcję *partition*, która dzieli listę l na listy elementów spełniających i niespełniających pewnego warunku boolowskiego.

Przykład:
partition even [1; 2; 3; 4] = ([2; 4], [1; 3])

`Lemma` *partition_spec* :
$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (l : *list* A),
partition p l = (*filter* p l, *filter* (`fun` $x \Rightarrow$ *negb* (p x)) l).

`Lemma` *partition_true* :
$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (l : *list* A),
partition (`fun` _ $\Rightarrow$ *true*) l = (l, []).

`Lemma` *partition_false* :
$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (l : *list* A),
partition (`fun` _ $\Rightarrow$ *false*) l = ([], l).

`Lemma` *partition_cons_true* :
$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (h : A) (t *l1* *l2* : *list* A),
p h = *true* $\rightarrow$ *partition* p t = (*l1*, *l2*) $\rightarrow$
partition p (h :: t) = (h :: *l1*, *l2*).

`Lemma` *partition_cons_false* :
$\forall$ (A : `Type`) (p : $A \rightarrow$ *bool*) (h : A) (t *l1* *l2* : *list* A),
p h = *false* $\rightarrow$ *partition* p t = (*l1*, *l2*) $\rightarrow$
partition p (h :: t) = (*l1*, h :: *l2*).

10.2.9 *findIndices*

Napisz funkcję *findIndices*, która znajduje indeksy wszystkich elementów listy, które spełniają predykat boolowski p.

Przykład:
findIndices even [1; 1; 2; 3; 5; 8; 13; 21; 34] = [2; 5; 8]

`Lemma` *findIndices_false* :
$\forall$ (A : `Type`) (l : *list* A),
findIndices (`fun` _ $\Rightarrow$ *false*) l = [].

Lemma *findIndices_true* :
  ∀ (*A* : Type) (*l* : *list A*),
    findIndices (fun _ ⇒ *true*) *l* =
    if *isEmpty l* then [] else *iterate S* (*length l*) 0.

Lemma *findIndices_isEmpty_true* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    isEmpty l = *true* → *findIndices p l* = [].

Lemma *isEmpty_findIndices_not_nil* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    findIndices p l ≠ [] → *isEmpty l* = *false*.

Lemma *length_findIndices* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    length (*findIndices p l*) = *count p l*.

Lemma *findIndices_snoc* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*x* : *A*) (*l* : *list A*),
    findIndices p (*snoc x l*) =
    if *p x*
    then *snoc* (*length l*) (*findIndices p l*)
    else *findIndices p l*.

Lemma *findIndices_app* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
    findIndices p (*l1* ++ *l2*) =
    findIndices p l1 ++ *map* (*plus* (*length l1*)) (*findIndices p l2*).

Lemma *findIndices_rev_aux* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    rev (*findIndices p* (*rev l*)) =
    map (fun *n* : *nat* ⇒ *length l* - *S n*) (*findIndices p l*).

Lemma *findIndices_rev* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    findIndices p (*rev l*) =
    rev (*map* (fun *n* : *nat* ⇒ *length l* - *S n*) (*findIndices p l*)).

Lemma *rev_findIndices* :
  ∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list A*),
    rev (*findIndices p l*) =
    map (fun *n* : *nat* ⇒ *length l* - *S n*) (*findIndices p* (*rev l*)).

Lemma *findIndices_map* :
  ∀ (*A B* : Type) (*f* : *A* → *B*) (*p* : *B* → *bool*) (*l* : *list A*),
    findIndices p (*map f l*) =
    findIndices (fun *x* : *A* ⇒ *p* (*f x*)) *l*.

Lemma *findIndices_replicate* :

$\forall$ (A : `Type`) (p : $A \to bool$) (n : nat) (x : A),
  findIndices p (*replicate n x*) =
  `match` n `with`
    | $0 \Rightarrow []$
    | S n' $\Rightarrow$ `if` p x `then` *iterate S n* 0 `else` []
  `end`.

`Lemma` *map_nth_findIndices* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*),
  map (`fun` n : *nat* $\Rightarrow$ *nth n l*) (*findIndices p l*) =
  map Some (*filter p l*).

`Lemma` *head_findIndices* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*),
  head (*findIndices p l*) = *findIndex p l*.

`Lemma` *tail_findIndices* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*),
  tail (*findIndices p l*) =
  `match` *removeFirst p l* `with`
    | *None* $\Rightarrow$ *None*
    | *Some* (_, *l'*) $\Rightarrow$ *Some* (*map S* (*findIndices p l'*))
  `end`.

`Lemma` *last_findIndices* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*),
  last (*findIndices p l*) =
  `match` *findIndex p* (*rev l*) `with`
    | *None* $\Rightarrow$ *None*
    | *Some n* $\Rightarrow$ *Some* (*length l* - *S n*)
  `end`.

`Lemma` *init_findIndices* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*),
  init (*findIndices p l*) =
  `match` *removeLast p l* `with`
    | *None* $\Rightarrow$ *None*
    | *Some* (_, *l'*) $\Rightarrow$ *Some* (*findIndices p l'*)
  `end`.

`Lemma` *findIndices_take* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*) (n : *nat*),
  findIndices p (*take n l*) =
  take (*count p* (*take n l*)) (*findIndices p l*).

`Lemma` *findIndices_insert* :
 $\forall$ (A : `Type`) (p : $A \to bool$) (l : *list A*) (n : *nat*) (x : A),

$findIndices\ p\ (insert\ l\ n\ x) =$
  $findIndices\ p\ (take\ n\ l) ++$
  $(\texttt{if}\ p\ x\ \texttt{then}\ [min\ (length\ l)\ n]\ \texttt{else}\ []) ++$
  $map\ (plus\ (S\ n))\ (findIndices\ p\ (drop\ n\ l)).$

`Lemma` *findIndices_replace* :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l\ l' : list\ A)\ (n : nat)\ (x : A),$
    $\texttt{replace}\ l\ n\ x = Some\ l' \to$
    $findIndices\ p\ l' =$
    $findIndices\ p\ (take\ n\ l) ++$
    $map\ (plus\ n)\ (findIndices\ p\ [x]) ++$
    $map\ (plus\ (S\ n))\ (findIndices\ p\ (drop\ (S\ n)\ l)).$

10.2.10 *takeWhile* i *dropWhile*

Zdefiniuj funkcje *takeWhile* oraz *dropWhile*, które, dopóki funkcja p zwraca *true*, odpowiednio biorą lub usuwają elementy z listy.

Przykład:
$takeWhile\ even$ [2; 4; 6; 1; 8; 10; 12] = [2; 4; 6]
$dropWhile\ even$ [2; 4; 6; 1; 8; 10; 12] = [1; 8; 10; 12]

`Lemma` *takeWhile_dropWhile_spec* :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $takeWhile\ p\ l ++ dropWhile\ p\ l = l.$

`Lemma` *takeWhile_false* :
  $\forall\ (A : \texttt{Type})\ (l : list\ A),$
    $takeWhile\ (\texttt{fun}\ \_ \Rightarrow false)\ l = [].$

`Lemma` *dropWhile_false* :
  $\forall\ (A : \texttt{Type})\ (l : list\ A),$
    $dropWhile\ (\texttt{fun}\ \_ \Rightarrow false)\ l = l.$

`Lemma` *takeWhile_andb* :
  $\forall\ (A : \texttt{Type})\ (p\ q : A \to bool)\ (l : list\ A),$
    $takeWhile\ (\texttt{fun}\ x : A \Rightarrow andb\ (p\ x)\ (q\ x))\ l =$
    $takeWhile\ p\ (takeWhile\ q\ l).$

`Lemma` *isEmpty_takeWhile* :
  $\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
    $isEmpty\ (takeWhile\ p\ l) =$
    $\texttt{match}\ l\ \texttt{with}$
      $|\ [] \Rightarrow true$
      $|\ h :: t \Rightarrow negb\ (p\ h)$
    $\texttt{end}.$

`Lemma` *isEmpty_dropWhile* :

$\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
  $isEmpty$ ($dropWhile\ p\ l$) = $all\ p\ l$.

Lemma $takeWhile\_snoc\_all$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (x : A) (l : $list\ A$),
    $all\ p\ l = true \rightarrow$
      $takeWhile\ p\ (snoc\ x\ l)$ = if $p\ x$ then $snoc\ x\ l$ else l.

Lemma $takeWhile\_idempotent$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $takeWhile\ p\ (takeWhile\ p\ l) = takeWhile\ p\ l$.

Lemma $dropWhile\_idempotent$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $dropWhile\ p\ (dropWhile\ p\ l) = dropWhile\ p\ l$.

Lemma $takeWhile\_replicate$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (n : nat) (x : A),
    $takeWhile\ p\ (replicate\ n\ x)$ =
    if $p\ x$ then $replicate\ n\ x$ else [].

Lemma $takeWhile\_iterate$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (f : $A \rightarrow A$) (n : nat) (x : A),
    ($\forall\ x : A,\ p\ (f\ x) = p\ x$) $\rightarrow$
      $takeWhile\ p\ (iterate\ f\ n\ x)$ =
      if $p\ x$ then $iterate\ f\ n\ x$ else [].

Lemma $dropWhile\_replicate$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (n : nat) (x : A),
    $dropWhile\ p\ (replicate\ n\ x)$ =
    if $p\ x$ then [] else $replicate\ n\ x$.

Lemma $dropWhile\_iterate$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (f : $A \rightarrow A$) (n : nat) (x : A),
    ($\forall\ x : A,\ p\ (f\ x) = p\ x$) $\rightarrow$
      $dropWhile\ p\ (iterate\ f\ n\ x)$ =
      if $p\ x$ then [] else $iterate\ f\ n\ x$.

Lemma $any\_takeWhile$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    $any\ p\ (takeWhile\ p\ l) = negb\ (isEmpty\ (takeWhile\ p\ l))$.

Lemma $any\_dropWhile$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),
    any (fun $x : A \Rightarrow negb\ (p\ x)$) ($dropWhile\ p\ l$) =
    $negb\ (isEmpty\ (dropWhile\ p\ l))$.

Lemma $any\_takeWhile\_dropWhile$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list\ A$),

$any\ p\ l = orb\ (any\ p\ (takeWhile\ p\ l))\ (any\ p\ (dropWhile\ p\ l)).$

Lemma *all_takeWhile* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
$all\ p\ (takeWhile\ p\ l) = true.$

Lemma *all_takeWhile'* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
$all\ p\ l = true \to takeWhile\ p\ l = l.$

Lemma *all_dropWhile* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
$all\ p\ (dropWhile\ p\ l) = all\ p\ l.$

Lemma *takeWhile_app_all* :
$\forall$ (A : Type) (p : $A \to bool$) ($l1\ l2$: $list\ A$),
$all\ p\ l1 = true \to takeWhile\ p\ (l1\ ++\ l2) = l1\ ++\ takeWhile\ p\ l2.$

Lemma *removeFirst_takeWhile* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
$removeFirst\ p\ (takeWhile\ p\ l) =$
match $takeWhile\ p\ l$ with
| [] $\Rightarrow$ *None*
| $h :: t \Rightarrow Some\ (h, t)$
end.

Lemma *removeLast_dropWhile* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
$removeFirst\ p\ (dropWhile$ (fun $x : A \Rightarrow negb\ (p\ x))\ l) =$
match $dropWhile$ (fun $x : A \Rightarrow negb\ (p\ x))\ l$ with
| [] $\Rightarrow$ *None*
| $h :: t \Rightarrow Some\ (h, t)$
end.

Lemma *findIndex_takeWhile* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) ($n\ m$: nat),
$findIndex\ p\ (takeWhile\ p\ l) = Some\ n \to$
$findIndex\ p\ l = Some\ n \to n \leq m.$

Lemma *findIndex_spec'* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (n : nat),
$findIndex\ p\ l = Some\ n \to$
$takeWhile$ (fun $x : A \Rightarrow negb\ (p\ x))\ l = take\ n\ l.$

Lemma *findIndex_dropWhile* :
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) ($n\ m$: nat),
$findIndex\ p\ (dropWhile\ p\ l) = Some\ m \to$
$findIndex\ p\ l = Some\ n \to n \leq m.$

Lemma *count_takeWhile* :

$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
  $count\ p\ (takeWhile\ p\ l) = length\ (takeWhile\ p\ l)$.

Lemma $count\_dropWhile$:
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
    $count\ p\ (dropWhile\ p\ l) \leq count\ p\ l$.

Lemma $count\_takeWhile\_dropWhile$:
  $\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
    $count\ p\ (takeWhile\ p\ l) + count\ p\ (dropWhile\ p\ l) = count\ p\ l$.

10.2.11 *span*

Zdefiniuj funkcję *span*, która dzieli listę l na listę b, której elementy nie spełniają predykatu p, element x, który spełnia p oraz listę e zawierającą resztę elementów l. Jeżeli na liście nie ma elementu spełniającego p, funkcja zwraca *None*.

Przykład:
$span\ even\ [1; 1; 2; 3; 5; 8] = Some\ ([1; 1], 2, [3; 5; 8])$
$span\ even\ [1; 3; 5] = None$

Lemma $isEmpty\_span$:
  $\forall$ (A : Type) (p : $A \to bool$) (x : A) ($l\ b\ e$: $list\ A$),
    $span\ p\ l = Some\ (b, x, e) \to$
      $isEmpty\ l = false$.

Lemma $length\_span$:
  $\forall$ (A : Type) (p : $A \to bool$) (x : A) ($l\ b\ e$: $list\ A$),
    $span\ p\ l = Some\ (b, x, e) \to length\ b + length\ e < length\ l$.

Lemma $length\_span'$:
  $\forall$ (A : Type) (p : $A \to bool$) (x : A) ($l\ b\ e$: $list\ A$),
    $span\ p\ l = Some\ (b, x, e) \to$
      $length\ b < length\ l\ \land$
      $length\ e < length\ l$.

Lemma $span\_snoc$:
  $\forall$ (A : Type) (p : $A \to bool$) (x : A) (l : $list\ A$),
    $span\ p\ (snoc\ x\ l) =$
    match $span\ p\ l$ with
        | $None \Rightarrow$ if $p\ x$ then $Some\ (l, x, [])$ else $None$
        | $Some\ (b, y, e) \Rightarrow Some\ (b, y, snoc\ x\ e)$
    end.

Lemma $span\_app$:
  $\forall$ (A : Type) (p : $A \to bool$) ($l1\ l2$: $list\ A$),
    $span\ p\ (l1\ {+}{+}\ l2) =$
    match $span\ p\ l1$, $span\ p\ l2$ with

```
        | Some (b, x, e), _ ⇒ Some (b, x, e ++ l2)
        | _, Some (b, x, e) ⇒ Some (l1 ++ b, x, e)
        | _, _ ⇒ None
    end.

Lemma span_map :
  ∀ (A B : Type) (f : A → B) (p : B → bool) (l : list A),
    span p (map f l) =
    match span (fun x : A ⇒ p (f x)) l with
        | None ⇒ None
        | Some (b, x, e) ⇒ Some (map f b, f x, map f e)
    end.

Lemma span_join :
  ∀ (A : Type) (p : A → bool) (lla : list (list A)),
    span p (join lla) =
    match span (any p) lla with
        | None ⇒ None
        | Some (bl, l, el) ⇒
            match span p l with
                | None ⇒ None
                | Some (b, x, e) ⇒ Some (join bl ++ b, x, e ++ join el)
            end
    end.

Lemma span_replicate :
  ∀ (A : Type) (p : A → bool) (n : nat) (x : A),
    span p (replicate n x) =
    if andb (1 <=? n) (p x)
    then Some ([], x, replicate (n - 1) x)
    else None.

Lemma span_any :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) → any p l = true.

Lemma span_all :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) →
      all p l = andb (beq_nat (length b) 0) (all p e).

Lemma span_find :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) → find p l = Some x.

Lemma span_removeFirst :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
```

    span p l = Some (b, x, e) →
      removeFirst p l = Some (x, b ++ e).

```
Lemma count_span_l :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) → count p b = 0.

Lemma count_span_r :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) →
      count p e < length l - length b.

Lemma span_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    span p (filter p l) =
    match filter p l with
        | [] ⇒ None
        | h :: t ⇒ Some ([], h, t)
    end.

Lemma filter_span_l :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) → filter p b = [].

Lemma takeWhile_span :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) →
      takeWhile (fun x : A ⇒ negb (p x)) l = b.

Lemma dropWhile_span :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) →
      dropWhile (fun x : A ⇒ negb (p x)) l = x :: e.
```

Związki *span* i *rev*

Zdefiniuj funkcję *naps*, która działa tak jak *span*, tyle że "od tyłu". Udowodnij twierdzenie *span_rev*.

```
Lemma span_rev_aux :
  ∀ (A : Type) (p : A → bool) (l : list A),
    span p l =
    match naps p (rev l) with
        | None ⇒ None
        | Some (b, x, e) ⇒ Some (rev e, x, rev b)
    end.

Lemma span_rev :
```

$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$),
  $span\ p\ (rev\ l) =$
  match $naps\ p\ l$ with
    | $None \Rightarrow None$
    | $Some\ (b, x, e) \Rightarrow Some\ (rev\ e, x, rev\ b)$
  end.

10.3 Sekcja mocno ad hoc

10.3.1 *pmap*

Zdefiniuj funkcję *pmap*, która mapuje funkcję $f : A \to option\ B$ po liście l, ale odpakowuje wyniki zawinięte w *Some*, a wyniki równe *None* usuwa.

Przykład:
$pmap$ (fun $n : nat \Rightarrow$ if $even\ n$ then $None$ else $Some\ (n + 42)$) $[1; 2; 3] = [43; 45]$

Lemma *isEmpty_pmap_false* :
$\forall$ (A B : Type) (f : $A \to option\ B$) (l : $list\ A$),
  $isEmpty\ (pmap\ f\ l) = false \to isEmpty\ l = false$.

Lemma *isEmpty_pmap_true* :
$\forall$ (A B : Type) (f : $A \to option\ B$) (l : $list\ A$),
  $isEmpty\ l = true \to isEmpty\ (pmap\ f\ l) = true$.

Lemma *length_pmap* :
$\forall$ (A B : Type) (f : $A \to option\ B$) (l : $list\ A$),
  $length\ (pmap\ f\ l) \leq length\ l$.

Lemma *pmap_snoc* :
$\forall$ (A B : Type) (f : $A \to option\ B$) (a : A) (l : $list\ A$),
  $pmap\ f\ (snoc\ a\ l) =$
  match $f\ a$ with
    | $None \Rightarrow pmap\ f\ l$
    | $Some\ b \Rightarrow snoc\ b\ (pmap\ f\ l)$
  end.

Lemma *pmap_app* :
$\forall$ (A B : Type) (f : $A \to option\ B$) ($l1$ $l2$: $list\ A$),
  $pmap\ f\ (l1 ++ l2) = pmap\ f\ l1 ++ pmap\ f\ l2$.

Lemma *pmap_rev* :
$\forall$ (A B : Type) (f : $A \to option\ B$) (l : $list\ A$),
  $pmap\ f\ (rev\ l) = rev\ (pmap\ f\ l)$.

Lemma *pmap_map* :
$\forall$ (A B C : Type) (f : $A \to B$) (g : $B \to option\ C$) (l : $list\ A$),
  $pmap\ g\ (map\ f\ l) = pmap$ (fun $x : A \Rightarrow g\ (f\ x)$) l.

```
Lemma pmap_join :
  ∀ (A B : Type) (f : A → option B) (l : list (list A)),
    pmap f (join l) = join (map (pmap f) l).

Lemma pmap_bind :
  ∀ (A B C : Type) (f : A → list B) (g : B → option C) (l : list A),
    pmap g (bind f l) = bind (fun x : A ⇒ pmap g (f x)) l.

Lemma pmap_replicate :
  ∀ (A B : Type) (f : A → option B) (n : nat) (x : A),
    pmap f (replicate n x) =
    match f x with
        | None ⇒ []
        | Some y ⇒ replicate n y
    end.

Definition isSome {A : Type} (x : option A) : bool :=
match x with
    | None ⇒ false
    | _ ⇒ true
end.

Lemma head_pmap :
  ∀ (A B : Type) (f : A → option B) (l : list A),
    head (pmap f l) =
    match find isSome (map f l) with
        | None ⇒ None
        | Some x ⇒ x
    end.

Lemma pmap_zip :
  ∃
    (A B C : Type)
    (fa : A → option C) (fb : B → option C)
    (la : list A) (lb : list B),
      pmap
        (fun '(a, b) ⇒
        match fa a, fb b with
            | Some a', Some b' ⇒ Some (a', b')
            | _, _ ⇒ None
        end)
        (zip la lb) ≠
      zip (pmap fa la) (pmap fb lb).

Lemma any_pmap :
  ∀ (A B : Type) (f : A → option B) (p : B → bool) (l : list A),
    any p (pmap f l) =
```

```
    any
      (fun x : A ⇒
      match f x with
          | Some b ⇒ p b
          | None ⇒ false
      end)
      l.

Lemma all_pmap :
  ∀ (A B : Type) (f : A → option B) (p : B → bool) (l : list A),
    all p (pmap f l) =
    all
      (fun x : A ⇒
      match f x with
          | Some b ⇒ p b
          | None ⇒ true
      end)
      l.

Lemma find_pmap :
  ∀ (A B : Type) (f : A → option B) (p : B → bool) (l : list A),
    find p (pmap f l) =
    let oa :=
      find (fun x : A ⇒ match f x with Some b ⇒ p b | _ ⇒ false end) l
    in
    match oa with
        | Some a ⇒ f a
        | None ⇒ None
    end.

Lemma findLast_pmap :
  ∀ (A B : Type) (f : A → option B) (p : B → bool) (l : list A),
    findLast p (pmap f l) =
    let oa :=
      findLast
        (fun x : A ⇒ match f x with Some b ⇒ p b | _ ⇒ false end) l
    in
    match oa with
        | Some a ⇒ f a
        | None ⇒ None
    end.

Lemma count_pmap :
  ∀ (A B : Type) (f : A → option B) (p : B → bool) (l : list A),
    count p (pmap f l) =
```

```
    count
      (fun x : A ⇒
      match f x with
           | Some b ⇒ p b
           | None ⇒ false
      end)
      l.

(*  TODO *) Definition aux {A B : Type} (p : B → bool) (f : A → option B)
  (dflt : bool) (x : A) : bool :=
match f x with
    | Some b ⇒ p b
    | None ⇒ dflt
end.

Lemma pmap_filter :
  ∀ (A B : Type) (p : B → bool) (f : A → option B) (l : list A),
    filter p (pmap f l) =
    pmap f (filter (aux p f false) l).

Lemma pmap_takeWhile :
  ∀ (A B : Type) (p : B → bool) (f : A → option B) (l : list A),
    takeWhile p (pmap f l) =
    pmap f
      (takeWhile
        (fun x : A ⇒ match f x with | Some b ⇒ p b | _ ⇒ true end)
        l).

Lemma pmap_dropWhile :
  ∀ (A B : Type) (p : B → bool) (f : A → option B) (l : list A),
    dropWhile p (pmap f l) =
    pmap f
      (dropWhile
        (fun x : A ⇒ match f x with | Some b ⇒ p b | _ ⇒ true end)
        l).

Lemma pmap_span :
  ∀ (A B : Type) (f : A → option B) (p : B → bool) (l : list A),
    match
      span
        (fun x : A ⇒ match f x with None ⇒ false | Some b ⇒ p b end)
        l
    with
        | None ⇒ True
        | Some (b, x, e) ⇒
            ∃ y : B, f x = Some y ∧
```

$span\ p\ (pmap\ f\ l) = Some\ (pmap\ f\ b,\ y,\ pmap\ f\ e)$

```
end.
```

Lemma *pmap_nth_findIndices* :
$\forall\ (A : \texttt{Type})\ (p : A \rightarrow bool)\ (l : list\ A),$
$pmap\ (\texttt{fun}\ n : nat \Rightarrow nth\ n\ l)\ (findIndices\ p\ l) =$
$filter\ p\ l.$

10.4 Bardziej skomplikowane funkcje

10.4.1 *intersperse*

Napisz funkcję *intersperse*, który wstawia element $x : A$ między każde dwa elementy z listy $l : list\ A$. Zastanów się dobrze nad przypadkami bazowymi.

Przykład:
$intersperse\ 42\ [1; 2; 3] = [1; 42; 2; 42; 3]$

Lemma *isEmpty_intersperse* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A),$
$isEmpty\ (intersperse\ x\ l) = isEmpty\ l.$

Lemma *length_intersperse* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A),$
$length\ (intersperse\ x\ l) = 2 \times length\ l\ \text{-}\ 1.$

Lemma *intersperse_snoc* :
$\forall\ (A : \texttt{Type})\ (x\ y : A)\ (l : list\ A),$
$intersperse\ x\ (snoc\ y\ l) =$
$\texttt{if}\ isEmpty\ l\ \texttt{then}\ [y]\ \texttt{else}\ snoc\ y\ (snoc\ x\ (intersperse\ x\ l)).$

Lemma *intersperse_app* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l1\ l2 : list\ A),$
$intersperse\ x\ (l1\ ++\ l2) =$
$\texttt{match}\ l1,\ l2\ \texttt{with}$
$|\ [],\ \_ \Rightarrow intersperse\ x\ l2$
$|\ \_,\ [] \Rightarrow intersperse\ x\ l1$
$|\ h1 :: t1,\ h2 :: t2 \Rightarrow$
$intersperse\ x\ l1\ ++\ x :: intersperse\ x\ l2$
$\texttt{end}.$

Lemma *intersperse_app_cons* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l1\ l2 : list\ A),$
$l1 \neq [] \rightarrow l2 \neq [] \rightarrow$
$intersperse\ x\ (l1\ ++\ l2) = intersperse\ x\ l1\ ++\ x :: intersperse\ x\ l2.$

Lemma *intersperse_rev* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A),$

$intersperse\ x\ (rev\ l) = rev\ (intersperse\ x\ l).$

Lemma $intersperse\_map$:
$\forall\ (A\ B : \texttt{Type})\ (f : A \to B)\ (l : list\ A)\ (a : A)\ (b : B),$
$f\ a = b \to intersperse\ b\ (map\ f\ l) = map\ f\ (intersperse\ a\ l).$

Lemma $head\_intersperse$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A),$
$head\ (intersperse\ x\ l) = head\ l.$

Lemma $last\_intersperse$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A),$
$last\ (intersperse\ x\ l) = last\ l.$

Lemma $tail\_intersperse$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A),$
$tail\ (intersperse\ x\ l) =$
match $tail\ l$ with
| $None \Rightarrow None$
| $Some\ [] \Rightarrow Some\ []$
| $Some\ (h :: t) \Rightarrow tail\ (intersperse\ x\ l)$
end.

Lemma $nth\_intersperse\_even$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n : nat),$
$n < length\ l \to$
$nth\ (2 \times n)\ (intersperse\ x\ l) = nth\ n\ l.$

Lemma $nth\_intersperse\_odd$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n : nat),$
$0 < n \to n < length\ l \to$
$nth\ (2 \times n - 1)\ (intersperse\ x\ l) = Some\ x.$

Lemma $intersperse\_take$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n : nat),$
$intersperse\ x\ (take\ n\ l) =$
$take\ (2 \times n - 1)\ (intersperse\ x\ l).$

Lemma $any\_intersperse$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (x : A)\ (l : list\ A),$
$any\ p\ (intersperse\ x\ l) =$
$orb\ (any\ p\ l)\ (andb\ (2 <=?\ length\ l)\ (p\ x)).$

Lemma $all\_intersperse$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (x : A)\ (l : list\ A),$
$all\ p\ (intersperse\ x\ l) =$
$all\ p\ l\ \&\&\ ((length\ l <=?\ 1)\ ||\ p\ x).$

Lemma $findIndex\_intersperse$:
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (x : A)\ (l : list\ A),$

```
findIndex p (intersperse x l) =
if p x
then
  match l with
      | [] ⇒ None
      | [h] ⇒ if p h then Some 0 else None
      | h :: t ⇒ if p h then Some 0 else Some 1
  end
else
  match findIndex p l with
      | None ⇒ None
      | Some n ⇒ Some (2 × n)
  end.
```

```
Lemma count_intersperse :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    count p (intersperse x l) =
    count p l + if p x then length l - 1 else 0.
```

```
Lemma filter_intersperse_false :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
    p x = false → filter p (intersperse x l) = filter p l.
```

```
Lemma pmap_intersperse :
  ∀ (A B : Type) (f : A → option B) (x : A) (l : list A),
    f x = None → pmap f (intersperse x l) = pmap f l.
```

10.5 Proste predykaty

10.5.1 *elem*

Zdefiniuj induktywny predykat *elem*. *elem x l* jest spełniony, gdy *x* jest elementem listy *l*.

```
Lemma elem_not_nil :
  ∀ (A : Type) (x : A), ¬ elem x [].
```

```
Lemma elem_not_cons :
  ∀ (A : Type) (x h : A) (t : list A),
    ¬ elem x (h :: t) → x ≠ h ∧ ¬ elem x t.
```

```
Lemma elem_cons' :
  ∀ (A : Type) (x h : A) (t : list A),
    elem x (h :: t) ↔ x = h ∨ elem x t.
```

```
Lemma elem_snoc :
  ∀ (A : Type) (x y : A) (l : list A),
    elem x (snoc y l) ↔ elem x l ∨ x = y.
```

Lemma *elem_app_l* :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    elem x l1 → elem x (l1 ++ l2).

Lemma *elem_app_r* :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    elem x l2 → elem x (l1 ++ l2).

Lemma *elem_or_app* :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    elem x l1 ∨ elem x l2 → elem x (l1 ++ l2).

Lemma *elem_app_or* :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    elem x (l1 ++ l2) → elem x l1 ∨ elem x l2.

Lemma *elem_app* :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    elem x (l1 ++ l2) ↔ elem x l1 ∨ elem x l2.

Lemma *elem_spec* :
  ∀ (A : Type) (x : A) (l : list A),
    elem x l ↔ ∃ l1 l2 : list A, l = l1 ++ x :: l2.

Lemma *elem_map* :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
    elem x l → elem (f x) (map f l).

Lemma *elem_map_conv* :
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
    elem y (map f l) ↔ ∃ x : A, f x = y ∧ elem x l.

Lemma *elem_map_conv'* :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
    (∀ x y : A, f x = f y → x = y) →
      elem (f x) (map f l) → elem x l.

Lemma *map_ext_elem* :
  ∀ (A B : Type) (f g : A → B) (l : list A),
    (∀ x : A, elem x l → f x = g x) → map f l = map g l.

Lemma *elem_join* :
  ∀ (A : Type) (x : A) (ll : list (list A)),
    elem x (join ll) ↔ ∃ l : list A, elem x l ∧ elem l ll.

Lemma *elem_replicate* :
  ∀ (A : Type) (n : nat) (x y : A),
    elem y (replicate n x) ↔ n ≠ 0 ∧ x = y.

Lemma *nth_elem* :
  ∀ (A : Type) (l : list A) (n : nat),

```
    n < length l → ∃ x : A, nth n l = Some x ∧ elem x l.

(* TOOD: ulepszyć? *) Lemma iff_elem_nth :
  ∀ (A : Type) (x : A) (l : list A),
    elem x l ↔ ∃ n : nat, nth n l = Some x.

Lemma elem_rev_aux :
  ∀ (A : Type) (x : A) (l : list A),
    elem x l → elem x (rev l).

Lemma elem_rev :
  ∀ (A : Type) (x : A) (l : list A),
    elem x (rev l) ↔ elem x l.

Lemma elem_remove_nth :
  ∀ (A : Type) (x : A) (l : list A) (n : nat),
    elem x l → nth n l ≠ Some x →
    match remove n l with
        | None ⇒ True
        | Some (_, l') ⇒ elem x l'
    end.

Lemma elem_take :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    elem x (take n l) → elem x l.

Lemma elem_drop :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    elem x (drop n l) → elem x l.

Lemma elem_splitAt' :
  ∀ (A : Type) (l l1 l2 : list A) (n : nat) (x y : A),
    splitAt n l = Some (l1, y, l2) →
      elem x l ↔ x = y ∨ elem x l1 ∨ elem x l2.

Lemma elem_insert :
  ∀ (A : Type) (l : list A) (n : nat) (x y : A),
    elem y (insert l n x) ↔ x = y ∨ elem y l.

Lemma elem_replace :
  ∀ (A : Type) (l l' : list A) (n : nat) (x y : A),
    replace l n x = Some l' →
      elem y l' ↔ elem y (take n l) ∨ x = y ∨ elem y (drop (S n) l).

Lemma elem_filter :
  ∀ (A : Type) (p : A → bool) (l : list A) (x : A),
    elem x (filter p l) ↔ p x = true ∧ elem x l.

Lemma elem_filter_conv :
  ∀ (A : Type) (p : A → bool) (x : A) (l : list A),
```

$elem\ x\ l \leftrightarrow$
$elem\ x\ (filter\ p\ l) \land p\ x = true \lor$
$elem\ x\ (filter\ (\texttt{fun}\ x : A \Rightarrow negb\ (p\ x))\ l) \land p\ x = false.$

```
Lemma elem_partition :
  ∀ (A : Type) (p : A → bool) (x : A) (l l1 l2 : list A),
    partition p l = (l1, l2) →
      elem x l ↔
      (elem x l1 ∧ p x = true) ∨ (elem x l2 ∧ p x = false).

Lemma elem_takeWhile :
  ∀ (A : Type) (p : A → bool) (l : list A) (x : A),
    elem x (takeWhile p l) → elem x l ∧ p x = true.

Lemma elem_dropWhile :
  ∀ (A : Type) (p : A → bool) (l : list A) (x : A),
    elem x (dropWhile p l) → elem x l.

Lemma elem_takeWhile_dropWhile :
  ∀ (A : Type) (p : A → bool) (l : list A) (x : A),
    elem x l ↔ elem x (takeWhile p l) ∨ elem x (dropWhile p l).

Lemma elem_dropWhile_conv :
  ∀ (A : Type) (p : A → bool) (l : list A) (x : A),
    elem x l → ¬ elem x (dropWhile p l) → p x = true.

(*  TODO: span i intersperse, groupBy *)

Lemma span_spec' :
  ∀ (A : Type) (p : A → bool) (l : list A),
    match span p l with
        | None ⇒ ∀ x : A, elem x l → p x = false
        | Some (b, x, e) ⇒
            b = takeWhile (fun x : A ⇒ negb (p x)) l ∧
            Some x = find p l ∧
            x :: e = dropWhile (fun x : A ⇒ negb (p x)) l ∧
            Some (x, b ++ e) = removeFirst p l
    end.

Lemma elem_span_None :
  ∀ (A : Type) (p : A → bool) (l : list A),
    span p l = None → ∀ x : A, elem x l → p x = false.

Lemma elem_span_Some :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) →
      ∀ y : A, elem y l ↔ elem y b ∨ y = x ∨ elem y e.

Lemma elem_span :
  ∀ (A : Type) (p : A → bool) (l : list A),
```

```
    match span p l with
        | None ⇒ ∀ x : A, elem x l → p x = false
        | Some (b, x, e) ⇒
            ∀ y : A, elem y l ↔ elem y b ∨ y = x ∨ elem y e
    end.
```

```
Lemma elem_removeFirst_None :
  ∀ (A : Type) (p : A → bool) (l : list A),
    removeFirst p l = None →
      ∀ x : A, elem x l → p x = false.
```

```
Lemma elem_zip :
  ∀ (A B : Type) (a : A) (b : B) (la : list A) (lb : list B),
    elem (a, b) (zip la lb) → elem a la ∧ elem b lb.
```

```
Lemma zip_not_elem :
  ∃ (A B : Type) (a : A) (b : B) (la : list A) (lb : list B),
    elem a la ∧ elem b lb ∧ ¬ elem (a, b) (zip la lb).
```

```
Lemma elem_findIndices :
  ∀ (A : Type) (p : A → bool) (l : list A) (n : nat),
    elem n (findIndices p l) →
      ∃ x : A, nth n l = Some x ∧ p x = true.
```

```
Lemma isEmpty_bind :
  ∀ (A B : Type) (f : A → list B) (l : list A),
    isEmpty (bind f l) = true ↔
    l = [] ∨ l ≠ [] ∧ ∀ x : A, elem x l → f x = [].
```

```
Lemma elem_pmap :
  ∀ (A B : Type) (f : A → option B) (l : list A) (a : A) (b : B),
    f a = Some b → elem a l → elem b (pmap f l).
```

```
Lemma elem_pmap' :
  ∀ (A B : Type) (f : A → option B) (l : list A) (b : B),
    (∃ a : A, elem a l ∧ f a = Some b) → elem b (pmap f l).
```

```
Lemma elem_pmap_conv :
  ∀ (A B : Type) (f : A → option B) (l : list A) (b : B),
    elem b (pmap f l) → ∃ a : A, elem a l ∧ f a = Some b.
```

```
Lemma elem_intersperse :
  ∀ (A : Type) (x y : A) (l : list A),
    elem x (intersperse y l) ↔ elem x l ∨ (x = y ∧ 2 ≤ length l).
```

10.5.2 *In*

Gratuluję, udało ci się zdefiniować predykat *elem* i dowieść wszystkich jego właściwości. To jednak nie koniec zabawy, gdyż predykaty możemy definiować nie tylko przez indukcję, ale także przez rekursję. Być może taki sposób definiowania jest nawet lepszy? Przyjrzyjmy się poniższej definicji — tak właśnie "bycie elementem" jest zdefiniowane w bibliotece standardowej.

Fixpoint In $\{A : \texttt{Type}\}$ $(x : A)$ $(l : list\ A)$: Prop :=
match l with
  | $[] \Rightarrow False$
  | $h :: t \Rightarrow x = h \lor In\ x\ t$
end.

Powyższa definicja jest bardzo podobna do tej induktywnej. $In\ x$ dla listy pustej redukuje się do $False$, co oznacza, że w pustej liście nic nie ma, zaś dla listy mającej głowę i ogon redukuje się do zdania "x jest głową lub jest elementem ogona".

Definicja taka ma swoje wady i zalety. Największą moim zdaniem wadą jest to, że nie możemy robić indukcji po dowodzie, gdyż dowód faktu $In\ x\ l$ nie jest induktywny. Największą zaletą zaś jest fakt, że nie możemy robić indukcji po dowodzie — im mniej potencjalnych rzeczy, po których można robić indukcję, tym mniej zastanawiania się. Przekonajmy się zatem na własnej skórze, która definicja jest "lepsza".

Lemma In_elem :
  $\forall$ $(A : \texttt{Type})$ $(x : A)$ $(l : list\ A)$,
    $In\ x\ l \leftrightarrow elem\ x\ l$.

Lemma In_not_nil :
  $\forall$ $(A : \texttt{Type})$ $(x : A)$, $\neg\ In\ x\ []$.

Lemma In_not_cons :
  $\forall$ $(A : \texttt{Type})$ $(x\ h : A)$ $(t : list\ A)$,
    $\neg\ In\ x\ (h :: t) \rightarrow x \neq h \land \neg\ In\ x\ t$.

Lemma In_cons :
  $\forall$ $(A : \texttt{Type})$ $(x\ h : A)$ $(t : list\ A)$,
    $In\ x\ (h :: t) \leftrightarrow x = h \lor In\ x\ t$.

Lemma In_snoc :
  $\forall$ $(A : \texttt{Type})$ $(x\ y : A)$ $(l : list\ A)$,
    $In\ x\ (snoc\ y\ l) \leftrightarrow In\ x\ l \lor x = y$.

Lemma In_app_l :
  $\forall$ $(A : \texttt{Type})$ $(x : A)$ $(l1\ l2 : list\ A)$,
    $In\ x\ l1 \rightarrow In\ x\ (l1\ {+}{+}\ l2)$.

Lemma In_app_r :
  $\forall$ $(A : \texttt{Type})$ $(x : A)$ $(l1\ l2 : list\ A)$,
    $In\ x\ l2 \rightarrow In\ x\ (l1\ {+}{+}\ l2)$.

Lemma In_or_app :
  $\forall$ $(A : \texttt{Type})$ $(x : A)$ $(l1\ l2 : list\ A)$,
    $In\ x\ l1 \lor In\ x\ l2 \rightarrow In\ x\ (l1\ {+}{+}\ l2)$.

Lemma In_app_or :
$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
In x ($l1$ ++ $l2$) $\rightarrow$ In x $l1$ $\lor$ In x $l2$.

Lemma In_app :
$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
In x ($l1$ ++ $l2$) $\leftrightarrow$ In x $l1$ $\lor$ In x $l2$.

Lemma In_spec :
$\forall$ (A : Type) (x : A) (l : $list$ A),
In x l $\leftrightarrow$ $\exists$ $l1$ $l2$: $list$ A, l = $l1$ ++ x :: $l2$.

Lemma In_map :
$\forall$ (A B : Type) (f : $A \rightarrow B$) (l : $list$ A) (x : A),
In x l $\rightarrow$ In (f x) (map f l).

Lemma In_map_conv :
$\forall$ (A B : Type) (f : $A \rightarrow B$) (l : $list$ A) (y : B),
In y (map f l) $\leftrightarrow$ $\exists$ x : A, f x = y $\land$ In x l.

Lemma In_map_conv' :
$\forall$ (A B : Type) (f : $A \rightarrow B$) (l : $list$ A) (x : A),
($\forall$ x y : A, f x = f y $\rightarrow$ x = y) $\rightarrow$
In (f x) (map f l) $\rightarrow$ In x l.

Lemma map_ext_In :
$\forall$ (A B : Type) (f g : $A \rightarrow B$) (l : $list$ A),
($\forall$ x : A, In x l $\rightarrow$ f x = g x) $\rightarrow$ map f l = map g l.

Lemma In_join :
$\forall$ (A : Type) (x : A) (ll : $list$ ($list$ A)),
In x ($join$ ll) $\leftrightarrow$
$\exists$ l : $list$ A, In x l $\land$ In l ll.

Lemma $In\_replicate$:
$\forall$ (A : Type) (n : nat) (x y : A),
In y ($replicate$ n x) $\leftrightarrow$ $n \neq 0$ $\land$ x = y.

Lemma $In\_iterate$:
$\forall$ (A : Type) (f : $A \rightarrow A$) (n : nat) (x y : A),
In y ($iterate$ f n x) $\leftrightarrow$ $\exists$ k : nat, $k < n$ $\land$ y = $iter$ f k x.

Lemma nth_In :
$\forall$ (A : Type) (l : $list$ A) (n : nat),
n < $length$ l $\rightarrow$ $\exists$ x : A, nth n l = $Some$ x $\land$ In x l.

Lemma iff_In_nth :
$\forall$ (A : Type) (x : A) (l : $list$ A),
In x l $\leftrightarrow$ $\exists$ n : nat, nth n l = $Some$ x.

Lemma In_rev_aux :

$\forall$ (A : Type) (x : A) (l : *list* A),
In x l $\rightarrow$ *In* x (*rev* l).

Lemma *In_rev* :
$\forall$ (A : Type) (x : A) (l : *list* A),
In x (*rev* l) $\leftrightarrow$ *In* x l.

Lemma *In_take* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*) (x : A),
In x (*take* n l) $\rightarrow$ *In* x l.

Lemma *In_drop* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*) (x : A),
In x (*drop* n l) $\rightarrow$ *In* x l.

Lemma *In_splitAt* :
$\forall$ (A : Type) (l b e : *list* A) (n : *nat*) (x y : A),
splitAt n l = *Some* (b, x, e) $\rightarrow$
In y l $\leftrightarrow$ *In* y b $\lor$ x = y $\lor$ *In* y e.

Lemma *In_insert* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*) (x y : A),
In y (*insert* l n x) $\leftrightarrow$ x = y $\lor$ *In* y l.

Lemma *In_replace* :
$\forall$ (A : Type) (l l' : *list* A) (n : *nat*) (x y : A),
replace l n x = *Some* l' $\rightarrow$
In y l' $\leftrightarrow$ *In* y (*take* n l) $\lor$ x = y $\lor$ *In* y (*drop* (S n) l).

Lemma *In_filter* :
$\forall$ (A : Type) (p : A $\rightarrow$ *bool*) (l : *list* A) (x : A),
In x (*filter* p l) $\leftrightarrow$ p x = *true* $\land$ *In* x l.

Lemma *In_filter_conv* :
$\forall$ (A : Type) (p : A $\rightarrow$ *bool*) (x : A) (l : *list* A),
In x l $\leftrightarrow$
In x (*filter* p l) $\land$ p x = *true* $\lor$
In x (*filter* (fun x : A $\Rightarrow$ *negb* (p x)) l) $\land$ p x = *false*.

Lemma *In_partition* :
$\forall$ (A : Type) (p : A $\rightarrow$ *bool*) (x : A) (l $l1$ $l2$: *list* A),
partition p l = ($l1$, $l2$) $\rightarrow$
In x l $\leftrightarrow$
(*In* x $l1$ $\land$ p x = *true*) $\lor$ (*In* x $l2$ $\land$ p x = *false*).

Lemma *In_takeWhile* :
$\forall$ (A : Type) (p : A $\rightarrow$ *bool*) (l : *list* A) (x : A),
In x (*takeWhile* p l) $\rightarrow$ *In* x l $\land$ p x = *true*.

Lemma *In_dropWhile* :

$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (x : A),
$In\ x\ (dropWhile\ p\ l) \to In\ x\ l$.

Lemma $In\_takeWhile\_dropWhile$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (x : A),
$In\ x\ l \to$
$In\ x\ (takeWhile\ p\ l) \lor$
$In\ x\ (dropWhile\ p\ l)$.

Lemma $In\_dropWhile\_conv$:
$\forall$ (A : Type) (p : $A \to bool$) (l : $list\ A$) (x : A),
$In\ x\ l \to \neg\ In\ x\ (dropWhile\ p\ l) \to p\ x = true$.

```
(* TODO: jak elem *)
```

Lemma In_span :
$\forall$ (A : Type) (p : $A \to bool$) ($x\ y$: A) ($l\ b\ e$: $list\ A$),
$span\ p\ l = Some\ (b, x, e) \to$
$In\ y\ l \leftrightarrow In\ y\ b \lor y = x \lor In\ y\ e$.

Lemma In_zip :
$\forall$ ($A\ B$: Type) (a : A) (b : B) (la : $list\ A$) (lb : $list\ B$),
$In\ (a, b)\ (zip\ la\ lb) \to In\ a\ la \land In\ b\ lb$.

Lemma zip_not_In :
$\exists$ ($A\ B$: Type) (a : A) (b : B) (la : $list\ A$) (lb : $list\ B$),
$In\ a\ la \land In\ b\ lb \land \neg\ In\ (a, b)\ (zip\ la\ lb)$.

Lemma $In\_intersperse$:
$\forall$ (A : Type) ($x\ y$: A) (l : $list\ A$),
$In\ x\ (intersperse\ y\ l) \leftrightarrow$
$In\ x\ l \lor (x = y \land 2 \leq length\ l)$.

10.5.3 *NoDup*

Zdefiniuj induktywny predykat *NoDup*. Zdanie *NoDup l* jest prawdziwe, gdy w *l* nie ma powtarzających się elementów. Udowodnij, że zdefiniowall przez ciebie predykat posiada pożądane właściwości.

Lemma $NoDup\_singl$:
$\forall$ (A : Type) (x : A), $NoDup\ [x]$.

Lemma $NoDup\_cons\_inv$:
$\forall$ (A : Type) (h : A) (t : $list\ A$),
$NoDup\ (h :: t) \to NoDup\ t$.

Lemma $NoDup\_length$:
$\forall$ (A : Type) (l : $list\ A$),
$\neg\ NoDup\ l \to 2 \leq length\ l$.

Lemma $NoDup\_snoc$:
  $\forall$ (A : Type) (x : A) (l : $list$ A),
    $NoDup$ ($snoc$ x l) $\leftrightarrow$ $NoDup$ l $\wedge$ $\neg$ $elem$ x l.

Lemma $NoDup\_app$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $NoDup$ ($l1$ ++ $l2$) $\leftrightarrow$
    $NoDup$ $l1$ $\wedge$
    $NoDup$ $l2$ $\wedge$
    ($\forall$ x : A, $elem$ x $l1$ $\rightarrow$ $\neg$ $elem$ x $l2$) $\wedge$
    ($\forall$ x : A, $elem$ x $l2$ $\rightarrow$ $\neg$ $elem$ x $l1$).

Lemma $NoDup\_app\_comm$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $NoDup$ ($l1$ ++ $l2$) $\leftrightarrow$ $NoDup$ ($l2$ ++ $l1$).

Lemma $NoDup\_rev$:
  $\forall$ (A : Type) (l : $list$ A),
    $NoDup$ (rev l) $\leftrightarrow$ $NoDup$ l.

Lemma $NoDup\_map$:
  $\forall$ (A B : Type) (f : $A \rightarrow B$) (l : $list$ A),
    $NoDup$ (map f l) $\rightarrow$ $NoDup$ l.

Lemma $NoDup\_map\_inj$:
  $\forall$ (A B : Type) (f : $A \rightarrow B$) (l : $list$ A),
    ($\forall$ x y : A, f x = f y $\rightarrow$ x = y) $\rightarrow$
      $NoDup$ l $\rightarrow$ $NoDup$ (map f l).

Lemma $NoDup\_replicate$:
  $\forall$ (A : Type) (n : nat) (x : A),
    $NoDup$ ($replicate$ n x) $\leftrightarrow$ $n = 0 \vee n = 1$.

Lemma $NoDup\_take$:
  $\forall$ (A : Type) (l : $list$ A) (n : nat),
    $NoDup$ l $\rightarrow$ $NoDup$ ($take$ n l).

Lemma $NoDup\_drop$:
  $\forall$ (A : Type) (l : $list$ A) (n : nat),
    $NoDup$ l $\rightarrow$ $NoDup$ ($drop$ n l).

Lemma $NoDup\_filter$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list$ A),
    $NoDup$ l $\rightarrow$ $NoDup$ ($filter$ p l).

Lemma $NoDup\_partition$:
  $\forall$ (A : Type) (p : $A \rightarrow bool$) (l $l1$ $l2$: $list$ A),
    $partition$ p l = ($l1$, $l2$) $\rightarrow$ $NoDup$ l $\leftrightarrow$ $NoDup$ $l1$ $\wedge$ $NoDup$ $l2$.

Lemma $NoDup\_takeWhile$:

$\forall$ (A : Type) (p : $A \to$ *bool*) (l : *list* A),
NoDup $l \to$ *NoDup* (*takeWhile* p l).

Lemma *NoDup_dropWhile* :
$\forall$ (A : Type) (p : $A \to$ *bool*) (l : *list* A),
NoDup $l \to$ *NoDup* (*dropWhile* p l).

Lemma *NoDup_zip* :
$\forall$ (A B : Type) (*la* : *list* A) (*lb* : *list* B),
NoDup la $\land$ *NoDup lb* $\to$ *NoDup* (*zip la lb*).

Lemma *NoDup_zip_conv* :
$\exists$ (A B : Type) (*la* : *list* A) (*lb* : *list* B),
NoDup (*zip la lb*) $\land \neg$ *NoDup la* $\land \neg$ *NoDup lb*.

Lemma *NoDup_pmap* :
$\exists$ (A B : Type) (f : $A \to$ *option* B) (l : *list* A),
NoDup $l \land \neg$ *NoDup* (*pmap* f l).

Lemma *NoDup_intersperse* :
$\forall$ (A : Type) (x : A) (l : *list* A),
NoDup (*intersperse* x l) $\to$ *length* $l \leq 2$.

Lemma *NoDup_spec* :
$\forall$ (A : Type) (l : *list* A),
$\neg$ *NoDup* $l \leftrightarrow$
$\exists$ (x : A) ($l1$ $l2$ $l3$: *list* A),
$l = l1 ++ x :: l2 ++ x :: l3$.

10.5.4 *Dup*

Powodem problemów z predykatem *NoDup* jest fakt, że jest on w pewnym sensie niekonstruktywny. Wynika to wprost z jego definicji: *NoDup* l zachodzi, gdy w l nie ma duplikatów. Parafrazując: *NoDup* l zachodzi, gdy *nieprawda*, że w l są duplikaty.

Jak widać, w naszej definicji implicité występuje negacja. Wobec tego jeżeli spróbujemy za pomocą *NoDup* wyrazić zdanie "na liście l są duplikaty", to tak naprawdę dostaniemy zdanie "nieprawda, że nieprawda, że l ma duplikaty".

Dostaliśmy więc po głowie nagłym atakiem podwójnej negacji. Nie ma się co dziwić w takiej sytuacji, że nasza "negatywna" definicja predykatu *NoDup* jest nazbyt klasyczna. Możemy jednak uratować sytuację, jeżeli zdefiniujemy predykat *Dup* i zanegujemy go.

Zdefiniuj predykat *Dup*, który jest spełniony, gdy na liście występują duplikaty.

Lemma *Dup_nil* :
$\forall$ A : Type, $\neg$ *Dup* (@*nil* A).

Lemma *Dup_cons* :
$\forall$ (A : Type) (x : A) (l : *list* A),
Dup (x :: l) $\leftrightarrow$ *elem* x $l \lor$ *Dup* l.

Lemma Dup_singl :
  $\forall$ (A : Type) (x : A), $\neg$ Dup $[x]$.

Lemma Dup_cons_inv :
  $\forall$ (A : Type) (h : A) (t : $list$ A),
    $\neg$ Dup (h :: t) $\rightarrow$ $\neg$ $elem$ h t $\wedge$ $\neg$ Dup t.

Lemma Dup_spec :
  $\forall$ (A : Type) (l : $list$ A),
    Dup l $\leftrightarrow$
    $\exists$ (x : A) ($l1$ $l2$ $l3$: $list$ A),
      $l = l1$ ++ x :: $l2$ ++ x :: $l3$.

Lemma Dup_NoDup :
  $\forall$ (A : Type) (l : $list$ A),
    $\neg$ Dup l $\leftrightarrow$ $NoDup$ l.

Lemma Dup_length :
  $\forall$ (A : Type) (l : $list$ A),
    Dup l $\rightarrow$ $2 \leq$ $length$ l.

Lemma Dup_snoc :
  $\forall$ (A : Type) (x : A) (l : $list$ A),
    Dup ($snoc$ x l) $\leftrightarrow$ Dup l $\vee$ $elem$ x l.

Lemma Dup_app_l :
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    Dup $l1$ $\rightarrow$ Dup ($l1$ ++ $l2$).

Lemma Dup_app_r :
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    Dup $l2$ $\rightarrow$ Dup ($l1$ ++ $l2$).

Lemma Dup_app_both :
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $elem$ x $l1$ $\rightarrow$ $elem$ x $l2$ $\rightarrow$ Dup ($l1$ ++ $l2$).

Lemma Dup_app :
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    Dup ($l1$ ++ $l2$) $\leftrightarrow$
    Dup $l1$ $\vee$ Dup $l2$ $\vee$ $\exists$ x : A, $elem$ x $l1$ $\wedge$ $elem$ x $l2$.

Lemma Dup_rev :
  $\forall$ (A : Type) (l : $list$ A),
    Dup (rev l) $\leftrightarrow$ Dup l.

Lemma Dup_map :
  $\forall$ (A B : Type) (f : $A \rightarrow B$) (l : $list$ A),
    Dup l $\rightarrow$ Dup (map f l).

Lemma Dup_map_conv :

$\forall$ (A B : Type) (f : $A \to B$) (l : *list* A),
  ($\forall$ x y : A, f x = f y $\to$ x = y) $\to$
    Dup (*map* f l) $\to$ *Dup* l.

Lemma *Dup_join* :
$\forall$ (A : Type) (ll : *list* (*list* A)),
  Dup (*join* ll) $\to$
  ($\exists$ l : *list* A, *elem* l ll $\land$ *Dup* l) $\lor$
  ($\exists$ (x : A) ($l1$ $l2$: *list* A),
    elem x $l1$ $\land$ *elem* x $l2$ $\land$ *elem* $l1$ ll $\land$ *elem* $l2$ ll).

Lemma *Dup_replicate* :
$\forall$ (A : Type) (n : *nat*) (x : A),
  Dup (*replicate* n x) $\to$ $2 \le n$.

Lemma *Dup_nth* :
$\forall$ (A : Type) (l : *list* A),
  Dup l $\leftrightarrow$
  $\exists$ (x : A) ($n1$ $n2$: *nat*),
    $n1 < n2$ $\land$ *nth* $n1$ l = *Some* x $\land$ *nth* $n2$ l = *Some* x.

Lemma *Dup_take* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*),
  Dup (*take* n l) $\to$ *Dup* l.

Lemma *Dup_drop* :
$\forall$ (A : Type) (l : *list* A) (n : *nat*),
  Dup (*drop* n l) $\to$ *Dup* l.

```
(*  TODO: Dup dla insert i replace *)
(*  TODO: findIndex, takeWhile, dropWhile dla replace *)
```

Lemma *Dup_filter* :
$\forall$ (A : Type) (p : $A \to$ *bool*) (l : *list* A),
  Dup (*filter* p l) $\to$ *Dup* l.

Lemma *Dup_filter_conv* :
$\forall$ (A : Type) (p : $A \to$ *bool*) (l : *list* A),
  Dup l $\to$
    Dup (*filter* p l) $\lor$
    Dup (*filter* (fun x : A $\Rightarrow$ *negb* (p x)) l).

Lemma *Dup_partition* :
$\forall$ (A : Type) (p : $A \to$ *bool*) (l $l1$ $l2$: *list* A),
  partition p l = ($l1$, $l2$) $\to$ *Dup* l $\leftrightarrow$ *Dup* $l1$ $\lor$ *Dup* $l2$.

Lemma *Dup_takeWhile* :
$\forall$ (A : Type) (p : $A \to$ *bool*) (l : *list* A),
  Dup (*takeWhile* p l) $\to$ *Dup* l.

Lemma *Dup_dropWhile* :

```
  ∀ (A : Type) (p : A → bool) (l : list A),
    Dup (dropWhile p l) → Dup l.

Lemma Dup_takeWhile_dropWhile_conv :
  ∀ (A : Type) (p : A → bool) (l : list A),
    Dup l →
      Dup (takeWhile p l) ∨
      Dup (dropWhile p l) ∨
      ∃ x : A,
        elem x (takeWhile p l) ∧ elem x (dropWhile p l).

Lemma Dup_span :
  ∀ (A : Type) (p : A → bool) (x : A) (l b e : list A),
    span p l = Some (b, x, e) →
      Dup l ↔ Dup b ∨ Dup e ∨ elem x b ∨ elem x e ∨
        ∃ y : A, elem y b ∧ elem y e.

(*  TODO: NoDup, Rep *)

Lemma Dup_zip :
  ∀ (A B : Type) (la : list A) (lb : list B),
    Dup (zip la lb) → Dup la ∧ Dup lb.

Lemma Dup_zip_conv :
  ∀ (A B : Type) (la : list A) (lb : list B),
    ¬ Dup la ∧ ¬ Dup lb → ¬ Dup (zip la lb).

Lemma Dup_pmap :
  ∃ (A B : Type) (f : A → option B) (l : list A),
    Dup l ∧ ¬ Dup (pmap f l).

Lemma Dup_intersperse :
  ∀ (A : Type) (x : A) (l : list A),
    Dup (intersperse x l) → 2 ≤ length l.
```

10.5.5 *Rep*

Jeżeli zastanowimy się chwilę, to dojdziemy do wniosku, że *Dup l* znaczy "istnieje x, który występuje na liście l co najmniej dwa razy". Widać więc, że *Dup* jest jedynie specjalnym przypadkiem pewngo bardziej ogólnego predykatu *Rep x n* dla dowolnego x oraz n = 2. Zdefiniuj relację *Rep*. Zdanie *Rep x n l* zachodzi, gdy element x występuje na liście l co najmnej n razy.

Zastanów się, czy lepsza będzie definicja induktywna, czy rekurencyjna. Jeżeli nie masz nic lepszego do roboty, zaimplementuj obie wersje i porównaj je pod względem łatwości w użyciu.

```
Lemma Rep_S_cons :
  ∀ (A : Type) (x y : A) (n : nat) (l : list A),
```

$Rep\ x\ (S\ n)\ (y :: l) \leftrightarrow (x = y \land Rep\ x\ n\ l) \lor Rep\ x\ (S\ n)\ l.$

Lemma *Rep_cons* :
$\forall$ (A : Type) ($x\ y$: A) (n : nat) (l : $list\ A$),
$Rep\ x\ n\ (y :: l) \leftrightarrow (x = y \land Rep\ x\ (n - 1)\ l) \lor Rep\ x\ n\ l.$

Lemma *elem_Rep* :
$\forall$ (A : Type) (x : A) (l : $list\ A$),
$elem\ x\ l \to Rep\ x\ 1\ l.$

Lemma *Rep_elem* :
$\forall$ (A : Type) (x : A) (n : nat) (l : $list\ A$),
$1 \le n \to Rep\ x\ n\ l \to elem\ x\ l.$

Lemma *Dup_Rep* :
$\forall$ (A : Type) (l : $list\ A$),
$Dup\ l \to \exists\ x : A,\ Rep\ x\ 2\ l.$

Lemma *Rep_Dup* :
$\forall$ (A : Type) (x : A) (n : nat) (l : $list\ A$),
$2 \le n \to Rep\ x\ n\ l \to Dup\ l.$

Lemma *Rep_le* :
$\forall$ (A : Type) (x : A) ($n\ m$: nat) (l : $list\ A$),
$n \le m \to Rep\ x\ m\ l \to Rep\ x\ n\ l.$

Lemma *Rep_S_inv* :
$\forall$ (A : Type) (x : A) (n : nat) (l : $list\ A$),
$Rep\ x\ (S\ n)\ l \to Rep\ x\ n\ l.$

Lemma *Rep_length* :
$\forall$ (A : Type) (x : A) (n : nat) (l : $list\ A$),
$Rep\ x\ n\ l \to n \le length\ l.$

Lemma *Rep_S_snoc* :
$\forall$ (A : Type) (x : A) (n : nat) (l : $list\ A$),
$Rep\ x\ n\ l \to Rep\ x\ (S\ n)\ (snoc\ x\ l).$

Lemma *Rep_snoc* :
$\forall$ (A : Type) ($x\ y$: A) (n : nat) (l : $list\ A$),
$Rep\ x\ n\ l \to Rep\ x\ n\ (snoc\ y\ l).$

Lemma *Rep_app_l* :
$\forall$ (A : Type) (x : A) (n : nat) ($l1\ l2$: $list\ A$),
$Rep\ x\ n\ l1 \to Rep\ x\ n\ (l1 +\!+\ l2).$

Lemma *Rep_app_r* :
$\forall$ (A : Type) (x : A) (n : nat) ($l1\ l2$: $list\ A$),
$Rep\ x\ n\ l2 \to Rep\ x\ n\ (l1 +\!+\ l2).$

Lemma *Rep_app* :
$\forall$ (A : Type) (x : A) ($n1\ n2$: nat) ($l1\ l2$: $list\ A$),

$Rep\ x\ n1\ l1 \rightarrow Rep\ x\ n2\ l2 \rightarrow Rep\ x\ (n1 + n2)\ (l1 ++ l2).$

Lemma Rep_app_conv :
$\forall\ (A : \texttt{Type})\ (x : A)\ (n : nat)\ (l1\ l2 : list\ A),$
$Rep\ x\ n\ (l1 ++ l2) \leftrightarrow$
$\exists\ n1\ n2 : nat,$
$Rep\ x\ n1\ l1 \wedge Rep\ x\ n2\ l2 \wedge n = n1 + n2.$

Lemma Rep_rev :
$\forall\ (A : \texttt{Type})\ (x : A)\ (n : nat)\ (l : list\ A),$
$Rep\ x\ n\ (rev\ l) \leftrightarrow Rep\ x\ n\ l.$

Lemma Rep_map :
$\forall\ (A\ B : \texttt{Type})\ (f : A \rightarrow B)\ (x : A)\ (n : nat)\ (l : list\ A),$
$Rep\ x\ n\ l \rightarrow Rep\ (f\ x)\ n\ (map\ f\ l).$

Lemma Rep_map_conv :
$\forall\ (A\ B : \texttt{Type})\ (f : A \rightarrow B)\ (x : A)\ (n : nat)\ (l : list\ A),$
$(\forall\ x\ y : A, f\ x = f\ y \rightarrow x = y) \rightarrow$
$Rep\ (f\ x)\ n\ (map\ f\ l) \rightarrow Rep\ x\ n\ l.$

Lemma $Rep\_replicate$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (n : nat),$
$Rep\ x\ n\ (replicate\ n\ x).$

Lemma $Rep\_replicate\_general$:
$\forall\ (A : \texttt{Type})\ (x : A)\ (n\ m : nat),$
$n \leq m \rightarrow Rep\ x\ n\ (replicate\ m\ x).$

Lemma Rep_take :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n\ m : nat),$
$Rep\ x\ n\ (take\ m\ l) \rightarrow Rep\ x\ n\ l.$

Lemma Rep_drop :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l : list\ A)\ (n\ m : nat),$
$Rep\ x\ n\ (drop\ m\ l) \rightarrow Rep\ x\ n\ l.$

Lemma Rep_filter :
$\forall\ (A : \texttt{Type})\ (p : A \rightarrow bool)\ (x : A)\ (n : nat)\ (l : list\ A),$
$Rep\ x\ n\ (filter\ p\ l) \rightarrow Rep\ x\ n\ l.$

Lemma Rep_filter_true :
$\forall\ (A : \texttt{Type})\ (p : A \rightarrow bool)\ (x : A)\ (n : nat)\ (l : list\ A),$
$p\ x = true \rightarrow Rep\ x\ n\ l \rightarrow Rep\ x\ n\ (filter\ p\ l).$

Lemma Rep_filter_false :
$\forall\ (A : \texttt{Type})\ (p : A \rightarrow bool)\ (x : A)\ (l : list\ A)\ (n : nat),$
$p\ x = false \rightarrow Rep\ x\ n\ (filter\ p\ l) \rightarrow n = 0.$

Lemma $Rep\_takeWhile$:
$\forall\ (A : \texttt{Type})\ (p : A \rightarrow bool)\ (x : A)\ (l : list\ A)\ (n : nat),$

$Rep\ x\ n\ (takeWhile\ p\ l) \rightarrow Rep\ x\ n\ l.$

Lemma $Rep\_dropWhile$:
$\forall\ (A : \texttt{Type})\ (p : A \rightarrow bool)\ (x : A)\ (l : list\ A)\ (n : nat),$
$Rep\ x\ n\ (dropWhile\ p\ l) \rightarrow Rep\ x\ n\ l.$

Lemma Rep_zip :
$\forall\ (A\ B : \texttt{Type})\ (a : A)\ (b : B)\ (la : list\ A)\ (lb : list\ B)\ (n : nat),$
$Rep\ (a, b)\ n\ (zip\ la\ lb) \rightarrow Rep\ a\ n\ la \wedge Rep\ b\ n\ lb.$

Lemma $Rep\_intersperse$:
$\forall\ (A : \texttt{Type})\ (x\ y : A)\ (n : nat)\ (l : list\ A),$
$Rep\ x\ n\ (intersperse\ y\ l) \leftrightarrow$
$Rep\ x\ n\ l \vee x = y \wedge Rep\ x\ (S\ n\ \text{-}\ length\ l)\ l.$

10.5.6 *Exists*

Zaimplementuj induktywny predykat *Exists*. *Exists* P l zachodzi, gdy lista l zawiera taki element, który spełnia predykat P.

Lemma $Exists\_spec$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (l : list\ A),$
$Exists\ P\ l \leftrightarrow \exists\ x : A,\ elem\ x\ l \wedge P\ x.$

Lemma $Exists\_nil$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop}),$
$Exists\ P\ [] \leftrightarrow False.$

Lemma $Exists\_cons$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (h : A)\ (t : list\ A),$
$Exists\ P\ (h :: t) \leftrightarrow P\ h \vee Exists\ P\ t.$

Lemma $Exists\_length$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (l : list\ A),$
$Exists\ P\ l \rightarrow 1 \leq length\ l.$

Lemma $Exists\_snoc$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (l : list\ A),$
$Exists\ P\ (snoc\ x\ l) \leftrightarrow Exists\ P\ l \vee P\ x.$

Lemma $Exists\_app$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (l1\ l2 : list\ A),$
$Exists\ P\ (l1 +\!+\ l2) \leftrightarrow Exists\ P\ l1 \vee Exists\ P\ l2.$

Lemma $Exists\_rev$:
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (l : list\ A),$
$Exists\ P\ (rev\ l) \leftrightarrow Exists\ P\ l.$

Lemma $Exists\_map$:
$\forall\ (A\ B : \texttt{Type})\ (P : B \rightarrow \texttt{Prop})\ (f : A \rightarrow B)\ (l : list\ A),$

```
    Exists P (map f l) → Exists (fun x : A ⇒ P (f x)) l.
Lemma Exists_join :
  ∀ (A : Type) (P : A → Prop) (ll : list (list A)),
    Exists P (join ll) ↔
    Exists (fun l : list A ⇒ Exists P l) ll.
Lemma Exists_replicate :
  ∀ (A : Type) (P : A → Prop) (n : nat) (x : A),
    Exists P (replicate n x) ↔ 1 ≤ n ∧ P x.
Lemma Exists_nth :
  ∀ (A : Type) (P : A → Prop) (l : list A),
    Exists P l ↔
    ∃ (n : nat) (x : A), nth n l = Some x ∧ P x.
Lemma Exists_remove :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Exists P l →
    match remove n l with
        | None ⇒ True
        | Some (x, l') ⇒ ¬ P x → Exists P l'
    end.
Lemma Exists_take :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Exists P (take n l) → Exists P l.
Lemma Exists_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Exists P (drop n l) → Exists P l.
Lemma Exists_take_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Exists P l → Exists P (take n l) ∨ Exists P (drop n l).
Lemma Exists_splitAt :
  ∀ (A : Type) (P : A → Prop) (l l1 l2 : list A) (n : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      Exists P l ↔ P x ∨ Exists P l1 ∨ Exists P l2.
Lemma Exists_insert :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat) (x : A),
    Exists P (insert l n x) ↔ P x ∨ Exists P l.
Lemma Exists_replace :
  ∀ (A : Type) (P : A → Prop) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      Exists P l' ↔
      Exists P (take n l) ∨ P x ∨ Exists P (drop (S n) l).
```

Lemma $Exists\_filter$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $Exists\ P\ (filter\ p\ l) \to Exists\ P\ l$.

Lemma $Exists\_filter\_conv$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $Exists\ P\ l \to$
      $Exists\ P\ (filter\ p\ l) \lor$
      $Exists\ P\ (filter\ ($fun $x : A \Rightarrow negb\ (p\ x))\ l)$.

Lemma $Exists\_filter\_compat$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $(\forall\ x : A,\ P\ x \leftrightarrow p\ x = false) \to \neg\ Exists\ P\ (filter\ p\ l)$.

Lemma $Exists\_partition$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) ($l\ l1\ l2$: $list\ A$),
    $partition\ p\ l = (l1,\ l2) \to$
      $Exists\ P\ l \leftrightarrow Exists\ P\ l1 \lor Exists\ P\ l2$.

Lemma $Exists\_takeWhile$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $Exists\ P\ (takeWhile\ p\ l) \to Exists\ P\ l$.

Lemma $Exists\_takeWhile\_compat$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $(\forall\ x : A,\ P\ x \leftrightarrow p\ x = false) \to \neg\ Exists\ P\ (takeWhile\ p\ l)$.

Lemma $Exists\_dropWhile$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $Exists\ P\ (dropWhile\ p\ l) \to Exists\ P\ l$.

Lemma $Exists\_takeWhile\_dropWhile$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list\ A$),
    $Exists\ P\ l \to Exists\ P\ (takeWhile\ p\ l) \lor Exists\ P\ (dropWhile\ p\ l)$.

Lemma $Exists\_span$:
  $\forall$
    (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (x : A) ($l\ b\ e$: $list\ A$),
      $(\forall\ x : A,\ P\ x \leftrightarrow p\ x = true) \to$
      $span\ p\ l = Some\ (b,\ x,\ e) \to$
        $Exists\ P\ l \leftrightarrow Exists\ P\ b \lor P\ x \lor Exists\ P\ e$.

Lemma $Exists\_interesting$:
  $\forall$ ($A\ B$: Type) (P : $A \times B \to$ Prop) (la : $list\ A$) (hb : B) (tb : $list\ B$),
    $Exists\ ($fun $a : A \Rightarrow Exists\ ($fun $b : B \Rightarrow P\ (a,\ b))\ tb)\ la \to$
    $Exists\ ($fun $a : A \Rightarrow Exists\ ($fun $b : B \Rightarrow P\ (a,\ b))\ (hb :: tb))\ la$.

Lemma $Exists\_zip$:
  $\forall$ ($A\ B$: Type) (P : $A \times B \to$ Prop) (la : $list\ A$) (lb : $list\ B$),
    $Exists\ P\ (zip\ la\ lb) \to$

$\quad\quad$ *Exists* (fun $a : A \Rightarrow$ *Exists* (fun $b : B \Rightarrow P\ (a, b))\ lb)\ la$.

Lemma *Exists_pmap* :
$\forall$ (A B : Type) ($f : A \to$ *option* B) ($P : B \to$ Prop) ($l :$ *list* A),
Exists P (*pmap* $f\ l$) $\leftrightarrow$
Exists (fun $x : A \Rightarrow$ match $f\ x$ with | *Some* $b \Rightarrow P\ b$ | _ $\Rightarrow$ *False* end) l.

Lemma *Exists_intersperse* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($x : A$) ($l :$ *list* A),
Exists P (*intersperse* $x\ l$) $\leftrightarrow$
Exists $P\ l \lor (P\ x \land 2 \leq$ *length* l).

10.5.7 *Forall*

Zaimplementuj induktywny predykat *Forall*. *Forall* $P\ l$ jest spełniony, gdy każdy element listy l spełnia predykat P.

Lemma *Forall_spec* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($l :$ *list* A),
Forall $P\ l \leftrightarrow \forall\ x : A$, *elem* $x\ l \to P\ x$.

Lemma *Forall_nil* :
$\forall$ (A : Type) ($P : A \to$ Prop),
Forall P [] $\leftrightarrow$ *True*.

Lemma *Forall_cons* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($h : A$) ($t :$ *list* A),
Forall P ($h :: t$) $\leftrightarrow P\ h \land$ *Forall* $P\ t$.

Lemma *Forall_snoc* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($x : A$) ($l :$ *list* A),
Forall P (*snoc* $x\ l$) $\leftrightarrow$ *Forall* $P\ l \land P\ x$.

Lemma *Forall_app* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($l1\ l2 :$ *list* A),
Forall P ($l1$ ++ $l2$) $\leftrightarrow$ *Forall* $P\ l1 \land$ *Forall* $P\ l2$.

Lemma *Forall_rev* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($l :$ *list* A),
Forall P (*rev* l) $\leftrightarrow$ *Forall* $P\ l$.

Lemma *Forall_map* :
$\forall$ (A B : Type) ($P : B \to$ Prop) ($f : A \to B$) ($l :$ *list* A),
Forall P (*map* $f\ l$) $\to$ *Forall* (fun $x : A \Rightarrow P\ (f\ x))\ l$.

Lemma *Forall_join* :
$\forall$ (A : Type) ($P : A \to$ Prop) ($ll :$ *list* (*list* A)),
Forall P (*join* ll) $\leftrightarrow$ *Forall* (fun $l :$ *list* $A \Rightarrow$ *Forall* $P\ l$) ll.

Lemma *Forall_replicate* :

```
  ∀ (A : Type) (P : A → Prop) (n : nat) (x : A),
    Forall P (replicate n x) ↔ n = 0 ∨ P x.

Lemma Forall_nth :
  ∀ (A : Type) (P : A → Prop) (l : list A),
    Forall P l ↔ ∀ n : nat, n < length l →
      ∃ x : A, nth n l = Some x ∧ P x.

Lemma Forall_remove :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Forall P l →
    match remove n l with
        | None ⇒ True
        | Some (x, l') ⇒ Forall P l'
    end.

Lemma Forall_take :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Forall P l → Forall P (take n l).

Lemma Forall_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Forall P l → Forall P (drop n l).

Lemma Forall_take_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    Forall P (take n l) → Forall P (drop n l) → Forall P l.

Lemma Forall_splitAt :
  ∀ (A : Type) (P : A → Prop) (l l1 l2 : list A) (n : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      Forall P l ↔ P x ∧ Forall P l1 ∧ Forall P l2.

Lemma Forall_insert :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat) (x : A),
      Forall P (insert l n x) ↔ P x ∧ Forall P l.

Lemma Forall_replace :
  ∀ (A : Type) (P : A → Prop) (l l' : list A) (n : nat) (x : A),
    replace l n x = Some l' →
      Forall P l' ↔
      Forall P (take n l) ∧ P x ∧ Forall P (drop (S n) l).

Lemma Forall_filter :
  ∀ (A : Type) (P : A → Prop) (p : A → bool) (l : list A),
    Forall P l → Forall P (filter p l).

Lemma Forall_filter_conv :
  ∀ (A : Type) (P : A → Prop) (p : A → bool) (l : list A),
    Forall P (filter p l) →
```

$Forall\ P\ (filter\ (\mathtt{fun}\ x : A \Rightarrow negb\ (p\ x))\ l) \rightarrow$
$Forall\ P\ l.$

Lemma $Forall\_filter\_compat$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (l : list\ A),$
$(\forall\ x : A,\ P\ x \leftrightarrow p\ x = true) \rightarrow Forall\ P\ (filter\ p\ l).$

Lemma $Forall\_partition$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (l\ l1\ l2 : list\ A),$
$partition\ p\ l = (l1,\ l2) \rightarrow$
$Forall\ P\ l \leftrightarrow Forall\ P\ l1 \wedge Forall\ P\ l2.$

Lemma $Forall\_takeWhile$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (l : list\ A),$
$Forall\ P\ l \rightarrow Forall\ P\ (takeWhile\ p\ l).$

Lemma $Forall\_takeWhile\_compat$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (l : list\ A),$
$(\forall\ x : A,\ P\ x \leftrightarrow p\ x = true) \rightarrow Forall\ P\ (takeWhile\ p\ l).$

Lemma $Forall\_dropWhile$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (l : list\ A),$
$Forall\ P\ l \rightarrow Forall\ P\ (dropWhile\ p\ l).$

Lemma $Forall\_takeWhile\_dropWhile$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (l : list\ A),$
$Forall\ P\ (takeWhile\ p\ l) \rightarrow Forall\ P\ (dropWhile\ p\ l) \rightarrow Forall\ P\ l.$

Lemma $Forall\_span$:
$\forall$
$(A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (p : A \rightarrow bool)\ (x : A)\ (l\ b\ e : list\ A),$
$(\forall\ x : A,\ P\ x \leftrightarrow p\ x = true) \rightarrow$
$span\ p\ l = Some\ (b,\ x,\ e) \rightarrow$
$Forall\ P\ l \leftrightarrow Forall\ P\ b \wedge P\ x \wedge Forall\ P\ e.$

Lemma $Forall\_zip$:
$\forall\ (A\ B : \mathtt{Type})\ (PA : A \rightarrow \mathtt{Prop})\ (PB : B \rightarrow \mathtt{Prop})$
$(la : list\ A)\ (lb : list\ B),$
$Forall\ PA\ la \rightarrow Forall\ PB\ lb \rightarrow$
$Forall\ (\mathtt{fun}\ '(a,\ b) \Rightarrow PA\ a \wedge PB\ b)\ (zip\ la\ lb).$

Lemma $Forall\_pmap$:
$\forall\ (A\ B : \mathtt{Type})\ (f : A \rightarrow option\ B)\ (P : B \rightarrow \mathtt{Prop})\ (l : list\ A),$
$Forall\ (\mathtt{fun}\ x : A \Rightarrow \mathtt{match}\ f\ x\ \mathtt{with}\ |\ Some\ b \Rightarrow P\ b\ |\ \_ \Rightarrow False\ \mathtt{end})\ l \rightarrow$
$Forall\ P\ (pmap\ f\ l).$

Lemma $Forall\_intersperse$:
$\forall\ (A : \mathtt{Type})\ (P : A \rightarrow \mathtt{Prop})\ (x : A)\ (l : list\ A),$
$Forall\ P\ (intersperse\ x\ l) \leftrightarrow$
$Forall\ P\ l \wedge (2 \leq length\ l \rightarrow P\ x).$

Lemma *Forall_impl* :
$\forall$ (A : Type) (P Q : $A \to$ Prop) (l : *list* A),
($\forall$ x : A , P $x \to Q$ x) $\to$
Forall P $l \to$ *Forall* Q l.

Lemma *Forall_Exists* :
$\forall$ (A : Type) (P : $A \to$ Prop) (l : *list* A),
Forall P $l \to \neg$ *Exists* (fun x : $A \Rightarrow \neg$ P x) l.

Lemma *Exists_Forall* :
$\forall$ (A : Type) (P : $A \to$ Prop) (l : *list* A),
Exists P $l \to \neg$ *Forall* (fun x : $A \Rightarrow \neg$ P x) l.

10.5.8 *AtLeast*

Czas uogólnić relację *Rep* oraz predykaty *Exists* i *Forall*. Zdefiniuj w tym celu relację *AtLeast*. Zdanie *AtLeast* P n l zachodzi, gdy na liście l jest co najmniej n elementów spełniających predykat P.

Lemma *AtLeast_cons* :
$\forall$ (A : Type) (P : $A \to$ Prop) (n : *nat*) (h : A) (t : *list* A),
AtLeast P n (h :: t) $\leftrightarrow$
AtLeast P n $t \lor P$ $h \land$ *AtLeast* P (n - 1) t.

Lemma *AtLeast_cons'* :
$\forall$ (A : Type) (P : $A \to$ Prop) (n : *nat*) (h : A) (t : *list* A),
AtLeast P (S n) (h :: t) $\to$ *AtLeast* P n t.

Lemma *AtLeast_length* :
$\forall$ (A : Type) (P : $A \to$ Prop) (n : *nat*) (l : *list* A),
AtLeast P n $l \to n \leq$ *length* l.

Lemma *AtLeast_snoc* :
$\forall$ (A : Type) (P : $A \to$ Prop) (n : *nat*) (x : A) (l : *list* A),
AtLeast P n $l \to$ *AtLeast* P n (*snoc* x l).

Lemma *AtLeast_S_snoc* :
$\forall$ (A : Type) (P : $A \to$ Prop) (n : *nat*) (x : A) (l : *list* A),
AtLeast P n $l \to P$ $x \to$ *AtLeast* P (S n) (*snoc* x l).

Lemma *AtLeast_Exists* :
$\forall$ (A : Type) (P : $A \to$ Prop) (l : *list* A),
AtLeast P 1 $l \leftrightarrow$ *Exists* P l.

Lemma *AtLeast_Forall* :
$\forall$ (A : Type) (P : $A \to$ Prop) (l : *list* A),
AtLeast P (*length* l) $l \leftrightarrow$ *Forall* P l.

Lemma *AtLeast_Rep* :

$\forall\ (A : \texttt{Type})\ (x : A)\ (n : nat)\ (l : list\ A),$
$\quad AtLeast\ (\texttt{fun}\ y : A \Rightarrow x = y)\ n\ l \leftrightarrow Rep\ x\ n\ l.$

Lemma *AtLeast_app_l* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (l1\ l2 : list\ A),$
$\quad AtLeast\ P\ n\ l1 \to AtLeast\ P\ n\ (l1 +\!+\ l2).$

Lemma *AtLeast_app_r* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (l1\ l2 : list\ A),$
$\quad AtLeast\ P\ n\ l2 \to AtLeast\ P\ n\ (l1 +\!+\ l2).$

Lemma *AtLeast_plus_app* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n1\ n2 : nat)\ (l1\ l2 : list\ A),$
$\quad AtLeast\ P\ n1\ l1 \to AtLeast\ P\ n2\ l2 \to$
$\quad\quad AtLeast\ P\ (n1 + n2)\ (l1 +\!+\ l2).$

Lemma *AtLeast_app_conv* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (l1\ l2 : list\ A),$
$\quad AtLeast\ P\ n\ (l1 +\!+\ l2) \to$
$\quad\quad \exists\ n1\ n2 : nat,\ AtLeast\ P\ n1\ l1 \land AtLeast\ P\ n2\ l2 \land n = n1 + n2.$

Lemma *AtLeast_app* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (l1\ l2 : list\ A),$
$\quad AtLeast\ P\ n\ (l1 +\!+\ l2) \leftrightarrow$
$\quad \exists\ n1\ n2 : nat,$
$\quad\quad AtLeast\ P\ n1\ l1 \land AtLeast\ P\ n2\ l2 \land n = n1 + n2.$

Lemma *AtLeast_app_comm* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (l1\ l2 : list\ A),$
$\quad AtLeast\ P\ n\ (l1 +\!+\ l2) \to AtLeast\ P\ n\ (l2 +\!+\ l1).$

Lemma *AtLeast_rev* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (l : list\ A),$
$\quad AtLeast\ P\ n\ (rev\ l) \leftrightarrow AtLeast\ P\ n\ l.$

Lemma *AtLeast_map* :
$\forall\ (A\ B : \texttt{Type})\ (P : B \to \texttt{Prop})\ (f : A \to B)\ (n : nat)\ (l : list\ A),$
$\quad AtLeast\ (\texttt{fun}\ x : A \Rightarrow P\ (f\ x))\ n\ l \to$
$\quad\quad AtLeast\ P\ n\ (map\ f\ l).$

Lemma *AtLeast_map_conv* :
$\forall\ (A\ B : \texttt{Type})\ (P : B \to \texttt{Prop})\ (f : A \to B)\ (n : nat)\ (l : list\ A),$
$\quad (\forall\ x\ y : A,\ f\ x = f\ y \to x = y) \to AtLeast\ P\ n\ (map\ f\ l) \to$
$\quad\quad AtLeast\ (\texttt{fun}\ x : A \Rightarrow P\ (f\ x))\ n\ l.$

Lemma *AtLeast_replicate* :
$\forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop})\ (n : nat)\ (x : A),$
$\quad n \neq 0 \to P\ x \to AtLeast\ P\ n\ (replicate\ n\ x).$

Lemma *AtLeast_replicate_conv* :

```
  ∀ (A : Type) (P : A → Prop) (n m : nat) (x : A),
    AtLeast P m (replicate n x) → m = 0 ∨ m ≤ n ∧ P x.

Lemma AtLeast_remove :
  ∀ (A : Type) (P : A → Prop) (l : list A) (m : nat),
    AtLeast P m l → ∀ n : nat,
      match remove n l with
          | None ⇒ True
          | Some (_, l') ⇒ AtLeast P (m - 1) l'
      end.

Lemma AtLeast_take :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n m : nat),
    AtLeast P m (take n l) → AtLeast P m l.

Lemma AtLeast_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n m : nat),
    AtLeast P m (drop n l) → AtLeast P m l.

Lemma AtLeast_take_drop :
  ∀ (A : Type) (P : A → Prop) (n m : nat) (l : list A),
    AtLeast P n l →
    ∃ n1 n2 : nat,
      AtLeast P n1 (take m l) ∧ AtLeast P n2 (drop m l) ∧ n = n1 + n2.

Lemma AtLeast_splitAt :
  ∀ (A : Type) (P : A → Prop) (l l1 l2 : list A) (n : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      ∀ m : nat,
        AtLeast P m l →
        ∃ m1 mx m2 : nat,
          AtLeast P m1 l1 ∧ AtLeast P mx [x] ∧ AtLeast P m2 l2 ∧
          m1 + mx + m2 = m.

Lemma AtLeast_insert :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n : nat),
    AtLeast P n l → ∀ (m : nat) (x : A),
      AtLeast P n (insert l m x).

Lemma AtLeast_replace :
  ∀ (A : Type) (P : A → Prop) (l l' : list A) (n m : nat) (x : A),
    replace l n x = Some l' → AtLeast P m l →
      AtLeast P (m - 1) l'.

Lemma AtLeast_replace' :
  ∀ (A : Type) (P : A → Prop) (l l' : list A) (n m : nat) (x : A),
    replace l n x = Some l' → AtLeast P m l → P x →
      AtLeast P m l'.
```

Lemma $AtLeast\_replace\_conv$:
$\forall$ (A : Type) (P : $A \to$ Prop) (l l' : $list$ A) (n m : nat) (x : A),
replace l n x = $Some$ l' $\to$ $AtLeast$ P m l' $\to$ $AtLeast$ P (m - 1) l.

Lemma $AtLeast\_replace\_conv'$:
$\forall$ (A : Type) (P : $A \to$ Prop) (l l' : $list$ A) (n m : nat) (x y : A),
replace l n x = $Some$ l' $\to$ nth n l = $Some$ y $\to$ P y $\to$
$AtLeast$ P m l' $\to$ $AtLeast$ P m l.

(* TODO: *Exactly*, *AtMost* dla replace *)

Lemma $AtLeast\_filter$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (n : nat) (l : $list$ A),
$AtLeast$ P n ($filter$ p l) $\to$ $AtLeast$ P n l.

Lemma $AtLeast\_filter\_compat\_true$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list$ A),
($\forall$ x : A, P x $\leftrightarrow$ p x = $true$) $\to$
$AtLeast$ P ($length$ ($filter$ p l)) ($filter$ p l).

Lemma $AtLeast\_filter\_compat\_false$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (n : nat) (l : $list$ A),
($\forall$ x : A, P x $\leftrightarrow$ p x = $false$) $\to$
$AtLeast$ P n ($filter$ p l) $\to$ n = 0.

Lemma $AtLeast\_takeWhile$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (n : nat) (l : $list$ A),
$AtLeast$ P n ($takeWhile$ p l) $\to$ $AtLeast$ P n l.

Lemma $AtLeast\_dropWhile$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (n : nat) (l : $list$ A),
$AtLeast$ P n ($dropWhile$ p l) $\to$ $AtLeast$ P n l.

Lemma $AtLeast\_takeWhile\_true$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list$ A),
($\forall$ x : A, P x $\leftrightarrow$ p x = $true$) $\to$
$AtLeast$ P ($length$ ($takeWhile$ p l)) ($takeWhile$ p l).

Lemma $AtLeast\_takeWhile\_false$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (n : nat) (l : $list$ A),
($\forall$ x : A, P x $\leftrightarrow$ p x = $false$) $\to$
$AtLeast$ P n ($takeWhile$ p l) $\to$ n = 0.

Lemma $AtLeast\_dropWhile\_true$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list$ A) (n : nat),
($\forall$ x : A, P x $\leftrightarrow$ p x = $true$) $\to$
$AtLeast$ P n l $\to$ $AtLeast$ P (n - $length$ ($takeWhile$ p l)) ($dropWhile$ p l).

Lemma $AtLeast\_dropWhile\_false$:
$\forall$ (A : Type) (P : $A \to$ Prop) (p : $A \to bool$) (l : $list$ A) (n : nat),

$(\forall\ x : A,\ P\ x \leftrightarrow p\ x = \mathit{false}) \rightarrow$
$\mathit{AtLeast}\ P\ n\ l \rightarrow \mathit{AtLeast}\ P\ n\ (\mathit{dropWhile}\ p\ l).$

Lemma *AtLeast_zip* :
$\forall\ (A\ B : \texttt{Type})\ (PA : A \rightarrow \texttt{Prop})\ (PB : B \rightarrow \texttt{Prop})$
$(la : \mathit{list}\ A)\ (lb : \mathit{list}\ B)\ (n : \mathit{nat}),$
$\mathit{AtLeast}\ (\texttt{fun}\ '(a, b) \Rightarrow PA\ a \land PB\ b)\ n\ (\mathit{zip}\ la\ lb) \rightarrow$
$\mathit{AtLeast}\ PA\ n\ la \land \mathit{AtLeast}\ PB\ n\ lb.$

Lemma *AtLeast_findIndices* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (p : A \rightarrow \mathit{bool})\ (l : \mathit{list}\ A)\ (n : \mathit{nat}),$
$(\forall\ x : A,\ P\ x \leftrightarrow p\ x = \mathit{true}) \rightarrow$
$\mathit{AtLeast}\ P\ n\ l \rightarrow n \leq \mathit{length}\ (\mathit{findIndices}\ p\ l).$

Lemma *AtLeast_1_elem* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (l : \mathit{list}\ A),$
$\mathit{AtLeast}\ P\ 1\ l \leftrightarrow \exists\ x : A,\ \mathit{elem}\ x\ l \land P\ x.$

Lemma *AtLeast_intersperse* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (l : \mathit{list}\ A),$
$P\ x \rightarrow \mathit{AtLeast}\ P\ (\mathit{length}\ l\ \text{-}\ 1)\ (\mathit{intersperse}\ x\ l).$

Lemma *AtLeast_intersperse'* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (n : \mathit{nat})\ (l : \mathit{list}\ A),$
$\mathit{AtLeast}\ P\ n\ l \rightarrow P\ x \rightarrow$
$\mathit{AtLeast}\ P\ (n + (\mathit{length}\ l\ \text{-}\ 1))\ (\mathit{intersperse}\ x\ l).$

Lemma *AtLeast_intersperse''* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (n : \mathit{nat})\ (l : \mathit{list}\ A),$
$\mathit{AtLeast}\ P\ n\ l \rightarrow \neg\ P\ x \rightarrow \mathit{AtLeast}\ P\ n\ (\mathit{intersperse}\ x\ l).$

10.5.9 *Exactly*

Zdefiniuj predykat *Exactly*. Zdanie *Exactly* P n l zachodzi, gdy na liście l występuje dokładnie n elementów spełniających predykat P.

Lemma *Exactly_0_cons* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (l : \mathit{list}\ A),$
$\mathit{Exactly}\ P\ 0\ (x :: l) \leftrightarrow \neg\ P\ x \land \mathit{Exactly}\ P\ 0\ l.$

Lemma *Exactly_S_cons* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : \mathit{nat})\ (x : A)\ (l : \mathit{list}\ A),$
$\mathit{Exactly}\ P\ (S\ n)\ (x :: l) \leftrightarrow$
$P\ x \land \mathit{Exactly}\ P\ n\ l \lor \neg\ P\ x \land \mathit{Exactly}\ P\ (S\ n)\ l.$

Lemma *Exactly_AtLeast* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : \mathit{nat})\ (l : \mathit{list}\ A),$
$\mathit{Exactly}\ P\ n\ l \rightarrow \mathit{AtLeast}\ P\ n\ l.$

Lemma $Exactly\_eq$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n m : nat) (l : $list$ A),
    $Exactly$ P n l $\to$ $Exactly$ P m l $\to$ $n = m$.

Lemma $Exactly\_length$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) (l : $list$ A),
    $Exactly$ P n l $\to$ $n \leq length$ l.

Lemma $Exactly\_snoc$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) (x : A) (l : $list$ A),
    $Exactly$ P n l $\to \neg$ P x $\to$ $Exactly$ P n ($snoc$ x l).

Lemma $Exactly\_S\_snoc$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) (x : A) (l : $list$ A),
    $Exactly$ P n l $\to$ P x $\to$ $Exactly$ P (S n) ($snoc$ x l).

Lemma $Exactly\_app$:
  $\forall$ (A : Type) (P : $A \to$ Prop) ($n1$ $n2$: nat) ($l1$ $l2$: $list$ A),
    $Exactly$ P $n1$ $l1$ $\to$ $Exactly$ P $n2$ $l2$ $\to$ $Exactly$ P ($n1 + n2$) ($l1$ ++ $l2$).

Lemma $Exactly\_app\_conv$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) ($l1$ $l2$: $list$ A),
    $Exactly$ P n ($l1$ ++ $l2$) $\to$
      $\exists$ $n1$ $n2$: nat,
        $Exactly$ P $n1$ $l1$ $\land$ $Exactly$ P $n2$ $l2$ $\land$ $n = n1 + n2$.

Lemma $Exactly\_app\_comm$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) ($l1$ $l2$: $list$ A),
    $Exactly$ P n ($l1$ ++ $l2$) $\to$ $Exactly$ P n ($l2$ ++ $l1$).

Lemma $Exactly\_rev$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) (l : $list$ A),
    $Exactly$ P n (rev l) $\leftrightarrow$ $Exactly$ P n l.

Lemma $Exactly\_map$:
  $\forall$ (A B : Type) (P : $B \to$ Prop) (f : $A \to B$) (n : nat) (l : $list$ A),
    ($\forall$ x y : A, f x = f y $\to$ $x = y$) $\to$
    $Exactly$ (fun x : $A \Rightarrow P$ (f x)) n l $\leftrightarrow$
      $Exactly$ P n (map f l).

Lemma $Exactly\_replicate$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) (x : A),
    P x $\to$ $Exactly$ P n ($replicate$ n x).

Lemma $Exactly\_replicate\_conv$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n : nat) (x : A),
    $Exactly$ P n ($replicate$ n x) $\to$ $n = 0 \lor P$ x.

Lemma $Exactly\_replicate'$:
  $\forall$ (A : Type) (P : $A \to$ Prop) (n m : nat) (x : A),

```
    Exactly P n (replicate m x) ↔
    n = 0 ∧ m = 0 ∨
    n = 0 ∧ ¬ P x ∨
    n = m ∧ P x.

Lemma Exactly_take :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n m1 m2 : nat),
    Exactly P m1 (take n l) → Exactly P m2 l → m1 ≤ m2.

Lemma Exactly_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n m1 m2 : nat),
    Exactly P m1 (drop n l) → Exactly P m2 l → m1 ≤ m2.

Lemma Exactly_take_drop :
  ∀ (A : Type) (P : A → Prop) (l : list A) (n m : nat),
    Exactly P n l → ∃ n1 n2 : nat,
      n = n1 + n2 ∧ Exactly P n1 (take m l) ∧ Exactly P n2 (drop m l).

Lemma Exactly_splitAt :
  ∀ (A : Type) (P : A → Prop) (l l1 l2 : list A) (n : nat) (x : A),
    splitAt n l = Some (l1, x, l2) →
      ∀ m : nat,
        Exactly P m l ↔
        ∃ m1 mx m2 : nat,
          Exactly P m1 l1 ∧ Exactly P mx [x] ∧ Exactly P m2 l2 ∧
          m1 + mx + m2 = m.

Lemma Exactly_filter :
  ∀ (A : Type) (P : A → Prop) (p : A → bool) (l : list A),
    (∀ x : A, P x ↔ p x = true) →
      Exactly P (length (filter p l)) (filter p l).

Lemma Exactly_takeWhile :
  ∀ (A : Type) (P : A → Prop) (p : A → bool) (l : list A),
    (∀ x : A, P x ↔ p x = true) →
      Exactly P (length (takeWhile p l)) (takeWhile p l).

Lemma Exactly_dropWhile :
  ∀ (A : Type) (P : A → Prop) (p : A → bool) (n : nat) (l : list A),
    (∀ x : A, P x ↔ p x = true) →
    Exactly P n l →
      Exactly P (n - length (takeWhile p l)) (dropWhile p l).

Lemma Exactly_span :
  ∀
    (A : Type) (P : A → Prop) (p : A → bool)
    (n : nat)(x : A) (l b e : list A),
      (∀ x : A, P x ↔ p x = true) →
```

$span\ p\ l = Some\ (b,\ x,\ e) \rightarrow$
$Exactly\ P\ n\ l \leftrightarrow$
$\exists\ n1\ n2 : nat,$
$Exactly\ P\ n1\ b \land Exactly\ P\ n2\ e \land$
`if` $p\ x$ `then` $S\ (n1 + n2) = n$ `else` $n1 + n2 = n.$

```
(*  TODO: span i AtLeast, AtMost *)
```

`Lemma` *Exactly_intersperse* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (n : nat)\ (l : list\ A),$
$Exactly\ P\ n\ l \rightarrow P\ x \rightarrow$
$Exactly\ P\ (n + (length\ l\ \text{-}\ 1))\ (intersperse\ x\ l).$

`Lemma` *Exactly_intersperse'* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (n : nat)\ (l : list\ A),$
$Exactly\ P\ n\ l \rightarrow \neg\ P\ x \rightarrow$
$Exactly\ P\ n\ (intersperse\ x\ l).$

10.5.10 *AtMost*

`Lemma` *AtMost_0* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (x : A)\ (l : list\ A),$
$AtMost\ P\ 0\ (x :: l) \leftrightarrow \neg\ P\ x \land AtMost\ P\ 0\ l.$

`Lemma` *AtMost_nil* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : nat),$
$AtMost\ P\ n\ [] \leftrightarrow True.$

`Lemma` *AtMost_le* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : nat)\ (l : list\ A),$
$AtMost\ P\ n\ l \rightarrow \forall\ m : nat,\ n \leq m \rightarrow AtMost\ P\ m\ l.$

`Lemma` *AtMost_S_cons* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : nat)\ (x : A)\ (l : list\ A),$
$AtMost\ P\ (S\ n)\ (x :: l) \leftrightarrow$
$(\tilde{}\ P\ x \land AtMost\ P\ (S\ n)\ l) \lor AtMost\ P\ n\ l.$

`Lemma` *AtMost_S_snoc* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : nat)\ (x : A)\ (l : list\ A),$
$AtMost\ P\ n\ l \rightarrow AtMost\ P\ (S\ n)\ (snoc\ x\ l).$

`Lemma` *AtMost_snoc* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : nat)\ (x : A)\ (l : list\ A),$
$AtMost\ P\ n\ l \rightarrow \neg\ P\ x \rightarrow AtMost\ P\ n\ (snoc\ x\ l).$

`Lemma` *AtMost_S* :
$\forall\ (A : \texttt{Type})\ (P : A \rightarrow \texttt{Prop})\ (n : nat)\ (l : list\ A),$
$AtMost\ P\ n\ l \rightarrow AtMost\ P\ (S\ n)\ l.$

10.6 Relacje między listami

```
(* TODO: zrób coś z tym *)
Inductive bool_le : bool → bool → Prop :=
    | ble_refl : ∀ b : bool, bool_le b b
    | ble_false_true : bool_le false true.
(*
   Definition bool_le (b1 b2 : bool) : Prop :=
   match b1, b2 with
    | false, _ => True
    | true, false => False
    | true, true => True
   end.
   *)
```

10.6.1 Listy jako termy

```
Lemma Sublist_length :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → length l1 < length l2.

Lemma Sublist_cons_l :
  ∀ (A : Type) (x : A) (l : list A), ¬ Sublist (x :: l) l.

Lemma Sublist_cons_l' :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Sublist (x :: l1) l2 → Sublist l1 l2.

Lemma Sublist_nil_cons :
  ∀ (A : Type) (x : A) (l : list A), Sublist [] (x :: l).

Lemma Sublist_irrefl :
  ∀ (A : Type) (l : list A), ¬ Sublist l l.

Lemma Sublist_antisym :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ¬ Sublist l2 l1.

Lemma Sublist_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    Sublist l1 l2 → Sublist l2 l3 → Sublist l1 l3.

Lemma Sublist_snoc :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Sublist l1 l2 → Sublist (snoc x l1) (snoc x l2).

Lemma Sublist_snoc_inv :
```

$\forall$ (A : Type) (x y : A) ($l1$ $l2$: $list$ A),
$Sublist$ ($snoc$ x $l1$) ($snoc$ y $l2$) $\rightarrow$ $Sublist$ $l1$ $l2$ $\land$ $x = y$.

Lemma $Sublist\_app\_l$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$l1 \neq []$ $\rightarrow$ $Sublist$ $l2$ ($l1$ ++ $l2$).

Lemma $Sublist\_app\_l'$:
$\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$ $Sublist$ $l1$ ($l3$ ++ $l2$).

Lemma $Sublist\_app\_r$:
$\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$ $Sublist$ ($l1$ ++ $l3$) ($l2$ ++ $l3$).

Lemma $Sublist\_map$:
$\forall$ (A B : Type) (f : $A \rightarrow B$) ($l1$ $l2$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$ $Sublist$ (map f $l1$) (map f $l2$).

Lemma $Sublist\_join$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ ($list$ A)),
$\neg$ $elem$ [] $l2$ $\rightarrow$ $Sublist$ $l1$ $l2$ $\rightarrow$ $Sublist$ ($join$ $l1$) ($join$ $l2$).

Lemma $Sublist\_replicate$:
$\forall$ (A : Type) (n m : nat) (x : A),
$Sublist$ ($replicate$ n x) ($replicate$ m x) $\leftrightarrow$ $n < m$.

Lemma $Sublist\_replicate'$:
$\forall$ (A : Type) (n m : nat) (x y : A),
$Sublist$ ($replicate$ n x) ($replicate$ m y) $\leftrightarrow$
$n < m \land (n \neq 0 \rightarrow x = y)$.

Lemma $Sublist\_iterate$:
$\forall$ (A : Type) (f : $A \rightarrow A$) (n m : nat) (x : A),
$Sublist$ ($iterate$ f n x) ($iterate$ f m x) $\rightarrow$ $False$.

Lemma $Sublist\_iterate'$:
$\forall$ (A : Type) (f : $A \rightarrow A$) (n m : nat) (x y : A),
$Sublist$ ($iterate$ f n x) ($iterate$ f m y) $\rightarrow$ $False$.

Lemma $Sublist\_tail$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$
$\forall$ $t1$ $t2$: $list$ A, $tail$ $l1$ = $Some$ $t1$ $\rightarrow$ $tail$ $l2$ = $Some$ $t2$ $\rightarrow$
$Sublist$ $t1$ $t2$.

Lemma $Sublist\_last$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$ $l1$ = [] $\lor$ $last$ $l1$ = $last$ $l2$.

```
(*  TODO: insert, remove, take *)
```

```
Lemma Sublist_spec :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 ↔
    ∃ n : nat,
      n < length l2 ∧ l1 = drop (S n) l2.

Lemma Sublist_drop_r :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ n : nat, Sublist (drop n l1) l2.

Lemma Sublist_drop :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ n : nat,
      n < length l2 → Sublist (drop n l1) (drop n l2).

Lemma Sublist_replace :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ (l1' l2' : list A) (n : nat) (x : A),
      replace l1 n x = Some l1' → replace l2 (n + length l1) x = Some l2' →
        Sublist l1' l2'.

Lemma Sublist_zip :
  ∃ (A B : Type) (la1 la2 : list A) (lb1 lb2 : list A),
    Sublist la1 la2 ∧ Sublist lb1 lb2 ∧
      ¬ Sublist (zip la1 lb1) (zip la2 lb2).

(* TODO: zipWith, unzip, unzipWith *)

Lemma Sublist_any_false :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 → any p l2 = false → any p l1 = false.

Lemma Sublist_any_true :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 → any p l1 = true → any p l2 = true.

(* TODO: Sublist_all *)

Lemma Sublist_findLast :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 → ∀ x : A,
      findLast p l1 = Some x → findLast p l2 = Some x.

Lemma Sublist_removeLast :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 →
      match removeLast p l1, removeLast p l2 with
          | None, None ⇒ True
          | None, Some (x, l2') ⇒ l1 = l2' ∨ Sublist l1 l2'
          | x, None ⇒ False
```

```
      | Some (x, l1'), Some (y, l2') ⇒ x = y ∧ Sublist l1' l2'
    end.

Lemma Sublist_findIndex :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 → ∀ n : nat,
      findIndex p l1 = Some n → ∃ m : nat,
        findIndex p l2 = Some m.

Lemma Sublist_filter :
  ∃ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 ∧ ¬ Sublist (filter p l1) (filter p l2).

Lemma Sublist_findIndices :
  ∃ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 ∧ ¬ Sublist (findIndices p l1) (findIndices p l2).

Lemma Sublist_takeWhile :
  ∃ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 ∧ ¬ Sublist (takeWhile p l1) (takeWhile p l2).

Lemma Sublist_dropWhile :
  ∃ (A : Type) (p : A → bool) (l1 l2 : list A),
    Sublist l1 l2 ∧ ¬ Sublist (dropWhile p l1) (dropWhile p l2).

Lemma Sublist_pmap :
  ∃ (A B : Type) (f : A → option B) (l1 l2 : list A),
    Sublist l1 l2 ∧ ¬ Sublist (pmap f l1) (pmap f l2).

Lemma Sublist_intersperse :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Sublist l1 l2 → Sublist (intersperse x l1) (intersperse x l2).

Lemma Sublist_elem :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ x : A, elem x l1 → elem x l2.

Lemma Sublist_In :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ x : A, In x l1 → In x l2.

Lemma Sublist_NoDup :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → NoDup l2 → NoDup l1.

Lemma Sublist_Dup :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → Dup l1 → Dup l2.

Lemma Sublist_Rep :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
```

```
    Sublist l1 l2 → ∀ n : nat, Rep x n l1 → Rep x n l2.
```

```
Lemma Sublist_Exists :
  ∀ (A : Type) (P : A → Prop) (l1 l2 : list A),
    Sublist l1 l2 → Exists P l1 → Exists P l2.
```

```
Lemma Sublist_Forall :
  ∀ (A : Type) (P : A → Prop) (l1 l2 : list A),
    Sublist l1 l2 → Forall P l2 → Forall P l1.
```

```
Lemma Sublist_AtLeast :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ (P : A → Prop) (n : nat),
      AtLeast P n l1 → AtLeast P n l2.
```

```
Lemma Sublist_AtMost :
  ∀ (A : Type) (l1 l2 : list A),
    Sublist l1 l2 → ∀ (P : A → Prop) (n : nat),
      AtMost P n l2 → AtMost P n l1.
```

10.6.2 Prefiksy

```
Inductive Prefix {A : Type} : list A → list A → Prop :=
    | Prefix_nil : ∀ l : list A, Prefix [] l
    | Prefix_cons :
        ∀ (x : A) (l1 l2 : list A),
          Prefix l1 l2 → Prefix (x :: l1) (x :: l2).
```

```
Lemma Prefix_spec :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 ↔ ∃ l3 : list A, l2 = l1 ++ l3.
```

```
Lemma Prefix_refl :
  ∀ (A : Type) (l : list A), Prefix l l.
```

```
Lemma Prefix_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    Prefix l1 l2 → Prefix l2 l3 → Prefix l1 l3.
```

```
Lemma Prefix_wasym :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 → Prefix l2 l1 → l1 = l2.
```

```
(* TODO: null *)
```

```
Lemma Prefix_length :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 → length l1 ≤ length l2.
```

```
Lemma Prefix_snoc :
```

$\exists$ (A : Type) ($l1$ $l2$: $list$ A),
  $Prefix$ $l1$ $l2$ $\land$ $\exists$ x : A, $\neg$ $Prefix$ ($snoc$ x $l1$) ($snoc$ x $l2$).

Lemma $Prefix\_app$:
  $\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
    $Prefix$ $l1$ $l2$ $\to$ $Prefix$ ($l3$ ++ $l1$) ($l3$ ++ $l2$).

Lemma $Prefix\_app\_r$:
  $\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
    $Prefix$ $l1$ $l2$ $\to$ $Prefix$ $l1$ ($l2$ ++ $l3$).

Lemma $Prefix\_rev\_l$:
  $\forall$ (A : Type) (l : $list$ A),
    $Prefix$ (rev l) l $\to$ l = rev l.

Lemma $Prefix\_rev\_r$:
  $\forall$ (A : Type) (l : $list$ A),
    $Prefix$ l (rev l) $\to$ l = rev l.

Lemma $Prefix\_map$:
  $\forall$ (A B : Type) (f : $A \to B$) ($l1$ $l2$: $list$ A),
    $Prefix$ $l1$ $l2$ $\to$ $Prefix$ (map f $l1$) (map f $l2$).

Lemma $Prefix\_join$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ ($list$ A)),
    $Prefix$ $l1$ $l2$ $\to$ $Prefix$ ($join$ $l1$) ($join$ $l2$).

Lemma $Prefix\_replicate$:
  $\forall$ (A : Type) (n m : nat) (x : A),
    $Prefix$ ($replicate$ n x) ($replicate$ m x) $\leftrightarrow$ $n \le m$.

Lemma $Prefix\_replicate\_inv\_l$:
  $\forall$ (A : Type) (l : $list$ A) (n : nat) (x : A),
    $Prefix$ l ($replicate$ n x) $\to$
      $\exists$ m : nat, $m \le n \land l$ = $replicate$ m x.

Lemma $Prefix\_replicate\_inv\_r$:
  $\exists$ (A : Type) (l : $list$ A) (n : nat) (x : A),
    $Prefix$ ($replicate$ n x) l $\land$
      $\neg$ $\exists$ m : nat, $n \le m \land l$ = $replicate$ m x.

Lemma $Prefix\_replicate'$:
  $\forall$ (A : Type) (n : nat) (x y : A),
    $Prefix$ ($replicate$ n x) ($replicate$ n y) $\leftrightarrow$ $n = 0 \lor x = y$.

Lemma $Prefix\_replicate''$:
  $\forall$ (A : Type) (n m : nat) (x y : A),
    $Prefix$ ($replicate$ n x) ($replicate$ m y) $\leftrightarrow$
    $n = 0 \lor n \le m \land x = y$.

Lemma $Prefix\_iterate$:

```
  ∀ (A : Type) (f : A → A) (n m : nat) (x : A),
    Prefix (iterate f n x) (iterate f m x) ↔ n ≤ m.

Lemma Prefix_insert :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 → ∀ (n : nat) (x : A),
      n ≤ length l1 → Prefix (insert l1 n x) (insert l2 n x).

Lemma Prefix_replace :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 → ∀ (n : nat) (x : A),
      match replace l1 n x, replace l2 n x with
          | Some l1', Some l2' ⇒ Prefix l1' l2'
          | _, _ ⇒ True
      end.

Lemma insert_length_ge :
  ∀ (A : Type) (l : list A) (n : nat) (x : A),
    length l ≤ n → insert l n x = snoc x l.

Lemma Prefix_insert' :
  ∃ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 ∧
    ∃ (n : nat) (x : A),
      length l1 < n ∧ ¬ Prefix (insert l1 n x) (insert l2 n x).

Lemma Prefix_take_l :
  ∀ (A : Type) (l : list A) (n : nat), Prefix (take n l) l.

Lemma Prefix_take :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 → ∀ n : nat, Prefix (take n l1) (take n l2).

Lemma Prefix_drop :
  ∀ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 → ∀ n : nat, Prefix (drop n l1) (drop n l2).

Lemma Prefix_zip :
  ∀ (A B : Type) (la1 la2 : list A) (lb1 lb2 : list B),
    Prefix la1 la2 → Prefix lb1 lb2 →
      Prefix (zip la1 lb1) (zip la2 lb2).

(* TODO: unzip, zipWith, unzipWith *)

Lemma Prefix_any_false :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Prefix l1 l2 → any p l2 = false → any p l1 = false.

Lemma Prefix_any :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
```

Prefix l1 l2 → *any p l1* = *true* → *any p l2* = *true*.

`Lemma` *Prefix_all_false* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *all p l1* = *false* → *all p l2* = *false*.

`Lemma` *Prefix_all_true* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *all p l2* = *true* → *all p l1* = *true*.

`Lemma` *Prefix_find_None* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *find p l2* = *None* → *find p l1* = *None*.

`Lemma` *Prefix_find_Some* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → ∀ *x* : *A*,
find p l1 = *Some x* → *find p l2* = *Some x*.

`Lemma` *Prefix_findIndex_None* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *findIndex p l2* = *None* → *findIndex p l1* = *None*.

`Lemma` *Prefix_findIndex_Some* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → ∀ *i* : *nat*,
findIndex p l1 = *Some i* → *findIndex p l2* = *Some i*.

`Lemma` *Prefix_count* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *count p l1* ≤ *count p l2*.

`Lemma` *Prefix_filter* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *Prefix* (*filter p l1*) (*filter p l2*).

`Lemma` *Prefix_findIndices* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *Prefix* (*findIndices p l1*) (*findIndices p l2*).

`Lemma` *Prefix_takeWhile* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l* : *list A*),
Prefix (*takeWhile p l*) *l*.

`Lemma` *Prefix_takeWhile_pres* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *Prefix* (*takeWhile p l1*) (*takeWhile p l2*).

`Lemma` *Prefix_dropWhile* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Prefix l1 l2 → *Prefix* (*dropWhile p l1*) (*dropWhile p l2*).

```
(* TODO: findLast, removeFirst i removeLast *)
```

Lemma *Prefix_pmap* :
$\forall$ $(A\ B : \texttt{Type})$ $(f : A \to option\ B)$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to Prefix\ (pmap\ f\ l1)\ (pmap\ f\ l2)$.

Lemma *Prefix_intersperse* :
$\forall$ $(A : \texttt{Type})$ $(x : A)$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to Prefix\ (intersperse\ x\ l1)\ (intersperse\ x\ l2)$.

```
(* TODO: groupBy *)
```

Lemma *Prefix_elem* :
$\forall$ $(A : \texttt{Type})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to \forall\ x : A, elem\ x\ l1 \to elem\ x\ l2$.

Lemma *Prefix_elem_conv* :
$\forall$ $(A : \texttt{Type})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to \forall\ x : A, \neg\ elem\ x\ l2 \to \neg\ elem\ x\ l1$.

```
(* TODO: In *)
```

Lemma *Prefix_NoDup* :
$\forall$ $(A : \texttt{Type})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to NoDup\ l2 \to NoDup\ l1$.

Lemma *Prefix_Dup* :
$\forall$ $(A : \texttt{Type})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to Dup\ l1 \to Dup\ l2$.

```
(* TODO: Rep *)
```

Lemma *Prefix_Exists* :
$\forall$ $(A : \texttt{Type})$ $(P : A \to \texttt{Prop})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to Exists\ P\ l1 \to Exists\ P\ l2$.

Lemma *Prefix_Forall* :
$\forall$ $(A : \texttt{Type})$ $(P : A \to \texttt{Prop})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to Forall\ P\ l2 \to Forall\ P\ l1$.

Lemma *Prefix_AtLeast* :
$\forall$ $(A : \texttt{Type})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to \forall\ (P : A \to \texttt{Prop})\ (n : nat)$,
$AtLeast\ P\ n\ l1 \to AtLeast\ P\ n\ l2$.

Lemma *Prefix_AtMost* :
$\forall$ $(A : \texttt{Type})$ $(l1\ l2 : list\ A)$,
$Prefix\ l1\ l2 \to \forall\ (P : A \to \texttt{Prop})\ (n : nat)$,
$AtMost\ P\ n\ l2 \to AtMost\ P\ n\ l1$.

```
(* TODO: Exactly - raczej nic z tego *)
```

Lemma *Sublist_Prefix* :

$\exists$ (A : Type) ($l1$ $l2$: $list$ A),
$Sublist$ $l1$ $l2$ $\land$ $\neg$ $Prefix$ $l1$ $l2$.

Lemma $Prefix\_spec'$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Prefix$ $l1$ $l2$ $\leftrightarrow$ $\exists$ n : nat, $l1$ = $take$ n $l2$.

10.6.3 Sufiksy

Inductive $Suffix$ {A : Type} : $list$ A $\to$ $list$ A $\to$ Prop :=
| $Suffix\_refl$:
$\forall$ l : $list$ A, $Suffix$ l l
| $Suffix\_cons$:
$\forall$ (x : A) ($l1$ $l2$: $list$ A),
$Suffix$ $l1$ $l2$ $\to$ $Suffix$ $l1$ (x :: $l2$).

Lemma $Suffix\_spec$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Suffix$ $l1$ $l2$ $\leftrightarrow$ $\exists$ $l3$: $list$ A, $l2$ = $l3$ ++ $l1$.

Lemma $Suffix\_cons\_inv$:
$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Suffix$ (x :: $l1$) $l2$ $\to$ $Suffix$ $l1$ $l2$.

Lemma $Suffix\_trans$:
$\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
$Suffix$ $l1$ $l2$ $\to$ $Suffix$ $l2$ $l3$ $\to$ $Suffix$ $l1$ $l3$.

Lemma $Suffix\_wasym$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Suffix$ $l1$ $l2$ $\to$ $Suffix$ $l2$ $l1$ $\to$ $l1$ = $l2$.

Lemma $Suffix\_nil\_l$:
$\forall$ (A : Type) (l : $list$ A), $Suffix$ [] l.

Lemma $Suffix\_snoc$:
$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Suffix$ $l1$ $l2$ $\to$ $Suffix$ ($snoc$ x $l1$) ($snoc$ x $l2$).

Lemma $Suffix\_Sublist$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Suffix$ $l1$ $l2$ $\leftrightarrow$ $Sublist$ $l1$ $l2$ $\lor$ $l1$ = $l2$.

Lemma $Prefix\_Suffix$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Prefix$ $l1$ $l2$ $\leftrightarrow$ $Suffix$ (rev $l1$) (rev $l2$).

10.6.4 Listy jako ciągi

```
Inductive Subseq {A : Type} : list A → list A → Prop :=
    | Subseq_nil :
        ∀ l : list A, Subseq [] l
    | Subseq_cons :
        ∀ (x : A) (l1 l2 : list A),
          Subseq l1 l2 → Subseq (x :: l1) (x :: l2)
    | Subseq_skip :
        ∀ (x : A) (l1 l2 : list A),
          Subseq l1 l2 → Subseq l1 (x :: l2).

Lemma Subseq_refl :
  ∀ (A : Type) (l : list A), Subseq l l.

Lemma Subseq_cons_inv :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Subseq (x :: l1) (x :: l2) → Subseq l1 l2.

Lemma Subseq_cons_inv_l :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Subseq (x :: l1) l2 → Subseq l1 l2.

Lemma Subseq_isEmpty_l :
  ∀ (A : Type) (l1 l2 : list A),
    isEmpty l1 = true → Subseq l1 l2.

Lemma Subseq_nil_r :
  ∀ (A : Type) (l : list A),
    Subseq l [] → l = [].

Lemma Subseq_length :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → length l1 ≤ length l2.

Lemma Subseq_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    Subseq l1 l2 → Subseq l2 l3 → Subseq l1 l3.

Lemma Subseq_cons_l_app :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Subseq (x :: l1) l2 →
      ∃ l21 l22 : list A, l2 = l21 ++ x :: l22 ∧ Subseq l1 l22.

Lemma Subseq_wasym :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → Subseq l2 l1 → l1 = l2.

Lemma Subseq_snoc :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
```

$\quad$ *Subseq l1 l2* → *Subseq l1* (*snoc x l2*).

Lemma *Subseq_snoc'* :
∀ (*A* : Type) (*x* : *A*) (*l1 l2* : *list A*),
Subseq l1 l2 → *Subseq* (*snoc x l1*) (*snoc x l2*).

Lemma *Subseq_app_l* :
∀ (*A* : Type) (*l1 l2 l3* : *list A*),
Subseq l1 l2 → *Subseq l1* (*l3* ++ *l2*).

Lemma *Subseq_app_r* :
∀ (*A* : Type) (*l1 l2 l3* : *list A*),
Subseq l1 l2 → *Subseq l1* (*l2* ++ *l3*).

Lemma *Subseq_app_l'* :
∀ (*A* : Type) (*l1 l2 l3* : *list A*),
Subseq l1 l2 → *Subseq* (*l3* ++ *l1*) (*l3* ++ *l2*).

Lemma *Subseq_app_r'* :
∀ (*A* : Type) (*l1 l2 l3* : *list A*),
Subseq l1 l2 → *Subseq* (*l1* ++ *l3*) (*l2* ++ *l3*).

Lemma *Subseq_app_not_comm* :
∃ (*A* : Type) (*l1 l2 l3* : *list A*),
Subseq l1 (*l2* ++ *l3*) ∧ ¬ *Subseq l1* (*l3* ++ *l2*).

Lemma *Subseq_map* :
∀ (*A B* : Type) (*f* : *A* → *B*) (*l1 l2* : *list A*),
Subseq l1 l2 → *Subseq* (*map f l1*) (*map f l2*).

Lemma *Subseq_join* :
∀ (*A* : Type) (*l1 l2* : *list* (*list A*)),
Subseq l1 l2 → *Subseq* (*join l1*) (*join l2*).

Lemma *Subseq_replicate* :
∀ (*A* : Type) (*n m* : *nat*) (*x* : *A*),
Subseq (*replicate n x*) (*replicate m x*) ↔ *n* ≤ *m*.

Lemma *Subseq_iterate* :
∀ (*A* : Type) (*f* : *A* → *A*) (*n m* : *nat*) (*x* : *A*),
Subseq (*iterate f n x*) (*iterate f m x*) ↔ *n* ≤ *m*.

Lemma *Subseq_tail* :
∀ (*A* : Type) (*l1 l2* : *list A*),
Subseq l1 l2 → ∀ *l1' l2'* : *list A*,
tail l1 = *Some l1'* → *tail l2* = *Some l2'* → *Subseq l1' l2'*.

Lemma *Subseq_uncons* :
∀ (*A* : Type) (*l1 l2* : *list A*),
Subseq l1 l2 → ∀ (*h1 h2* : *A*) (*t1 t2* : *list A*),
uncons l1 = *Some* (*h1*, *t1*) → *uncons l2* = *Some* (*h2*, *t2*) →

```
        h1 = h2 ∨ Subseq l1 t2.

Lemma Subseq_init :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → ∀ l1' l2' : list A,
      init l1 = Some l1' → init l2 = Some l2' → Subseq l1' l2'.

Lemma Subseq_unsnoc :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → ∀ (h1 h2 : A) (t1 t2 : list A),
      unsnoc l1 = Some (h1, t1) → unsnoc l2 = Some (h2, t2) →
        h1 = h2 ∨ Subseq l1 t2.

Lemma Subseq_nth :
  ∀ (A : Type) (l1 l2 : list A) (n : nat) (x : A),
    Subseq l1 l2 → nth n l1 = Some x →
      ∃ m : nat, nth m l2 = Some x ∧ n ≤ m.

Lemma Subseq_insert :
  ∀ (A : Type) (l1 l2 : list A) (n : nat) (x : A),
    Subseq l1 l2 → Subseq l1 (insert l2 n x).

Lemma Subseq_replace :
  ∀ (A : Type) (l1 l1' l2 : list A) (n : nat) (x : A),
    Subseq l1 l2 → replace l1 n x = Some l1' →
      ∃ (m : nat) (l2' : list A),
        replace l2 m x = Some l2' ∧ Subseq l1' l2'.

Lemma Subseq_remove' :
  ∀ (A : Type) (l1 l2 : list A) (n : nat),
    Subseq l1 l2 → Subseq (remove' n l1) l2.

Lemma Subseq_take :
  ∀ (A : Type) (l : list A) (n : nat),
    Subseq (take n l) l.

Lemma Subseq_drop :
  ∀ (A : Type) (l : list A) (n : nat),
    Subseq (drop n l) l.

Lemma Subseq_zip :
  ∃ (A B : Type) (la1 la2 : list A) (lb1 lb2 : list B),
    Subseq la1 la2 ∧ Subseq lb1 lb2 ∧
      ¬ Subseq (zip la1 lb1) (zip la2 lb2).

Lemma Subseq_all :
  ∀ (A : Type) (p : A → bool) (l1 l2 : list A),
    Subseq l1 l2 → bool_le (all p l2) (all p l1).

Lemma Subseq_any :
```

∀ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *bool_le* (*any* *p* *l1*) (*any* *p* *l2*).

Lemma *Subseq_all'* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *all* *p* *l2* = *true* → *all* *p* *l1* = *true*.

Lemma *Subseq_any'* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *any* *p* *l2* = *false* → *any* *p* *l1* = *false*.

Lemma *Subseq_findIndex* :
∃ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*) (*n* *m* : *nat*),
Subseq *l1* *l2* ∧ *findIndex* *p* *l1* = *Some* *n* ∧
findIndex *p* *l2* = *Some* *m* ∧ *m* < *n*.

Lemma *Subseq_count* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *count* *p* *l1* ≤ *count* *p* *l2*.

Lemma *Subseq_filter* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *Subseq* (*filter* *p* *l1*) (*filter* *p* *l2*).

Lemma *Subseq_filter'* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list* *A*),
Subseq (*filter* *p* *l*) *l*.

Lemma *Subseq_takeWhile* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list* *A*),
Subseq (*takeWhile* *p* *l*) *l*.

Lemma *Subseq_takeWhile'* :
∃ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* ∧ ¬ *Subseq* (*takeWhile* *p* *l1*) (*takeWhile* *p* *l2*).

Lemma *Subseq_dropWhile* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l* : *list* *A*),
Subseq (*dropWhile* *p* *l*) *l*.

Lemma *Subseq_dropWhile'* :
∀ (*A* : Type) (*p* : *A* → *bool*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *Subseq* (*dropWhile* *p* *l1*) (*dropWhile* *p* *l2*).

Lemma *Subseq_pmap* :
∀ (*A* *B* : Type) (*f* : *A* → *option* *B*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *Subseq* (*pmap* *f* *l1*) (*pmap* *f* *l2*).

Lemma *Subseq_map_pmap* :
∀ (*A* *B* : Type) (*f* : *A* → *option* *B*) (*l1* *l2* : *list* *A*),
Subseq *l1* *l2* → *Subseq* (*map* *Some* (*pmap* *f* *l1*)) (*map* *f* *l2*).

```
Lemma Subseq_intersperse :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    Subseq l1 l2 → Subseq (intersperse x l1) (intersperse x l2).

(* TODO *) Fixpoint intercalate {A : Type} (l : list A) (ll : list (list A)) : list A :=
match l, ll with
    | [], _ ⇒ join ll
    | _, [] ⇒ l
    | h :: t, l :: ls ⇒ h :: l ++ intercalate t ls
end.

Lemma Subseq_spec :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 →
      ∃ ll : list (list A),
        l2 = intercalate l1 ll.

Lemma Subseq_In :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 →
      ∀ x : A, In x l1 → In x l2.

Lemma Subseq_NoDup :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → NoDup l2 → NoDup l1.

Lemma Subseq_Dup :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → Dup l1 → Dup l2.

Lemma Subseq_Rep :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → ∀ (x : A) (n : nat),
      Rep x n l1 → Rep x n l2.

Lemma Subseq_Exists :
  ∀ (A : Type) (P : A → Prop) (l1 l2 : list A),
    Subseq l1 l2 → Exists P l1 → Exists P l2.

Lemma Subseq_Forall :
  ∀ (A : Type) (P : A → Prop) (l1 l2 : list A),
    Subseq l1 l2 → Forall P l2 → Forall P l1.

Lemma Subseq_AtLeast :
  ∀ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 → ∀ (P : A → Prop) (n : nat),
      AtLeast P n l1 → AtLeast P n l2.

Lemma Subseq_AtMost :
  ∀ (A : Type) (l1 l2 : list A),
```

Subseq l1 l2 → ∀ (*P* : *A* → `Prop`) (*n* : *nat*),
AtMost P n l2 → *AtMost P n l1*.

`Lemma` *Sublist_Subseq* :
∀ (*A* : `Type`) (*l1 l2* : *list A*),
Sublist l1 l2 → *Subseq l1 l2*.

`Lemma` *Subseq_Sublist* :
∃ (*A* : `Type`) (*l1 l2* : *list A*),
Subseq l1 l2 ∧ ¬ *Sublist l1 l2*.

`Lemma` *Prefix_Subseq* :
∀ (*A* : `Type`) (*l1 l2* : *list A*),
Prefix l1 l2 → *Subseq l1 l2*.

`Lemma` *Subseq_Prefix* :
∃ (*A* : `Type`) (*l1 l2* : *list A*),
Subseq l1 l2 ∧ ¬ *Prefix l1 l2*.

10.6.5 Zawieranie

`Definition` *Incl* {*A* : `Type`} (*l1 l2* : *list A*) : `Prop` :=
∀ *x* : *A*, *elem x l1* → *elem x l2*.

Przyjrzyjmy się powyższej definicji. Intuicyjnie można ją rozumieć tak, że *Incl l1 l2* zachodzi, gdy każdy element listy *l1* choć raz występuje też na liście *l2*. Udowodnij, że relacja ta ma poniższe właściwości.

`Lemma` *Incl_nil* :
∀ (*A* : `Type`) (*l* : *list A*), *Incl* [] *l*.

`Lemma` *Incl_nil_inv* :
∀ (*A* : `Type`) (*l* : *list A*),
Incl l [] → *l* = [].

`Lemma` *Incl_cons* :
∀ (*A* : `Type`) (*h* : *A*) (*t1 t2* : *list A*),
Incl t1 t2 → *Incl* (*h* :: *t1*) (*h* :: *t2*).

`Lemma` *Incl_cons'* :
∀ (*A* : `Type`) (*h* : *A*) (*t* : *list A*),
Incl t (*h* :: *t*).

`Lemma` *Incl_cons''* :
∀ (*A* : `Type`) (*h* : *A*) (*t l* : *list A*),
Incl l t → *Incl l* (*h* :: *t*).

`Lemma` *Incl_cons_conv* :
∃ (*A* : `Type`) (*x* : *A*) (*l1 l2* : *list A*),
Incl (*x* :: *l1*) (*x* :: *l2*) ∧ ¬ *Incl l1 l2*.

Lemma $Incl\_refl$:
  $\forall$ (A : Type) (l : $list$ A), $Incl$ l l.

Lemma $Incl\_trans$:
  $\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
    $Incl$ $l1$ $l2$ $\rightarrow$ $Incl$ $l2$ $l3$ $\rightarrow$ $Incl$ $l1$ $l3$.

(* TODO *)
Lemma $Incl\_length$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $\neg$ Dup $l1$ $\rightarrow$ $Incl$ $l1$ $l2$ $\rightarrow$ $length$ $l1$ $\leq$ $length$ $l2$.

Lemma $Incl\_snoc$:
  $\forall$ (A : Type) (x : A) (l : $list$ A),
    $Incl$ l ($snoc$ x l).

Lemma $Incl\_singl\_snoc$:
  $\forall$ (A : Type) (x : A) (l : $list$ A),
    $Incl$ [x] ($snoc$ x l).

Lemma $Incl\_snoc\_snoc$:
  $\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $Incl$ $l1$ $l2$ $\rightarrow$ $Incl$ ($snoc$ x $l1$) ($snoc$ x $l2$).

Lemma $Incl\_app\_rl$:
  $\forall$ (A : Type) (l $l1$ $l2$: $list$ A),
    $Incl$ l $l2$ $\rightarrow$ $Incl$ l ($l1$ ++ $l2$).

Lemma $Incl\_app\_rr$:
  $\forall$ (A : Type) (l $l1$ $l2$: $list$ A),
    $Incl$ l $l1$ $\rightarrow$ $Incl$ l ($l1$ ++ $l2$).

Lemma $Incl\_app\_l$:
  $\forall$ (A : Type) ($l1$ $l2$ $l3$: $list$ A),
    $Incl$ ($l1$ ++ $l2$) $l3$ $\leftrightarrow$ $Incl$ $l1$ $l3$ $\wedge$ $Incl$ $l2$ $l3$.

Lemma $Incl\_app\_sym$:
  $\forall$ (A : Type) ($l1$ $l2$ l : $list$ A),
    $Incl$ ($l1$ ++ $l2$) l $\rightarrow$ $Incl$ ($l2$ ++ $l1$) l.

Lemma $Incl\_rev$:
  $\forall$ (A : Type) (l : $list$ A), $Incl$ (rev l) l.

Lemma $Incl\_map$:
  $\forall$ (A B : Type) (f : $A \rightarrow B$) ($l1$ $l2$: $list$ A),
    $Incl$ $l1$ $l2$ $\rightarrow$ $Incl$ (map f $l1$) (map f $l2$).

Lemma $Incl\_join$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ ($list$ A)),
    $Incl$ $l1$ $l2$ $\rightarrow$ $Incl$ ($join$ $l1$) ($join$ $l2$).

Lemma $Incl\_elem\_join$:

```
  ∀ (A : Type) (ll : list (list A)) (l : list A),
    elem l ll → Incl l (join ll).

Lemma Incl_replicate :
  ∀ (A : Type) (n : nat) (x : A) (l : list A),
    elem x l → Incl (replicate n x) l.

Lemma Incl_replicate' :
  ∀ (A : Type) (n m : nat) (x : A),
    n ≠ 0 → Incl (replicate m x) (replicate n x).

Lemma Incl_nth :
  ∀ (A : Type) (l1 l2 : list A),
    Incl l1 l2 ↔
    ∀ (n1 : nat) (x : A), nth n1 l1 = Some x →
      ∃ n2 : nat, nth n2 l2 = Some x.

Lemma Incl_remove :
  ∀ (A : Type) (l : list A) (n : nat),
    match remove n l with
        | None ⇒ True
        | Some (_, l') ⇒ Incl l' l
    end.

Lemma Incl_take :
  ∀ (A : Type) (l : list A) (n : nat),
    Incl (take n l) l.

Lemma Incl_drop :
  ∀ (A : Type) (l : list A) (n : nat),
    Incl (drop n l) l.

Lemma Incl_insert :
  ∀ (A : Type) (l1 l2 : list A) (n m : nat) (x : A),
    Incl l1 l2 → Incl (insert l1 n x) (insert l2 m x).

Lemma Incl_replace :
  ∀ (A : Type) (l1 l1' l2 : list A) (n : nat) (x : A),
    Incl l1 l2 → replace l1 n x = Some l1' →
      ∃ (m : nat) (l2' : list A),
        replace l2 m x = Some l2' ∧ Incl l1' l2'.

Lemma Incl_splitAt :
  ∀ (A : Type) (l : list A) (n : nat),
    match splitAt n l with
        | None ⇒ True
        | Some (l1, _, l2) ⇒ Incl l1 l ∧ Incl l2 l
    end.

Lemma Incl_filter :
```

$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$\quad Incl\ (filter\ p\ l)\ l.$

`Lemma` *Incl_filter_conv* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$\quad Incl\ l\ (filter\ p\ l) \leftrightarrow filter\ p\ l = l.$

`Lemma` *Incl_takeWhile* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$\quad Incl\ (takeWhile\ p\ l)\ l.$

`Lemma` *Incl_takeWhile_conv* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$\quad Incl\ l\ (takeWhile\ p\ l) \leftrightarrow takeWhile\ p\ l = l.$

`Lemma` *Incl_dropWhile* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (l : list\ A),$
$\quad Incl\ (dropWhile\ p\ l)\ l.$

`Lemma` *Incl_span* :
$\forall\ (A : \texttt{Type})\ (p : A \to bool)\ (x : A)\ (l\ b\ e : list\ A),$
$\quad span\ p\ l = Some\ (b, x, e) \to$
$\quad\quad Incl\ b\ l \land elem\ x\ l \land Incl\ e\ l.$

`(* TODO: span i Sublist, palindromy *)`

`Lemma` *Incl_pmap* :
$\forall\ (A\ B : \texttt{Type})\ (f : A \to option\ B)\ (l : list\ A),$
$\quad Incl\ (map\ Some\ (pmap\ f\ l))\ (map\ f\ l).$

`Lemma` *Incl_intersperse* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l1\ l2 : list\ A),$
$\quad Incl\ l1\ (intersperse\ x\ l2) \to Incl\ l1\ (x :: l2).$

`Lemma` *Incl_intersperse_conv* :
$\forall\ (A : \texttt{Type})\ (x : A)\ (l1\ l2 : list\ A),$
$\quad 2 \leq length\ l2 \to Incl\ l1\ (x :: l2) \to Incl\ l1\ (intersperse\ x\ l2).$

`Lemma` *Incl_In* :
$\forall\ (A : \texttt{Type})\ (l1\ l2 : list\ A),$
$\quad Incl\ l1\ l2 \to \forall\ x : A, In\ x\ l1 \to In\ x\ l2.$

`Lemma` *Incl_NoDup* :
$\exists\ (A : \texttt{Type})\ (l1\ l2 : list\ A),$
$\quad Incl\ l1\ l2 \land NoDup\ l2 \land \neg\ NoDup\ l1.$

`Lemma` *Incl_Dup* :
$\exists\ (A : \texttt{Type})\ (l1\ l2 : list\ A),$
$\quad Incl\ l1\ l2 \land Dup\ l1 \land \neg\ Dup\ l2.$

`Lemma` *Incl_Rep* :

$\exists$ (A : Type) (x : A) (n : nat) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ Rep x n $l1$ $\wedge$ $\neg$ Rep x n $l2$.

Lemma *Incl_Exists* :
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\rightarrow \forall$ P : $A \rightarrow$ Prop,
$Exists$ P $l1$ $\rightarrow$ $Exists$ P $l2$.

Lemma *Incl_Forall* :
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\rightarrow \forall$ P : $A \rightarrow$ Prop,
$Forall$ P $l2$ $\rightarrow$ $Forall$ P $l1$.

Lemma *Incl_AtLeast* :
$\exists$ (A : Type) (P : $A \rightarrow$ Prop) (n : nat) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ $AtLeast$ P n $l1$ $\wedge$ $\neg$ $AtLeast$ P n $l2$.

Lemma *Incl_Exactly* :
$\exists$ (A : Type) (P : $A \rightarrow$ Prop) (n : nat) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ $Exactly$ P n $l1$ $\wedge$ $\neg$ $Exactly$ P n $l2$.

Lemma *Incl_AtMost* :
$\exists$ (A : Type) (P : $A \rightarrow$ Prop) (n : nat) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ $AtMost$ P n $l2$ $\wedge$ $\neg$ $AtMost$ P n $l1$.

Lemma *Sublist_Incl* :
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$ $Incl$ $l1$ $l2$.

Lemma *Incl_Sublist* :
$\exists$ (A : Type) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ $\neg$ $Sublist$ $l1$ $l2$.

Lemma *Prefix_Incl* :
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Prefix$ $l1$ $l2$ $\rightarrow$ $Incl$ $l1$ $l2$.

Lemma *Incl_Prefix* :
$\exists$ (A : Type) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ $\neg$ $Prefix$ $l1$ $l2$.

Lemma *Subseq_Incl* :
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Subseq$ $l1$ $l2$ $\rightarrow$ $Incl$ $l1$ $l2$.

Lemma *Incl_Subseq* :
$\exists$ (A : Type) ($l1$ $l2$: $list$ A),
$Incl$ $l1$ $l2$ $\wedge$ $\neg$ $Subseq$ $l1$ $l2$.

10.6.6 Listy jako zbiory

```
Definition SetEquiv {A : Type} (l1 l2 : list A) : Prop :=
  ∀ x : A, elem x l1 ↔ elem x l2.

Lemma SetEquiv_Incl :
  ∀ (A : Type) (l1 l2 : list A),
    SetEquiv l1 l2 ↔ Incl l1 l2 ∧ Incl l2 l1.

Lemma SetEquiv_refl :
  ∀ (A : Type) (l : list A), SetEquiv l l.

Lemma SetEquiv_sym :
  ∀ (A : Type) (l1 l2 : list A),
    SetEquiv l1 l2 ↔ SetEquiv l2 l1.

Lemma SetEquiv_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    SetEquiv l1 l2 → SetEquiv l2 l3 → SetEquiv l1 l3.

Lemma SetEquiv_nil_l :
  ∀ (A : Type) (l : list A),
    SetEquiv [] l → l = [].

Lemma SetEquiv_nil_r :
  ∀ (A : Type) (l : list A),
    SetEquiv l [] → l = [].

Lemma SetEquiv_cons :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    SetEquiv l1 l2 → SetEquiv (x :: l1) (x :: l2).

Lemma SetEquiv_cons_conv :
  ∃ (A : Type) (x : A) (l1 l2 : list A),
    SetEquiv (x :: l1) (x :: l2) ∧ ¬ SetEquiv l1 l2.

Lemma SetEquiv_cons' :
  ∃ (A : Type) (x : A) (l : list A),
    ¬ SetEquiv l (x :: l).

Lemma SetEquiv_snoc_cons :
  ∀ (A : Type) (x : A) (l : list A),
    SetEquiv (snoc x l) (x :: l).

Lemma SetEquiv_snoc :
  ∀ (A : Type) (x : A) (l1 l2 : list A),
    SetEquiv l1 l2 → SetEquiv (snoc x l1) (snoc x l2).

Lemma SetEquiv_app_comm :
  ∀ (A : Type) (l1 l2 : list A),
    SetEquiv (l1 ++ l2) (l2 ++ l1).
```

```
Lemma SetEquiv_app_l :
  ∀ (A : Type) (l1 l2 l : list A),
    SetEquiv l1 l2 → SetEquiv (l ++ l1) (l ++ l2).

Lemma SetEquiv_app_r :
  ∀ (A : Type) (l1 l2 l : list A),
    SetEquiv l1 l2 → SetEquiv (l1 ++ l) (l2 ++ l).

Lemma SetEquiv_rev :
  ∀ (A : Type) (l : list A), SetEquiv (rev l) l.

Lemma SetEquiv_map :
  ∀ (A B : Type) (f : A → B) (l1 l2 : list A),
    SetEquiv l1 l2 → SetEquiv (map f l1) (map f l2).

Lemma SetEquiv_join :
  ∀ (A : Type) (l1 l2 : list (list A)),
    SetEquiv l1 l2 → SetEquiv (join l1) (join l2).

Lemma SetEquiv_replicate :
  ∀ (A : Type) (n : nat) (x : A),
    SetEquiv (replicate n x) (if isZero n then [] else [x]).

Lemma SetEquiv_replicate' :
  ∀ (A : Type) (n m : nat) (x : A),
    m ≠ 0 → n ≠ 0 → SetEquiv (replicate m x) (replicate n x).

(* TODO *) Lemma SetEquiv_nth :
  ∀ (A : Type) (l1 l2 : list A),
    SetEquiv l1 l2 ↔
    (∀ n : nat, ∃ m : nat, nth n l1 = nth m l2) ∧
    (∀ n : nat, ∃ m : nat, nth m l1 = nth n l2).

Lemma SetEquiv_take :
  ∀ (A : Type) (l : list A) (n : nat),
    SetEquiv (take n l) l ↔ Incl (drop n l) (take n l).

Lemma SetEquiv_drop :
  ∀ (A : Type) (n : nat) (l : list A),
    SetEquiv (drop n l) l ↔ Incl (take n l) (drop n l).

Lemma SetEquiv_filter :
  ∀ (A : Type) (p : A → bool) (l : list A),
    SetEquiv (filter p l) l ↔ all p l = true.

Lemma SetEquiv_takeWhile :
  ∀ (A : Type) (p : A → bool) (l : list A),
    SetEquiv (takeWhile p l) l ↔ all p l = true.

Lemma SetEquiv_dropWhile :
  ∃ (A : Type) (p : A → bool) (l : list A),
```

```
    SetEquiv (dropWhile p l) l ∧ any p l = true.

Lemma SetEquiv_pmap :
  ∃ (A B : Type) (f : A → option B) (l : list A),
    ¬ SetEquiv (map Some (pmap f l)) (map f l).

Lemma SetEquiv_intersperse :
  ∀ (A : Type) (x : A) (l : list A),
    2 ≤ length l → SetEquiv (intersperse x l) (x :: l).

Lemma SetEquiv_intersperse_conv :
  ∀ (A : Type) (x : A) (l : list A),
    SetEquiv (intersperse x l) (x :: l) →
      elem x l ∨ 2 ≤ length l.

Lemma SetEquiv_singl :
  ∀ (A : Type) (l : list A) (x : A),
    SetEquiv [x] l → ∀ y : A, elem y l → y = x.

Lemma SetEquiv_pres_intersperse :
  ∃ (A : Type) (x : A) (l1 l2 : list A),
    SetEquiv l1 l2 ∧ ¬ SetEquiv (intersperse x l1) (intersperse x l2).
```

10.6.7 Listy jako multizbiory

```
Require Export Coq.Classes.SetoidClass.
Require Import Coq.Classes.RelationClasses.

Inductive Permutation {A : Type} : list A → list A → Prop :=
    | perm_nil : Permutation [] []
    | perm_skip : ∀ (x : A) (l l' : list A),
        Permutation l l' → Permutation (x :: l) (x :: l')
    | perm_swap : ∀ (x y : A) (l : list A),
        Permutation (y :: x :: l) (x :: y :: l)
    | perm_trans : ∀ l l' l'' : list A,
        Permutation l l' → Permutation l' l'' → Permutation l l''.

Hint Constructors Permutation.

Lemma Permutation_refl :
  ∀ (A : Type) (l : list A),
    Permutation l l.

Lemma Permutation_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    Permutation l1 l2 → Permutation l2 l3 → Permutation l1 l3.

Lemma Permutation_sym :
  ∀ (A : Type) (l1 l2 : list A),
```

```
    Permutation l1 l2 → Permutation l2 l1.
Instance Permutation_Equivalence:
  ∀ A : Type, RelationClasses.Equivalence (Permutation (A := A)).
Lemma Permutation_isEmpty :
  ∀ (A : Type) (l1 l2 : list A),
    Permutation l1 l2 → isEmpty l1 = isEmpty l2.
Lemma Permutation_nil_l :
  ∀ (A : Type) (l : list A),
    Permutation [] l → l = [].
Lemma Permutation_nil_r :
  ∀ (A : Type) (l : list A),
    Permutation l [] → l = [].
Lemma Permutation_nil_cons_l :
  ∀ (A : Type) (l : list A) (x : A),
    ¬ Permutation [] (x :: l).
Lemma Permutation_nil_cons_r :
  ∀ (A : Type) (l : list A) (x : A),
    ¬ Permutation (x :: l) [].
Lemma Permutation_nil_app_cons_l :
  ∀ (A : Type) (l l' : list A) (x : A),
    ¬ Permutation [] (l ++ x :: l').
Instance Permutation_cons :
  ∀ A : Type,
    Proper (eq ==> @Permutation A ==> @Permutation A) (@cons A).
Lemma Permutation_ind' :
  ∀ (A : Type) (P : list A → list A → Prop),
    P [] [] →
    (∀ x l l', Permutation l l' → P l l' → P (x :: l) (x :: l')) →
    (∀ x y l l', Permutation l l' → P l l' →
      P (y :: x :: l) (x :: y :: l')) →
    (∀ l l' l'', Permutation l l' → P l l' → Permutation l' l'' →
      P l' l'' → P l l'') →
    ∀ l l', Permutation l l' → P l l'.
Inductive Elem {A : Type} (x : A) : list A → list A → Prop :=
    | es_here :
        ∀ l : list A, Elem x l (x :: l)
    | es_there :
        ∀ (y : A) (l1 l2 : list A),
          Elem x l1 l2 → Elem x (y :: l1) (y :: l2).
Lemma Elem_spec :
```

$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Elem$ x $l1$ $l2$ $\rightarrow \exists$ $l21$ $l22$: $list$ A,
$l2 = l21$ ++ x :: $l22 \land l1 = l21$ ++ $l22$.

Lemma $Permutation\_Elem$:
$\forall$ (A : Type) (x : A) (l l' : $list$ A),
$Elem$ x l l' $\rightarrow$ $Permutation$ l' (x :: l).

Lemma $Elem\_Permutation$:
$\forall$ (A : Type) (x : A) ($l1$ $l1'$: $list$ A),
$Elem$ x $l1$ $l1'$ $\rightarrow \forall$ $l2'$: $list$ A,
$Permutation$ $l1'$ $l2'$ $\rightarrow \exists$ $l2$: $list$ A, $Elem$ x $l2$ $l2'$.

Lemma $Permutation\_Elems$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Permutation$ $l1$ $l2$ $\rightarrow \forall$ (x : A) ($l1'$ $l2'$: $list$ A),
$Elem$ x $l1'$ $l1$ $\rightarrow$ $Elem$ x $l2'$ $l2$ $\rightarrow$
$Permutation$ $l1'$ $l2'$.

Lemma $Permutation\_cons\_inv$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A) (x : A),
$Permutation$ (x :: $l1$) (x :: $l2$) $\rightarrow$ $Permutation$ $l1$ $l2$.

Lemma $Permutation\_length$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Permutation$ $l1$ $l2$ $\rightarrow$ $length$ $l1$ = $length$ $l2$.

Lemma $Permutation\_length'$:
$\forall$ A : Type,
$Proper$ (@$Permutation$ A ==> eq) (@$length$ A).

Lemma $Permutation\_length\_1$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$length$ $l1$ = 1 $\rightarrow$ $Permutation$ $l1$ $l2$ $\rightarrow$ $l1$ = $l2$.

Lemma $Permutation\_singl$:
$\forall$ (A : Type) (a b : A),
$Permutation$ [a] [b] $\rightarrow$ a = b.

Lemma $Permutation\_length\_1\_inv$:
$\forall$ (A : Type) (x : A) (l : $list$ A),
$Permutation$ [x] l $\rightarrow$ l = [x].

Lemma $Permutation\_snoc$:
$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Permutation$ $l1$ $l2$ $\rightarrow$ $Permutation$ ($snoc$ x $l1$) ($snoc$ x $l2$).

Lemma $Permutation\_cons\_snoc$:
$\forall$ (A : Type) (l : $list$ A) (x : A),
$Permutation$ (x :: l) ($snoc$ x l).

Lemma *Permutation_cons_snoc'* :
  ∀ (*A* : Type) (*l* : *list A*) (*x* : *A*),
    Permutation (*x* :: *l*) (*l* ++ [*x*]).

Lemma *Permutation_app_l* :
  ∀ (*A* : Type) (*l1 l2 l3* : *list A*),
    Permutation l1 l2 → *Permutation* (*l3* ++ *l1*) (*l3* ++ *l2*).

Lemma *Permutation_app_r* :
  ∀ (*A* : Type) (*l1 l2 l3* : *list A*),
    Permutation l1 l2 → *Permutation* (*l1* ++ *l3*) (*l2* ++ *l3*).

Lemma *Permutation_app* :
  ∀ (*A* : Type) (*l1 l1' l2 l2'* : *list A*),
    Permutation l1 l1' → *Permutation l2 l2'* →
      Permutation (*l1* ++ *l2*) (*l1'* ++ *l2'*).

Instance *Permutation_app'*:
  ∀ *A* : Type,
    Proper (@*Permutation A* ==> @*Permutation A* ==> @*Permutation A*) (@*app A*).

Lemma *Permutation_add_inside* :
  ∀ (*A* : Type) (*x* : *A*) (*l1 l2 l1' l2'* : *list A*),
    Permutation l1 l1' → *Permutation l2 l2'* →
      Permutation (*l1* ++ *x* :: *l2*) (*l1'* ++ *x* :: *l2'*).

Lemma *Permutation_cons_middle* :
  ∀ (*A* : Type) (*l1 l2 l3* : *list A*) (*x* : *A*),
    Permutation l1 (*l2* ++ *l3*) → *Permutation* (*x* :: *l1*) (*l2* ++ *x* :: *l3*).

Lemma *Permutation_middle* :
  ∀ (*A* : Type) (*l1 l2* : *list A*) (*x* : *A*),
    Permutation (*l1* ++ *x* :: *l2*) (*x* :: (*l1* ++ *l2*)).

Lemma *Permutation_app_comm* :
  ∀ (*A* : Type) (*l1 l2* : *list A*),
    Permutation (*l1* ++ *l2*) (*l2* ++ *l1*).

Lemma *Permutation_app_inv_l* :
  ∀ (*A* : Type) (*l1 l2 l3* : *list A*),
    Permutation (*l1* ++ *l2*) (*l1* ++ *l3*) → *Permutation l2 l3*.

Lemma *Permutation_app_inv_r* :
  ∀ (*A* : Type) (*l1 l2 l3* : *list A*),
    Permutation (*l1* ++ *l3*) (*l2* ++ *l3*) → *Permutation l1 l2*.

Lemma *Permutation_rev* :
  ∀ (*A* : Type) (*l* : *list A*),
    Permutation (*rev l*) *l*.

Instance *Permutation_rev'* :

∀ A : Type,
  Proper (@*Permutation* A ==> @*Permutation* A) (@*rev* A).

Lemma *Permutation_map*:
∀ (A B : Type) (f : $A \to B$) (*l1* *l2* : *list* A),
  Permutation *l1* *l2* → *Permutation* (*map* f *l1*) (*map* f *l2*).

Lemma *Permutation_map'*:
∀ (A B : Type) (f : $A \to B$),
  Morphisms.Proper
    (*Morphisms.respectful* (*Permutation* (A:=A)) (*Permutation* (A:=B)))
    (*map* f).

Lemma *Permutation_join* :
∀ (A : Type) (*l1* *l2* : *list* (*list* A)),
  Permutation *l1* *l2* → *Permutation* (*join* *l1*) (*join* *l2*).

Lemma *Permutation_join_rev* :
∃ (A : Type) (*l1* *l2* : *list* (*list* A)),
  Permutation (*join* *l1*) (*join* *l2*) ∧ ¬ *Permutation* *l1* *l2*.

Lemma *Permutation_replicate* :
∀ (A : Type) (n m : *nat*) (x : A),
  Permutation (*replicate* n x) (*replicate* m x) ↔ $n = m$.

Lemma *Permutation_In* :
∀ (A : Type) (*l1* *l2* : *list* A),
  Permutation *l1* *l2* → (∀ x : A, *In* x *l1* ↔ *In* x *l2*).

Lemma *Permutation_in* :
∀ (A : Type) (l l' : *list* A) (x : A),
  Permutation l l' → *In* x l → *In* x l'.

Lemma *Permutation_in'* :
∀ A : Type,
  Proper (*eq* ==> @*Permutation* A ==> *iff*) (@*In* A).

Lemma *Permutation_replicate'* :
∀ (A : Type) (n : *nat*) (x y : A),
  Permutation (*replicate* n x) (*replicate* n y) ↔ $n = 0 \lor x = y$.

Lemma *Permutation_iterate* :
∀ (A : Type) (f : $A \to A$) (n m : *nat*) (x : A),
  Permutation (*iterate* f n x) (*iterate* f m x) ↔ $n = m$.

Lemma *Permutation_iterate'* :
∀ (A : Type) (f : $A \to A$) (n : *nat*) (x y : A),
  Permutation (*iterate* f n x) (*iterate* f n y) →
    $n = 0 \lor \exists\, k$: *nat*, $k < n \land$ *iter* f k $y = x$.

Lemma *Permutation_insert* :

$\forall$ (A : Type) (l : *list* A) (n : *nat*) (x : A),
  Permutation (*insert* l n x) (x :: l).

Lemma *Permutation_take* :
  $\exists$ (A : Type) (*l1* *l2* : *list* A),
    Permutation *l1* *l2* $\wedge$
      $\exists$ n : *nat*, $\neg$ *Permutation* (*take* n *l1*) (*take* n *l2*).

Lemma *Permutation_drop* :
  $\exists$ (A : Type) (*l1* *l2* : *list* A),
    Permutation *l1* *l2* $\wedge$
      $\exists$ n : *nat*, $\neg$ *Permutation* (*drop* n *l1*) (*drop* n *l2*).

Lemma *Permutation_length_2_inv*:
  $\forall$ (A : Type) (x y : A) (l : *list* A),
    Permutation [x; y] $l \rightarrow l$ = [x; y] $\vee$ l = [y; x].

Lemma *Permutation_length_2*:
  $\forall$ (A : Type) (*x1* *x2* *y1* *y2* : A),
    Permutation [*x1*; *x2*] [*y1*; *y2*] $\rightarrow$
      x1 = *y1* $\wedge$ *x2* = *y2* $\vee$ *x1* = *y2* $\wedge$ *x2* = *y1*.

Lemma *Permutation_zip* :
  $\exists$ (A B : Type) (*la1* *la2* : *list* A) (*lb1* *lb2* : *list* B),
    Permutation *la1* *la2* $\wedge$ *Permutation* *lb1* *lb2* $\wedge$
      $\neg$ *Permutation* (*zip* *la1* *lb1*) (*zip* *la2* *lb2*).

Lemma *Permutation_zipWith* :
  $\exists$
    (A B C : Type) (f : $A \rightarrow B \rightarrow C$)
    (*la1* *la2* : *list* A) (*lb1* *lb2* : *list* B),
      Permutation *la1* *la2* $\wedge$
      Permutation *lb1* *lb2* $\wedge$
      $\neg$ *Permutation* (*zipWith* f *la1* *lb1*) (*zipWith* f *la2* *lb2*).

Lemma *Permutation_any* :
  $\forall$ (A : Type) (p : $A \rightarrow$ *bool*) (*l1* *l2* : *list* A),
    Permutation *l1* *l2* $\rightarrow$ *any* p *l1* = *any* p *l2*.

Lemma *Permutation_all* :
  $\forall$ (A : Type) (p : $A \rightarrow$ *bool*) (*l1* *l2* : *list* A),
    Permutation *l1* *l2* $\rightarrow$ *all* p *l1* = *all* p *l2*.

Lemma *Permutation_count* :
  $\forall$ (A : Type) (p : $A \rightarrow$ *bool*) (*l1* *l2* : *list* A),
    Permutation *l1* *l2* $\rightarrow$ *count* p *l1* = *count* p *l2*.

Lemma *Permutation_count_replicate_filter* :
  $\forall$ (A : Type) (p : $A \rightarrow$ *bool*) (l b e : *list* A) (x : A),
    span p l = *Some* (b, x, e) $\rightarrow$

Permutation l (replicate (count p l) x ++ b ++ e).

`Lemma` *Permutation_count_conv* :
∀ (*A* : `Type`) (*l1 l2* : *list A*),
(∀ *p* : *A* → *bool*, *count p l1* = *count p l2*) → *Permutation l1 l2*.

`Lemma` *Permutation_filter* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Permutation l1 l2 → *Permutation* (*filter p l1*) (*filter p l2*).

`Lemma` *Permutation_takeWhile* :
∃ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Permutation l1 l2 ∧
¬ *Permutation* (*takeWhile p l1*) (*takeWhile p l2*).

`Lemma` *Permutation_dropWhile* :
∃ (*A* : `Type`) (*p* : *A* → *bool*) (*l1 l2* : *list A*),
Permutation l1 l2 ∧
¬ *Permutation* (*dropWhile p l1*) (*dropWhile p l2*).

`Lemma` *Permutation_span* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l b e* : *list A*) (*x* : *A*),
span p l = *Some* (*b*, *x*, *e*) → *Permutation l* (*b* ++ *x* :: *e*).

`Lemma` *Permutation_removeFirst* :
∀ (*A* : `Type`) (*p* : *A* → *bool*) (*l l'* : *list A*) (*x* : *A*),
removeFirst p l = *Some* (*x*, *l'*) → *Permutation l* (*x* :: *l'*).

`Lemma` *Permutation_intersperse_replicate* :
∀ (*A* : `Type`) (*x* : *A*) (*l* : *list A*),
Permutation (*intersperse x l*) (*replicate* (*length l* - 1) *x* ++ *l*).

`Lemma` *Permutation_intersperse* :
∀ (*A* : `Type`) (*x* : *A*) (*l1 l2* : *list A*),
Permutation (*intersperse x l1*) (*intersperse x l2*) ↔
Permutation l1 l2.

`Lemma` *Permutation_pmap* :
∀ (*A B* : `Type`) (*f* : *A* → *option B*) (*l1 l2* : *list A*),
Permutation l1 l2 → *Permutation* (*pmap f l1*) (*pmap f l2*).

`Lemma` *Permutation_elem* :
∀ (*A* : `Type`) (*l1 l2* : *list A*),
Permutation l1 l2 →
∀ *x* : *A*, *elem x l1* ↔ *elem x l2*.

`Lemma` *Permutation_replicate''* :
∀ (*A* : `Type`) (*n m* : *nat*) (*x y* : *A*),
Permutation (*replicate n x*) (*replicate m y*) ↔
$n = m \land (n \neq 0 \to m \neq 0 \to x = y)$.

Lemma $NoDup\_Permutation$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $NoDup$ $l1$ $\rightarrow$ $NoDup$ $l2$ $\rightarrow$
      ($\forall$ x : A, In x $l1$ $\leftrightarrow$ In x $l2$) $\rightarrow$ $Permutation$ $l1$ $l2$.

Lemma $NoDup\_Permutation\_bis$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $NoDup$ $l1$ $\rightarrow$ $NoDup$ $l2$ $\rightarrow$ $length$ $l2$ $\leq$ $length$ $l1$ $\rightarrow$
      $Incl$ $l1$ $l2$ $\rightarrow$ $Permutation$ $l1$ $l2$.

Lemma $Permutation\_NoDup$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$ $NoDup$ $l1$ $\rightarrow$ $NoDup$ $l2$.

Lemma $Permutation\_NoDup'$:
  $\forall$ A : Type,
    $Morphisms.Proper$
      ($Morphisms.respectful$ ($Permutation$ (A:=A)) iff)
      ($NoDup$ (A:=A)).

Lemma $Permutation\_Dup$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$ Dup $l1$ $\leftrightarrow$ Dup $l2$.

Lemma $Permutation\_Rep$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$
      $\forall$ (x : A) (n : nat), Rep x n $l1$ $\leftrightarrow$ Rep x n $l2$.

Lemma $Permutation\_Exists$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$
      $\forall$ P : A $\rightarrow$ Prop, $Exists$ P $l1$ $\leftrightarrow$ $Exists$ P $l2$.

Lemma $Permutation\_Exists\_conv$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    ($\forall$ P : A $\rightarrow$ Prop, $Exists$ P $l1$ $\leftrightarrow$ $Exists$ P $l2$) $\wedge$
    $\neg$ $Permutation$ $l1$ $l2$.

Lemma $Permutation\_Forall$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$
      $\forall$ P : A $\rightarrow$ Prop, $Forall$ P $l1$ $\leftrightarrow$ $Forall$ P $l2$.

Lemma $Permutation\_AtLeast$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$
      $\forall$ (P : A $\rightarrow$ Prop) (n : nat), $AtLeast$ P n $l1$ $\leftrightarrow$ $AtLeast$ P n $l2$.

Lemma $Permutation\_Exactly$:

$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Permutation$ $l1$ $l2$ $\rightarrow$
    $\forall$ (P : $A \rightarrow$ Prop) (n : nat), $Exactly$ P n $l1$ $\leftrightarrow$ $Exactly$ P n $l2$.

Lemma $Permutation\_AtMost$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$
      $\forall$ (P : $A \rightarrow$ Prop) (n : nat), $AtMost$ P n $l1$ $\leftrightarrow$ $AtMost$ P n $l2$.

Lemma $Permutation\_Sublist$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\land \neg$ $Sublist$ $l1$ $l2$.

Lemma $Sublist\_Permutation$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    $Sublist$ $l1$ $l2$ $\land \neg$ $Permutation$ $l1$ $l2$.

Lemma $Permutation\_Prefix$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\land \neg$ $Prefix$ $l1$ $l2$.

Lemma $Prefix\_Permutation$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    $Prefix$ $l1$ $l2$ $\land \neg$ $Permutation$ $l1$ $l2$.

Lemma $Permutation\_Subseq$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\land \neg$ $Subseq$ $l1$ $l2$.

Lemma $Subseq\_Permutation$:
  $\exists$ (A : Type) ($l1$ $l2$: $list$ A),
    $Subseq$ $l1$ $l2$ $\land \neg$ $Permutation$ $l1$ $l2$.

Lemma $Permutation\_Incl$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$ $Incl$ $l1$ $l2$.

Lemma $Permutation\_SetEquiv$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Permutation$ $l1$ $l2$ $\rightarrow$ $SetEquiv$ $l1$ $l2$.

10.6.8 Listy jako cykle

Inductive $Cycle$ {A : Type} : $list$ A $\rightarrow$ $list$ A $\rightarrow$ Prop :=
  | $Cycle\_refl$: $\forall$ l : $list$ A, $Cycle$ l l
  | $Cycle\_cyc$:
      $\forall$ (x : A) ($l1$ $l2$: $list$ A),
        $Cycle$ $l1$ ($snoc$ x $l2$) $\rightarrow$ $Cycle$ $l1$ (x :: $l2$).

Lemma lt_plus_S :
$\forall$ n m : nat,
$n < m \rightarrow \exists$ k : nat, $m = S$ $(n + k)$.

Lemma $Cycle\_spec$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Cycle$ $l1$ $l2$ $\leftrightarrow$
$\exists$ n : nat, $n \leq length$ $l1$ $\wedge$ $l1$ $=$ $drop$ n $l2$ $++$ $take$ n $l2$.

Lemma $Cycle\_isEmpty$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Cycle$ $l1$ $l2$ $\rightarrow$ $isEmpty$ $l1$ $=$ $isEmpty$ $l2$.

Lemma $Cycle\_nil\_l$:
$\forall$ (A : Type) (l : $list$ A),
$Cycle$ [] $l \rightarrow l =$ [].

Lemma $Cycle\_nil\_r$:
$\forall$ (A : Type) (l : $list$ A),
$Cycle$ l [] $\rightarrow l =$ [].

Lemma $Cycle\_length$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Cycle$ $l1$ $l2$ $\rightarrow$ $length$ $l1$ $=$ $length$ $l2$.

Lemma $Cycle\_cons$:
$\exists$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Cycle$ $l1$ $l2$ $\wedge \neg$ $Cycle$ (x :: $l1$) (x :: $l2$).

Lemma $Cycle\_cons\_inv$:
$\exists$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Cycle$ (x :: $l1$) (x :: $l2$) $\wedge \neg$ $Cycle$ $l1$ $l2$.

Lemma $Cycle\_snoc$:
$\exists$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
$Cycle$ $l1$ $l2$ $\wedge \neg$ $Cycle$ ($snoc$ x $l1$) ($snoc$ x $l2$).

Lemma $Cycle\_sym$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Cycle$ $l1$ $l2$ $\rightarrow$ $Cycle$ $l2$ $l1$.

Lemma $Cycle\_snoc\_cons$:
$\forall$ (A : Type) (x : A) (l : $list$ A),
$Cycle$ ($snoc$ x l) (x :: l).

Lemma $Cycle\_cons\_snoc$:
$\forall$ (A : Type) (x : A) (l : $list$ A),
$Cycle$ (x :: l) ($snoc$ x l).

Lemma $Cycle\_cons\_snoc'$:
$\forall$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),

```
    Cycle l1 (x :: l2) → Cycle l1 (snoc x l2).
Lemma Cycle_trans :
  ∀ (A : Type) (l1 l2 l3 : list A),
    Cycle l1 l2 → Cycle l2 l3 → Cycle l1 l3.
Lemma Cycle_app :
  ∀ (A : Type) (l1 l2 l3 : list A),
    Cycle l1 (l2 ++ l3) → Cycle l1 (l3 ++ l2).
Lemma Cycle_rev :
  ∀ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 → Cycle (rev l1) (rev l2).
Lemma Cycle_map :
  ∀ (A B : Type) (f : A → B) (l1 l2 : list A),
    Cycle l1 l2 → Cycle (map f l1) (map f l2).
Lemma Cycle_join :
  ∀ (A : Type) (l1 l2 : list (list A)),
    Cycle l1 l2 → Cycle (join l1) (join l2).
Lemma Cycle_replicate :
  ∀ (A : Type) (n m : nat) (x : A),
    Cycle (replicate n x) (replicate m x) ↔ n = m.
Lemma Cycle_iterate :
  ∀ (A : Type) (f : A → A) (n m : nat) (x : A),
    Cycle (iterate f n x) (iterate f m x) ↔ n = m.
Lemma Cycle_iterate' :
  ∀ (A : Type) (f : A → A) (n m : nat) (x y : A),
    Cycle (iterate f n x) (iterate f m y) ↔ n = m.
(* TODO: head tail etc *)
Lemma Cycle_nth :
  ∀ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 → ∀ (n : nat) (x : A),
      nth n l1 = Some x → ∃ m : nat, nth m l2 = Some x.
Lemma Cycle_insert :
  ∀ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 → ∀ (n : nat) (x : A),
      ∃ m : nat, Cycle (insert l1 n x) (insert l2 m x) .
Lemma Cycle_zip :
  ∃ (A B : Type) (la1 la2 : list A) (lb1 lb2 : list B),
    Cycle la1 la2 ∧ Cycle lb1 lb2 ∧ ¬ Cycle (zip la1 lb1) (zip la2 lb2).
(* TODO: zipW, unzip, unzipW *)
Lemma Cycle_any :
```

$\forall$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\to$ any p $l1$ = any p $l2$.

Lemma $Cycle\_all$:
  $\forall$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\to$ all p $l1$ = all p $l2$.

Lemma $Cycle\_find$:
  $\exists$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A) (x : A),
    $Cycle$ $l1$ $l2$ $\land$ $find$ p $l1$ = $Some$ x $\land$ $find$ p $l2$ $\neq$ $Some$ x.

```
(*  TODO: findLast, removeFirst, removeLast *)
```

Lemma $Cycle\_findIndex$:
  $\exists$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A) (n : nat),
    $Cycle$ $l1$ $l2$ $\land$ $findIndex$ p $l1$ = $Some$ n $\land$ $findIndex$ p $l2$ $\neq$ $Some$ n.

Lemma $Cycle\_count$:
  $\forall$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\to$ $count$ p $l1$ = $count$ p $l2$.

Lemma $Cycle\_filter$:
  $\forall$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\to$ $Cycle$ ($filter$ p $l1$) ($filter$ p $l2$).

```
(*  TODO: findIndices *)
```

Lemma $Cycle\_takeWhile$:
  $\exists$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\land$ $\neg$ $Cycle$ ($takeWhile$ p $l1$) ($takeWhile$ p $l2$).

Lemma $Cycle\_dropWhile$:
  $\exists$ (A : Type) (p : $A \to bool$) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\land$ $\neg$ $Cycle$ ($dropWhile$ p $l1$) ($dropWhile$ p $l2$).

Lemma $Cycle\_pmap$:
  $\forall$ (A B : Type) (f : $A \to option$ B) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\to$ $Cycle$ ($pmap$ f $l1$) ($pmap$ f $l2$).

Lemma $Cycle\_intersperse$:
  $\exists$ (A : Type) (x : A) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\land$ $\neg$ $Cycle$ ($intersperse$ x $l1$) ($intersperse$ x $l2$).

Lemma $Cycle\_Permutation$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\to$ $Permutation$ $l1$ $l2$.

Lemma $Cycle\_elem$:
  $\forall$ (A : Type) ($l1$ $l2$: $list$ A),
    $Cycle$ $l1$ $l2$ $\to$ $\forall$ x : A, $elem$ x $l1$ $\leftrightarrow$ $elem$ x $l2$.

Lemma $Cycle\_replicate'$:

$\forall$ (A : Type) (n m : nat) (x y : A),
  $Cycle$ ($replicate$ n x) ($replicate$ m y) $\leftrightarrow$
  $n = m \land (n \neq 0 \rightarrow m \neq 0 \rightarrow x = y)$.

Lemma $Cycle\_In$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow \forall$ x : A, In x $l1$ $\leftrightarrow$ In x $l2$.

Lemma $Cycle\_NoDup$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow$ $NoDup$ $l1$ $\rightarrow$ $NoDup$ $l2$.

Lemma $Cycle\_Dup$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow$ Dup $l1$ $\rightarrow$ Dup $l2$.

Lemma $Cycle\_Rep$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow \forall$ (x : A) (n : nat),
    Rep x n $l1$ $\rightarrow$ Rep x n $l2$.

Lemma $Cycle\_Exists$:
$\forall$ (A : Type) (P : $A \rightarrow$ Prop) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow$ $Exists$ P $l1$ $\rightarrow$ $Exists$ P $l2$.

Lemma $Cycle\_Forall$:
$\forall$ (A : Type) (P : $A \rightarrow$ Prop) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow$ $Forall$ P $l1$ $\rightarrow$ $Forall$ P $l2$.

Lemma $Cycle\_AtLeast$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow \forall$ (P : $A \rightarrow$ Prop) (n : nat),
    $AtLeast$ P n $l1$ $\rightarrow$ $AtLeast$ P n $l2$.

Lemma $Cycle\_Exactly$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow \forall$ (P : $A \rightarrow$ Prop) (n : nat),
    $Exactly$ P n $l1$ $\rightarrow$ $Exactly$ P n $l2$.

Lemma $Cycle\_AtMost$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\rightarrow \forall$ (P : $A \rightarrow$ Prop) (n : nat),
    $AtMost$ P n $l1$ $\rightarrow$ $AtMost$ P n $l2$.

Lemma $Cycle\_Sublist$:
$\exists$ (A : Type) ($l1$ $l2$: $list$ A),
  $Cycle$ $l1$ $l2$ $\land \neg$ $Sublist$ $l1$ $l2$.

Lemma $Sublist\_Cycle$:
$\exists$ (A : Type) ($l1$ $l2$: $list$ A),

```
    Sublist l1 l2 ∧ ¬ Cycle l1 l2.

Lemma Cycle_Prefix :
  ∃ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 ∧ ¬ Prefix l1 l2.

Lemma Prefix_Cycle :
  ∃ (A : Type) (l1 l2 : list A),
    Prefix l1 l2 ∧ ¬ Cycle l1 l2.

Lemma Cycle_Subseq :
  ∃ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 ∧ ¬ Subseq l1 l2.

Lemma Subseq_Cycle :
  ∃ (A : Type) (l1 l2 : list A),
    Subseq l1 l2 ∧ ¬ Cycle l1 l2.

Lemma Cycle_Incl :
  ∀ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 → Incl l1 l2.

Lemma Incl_Cycle :
  ∃ (A : Type) (l1 l2 : list A),
    Incl l1 l2 ∧ ¬ Cycle l1 l2.

Lemma Cycle_SetEquiv :
  ∀ (A : Type) (l1 l2 : list A),
    Cycle l1 l2 → SetEquiv l1 l2.

Lemma SetEquiv_Cycle :
  ∃ (A : Type) (l1 l2 : list A),
    SetEquiv l1 l2 ∧ ¬ Cycle l1 l2.
```

10.7 Niestandardowe reguły indukcyjne

Wyjaśnienia nadejdą już wkrótce.

```
Fixpoint list_ind_2
  (A : Type) (P : list A → Prop)
  (H0 : P [])
  (H1 : ∀ x : A, P [x])
  (H2 : ∀ (x y : A) (l : list A), P l → P (x :: y :: l))
    (l : list A) : P l.

Lemma list_ind_rev :
  ∀ (A : Type) (P : list A → Prop)
    (Hnil : P [])
    (Hsnoc : ∀ (h : A) (t : list A), P t → P (t ++ [h]))
```

(l : list A), P l.

```
Lemma list_ind_app_l :
  ∀ (A : Type) (P : list A → Prop)
  (Hnil : P []) (IH : ∀ l l' : list A, P l → P (l' ++ l))
    (l : list A), P l.
```

```
Lemma list_ind_app_r :
  ∀ (A : Type) (P : list A → Prop)
  (Hnil : P []) (IH : ∀ l l' : list A, P l → P (l ++ l'))
    (l : list A), P l.
```

```
Lemma list_ind_app :
  ∀ (A : Type) (P : list A → Prop)
  (Hnil : P []) (Hsingl : ∀ x : A, P [x])
  (IH : ∀ l l' : list A, P l → P l' → P (l ++ l'))
    (l : list A), P l.
```

```
Lemma list_app_ind :
  ∀ (A : Type) (P : list A → Prop),
    P [] →
    (∀ (l l1 l2 : list A), P l → P (l1 ++ l ++ l2)) →
      ∀ l : list A, P l.
```

10.7.1 Palindromy

Palindrom to słowo, które czyta się tak samo od przodu jak i od tyłu.

Zdefiniuj induktywny predykat *Palindrome*, które odpowiada powyższemu pojęciu palindromu, ale dla list elementów dowolnego typu, a nie tylko słów.

```
Lemma Palindrome_inv_2 :
  ∀ (A : Type) (x y : A),
    Palindrome [x; y] → x = y.
```

```
Lemma Palindrome_inv_3 :
  ∀ (A : Type) (x y : A) (l : list A),
    Palindrome (x :: l ++ [y]) → x = y.
```

```
Lemma nat_ind_2 :
  ∀ P : nat → Prop,
    P 0 → P 1 → (∀ n : nat, P n → P (S (S n))) →
      ∀ n : nat, P n.
```

```
Lemma Palindrome_length :
  ∀ (A : Type) (x : A) (n : nat),
    ∃ l : list A, Palindrome l ∧ n ≤ length l.
```

```
Lemma Palindrome_cons_snoc :
```

```
  ∀ (A : Type) (x : A) (l : list A),
    Palindrome l → Palindrome (x :: snoc x l).

Lemma Palindrome_app :
  ∀ (A : Type) (l1 l2 : list A),
    Palindrome l1 → Palindrome l2 → Palindrome (l1 ++ l2 ++ rev l1).

Lemma Palindrome_app' :
  ∀ (A : Type) (l1 l2 : list A),
    Palindrome l2 → Palindrome (l1 ++ l2 ++ rev l1).

Lemma Palindrome_rev :
  ∀ (A : Type) (l : list A),
    Palindrome l ↔ Palindrome (rev l).

Definition lengthOrder {A : Type} (l1 l2 : list A) : Prop :=
  length l1 < length l2.

Lemma lengthOrder_wf :
  ∀ A : Type, well_founded (@lengthOrder A).

(*  TODO: spec bez używania indukcji dobrze ufundowanej *)

Lemma Palindrome_spec :
  ∀ (A : Type) (l : list A),
    Palindrome l ↔ l = rev l.

Lemma Palindrome_spec' :
  ∀ (A : Type) (l : list A),
    Palindrome l → ∃ l1 l2 : list A,
      l = l1 ++ l2 ++ rev l1 ∧ length l2 ≤ 1.

Lemma Palindrome_map :
  ∀ (A B : Type) (f : A → B) (l : list A),
    Palindrome l → Palindrome (map f l).

Lemma replicate_S :
  ∀ (A : Type) (n : nat) (x : A),
    replicate (S n) x = x :: replicate n x.

Lemma Palindrome_replicate :
  ∀ (A : Type) (n : nat) (x : A),
    Palindrome (replicate n x).

Lemma Palindrome_cons_replicate :
  ∀ (A : Type) (n : nat) (x y : A),
    Palindrome (x :: replicate n y) → n = 0 ∨ x = y.

Lemma Palindrome_iterate :
  ∀ (A : Type) (f : A → A),
    (∀ (n : nat) (x : A), Palindrome (iterate f n x)) →
      ∀ x : A, f x = x.
```

Lemma $Palindrome\_nth$:
$\forall$ (A : Type) (l : $list$ A),
$Palindrome$ $l \rightarrow \forall$ n : nat,
$n <$ $length$ $l \rightarrow$ nth n l = nth ($length$ l - S n) l.

Lemma $Palindrome\_drop$:
$\forall$ (A : Type) (l : $list$ A),
($\forall$ n : nat, $Palindrome$ ($drop$ n l)) $\rightarrow$
l = [] $\vee$ $\exists$ (n : nat) (x : A), l = $replicate$ n x.

Lemma $Palindrome\_take$:
$\forall$ (A : Type) (l : $list$ A),
($\forall$ n : nat, $Palindrome$ ($take$ n l)) $\rightarrow$
l = [] $\vee$ $\exists$ (n : nat) (x : A), l = $replicate$ n x.

Lemma $replace\_Palindrome$:
$\forall$ (A : Type) (l l' : $list$ A) (n : nat) (x : A),
`replace` l n x = $Some$ l' $\rightarrow$ $Palindrome$ l $\rightarrow$
$Palindrome$ l' $\leftrightarrow$ $length$ l = 1 $\wedge$ n = 0 $\vee$ nth n l = $Some$ x.

Lemma $Palindrome\_zip$:
$\exists$ (A B : Type) (la : $list$ A) (lb : $list$ B),
$Palindrome$ la $\wedge$ $Palindrome$ lb $\wedge$ $\neg$ $Palindrome$ (zip la lb).

(* TODO: unzip, zipWith, unzipWith *)

Lemma $Palindrome\_find\_findLast$:
$\forall$ (A : Type) (p : $A \rightarrow bool$) (l : $list$ A),
$Palindrome$ $l \rightarrow$ $find$ p l = $findLast$ p l.

Lemma $Palindrome\_pmap$:
$\forall$ (A B : Type) (f : $A \rightarrow option$ B) (l : $list$ A),
$Palindrome$ $l \rightarrow$ $Palindrome$ ($pmap$ f l).

Lemma $Palindrome\_intersperse$:
$\forall$ (A : Type) (x : A) (l : $list$ A),
$Palindrome$ $l \rightarrow$ $Palindrome$ ($intersperse$ x l).

(* TODO: groupBy *)

Lemma $Palindrome\_Dup$:
$\forall$ (A : Type) (l : $list$ A),
$Palindrome$ $l \rightarrow$ $length$ $l \leq 1 \vee$ Dup l.

(* TODO: Incl, Sublist, subseq *)

Lemma $Sublist\_Palindrome$:
$\forall$ (A : Type) ($l1$ $l2$: $list$ A),
$Sublist$ $l1$ $l2$ $\rightarrow$ $Palindrome$ $l1$ $\rightarrow$ $Palindrome$ $l2$ $\rightarrow$ $l1$ = [].

Lemma $Prefix\_Palindrome$:
$\forall$ (A : Type) (l : $list$ A),

Prefix (*rev* *l*) *l* ↔ *Palindrome* *l*.

`Lemma` *Subseq_rev_l* :
  ∀ (*A* : `Type`) (*l* : *list* *A*),
    Subseq (*rev* *l*) *l* ↔ *Palindrome* *l*.

`Lemma` *Subseq_rev_r* :
  ∀ (*A* : `Type`) (*l* : *list* *A*),
    Subseq *l* (*rev* *l*) ↔ *Palindrome* *l*.

Rozdział 11

X31: Złożoność obliczeniowa

(Uwaga: numeracja jest chwilowa, bo nie wiem jak zrobić kolejność. To jest tylko draft. Chyba będę pisał rozdziały chaotycznie.)

Prerekwizyty:

- rekursja strukturalna
- dowodzenie przez indukcję
- listy
- teoria relacji

```
Require Import CoqBookPL.book.X3.
Require Import Omega.
Require Import Nat.
```

Zapoznaliśmy się już z rekursją strukturalną, dzięki której możemy definiować proste funkcje, oraz z techniką dowodzenia przez indukcję, dzięki której możemy stwierdzić ponad wszelką wątpliwość, że nasze funkcje robią to, czego od nich wymagamy. Skoro tak, to czas zapoznać się z kolejnym istotnym elementem układanki, jakim jest złożoność obliczeniowa.

W tym rozdziale nauczysz się analizować proste algorytmy pod względem czasu ich działania. Poznasz też technikę, która pozwala napisać niektóre funkcje rekurencyjne w dużo wydajniejszy sposób.

11.1 Czas działania programu

Cel naszych rozważań w tym rozdziale jest prosty: chcemy zbadać, jak długo będą wykonywać się nasze programy.

Jest to z pozoru proste zadanie: wystarczy włączyć zegar, odpalić program i wyłączyć zegar, gdy program się wykona. Takie podejście ma jednak spore wady, gdyż zmierzony w ten sposób czas:

- Zależy od sprzętu. Im lepszy sprzęt, tym krótszy czas.
- Jest w pewnym sensie losowy. Za każdym wykonaniem programu czas jego działania będzie nieco inny. Wobec tego musielibyśmy puszczać nasz program wielokrotnie, co spowolniłoby mierzenie czasu jego wykonania. Musielibyśmy też, zamiast "zwykłego" czasu działania, posługiwać się średnim czasem działania, co rodzi obawy natury statystycznej.
- Jest trudny do zmierzenia. Co, jeżeli wykonanie programu jest dłuższe, niż przewidywany czas istnienia wszechświata?

Wobec powyższego mierzenie czasu za pomocą zegarka należy odrzucić. Innym z pozoru dobrym pomysłem jest zastępienie pojęcia "czasu" pojęciem "ilości taktów procesora". Jednak i ono ma swoje wady:

- Zależy od sprzętu. Niektóre procesory mogą np. wykonywać wiele operacji na raz (wektoryzacja), inne zaś mają po kilka rdzeni i być może zechcą wykonać nasz kod współbieżnie na kilku z nich.
- Zależy od implementacji języka, którym się posługujemy. W Coqu jest możliwość ekstrakcji kodu do kilku innych języków (Haskell, Ocaml, Scheme), a kod wyekstraktowany do Haskella najpewniej miałby inny czas działania, niż kod wyekstraktowany do Ocamla.
- Również jest trudny do zmierzenia.

Jak widać, mierzenie czasu za pomocą taktów procesora też nie jest zbyt dobrym pomysłem. Prawdę mówiąc, wszelkie podejścia oparte na mierzeniu czegokolwiek będą się wiązały z takimi nieprzyjemnościami, jak błędy pomiaru, problemy z mierzeniem, czy potencjalna konieczność posługiwania się uśrednieniami.

11.2 Złożoność obliczeniowa

Zdecydujemy się zatem na podejście bardziej abstrakcyjne: będziemy liczyć, ile operacji wykonuje nasz program w zależności od rozmiaru danych wejściowych. Niech cię nie zmyli słowo "rozmiar": nie ma ono nic wspólnego z mierzeniem.

Żeby za dużo nie gdakać, rzućmy okiem na przykład.

```
Print head.
(*  ===> head =
        fix head (A : Type) (l : list A) {struct l} : option A :=
        match l with
            | [] => None
            | h :: _ => Some h
        end
```

```
: forall A : Type, list A -> option A *)
```

Tak powinna wyglądać definicja funkcji *head*, której napisanie było w poprzednim rozdziale jednym z twoich zadań.

Pierwszym krokiem naszej analizy jest ustalenie, czym są dane wejściowe. Dane wejściowe to po prostu argumenty funkcji *head*, czyli A : `Type` oraz l : *list* A.

Drugim krokiem jest ustalanie, które argumenty mają wpływ na czas działania funkcji i jaki jest ich rozmiar. Z pewnością wpływu na wynik nie może mieć typ A, gdyż dla każdego typu robi ona to samo — zmienia się tylko typ danych, na których operuje. Wobec tego jedynym argumentem, którego rozmiar może mieć znaczenie, jest l : *list* A.

Kolejnym krokiem jest ustalenie, jaki jest rozmiar listy l, ale zanim będzie to w ogóle możliwe, musimy zadać sobie bardziej fundamentalne pytanie: czym właściwie jest rozmiar? Przez rozmiar rozumieć będziemy zawsze pewną liczbę naturalną, która intuicyjnie mówi nam, jak duży i skomplikowany jest dany obiekt.

W przypadku typów induktywnych powinno to być dość jasne. Jako że funkcje na obiektach takich typów definiujemy przez rekursję, która stopniowo "pożera" swój argument, spodziewamy się, że obliczenie funkcji na "większym" obiekcie będzie wymagało wykonania większej ilości wywołań rekurencyjnych, co oznacza dłuższy "czas" wykonania ("czas" jest w cudzysłowie, gdyż tak naprawdę nie badamy już dosłownie czasu działania programu, a jedynie ilość wykonywanych przez niego operacji).

Czymże może być rozmiar listy? Cóż, potencjalnych miar rozmiaru list jest zapewne nieskończenie wiele, ale najsensowniejszym pomysłem, który powinien od razu przyjść ci na myśl, jest jej długość (ta sama, którą obliczamy za pomocą funkcji *length*).

W ostatnim kroku pozostaje nam policzyć na palcach, ile operacji wykonuje nasza funkcja. Pierwszą operacją jest pattern matching. Druga to zwrócenie wyniku. Hmmm, czyżby nasza funkcja wykonywała tylko dwie operacje?

Przypomnij sobie, że wzorce są dopasowywane w kolejności od góry do dołu. Wobec tego jeżeli lista nie jest pusta, to wykonujemy dwa dopasowania, a nie jedno. Wobec dla pustej listy wykonujemy dwie operacje, a dla niepustej trzy.

Ale czy aby na pewno? A może zwrócenie wyniku nie jest operacją? A może jego koszt jest inny niż koszt wykonania dopasowania? Być może nie podoba ci się forma naszego wyniku: "jeżeli pusta to 2, jeżeli nie to 3".

Powyższe wątpliwości wynikają w znacznej mierze z tego, że wynik naszej analizy jest zbyt szczegółowy. Nasze podejście wymaga jeszcze jednego, ostatniego już ulepszenia: zamiast analizy dokładnej posłużymy się analizą asymptotyczną.

11.3 Złożoność asymptotyczna

Za określeniem "złożoność asymptotyczna" kryje się prosta idea: nie interesuje nas dokładna ilość operacji, jakie program wykonuje, a tylko w jaki sposób zwiększa się ona w zależności od rozmiaru danych. Jeżeli przełożymy naszą odpowiedź na język złożoności asymptotycznej, zabrzmi ona: funkcja *head* działa w czasie stałym (co nieformalnie będziemy oznaczać przez O(1)).

Co znaczy określenie “czas stały”? Przede wszystkim nie odnosi się ono do czasu, lecz do ilości operacji. Przywyknij do tej konwencji — gdy chodzi o złożoność, “czas” znaczy “ilość operacji”. Odpowiadając na pytanie: jeżeli funkcja “działa w czasie stałym” to znaczy, że wykonuje ona taką samą ilość operacji niezależnie od rozmiaru danych.

Uzyskana odpowiedź nie powinna nas dziwić — ustaliliśmy wszakże, że funkcja *head* oblicza wynik za pomocą góra dwóch dopasowań do wzorca. Nawet jeżeli przekażemy do niej listę o długości milion, to nie dotyka ona jej ogona o długości 999999.

Co dokładnie oznacza stwierdzenie “taką samą ilość operacji”? Mówiąc wprost: ile konkretnie? O tym informuje nas nasze nieformalne oznaczenie O(1), które niedługo stanie się dla nas jasne. Przedtem jednak należy zauważyć, że istnieją trzy podstawowe sposoby analizowania złożoności asymptotycznej:

- optymistyczny, polegający na obliczeniu najkrótszego możliwego czasu działania programu
- średni, który polega na oszacowaniu przeciętnego czasu działania algorytmu, czyli czasu działania dla “typowych” danych wejściowych
- pesymistyczny, polegający na obliczeniu najgorszego możliwego czasu działania algorytmu.

Analizy optymistyczna i pesymistyczna są w miarę łatwe, a średnia — dość trudna. Jest tak dlatego, że przy dwóch pierwszych sposobach interesuje nas dokładnie jeden przypadek (najbardziej lub najmniej korzystny), a przy trzecim — przypdek “średni”, a do uporania się z nim musimy przeanalizować wszystkie przypadki.

Analizy średnia i pesymistyczna są w miarę przydatne, a optymistyczna — raczej nie. Optymizm należy odrzucić choćby ze względu na prawa Murphy’ego, które głoszą, że “jeżeli coś może się nie udać, to na pewno się nie uda”.

Wobec powyższych rozważań skupimy się na analizie pesymistycznej, gdyż ona jako jedyna z trzech możliwości jest zarówno użyteczna, jak i w miarę łatwa.

11.4 Duże O

11.4.1 Definicja i intuicja

Nadszedł wreszcie czas, aby formalnie zdefiniować “notację” duże O. Wziąłem słowo “notacja” w cudzysłów, gdyż w ten właśnie sposób byt ten jest nazywany w literaturze; w Coqu jednak słowo “notacja” ma zupełnie inne znaczenie, nijak niezwiązane z dużym O. Zauważmy też, że zbieżność nazwy O z identyczną nazwą konstruktora O : *nat* jest jedynie smutnym przypadkiem.

```
Definition O (f g : nat → nat) : Prop :=
  ∃ c n : nat,
    ∀ n' : nat, n ≤ n' → f n' ≤ c × g n'.
```

Zdanie $O\ f\ g$ można odczytać jako "f rośnie nie szybciej niż g" lub "f jest asymptotycznie mniejsze od g", gdyż O jest pewną formą porządku. Jest to jednak porządek specyficzny:

- Po pierwsze, funkcje f i g porównujemy porównując wyniki zwracane przez nie dla danego argumentu.
- Po drugie, nie porównujemy ich na wszystkich argumentach, lecz jedynie na wszystkich argumentach większych od pewnego $n : nat$. Oznacza to, że f może być "większe" od g na skończonej ilości argumentów od 0 do n, a mimo tego i tak być od g asymptotycznie mniejsze.
- Po trzecie, nie porównujemy $f\ n'$ bezpośrednio do $g\ n'$, lecz do $c \times g\ n'$. Można to intuicyjnie rozumieć tak, że nie interesują nas konkretne postaci funkcji f i g lecz jedynie ich komponenty najbardziej znaczące, czyli najbardziej wpływające na wynik. Przykład: jeżeli f(n) = 4n^2, a g(n) = 42n^42, to nie interesują nas stałe 4 i 42. Najbardziej znaczącym komponentem f jest n^2, zaś g — n^42.

Poszukaj w Internecie wizualizacji tej idei — ja niestety mam bardzo ograniczone możliwości osadzania multimediów w niniejszej książce (TODO: postaram się coś na to poradzić).

11.4.2 Złożoność formalna i nieformalna

Ostatecznie nasze nieformalne stwierdzenie, że złożoność funkcji *head* to O(1) możemy rozumieć tak: "ilość operacji wykonywanych przez funkcję *head* jest stała i nie zależy w żaden sposób od długości listy, która jest jej argumentem". Nie musimy przy tym zastanawiać się, ile dokładnie operacji wykonuje *head*: może 2, może 3, a może nawet 4, ale na pewno mniej niż, powiedzmy, 1000, więc taką wartość możemy przyjąć za c.

To nieformalne stwierdzenie moglibyśmy przy użyciu naszej formalnej definicji zapisać jako $O\ f$ (`fun _` ⇒ 1), gdzie f oznaczałoby ilość operacji wykonywanych przez funkcję *head*.

Moglibyśmy, ale nie możemy, gdyż zdania dotyczące złożoności obliczeniowej funkcji *head*, i ogólnie wszystkich funkcji możliwych do zaimplementowania w Coqu, nie są zdaniami Coqa (czyli termami typu `Prop`), lecz zdaniami o Coqu, a więc zdaniami wyrażonymi w metajęzyku (którym jest tutaj język polski).

Jest to bardzo istotne spostrzeżenie, więc powtórzmy je, tym razem nieco dobitniej: jest niemożliwe, aby w Coqu udowodnić, że jakaś funkcja napisana w Coqu ma jakąś złożoność obliczeniową.

Z tego względu nasza definicja O oraz ćwiczenia jej dotyczące mają jedynie charakter pomocniczy. Ich celem jest pomóc ci zrozumieć, czym jest złożoność asymptotyczna. Wszelkie dowodzenie złożoności obliczeniowej będziemy przeprowadzać w sposób tradycyjny, czyli "na kartce" (no, może poza pewną sztuczką, ale o tym później).

Ćwiczenie Udowodnij, że O jest relacją zwrotną i przechodnią. Pokaż też, że nie jest ani symetryczna, ani słabo antysymetryczna.

`Lemma` O_refl :
  $\forall\ f\ :\ nat \to nat,\ O\ f\ f.$

`Lemma` O_trans :
  $\forall\ f\ g\ h\ :\ nat \to nat,$
    $O\ f\ g \to O\ g\ h \to O\ f\ h.$

`Lemma` O_asym :
  $\exists\ f\ g\ :\ nat \to nat,\ O\ f\ g \land \neg\ O\ g\ f.$

`Lemma` $O\_not\_weak\_antisym$:
  $\exists\ f\ g\ :\ nat \to nat,\ O\ f\ g \land\ O\ g\ f \land f \neq g.$

11.4.3 Duże Omega

`Definition` $Omega\ (f\ g\ :\ nat \to nat)$: `Prop` $:=\ O\ g\ f.$

Omega to O z odwróconymi argumentami. Skoro $O\ f\ g$ oznacza, że f rośnie nie szybciej niż g, to $Omega\ g\ f$ musi znaczyć, ż g rośnie nie wolniej niż f. O oznacza więc ograniczenie górne, a *Omega* ograniczenie dolne.

`Lemma` $Omega\_refl$:
  $\forall\ f\ :\ nat \to nat,\ Omega\ f\ f.$

`Lemma` $Omega\_trans$:
  $\forall\ f\ g\ h\ :\ nat \to nat,$
    $Omega\ f\ g \to Omega\ g\ h \to Omega\ f\ h.$

`Lemma` $Omega\_not\_weak\_antisym$:
  $\exists\ f\ g\ :\ nat \to nat,\ O\ f\ g \land\ O\ g\ f \land f \neq g.$

11.5 Duże Theta

`Definition` $Theta\ (f\ g\ :\ nat \to nat)$: `Prop` $:=\ O\ f\ g \land\ O\ g\ f.$

Definicja $Theta\ f\ g$ głosi, że $O\ f\ g$ i $O\ g\ f$. Przypomnijmy, że $O\ f\ g$ możemy rozumieć jako "f rośnie asymptotycznie nie szybciej niż g", zaś $O\ g\ f$ analogicznie jako "g rośnie asymptotycznie nie szybciej niż f". Wobec tego interpretacja $Theta\ f\ g$ nasuwa się sama: "f i g rosną asymptotycznie w tym samym tempie".

Theta jest relacją równoważności, która oddaje nieformalną ideę najbardziej znaczącego komponentu funkcji, którą posłużyliśmy się opisując intuicje dotyczące O. Parafrazując:

- $O\ f\ g$ znaczy tyle, co "najbardziej znaczący komponent f jest mniejszy lub równy najbardziej znaczącemu komponentowi g"
- $Theta\ f\ g$ znaczy "najbardziej znaczące komponenty f i g są sobie równe".

Ćwiczenie `Theorem` *Theta_refl* :
  ∀ *f* : *nat* → *nat*, *Theta f f*.

`Theorem` *Theta_trans* :
  ∀ *f g h* : *nat* → *nat*,
    Theta f g → *Theta g h* → *Theta f h*.

`Theorem` *Theta_sym* :
  ∀ *f g* : *nat* → *nat*,
    Theta f g → *Theta g f*.

11.6 Złożoność typowych funkcji na listach

11.6.1 Analiza nieformalna

Skoro rozumiesz już, na czym polegają *O* oraz *Theta*, przeanalizujemy złożoność typowej funkcji operującej na listach. Zapoznamy się też z dwoma sposobami na sprawdzenie poprawności naszej analizy: mimo, że w Coqu nie można udowodnić, że dana funkcja ma jakąś złożoność obliczeniową, możemy użyć Coqa do upewnienia się, że nie popełniliśmy w naszej analizie nieformalnej pewnych rodzajów błędów.

Naszą ofiarą będzie funkcja *length*.

```
Print length.
(*  ===> length =
        fix length (A : Type) (l : list A) {struct l} : nat :=
        match l with
            | [] => 0
            | _ :: t => S (length A t)
        end
        : forall A : Type, list A -> nat *)
```

Oznaczmy złożoność tej funkcji w zależności o rozmiaru (długości) listy *l* przez T(n) (pamiętaj, że jest to oznaczenie nieformalne, które nie ma nic wspólnego z Coqiem). Jako, że nasza funkcja wykonuje dopasowanie *l*, rozważmy dwa przypadki:

- *l* ma postać []. Wtedy rozmiar *l* jest równy 0, a jedyne co robi nasza funkcja, to zwrócenie wyniku, które policzymy jako jedna operacja. Wobec tego T(0) = 1.

- *l* ma postać *h* :: *t*. Wtedy rozmiar *l* jest równy n + 1, gdzie n jest rozmiarem *t*. Nasza funkcja robi dwie rzeczy: rekurencyjnie wywołuje się z argumentem *t*, co kosztuje nas T(n) operacji , oraz dostawia do wyniku tego wywołania *S*, co kosztuje nas 1 operację. Wobec tego T(n + 1) = T(n) + 1.

Otrzymaliśmy więc odpowiedź w postaci równania rekurencyjnego T(0) = 1; T(n + 1) = T(n) + 1. Widać na oko, że T(n) = n + 1, a zatem złożoność funkcji *length* to O(n).

11.6.2 Formalne sprawdzenie

Ćwiczenie Żeby przekonać się, że powyższy akapit nie kłamie, zaimplementuj T w Coqu i udowodnij, że rzeczywiście rośnie ono nie szybciej niż `fun` $n \Rightarrow n$.

```
Theorem T_spec_0 : T 0 = 1.

Theorem T_spec_S : ∀ n : nat, T (S n) = 1 + T n.

Theorem T_sum : ∀ n : nat, T n = n + 1.

Theorem O_T_n : O T (fun n ⇒ n).
```

Prześledźmy jeszcze raz całą analizę, krok po kroku:

- oznaczamy złożoność analizowanej funkcji przez T
- patrząc na definicję analizowanej funkcji definiujemy T za pomocą równań T(0) = 1 i T(n + 1) = T(n) + 1
- rozwiązujemy równanie rekurencyjne i dostajemy T(n) = n + 1
- konkludujemy, że złożoność analizowanej funkcji to O(n)

W celu sprawdzenia analizy robimy następujące rzeczy:

- implementujemy T w Coqu
- dowodzimy, że rozwiązaliśmy równanie rekurencyjne poprawnie
- pokazujemy, że $O\ T$ (`fun` $n \Rightarrow n$) zachodzi

Dzięki powyższej procedurze udało nam się wyeliminować podejrzenie co do tego, że źle rozwiązaliśmy równanie rekurencyjne lub że źle podaliśmy złożoność za pomocą dużego O. Należy jednak po raz kolejny zaznaczyć, że nasza analiza nie jest formalnym dowodem tego, że funkcja *length* ma złożoność O(n). Jest tak dlatego, że pierwsza część naszej analizy jest nieformalna i nie może zostać w Coqu sformalizowana.

Jest jeszcze jeden sposób, żeby sprawdzić naszą nieformalną analizę. Mianowicie możemy sprawdzić nasze mniemanie, że T(n + 1) = T(n) + 1, dowodząc formalnie w Coqu, że pewna wariacja funkcji *length* wykonuje co najwyżej n wywołań rekurencyjnych, gdzie n jest rozmiarem jej argumentu.

```
Fixpoint length' {A : Type} (fuel : nat) (l : list A) : option nat :=
match fuel, l with
    | 0, _ ⇒ None
    | _, [] ⇒ Some 0
    | S fuel', _ :: t ⇒
        match length' fuel' t with
            | None ⇒ None
```

```
        | Some n ⇒ Some (S n)
    end
end.
```

Pomysł jest prosty: zdefiniujemy wariację funkcji *length* za pomocą techniki, którą nazywam "rekursją po paliwie". W porównaniu do *length*, której argumentem głównym jest l : *list A*, *length'* ma jeden dodatkowy argument *fuel* : *nat*, który będziemy zwać paliwem, a który jest jej argumentem głównym

Nasza rekursja wygląda tak, że każde wywołanie rekurencyjne zmniejsza zapasy paliwa o 1, ale z pozostałymi argumentami możemy robić dowolne cuda. Żeby uwzględnić możliwość wyczerpania się paliwa, nasza funkcja zwraca wartość typu *option nat* zamiast samego *nat*. Wyczerpaniu się paliwa odpowiada wynik *None*, zaś *Some* oznacza, że funkcja zakończyła się wykonywać przed wyczerpaniem się paliwa.

Paliwo jest więc tak naprawdę maksymalną ilością wywołań rekurencyjnych, które funkcja może wykonać. Jeżeli uda nam się udowodnić, że dla pewnej ilości paliwa funkcja zawsze zwraca *Some*, będzie to znaczyło, że znaleźliśmy górne ograniczenie ilości wywołań rekurencyjnych niezbędnych do poprawnego wykonania się funkcji.

Ćwiczenie Uwaga, trudne.

```
Theorem length'_rec_depth :
  ∀ (A : Type) (l : list A),
    length' (S (length l)) l = Some (length l).
```

Twierdzenie to wygląda dość kryptycznie głównie ze względu na fakt, że *length l* jest zarówno analizowaną przez nas funkcją, jaki i funkcją obliczającą rozmiar listy l.

Żeby lepiej zrozumieć, co się stało, spróbujmy zinterpretować powyższe twierdzenie. Mówi ono, że dla dowolnego A : `Type` i l : *list A*, jeżeli wywołamy *length'* na l dając jej S (*length l*) paliwa, to zwróci ona *Some* (*length l*).

Innymi słowy, S (*length l*) paliwa to dostatecznie dużo, aby funkcja wykonała się poprawnie. Dodatkowo *Some* (*length l*) jest pewną formą specyfikacji dla funkcji *length'*, która mówi, że jeżeli *length'* ma dostatecznie dużo paliwa, to wywołanie jej na l daje taki sam wynik jak *length l*, czyli nie pomyliliśmy się przy jej definiowaniu (chcieliśmy, żeby była to "wariacja" *length*, która daje takie same wyniki, ale jest zdefiniowana przez rekursję po paliwie).

Na tym kończy się nasz worek sztuczek formalnych, które pomagają nam upewnić się w poprawności naszej analizy nieformalnej.

11.7 Złożoność problemu

Dotychczas zajmowaliśmy się złożonością obliczeniową funkcji. Złożoność ta oznacza faktycznie złożoność sposobu rozwiązania pewnego problemu — w naszym przypadku były to problemy zwrócenia głowy listy (funkcja *head*) oraz obliczenia jej długości (funkcja *length*).

Złożoność ta nie mówi jednak nic o innych sposobach rozwiązania tego samego problemu. Być może istnieje szybszy sposób obliczania długości listy? Zajmijmy się więc przez krótką chwilę koncepcją pokrewną koncepcji złożoności obliczeniowej programu — jest nią koncepcja złożoności obliczeniowej problemu.

Na początku rozdziału stwierdziliśmy, że naszym celem będzie badanie "czasu działania programu". Taki cel może jednak budzić pewien niesmak: dlaczego mielibyśmy robić coś takiego?

Czas (także w swym informatycznym znaczeniu, jako ilość operacji) jest cennym zasobem i nie chcielibyśmy używać go nadaremnie ani marnować. Jeżeli poznamy złożoność obliczeniową zarówno problemu, jak i jego rozwiązania, to będziemy mogli stwierdzić, czy nasze rozwiązanie jest optymalne (w sensie asymptotycznym, czyli dla instancji problemu, w której rozmiary argumentów są bardzo duże).

Na nasze potrzeby zdefiniujmy złożoność problemu jako złożoność najszybszego programu, który rozwiązuje ten problem. Podobnie jak pojęcie złożoności obliczeniowej programu, jest niemożliwe, aby pojęcie to sformalizować w Coqu, będziemy się więc musieli zadowolić dywagacjami nieformalnymi.

Zacznijmy od *head* i problemu zwrócenia głowy listy. Czy można to zrobić szybciej, niż w czasie stałym? Oczywiście nie. Czas stały to najlepsze, co możemy uzyskać (zastanów się przez chwilę nad tym, dlaczego tak jest). Oczywiście należy to zdanie rozumieć w sensie asymptotycznym: jeżeli chodzi o dokładną złożoność, to różne funkcje działające w czasie stałym mogą wykonywać różną ilość operacji — zarówno "jeden" jak i "milion" oznaczają czas stały. Wobec tego złożoność problemu zwrócenia głowy listy to O(1).

A co z obliczaniem długości listy? Czy można to zrobić szybciej niż w czasie O(n)? Tutaj również odpowiedź brzmi "nie". Jest dość oczywiste, że w celu obliczenia długość całej listy musimy przejść ją całą. Jeżeli przejdziemy tylko pół, to obliczymy długość jedynie połowy listy.

11.8 Przyspieszanie funkcji rekurencyjnych

11.8.1 Złożoność *rev*

Przyjrzyjmy się złożoności funkcji *rev*.

```
Print rev.
(*  ===> rev =
        fix rev (A : Type) (l : list A) {struct l} : list A :=
        match l with
            |  =>
            | h :: t => rev A t ++ h
        end
        : forall A : Type, list A -> list A *)
```

Oznaczmy szukaną złożoność przez T(n). Z przypadku gdy l jest postaci [] uzyskujemy $T(0) = 1$. W przypadku gdy l jest postaci $h :: t$ mamy wywołanie rekurencyjne o koszcie

T(n); dostawiamy też *h* na koniec odwróconego ogona. Jaki jest koszt tej operacji? Aby to zrobić, musimy przebyć *rev t* od początku do końca, a więc koszt ten jest równy długości listy *l*. Stąd T(n + 1) = T(n) + n.

Pozostaje nam rozwiązać równanie. Jeżeli nie potrafisz tego zrobić, dla prostych równań pomocna może być strona https://www.wolframalpha.com/. Rozwijając to równanie mamy T(n) = n + (n - 1) + (n - 2) + ... + 1, więc T jest rzędu O(n^2).

A jaka jest złożoność problemu odwracania listy? Z pewnością nie można tego zrobić, jeżeli nie dotkniemy każdego elementu listy. Wobec tego możemy ją oszacować z dołu przez Omega(n).

Z taką sytuacją jeszcze się nie spotkaliśmy: wiemy, że asymptotycznie problem wymaga Omega(n) operacji, ale nasze rozwiązanie wykonuje O(n^2) operacji. Być może zatem możliwe jest napisanie funkcji *rev* wydajniej.

11.8.2 Pamięć

Przyjrzyj się jeszcze raz definicji funkcji *rev*. Funkcja *rev* nie ma pamięci — nie pamięta ona, jaką część wyniku już obliczyła. Po prostu wykonuje dopasowanie na swym argumencie i wywołuje się rekurencyjnie.

Funkcję *rev* będziemy mogli przyspieszyć, jeżeli dodamy jej pamięć. Na potrzeby tego rozdziału nie będziemy traktować pamięci jak zasobu, lecz jako pewną abstrakcyjną ideę. Przyjrzyjmy się poniższej, alternatywnej implementacji funkcji odwracającej listę.

```
Fixpoint rev_aux {A : Type} (l acc : list A) : list A :=
match l with
    | [] ⇒ acc
    | h :: t ⇒ rev_aux t (h :: acc)
end.

Fixpoint rev' {A : Type} (l : list A) : list A := rev_aux l [].
```

Funkcja *rev_aux* to serce naszej nowej implementacji. Mimo, że odwraca ona listę *l*, ma aż dwa argumenty — poza *l* ma też argument *acc* : *list A*, który nazywać będziemy akumulatorem. To właśnie on jest pamięcią tej funkcji. Jednak jego "bycie pamięcią" nie wynika z jego nazwy, a ze sposobu, w jaki użyliśmy go w definicji *rev_aux*.

Gdy *rev_aux* natrafi na pustą listę, zwraca wartość swego akumulatora. Nie powinno nas to dziwić — wszakże ma w nim zapamiętany cały wynik (bo zjadła już cały argument *l*). Jeżeli napotyka listę postaci *h* :: *t*, to wywołuje się rekurencyjnie na ogonie *t*, ale z akumulatorem, do którego dostawia na początek *h*.

```
Compute rev_aux [1; 2; 3; 4; 5] [].
(*  ===> = 5; 4; 3; 2; 1 : list nat *)
```

Widzimy więc na własne oczy, że *rev_aux* rzeczywiście odwraca listę. Robi to przerzucając swój argument główy kawałek po kawałku do swojego akumulatora — głowa *l* trafia do akumulatora na samym początku, a więc znajdzie się na samym jego końcu, gdyż przykryją ją dalsze fragmenty listy *l*.

```
Compute rev_aux [1; 2; 3; 4; 5] [6; 6; 6].
```

Trochę cię okłamałem twierdząc, że *rev_aux* odwraca *l*. Tak naprawdę oblicza ona odwrotność *l* z doklejonym na końcu akumulatorem. Tak więc wynik zwracany przez *rev_aux* zależy nie tylko od *l*, ale także od akumulatora *acc*. Właściwą funkcję *rev'* uzyskujemy, inicjalizując wartość akumulatora w *rev_aux* listą pustą.

Ćwiczenie Udowodnij poprawność funkcji *rev'*.

```
Lemma rev_aux_spec :
  ∀ (A : Type) (l acc : list A),
    rev_aux l acc = rev l ++ acc.

Theorem rev'_spec :
  ∀ (A : Type) (l : list A), rev' l = rev l.
```

Skoro już wiemy, że udało nam się poprawnie zdefiniować *rev'*, czyli alternatywne rozwiązanie problemu odwracania listy, pozostaje nam tylko sprawdzić, czy rzeczywiście jest ono szybsze niż *rev*. Zanim dokonamy analizy, spróbujemy sprawdzić naszą hipotezę empirycznie — w przypadku zejścia z O(n^2) do O(n) przyspieszenie powinno być widoczne gołym okiem.

Ćwiczenie Zdefiniuj funkcje *to0*, gdzie *to0 n* jest listą liczb od *n* do 0. Udowodnij poprawność zdefiniowanej funkcji.

```
Theorem to0_spec :
  ∀ n k : nat, k ≤ n → elem k (to0 n).

Time Eval compute in rev (to0 2000).
(*  ===> (...) Finished transaction in 7. secs (7.730824u,0.s) *)

Time Eval compute in rev' (to0 2000).
(*  ===> (...) Finished transaction in 4. secs (3.672441u,0.s) *)
```

Nasze mierzenie przeprowadzić możemy za pomocą komendy `Time`. Odwrócenie listy 2000 elementów na moim komputerze zajęło *rev* 7.73 sekundy, zaś *rev'* 3.67 sekundy, a więc jest ona w tym przypadku ponad dwukrotnie szybsza. Należy jednak zaznaczyć, że empiryczne próby badania szybkości programów w Coqu nie są dobrym pomysłem, gdyż nie jest on przystosowany do szybkiego wykonywania programów — jest on wszakże głównie asystentem dowodzenia.

Zakończmy analizą teoretyczną złożoności *rev'*. Oznaczmy czas działania *rev_aux* przez T(n). Dla [] zwraca ona jedynie akumulator, a zatem T(0) = 1. Dla *h* :: *t* przekłada ona głowę argumentu do akumulatora i wywołuje się rekurencyjnie, czyli T(n + 1) = T(n) + 1. Rozwiązując równanie rekurencyjne dostajemy T(n) = n + 1, a więc złożoność *rev_aux* to O(n). Jako, że *rev'* wywołuje *rev_aux* z pustym akumulatorem, to również jej złożoność wynosi O(n).

11.9 Podsumowanie

W tym rozdziale postawiliśmy sobie za cel mierzenie "czasu" działania programu. Szybko zrezygnowaliśmy z tego celu i zamieniliśmy go na analizę złożonóści obliczeniowej, choć bezpośrednie mierzenie nie jest niemożliwe.

Nauczyliśmy się analizować złożoność funkcji rekurencyjnych napisanych w Coqu, a także analizować złożoność samych problemów, które owe funkcje rozwiązują. Poznaliśmy też kilka sztuczek, w których posłużyliśmy się Coqiem do upewnienia się w naszych analizach.

Następnie porównując złożoność problemu odwracania listy ze złożonością naszego rozwiązania zauważyliśmy, że moglibyśmy rozwiązać go wydajniej. Poznaliśmy abstrakcyjne pojęcie pamięci i przyspieszyliśmy za jego pomocą funkcję *rev*.

Zdobytą wiedzę będziesz mógł od teraz wykorzystać w praktyce — za każdym razem, kiedy wyda ci się, że jakaś funcja "coś wolno działa", zbadaj jej złożoność obliczeniową i porównaj ze złożonością problemu, który rozwiązuje. Być może uda ci się znaleźć szybsze rozwiązanie.

Rozdział 12

X4: Funkcje

`Require Import` *Arith*.

Prerekwizyty:

- *Empty_set*, *unit*, *prod*, *sum* i funkcje
- właściwości konstruktorów?
- ∃!
- równość *eq*

W tym rozdziale zapoznamy się z najważniejszymi rodzajami funkcji. Trzeba przyznać na wstępie, że rozdział będzie raczej matematyczny.

12.1 Funkcje

Potrafisz już posługiwać się funkcjami. Mimo tego zróbmy krótkie przypomnienie.

Typ funkcji (niezależnych) z A w B oznaczamy przez $A \to B$. W Coqu funkcje możemy konstruować za pomocą abstrakcji (np. `fun` n : $nat \Rightarrow n + n$) albo za pomocą rekursji strukturalnej. Eliminować zaś możemy je za pomocą aplikacji: jeżeli $f : A \to B$ oraz $x : A$, to $f\ x : B$.

Funkcje wyrażają ideę przyporządkowania: każdemu elementowi dziedziny funkcja przyporządkowuje element przeciwdziedziny. Jednak status dziedziny i przeciwdziedziny nie jest taki sam: każdemu elementowi dziedziny coś odpowiada, jednak mogą istnieć elementy przeciwdziedziny, które nie są obrazem żadnego elementu dziedziny.

Co więcej, w Coqu wszystkie funkcje są konstruktywne, tzn. mogą zostać obliczone. Jest to coś, co bardzo mocno odróżnia Coqa oraz rachunek konstrukcji (jego teoretyczną podstawę) od innych systemów formalnych.

`Definition` *app* $\{A\ B$: `Type`$\}$ $(f : A \to B)$ $(x : A) : B := f\ x$.

```
Definition app' {A B : Type} (x : A) (f : A → B) : B := f x.
Notation "f $ x":= (app f x) (left associativity, at level 110).
Notation "x $> f":= (app' x f) (right associativity, at level 60).
Check plus (2 + 2) (3 + 3).
Check plus $ 2 + 2 $ 3 + 3.
Check (fun n : nat ⇒ n + n) 21.
Check 21 $> fun n : nat ⇒ n + n.
```

Najważniejszą rzeczą, jaką możemy zrobić z funkcją, jest zaaplikowanie jej do argumentu. Jest to tak częsta operacja, że zdefiniujemy sobie dwie notacje, które pozwolą nam zaoszczędzić kilka stuknięć w klawiaturę.

Uwaga techniczna: nie możemy przypisać notacji do wyrażenia "f x", gdyż zepsuło by to wyświetanie. Z tego powodu musimy napisać dwie osobne funkcje *app* i *app'* i do nich przypisać notacje.

Notacja $ będzie nam służyć do niepisania nawiasów: jeżeli argumentami funkcji będą skomplikowane termy, zamiast pisać wokół nich parę nawiasów, będziemy mogli wstawić tylko jeden symbol dolara "$". Dzięki temu zamiast 2n nawiasów napiszemy tylko n znaków "$" (choć trzeba przyznać, że będziemy musieli pisać więcej spacji).

Notacja $> umożliwi nam pisanie aplikacji w odwrotnej kolejności. Dzięki temu będziemy mogli np. pomijać nawiasy w abstrakcji. Jako, że nie da się zrobić notacji "x f", jest to najlepsze dostępne nam rozwiązanie.

```
Definition comp {A B C : Type} (f : A → B) (g : B → C)
  : A → C := fun x : A ⇒ g (f x).
Notation "f .> g":= (comp f g) (left associativity, at level 40).
```

Najważniejszą operacją, jaką możemy wykonywać na funkcjach, jest złożenie. Jedynym warunkiem jest, aby przeciwdziedzina pierwszej funkcji była taka sama, jak dziedzina drugiej funkcji. Składanie funkcji jest łączne.

Uwaga techniczna: jeżeli prezentuję jakieś twierdzenie bez dowodu, to znaczy, że dowód jest ćwiczeniem.

```
Theorem comp_assoc :
  ∀ (A B C D : Type) (f : A → B) (g : B → C) (h : C → D),
    (f .> g) .> h = f .> (g .> h).
Definition id (A : Type) : A → A := fun x : A ⇒ x.
```

Najważniejszą funkcją w całym kosmosie jest identyczność. Jest to funkcja, która nie robi zupełnie nic. Jej waga jest w tym, że jest ona elementem neutralnym składania funkcji.

```
Theorem id_left :
  ∀ (A B : Type) (f : A → B), id A .> f = f.
Theorem id_right :
  ∀ (A B : Type) (f : A → B), f .> id B = f.
```

```
Definition const {A B : Type} (b : B) : A → B := fun _ ⇒ b.
```

Funkcja stała to funkcja, która ignoruje swój drugi argument i zawsze zwraca pierwszy argument.

```
Definition flip
  {A B C : Type} (f : A → B → C) : B → A → C :=
    fun (b : B) (a : A) ⇒ f a b.
```

flip to całkiem przydatny kombinator (funkcja wyższego rzędu), który zamienia miejscami argumenty funkcji dwuargumentowej.

```
Fixpoint iter {A : Type} (n : nat) (f : A → A) : A → A :=
match n with
    | 0 ⇒ id A
    | S n' ⇒ f .> iter n' f
end.
```

Ostatnim przydatnim kombinatorem jest *iter*. Służy on do składania funkcji samej ze sobą n razy. Oczywiście funkcja, aby można ją było złożyć ze sobą, musi mieć identyczną dziedzinę i przeciwdziedzinę.

12.2 Aksjomat ekstensjonalności

Ważną kwestią jest ustalenie, kiedy dwie funkcje są równe. Zacznijmy od tego, że istnieją dwie koncepcje równości:

- intensjonalna — funkcje są zdefiniowane przez identyczne (czyli konwertowalne) wyrażenia
- ekstensjonalna — wartości funkcji dla każdego argumentu są równe

```
Print eq.
(*  ===>
    Inductive eq (A : Type) (x : A) : A -> Prop :=
        | eq_refl : x = x
   *)
```

Podstawowym i domyślnym rodzajem równości w Coqu jest równość intensjonalna, której właściwości już znasz. Każda funkcja, na mocy konstruktora *eq_refl*, jest równa samej sobie. Prawdą jest też mniej oczywisty fakt: każda funkcja jest równa swojej ekspansji eta.

```
Theorem eta_expansion :
  ∀ (A B : Type) (f : A → B), f = fun x : A ⇒ f x.
Proof.
  trivial.
Qed.
```

```
Print Assumptions eta_expansion.
(*  ===> Closed under the global context *)
```

Ekspansja eta funkcji f to nic innego, jak funkcja anonimowa, która bierze x i zwraca $f\ x$. Nazwa pochodzi od greckiej litery (eta). Powyższe twierdzenie jest trywialne, gdyż równość zachodzi na mocy konwersji.

Warto podkreślić, że jego prawdziwość nie zależy od żadnych aksjomatów. Stwierdzenie to możemy zweryfikować za pomocą komendy `Print Assumptions`, która wyświetla listę aksjomatów, które zostały wykorzystane w definicji danego termu. Napis "Closed under the global context" oznacza, że żadnego aksjomatu nie użyto.

```
Theorem plus_1_eq :
  (fun n : nat => 1 + n) = (fun n : nat => n + 1).
Proof.
  trivial.
  Fail rewrite plus_comm. (*  No i co teraz? *)
Abort.
```

Równość intensjonalna ma jednak swoje wady. Główną z nich jest to, że jest ona bardzo restrykcyjna. Widać to dobrze na powyższym przykładzie: nie jesteśmy w stanie udowodnić, że funkcje `fun` $n : nat \Rightarrow 1 + n$ oraz `fun` $n : nat \Rightarrow n + 1$ są równe, gdyż zostały zdefiniowane za pomocą innych termów. Mimo, że termy te są równe, to nie są konwertowalne, a zatem funkcje też nie są konwertowalne. Nie znaczy to jednak, że nie są równe — po prostu nie jesteśmy w stanie w żaden sposób pokazać, że są.

```
Require Import FunctionalExtensionality.

Check @functional_extensionality.
(*  ===> @functional_extensionality
          : forall (A B : Type) (f g : A -> B),
              (forall x : A, f x = g x) -> f = g *)
```

Z tarapatów wybawić nas może jedynie aksjomat ekstensjonalności dla funkcji, zwany w Coqu *functional_extensionality* (dla funkcji, które nie są zależne) lub *functional_extensionality_dep* (dla funkcji zależnych).

Aksjomat ten głosi, że f i g są równe, jeżeli są równe dla wszystkich argumentów. Jest on bardzo użyteczny, a przy tym nie ma żadnych smutnych konsekwencji i jest kompatybilny z wieloma innymi aksjomatami. Z tych właśnie powodów jest on jednym z najczęściej używanych w Coqu aksjomatów. My też będziemy go wykorzystywać.

```
Theorem plus_1_eq :
  (fun n : nat => 1 + n) = (fun n : nat => n + 1).
Proof.
  extensionality n. rewrite plus_comm. trivial.
Qed.
```

Sposób użycia aksjomatu jest banalnie prosty. Jeżeli mamy cel postaci $f = g$, to taktyka `extensionality` x przekształca go w cel postaci $f\ x = g\ x$, o ile tylko nazwa x nie jest już wykorzystana na coś innego.

Dzięki zastosowaniu aksjomatu nie musimy już polegać na konwertowalności termów definiujących funkcje. Wystarczy udowodnić, że są one równe. W tym przypadku robimy to za pomocą twierdzenia *plus_comm*.

Ćwiczenie Użyj aksjomatu ekstensjonalności, żeby pokazać, że dwie funkcje binarne są równe wtedy i tylko wtedy, gdy ich wartości dla wszystkich argumentów są równe.

```
Theorem binary_funext :
  ∀ (A B C : Type) (f g : A → B → C),
    f = g ↔ ∀ (a : A) (b : B), f a b = g a b.
```

12.3 Injekcje

TODO ACHTUNG: to pojęcie zostało użyte implicite przy opisywaniu właściciwości konstruktorów.

```
Definition injective {A B : Type} (f : A → B) : Prop :=
  ∀ x x' : A, f x = f x' → x = x'.
```

Objaśnienia zacznijmy od nazwy. Po łacinie "iacere" znaczy "rzucać", zaś "in" znaczy "w, do". W językach romańskich samo słowo "injekcja" oznacza zaś zastrzyk. Bliższym matematycznemu znaczeniu byłoby jednak tłumaczenie "wstrzyknięcie". Jeżeli funkcja jest injekcją, to możemy też powiedzieć, że jest "injektywna". Inną nazwą jest "funkcja różnowartościowa".

en.wikipedia.org/wiki/Bijection,%20injection%20and%20surjection

Tutaj można zapoznać się z obrazkami poglądowymi.

Podstawowa idea jest prosta: jeżeli funkcja jest injekcją, to identyczne jej wartości pochodzą od równych argumentów.

Przekonajmy się na przykładzie.

```
Goal injective (fun n : nat ⇒ 2 + n).
Proof.
  unfold injective; intros. destruct x, x'; cbn in *.
    trivial.
    inversion H.
    inversion H.
    inversion H. trivial.
Qed.
```

Funkcja `fun` n : *nat* ⇒ 2 + n, czyli dodanie 2 z lewej strony, jest injekcją, gdyż jeżeli 2 + n = 2 + n', to rozwiązując równanie dostajemy $n = n'$. Jeżeli wartości funkcji są równe, to argumenty również muszą być równe.

Zobaczmy też kontrprzykład.

```
Goal ¬ injective (fun n : nat ⇒ n × n - n).
Proof.
  unfold injective, not; intros.
```

```
  specialize (H 0 1). simpl in H. specialize (H eq_refl). inversion H.
Qed.
```

Funkcja f(n) = n^2 - n nie jest injekcją, gdyż mamy zarówno f(0) = 0 jak i f(1) = 0. Innymi słowy: są dwa nierówne argumenty (0 i 1), dla których wartość funkcji jest taka sama (0).

A oto alternatywna definicja.

```
Definition injective' {A B : Type} (f : A → B) : Prop :=
  ∀ x x' : A, x ≠ x' → f x ≠ f x'.
```

Głosi ona, że funkcja injektywna to funkcja, która dla różnych argumentów przyjmuje różne wartości. Innymi słowy, injekcja to funkcja, która zachowuje relację ≠. Przykład 1 możemy sparafrazować następująco: jeżeli *n* jest różn od *n'*, to wtedy 2 + *n* jest różne od 2 + *n'*.

Definicja ta jest równoważna poprzedniej, ale tylko pod warunkiem, że przyjmiemy logikę klasyczną. W logice konstruktywnej pierwsza definicja jest ogólniejsza od drugiej.

Ćwiczenie Pokaż, że *injective* jest mocniejsze od *injective'*. Pokaż też, że w logice klasycznej są one równoważne.

```
Theorem injective_injective' :
  ∀ (A B : Type) (f : A → B),
    injective f → injective' f.

Theorem injective'_injective :
  (∀ P : Prop, ¬ ¬ P → P) →
  ∀ (A B : Type) (f : A → B),
    injective' f → injective f.
```

Udowodnij, że różne funkcje są lub nie są injektywne.

```
Theorem id_injective :
  ∀ A : Type, injective (id A).

Theorem S_injective : injective S.

Theorem const_unit_inj :
  ∀ (A : Type) (a : A),
    injective (fun _ : unit ⇒ a).

Theorem add_k_left_inj :
  ∀ k : nat, injective (fun n : nat ⇒ k + n).

Theorem mul_k_inj :
  ∀ k : nat, k ≠ 0 → injective (fun n : nat ⇒ k × n).

Theorem const_2elem_not_inj :
  ∀ (A B : Type) (b : B),
    (∃ a a' : A, a ≠ a') → ¬ injective (fun _ : A ⇒ b).

Theorem mul_k_0_not_inj :
```

$\neg$ *injective* (`fun` n : $nat \Rightarrow 0 \times n$).

`Theorem` *pred_not_injective* : $\neg$ *injective pred.*

Jedną z ważnych właściwości injekcji jest to, że są składalne: złożenie dwóch injekcji daje injekcję.

`Theorem` *inj_comp* :
$\forall$ (A B C : `Type`) (f : $A \rightarrow B$) (g : $B \rightarrow C$),
injective $f \rightarrow$ *injective* $g \rightarrow$ *injective* (f .> g).

Ta właściwość jest dziwna. Być może kiedyś wymyślę dla niej jakąś bajkę.

`Theorem` *LOLWUT* :
$\forall$ (A B C : `Type`) (f : $A \rightarrow B$) (g : $B \rightarrow C$),
injective (f .> g) $\rightarrow$ *injective* f.

Na zakończenie należy dodać do naszej interpretacji pojęcia "injekcja" jeszcze jedną ideę. Mianowicie jeżeli istnieje injekcja $f : A \rightarrow B$, to ilość elementów typu A jest mniejsza lub równa liczbie elementów typu B, a więc typ A jest w pewien sposób mniejszy od B.

f musi przyporządkować każdemu elementowi A jakiś element B. Gdy elementów A jest więcej niż B, to z konieczności któryś z elementów B będzie obrazem dwóch lub więcej elementów A.

Wobec powyższego stwierdzenie "złożenie injekcji jest injekcją" możemy zinterpretować po prostu jako stwierdzenie, że relacja porządku, jaką jest istnienie injekcji, jest przechodnia. (TODO: to wymagałoby relacji jako prerekwizytu).

Ćwiczenie Udowodnij, że nie istnieje injekcja z *bool* w *unit*. Znaczy to, że *bool* ma więcej elementów, czyli jest większy, niż *unit*.

`Theorem` *no_inj_bool_unit* :
$\neg \exists f : bool \rightarrow unit$, *injective* f.

Pokaż, że istnieje injekcja z typu pustego w każdy inny. Znaczy to, że *Empty_set* ma nie więcej elementów, niż każdy inny typ (co nie powinno nas dziwić, gdyż *Empty_set* nie ma żadnych elementów).

`Theorem` *inj_Empty_set_A* :
$\forall$ A : `Type`, $\exists f$: *Empty_set* $\rightarrow A$, *injective* f.

12.4 Surjekcje

Drugim ważnym rodzajem funkcji są surjekcje.

`Definition` *surjective* $\{A\ B$: `Type`$\}$ (f : $A \rightarrow B$) : `Prop` :=
$\forall b : B, \exists a : A, f\ a = b$.

I znów zacznijmy od nazwy. Po francusku "sur" znaczy "na", zaś słowo "iacere" już znamy (po łac. "rzucać"). Słowo "surjekcja" moglibyśmy więc przetłumaczyć jako "pokrycie". Tym bowiem w istocie jest surjekcja — jest to funkcja, która "pokrywa" całą swoją przeciwdziedzinę.

Owo "pokrywanie" w definicji wyraziliśmy w ten sposób: dla każdego elementu b przeciwdziedziny B istnieje taki element a dziedziny A, że $f\ a = b$.

Zobaczmy przykład i kontrprzykład.

```
Theorem pred_surjective : surjective pred.
Proof.
  unfold surjective; intros. ∃ (S b). cbn. trivial.
Qed.
```

TODO Uwaga techniczna: od teraz do upraszczania zamiast taktyki `simpl` używać będziemy taktyki *cbn*. Różni się ona nieznacznie od `simpl`, ale jej główną zaletą jest nazwa — *cbn* to trzy litery, a `simpl` aż pięć, więc zaoszczędzimy sobie pisania.

Powyższe twierdzenie głosi, że "funkcja *pred* jest surjekcją", czyli, parafrazując, "każda liczba naturalna jest poprzednikiem innej liczby naturalnej". Nie powinno nas to zaskakiwać, wszakże każda liczba naturalna jest poprzednikiem swojego następnika, tzn. $pred\ (S\ n) = n$.

```
Theorem S_not_surjective : ¬ surjective S.
Proof.
  unfold surjective; intro. destruct (H 0). inversion H0.
Qed.
```

Surjekcją nie jest za to konstruktor S. To również nie powinno nas dziwić: istnieje przecież liczba naturalna, która nie jest następnikiem żadnej innej. Jest nią oczywiście zero.

Surjekcje cieszą się właściwościami podobnymi do tych, jakie są udziałem injekcji.

Ćwiczenie Pokaż, że złożenie surjekcji jest surjekcją. Udowodnij też "dziwną właściwość" surjekcji.

```
Theorem sur_comp :
  ∀ (A B C : Type) (f : A → B) (g : B → C),
    surjective f → surjective g → surjective (f .> g).

Theorem LOLWUT_sur :
  ∀ (A B C : Type) (f : A → B) (g : B → C),
    surjective (f .> g) → surjective g.
```

Ćwiczenie Zbadaj, czy wymienione funkcje są surjekcjami. Sformułuj i udowodnij odpowiednie twierdzenia.

Funkcje: identyczność, dodawanie (rozważ zero osobno), odejmowanie, mnożenie (rozważ 1 osobno).

Tak jak istnienie injekcji $f : A \to B$ oznacza, że A jest mniejszy od B, gdyż ma mniej (lub tyle samo) elementów, tak istnieje surjekcji $f : A \to B$ oznacza, że A jest większy niż B, gdyż ma więcej (lub tyle samo) elementów.

Jest tak na mocy samej definicji: każdy element przeciwdziedziny jest obrazem jakiegoś elementu dziedziny. Nie jest powiedziane, ile jest tych elementów, ale wiadomo, że co najmniej jeden.

Podobnie jak w przypadku injekcji, fakt że złożenie surjekcji jest surjekcją możemy traktować jako stwierdzenie, że porządek, jakim jest istnienie surjekcji, jest przechodni. (TODO)

Ćwiczenie Pokaż, że nie istnieje surjekcja z *unit* w *bool*. Oznacza to, że *unit* nie jest większy niż *bool*.

```
Theorem no_sur_unit_bool :
  ¬ ∃ f : unit → bool, surjective f.
```

Pokaż, że istnieje surjekcja z każdego typu niepustego w *unit*. Oznacza to, że każdy typ niepusty ma co najmniej tyle samo elementów, co *unit*, tzn. każdy typ nie pusty ma co najmniej jeden element.

```
Theorem sur_A_unit :
  ∀ (A : Type) (nonempty : A), ∃ f : A → unit, surjective f.
```

12.5 Bijekcje

```
Definition bijective {A B : Type} (f : A → B) : Prop :=
  injective f ∧ surjective f.
```

Po łacinie przedrostek "bi-" oznacza "dwa". Bijekcja to funkcja, która jest zarówno injekcją, jak i surjekcją.

```
Theorem id_bij : ∀ A : Type, bijective (@id A).
Proof.
  split; intros.
    apply id_injective.
    apply id_sur.
Qed.
```

```
Theorem S_not_bij : ¬ bijective S.
Proof.
  unfold bijective; intro. destruct H.
  apply S_not_surjective. assumption.
Qed.
```

Pozostawię przykłady bez komentarza — są one po prostu konsekwencją tego, co już wiesz na temat injekcji i surjekcji.

Ponieważ bijekcja jest surjekcją, to każdy element jej przeciwdziedziny jest obrazem jakiegoś elementu jej dziedziny (obraz elementu x to po prostu $f\ x$). Ponieważ jest injekcją, to element ten jest unikalny.

Bijekcja jest więc taką funkcją, że każdy element jej przeciwdziedziny jest obrazem dokładnie jednego elementu jej dziedziny. Ten właśnie fakt wyraża poniższa definicja alternatywna.

TODO: ∃! nie zostało dotychczas opisane, a chyba nie powinno być opisane tutaj.

```
Definition bijective' {A B : Type} (f : A → B) : Prop :=
  ∀ b : B, ∃! a : A, f a = b.
```

Ćwiczenie Udowodnij, że obie definicje są równoważne.

```
Theorem bijective_bijective' :
  ∀ (A B : Type) (f : A → B),
    bijective f ↔ bijective' f.
```

Ćwiczenie

```
Require Import List.
Import ListNotations.

Fixpoint unary (n : nat) : list unit :=
match n with
    | 0 ⇒ []
    | S n' ⇒ tt :: unary n'
end.
```

Funkcja *unary* reprezentuje liczbę naturalną n za pomocą listy zawierającej n kopii termu *tt*. Udowodnij, że *unary* jest bijekcją.

```
Theorem unary_bij : bijective unary.
```

Jak już powiedzieliśmy, bijekcje dziedziczą właściwości, które mają zarówno injekcje, jak i surjekcje. Wobec tego możemy skonkludować, że złożenie bijekcji jest bijekcją. Nie mają one jednak "dziwnej własciwości".

TODO UWAGA: od teraz twierdzenia, które pozostawię bez dowodu, z automatu stają się ćwiczeniami.

```
Theorem bij_comp :
  ∀ (A B C : Type) (f : A → B) (g : B → C),
    bijective f → bijective g → bijective (f .> g).
```

Bijekcje mają też interpretacje w idei rozmiaru oraz ilości elementów. Jeżeli istnieje bijekcja $f : A \to B$, to znaczy, że typy A oraz B mają dokładnie tyle samo elementów, czyli są "tak samo duże".

Nie powinno nas zatem dziwić, że relacja istnienia bijekcji jest relacją równoważności:

- każdy typ ma tyle samo elementów, co on sam
- jeżeli typ A ma tyle samo elementów co B, to B ma tyle samo elementów, co A
- jeżeli A ma tyle samo elementów co B, a B tyle samo elementów co C, to A ma tyle samo elementów co C

Ćwiczenie Jeżeli między A i B istnieje bijekcja, to mówimy, że A i B są równoliczne (ang. equipotent). Pokaż, że relacja równoliczności jest relacją równoważności. TODO: prerekwizyt: relacje równoważności

Definition *equipotent* $(A\ B$: Type) : Prop :=
  $\exists\, f\ :\ A \to B$, *bijective f*.

Notation Ą ˜ B":= (*equipotent* $A\ B$) (at level 40).

Równoliczność A i B będziemy oznaczać przez $A \neg B$. Nie należy notacji ¬ mylić z notacją ¬ oznaczającej negację logiczną. Ciężko jednak jest je pomylić, gdyż pierwsza zawsze bierze dwa argumenty, a druga tylko jeden.

Theorem *equipotent_refl* :
  $\forall\ A$: Type, $A \neg A$.

Theorem *equipotent_sym* :
  $\forall\ A\ B$: Type, $A \neg B \to B \neg A$.

Theorem *equipotent_trans* :
  $\forall\ A\ B\ C$: Type, $A \neg B \to B \neg C \to A \neg C$.

12.6 Inwolucje

Definition *involutive* $\{A$: Type$\}$ $(f\ :\ A \to A)$: Prop :=
  $\forall\ x\ :\ A,\ f\ (f\ x) = x$.

Kolejnym ważnym (choć nie aż tak ważnym) rodzajem funkcji są inwolucje. Po łacinie “volvere” znaczy “obracać się”. Inwolucja to funkcja, która niczym Chuck Norris wykonuje półobrót — w tym sensie, że zaaplikowanie jej dwukrotnie daje cały obrót, a więc stan wyjściowy.

Mówiąc bardziej po ludzku, inwolucja to funkcja, która jest swoją własną odwrotnością. Spotkaliśmy się już z przykładami inwolucji: najbardziej trywialnym z nich jest funkcja identycznościowa, bardziej oświecającym zaś funkcja *rev*, która odwraca listę — odwrócenie listy dwukrotnie daje wyjściową listę. Inwolucją jest też *negb*.

Theorem *id_inv* :
  $\forall\ A$: Type, *involutive* (*id* A).

Theorem *rev_inv* :
  $\forall\ A$: Type, *involutive* (@*rev* A).

Theorem *negb_inv* : *involutive negb*.

Żeby nie odgrzewać starych kotletów, przyjrzyjmy się funkcji *weird*.

Fixpoint *weird* $\{A$: Type$\}$ $(l\ :\ $*list* $A)$: *list* A :=
match l with
    | [] ⇒ []
    | [x] ⇒ [x]

```
    | x :: y :: t ⇒ y :: x :: weird t
end.
```

```
Theorem weird_inv :
  ∀ A : Type, involutive (@weird A).
```

Funkcja ta zamienia miejscami bloki elementów listy o długości dwa. Nietrudno zauważyć, że dwukrotne takie przestawienie jest identycznością. UWAGA TODO: dowód wymaga specjalnej reguły indukcyjnej.

```
Theorem flip_inv :
  ∀ A : Type, involutive (@flip A A A).
```

Inwolucją jest też kombinator *flip*, który poznaliśmy na początku rozdziału. Przypomnijmy, że zamienia on miejscami argumenty funkcji binarnej. Nie dziwota, że dwukrotna taka zamiana daje oryginalną funkcję.

```
Goal ¬ involutive (@rev nat .> weird).
```

Okazuje się, że złożenie inwolucji wcale nie musi być inwolucją. Wynika to z faktu, że funcje *weird* i *rev* są w pewien sposób niekompatybilne — pierwsze wywołanie każdej z nich przeszkadza drugiemu wywołaniu drugiej z nich odwrócić efekt pierwszego wywołania.

```
Theorem comp_inv :
  ∀ (A : Type) (f g : A → A),
    involutive f → involutive g → f .> g = g .> f → involutive (f .> g).
```

Kryterium to jest rozstrzygające — jeżeli inwolucje komutują ze sobą (czyli są "kompatybilne", f .> g = g .> f), to ich złożenie również jest inwolucją.

```
Theorem inv_bij :
  ∀ (A : Type) (f : A → A), involutive f → bijective f.
```

Ponieważ każda inwolucja ma odwrotność (którą jest ona sama), każda inwolucja jest z automatu bijekcją.

Ćwiczenie Rozważmy funkcje rzeczywiste f(x) = ax^n, f(x) = ax^(-n), f(x) = sin(x), f(x) = cos(x), f(x) = a/x, f(x) = a - x, f(x) = e^x. Które z nich są inwolucjami?

12.7 Uogólnione inwolucje

Pojęcie inwolucji można nieco uogólnić. Żeby to zrobić, przeformułujmy najpierw definicję inwolucji.

```
Definition involutive' {A : Type} (f : A → A) : Prop :=
  f .> f = id A.
```

Nowa definicja głosi, że inwolucja to taka funkcja, że jej złożenie ze sobą jest identycznością. Jeżeli funkcje f .> f i *id A* zaaplikujemy do argumentu x, otrzymamy oryginalną definicję. Nowa definicja jest równoważna starej na mocy aksjomatu ekstensjonalności dla funkcji.

```
Theorem involutive_involutive' :
  ∀ (A : Type) (f : A → A), involutive f ↔ involutive' f.
```

Pójdźmy o krok dalej. Zamiast składania .> użyjmy kombinatora *iter* 2, który ma taki sam efekt.

```
Definition involutive'' {A : Type} (f : A → A) : Prop :=
  iter 2 f = id A.

Theorem involutive'_involutive'' :
  ∀ (A : Type) (f : A → A),
    involutive' f ↔ involutive'' f.
```

Droga do uogólnienia została już prawie przebyta. Nasze dotychczasowe inwolucje nazwiemy uogólnionymi inwolucjami rzędu 2. Definicję uogólnionej inwolucji otrzymamy, zastępując w definicji 2 przez n.

```
Definition gen_involutive {A : Type} (n : nat) (f : A → A)
  : Prop := iter n f = id A.
```

Nie żeby pojęcie to było jakoś szczególnie często spotykane lub nawet przydatne — wymyśliłem je na poczekaniu. Spróbujmy znaleźć jakąś uogólnioną inwolucję o rzędzie większym niż 2.

```
Fixpoint weirder {A : Type} (l : list A) : list A :=
match l with
    | [] ⇒ []
    | [x] ⇒ [x]
    | [x; y] ⇒ [x; y]
    | x :: y :: z :: t ⇒ y :: z :: x :: weirder t
end.

Compute weirder [1; 2; 3; 4; 5].
(*  ===> = 2; 3; 1; 4; 5 : list nat *)

Compute iter 3 weirder [1; 2; 3; 4; 5].
(*  ===> = 1; 2; 3; 4; 5 : list nat *)

Theorem weirder_inv_3 :
  ∀ A : Type, gen_involutive 3 (@weirder A).
```

12.8 Idempotencja

```
Definition idempotent {A : Type} (f : A → A) : Prop :=
  ∀ x : A, f (f x) = f x.
```

Kolejnym rodzajem funkcji są funkcje idempotente. Po łacinie "idem" znaczy "taki sam", zaś "potentia" oznacza "moc". Funkcja idempotentna to taka, której wynik jest taki sam niezależnie od tego, ile razy zostanie zaaplikowana.

Przykłady można mnożyć. Idempotentne jest wciśnięcie guzika w windzie — jeżeli np. wciśniemy "2", to po wjechaniu na drugi piętro kolejne wciśnięcia guzika "2" nie będą miały żadnego efektu.

Idempotentne jest również sortowanie. Jeżeli posortujemy listę, to jest ona posortowana i kolejne sortowania niczego w niej nie zmienią. Problemem sortowania zajmiemy się w przyszłych rozdziałach.

```
Theorem id_idem :
  ∀ A : Type, idempotent (id A).

Theorem const_idem :
  ∀ (A B : Type) (b : B), idempotent (const b).

Theorem take_idem :
  ∀ (A : Type) (n : nat), idempotent (@take A n).
```

Identyczność jest idempotentna — niezrobienie niczego dowolną ilość razy jest wszakże ciągle niezrobieniem niczego. Podobnież funkcja stała jest idempotentna — zwracanie tej samej wartości daje zawsze ten sam efekt, niezależnie od ilości powtórzeń.

Ciekawszym przykładem, który jednak nie powinien cię zaskoczyć, jest funkcja *take* dla dowolnego n : *nat*. Wzięcie n elementów z listy l daje nam listę mającą co najwyżej n elementów. Próba wzięcia n elementów z takiej listy niczego nie zmieni, gdyż jej długość jest mniejsza lub równa ilości elementów, które chcemy wziąć.

```
Theorem comp_idem :
  ∀ (A : Type) (f g : A → A),
    idempotent f → idempotent g → f .> g = g .> f →
      idempotent (f .> g).
```

Jeżeli chodzi o składanie funkcji idempotentnych, sytuacja jest podobna do tej, jaka jest udziałem inwolucji.

Rozdział 13

X5: Relacje

```
Require Import X4.
Require Import FunctionalExtensionality.
Require Import Nat.
Require Import List.
Import ListNotations.
```

Prerekwizyty:

- definicje induktywne
- klasy (?)

W tym rozdziale zajmiemy się badaniem relacji. Poznamy podstawowe rodzaje relacji, ich właściwości, a także zależności i przekształcenia między nimi. Rozdział będzie raczej matematyczny.

13.1 Relacje binarne

Zacznijmy od przypomnienia klasyfikacji zdań, predykatów i relacji:

- zdania to obiekty typu `Prop`. Twierdzą one coś na temat świata: "niebo jest niebieskie", $P \rightarrow Q$ etc. W uproszczeniu możemy myśleć o nich, że są prawdziwe lub fałszywe, co nie znaczy wcale, że można to automatycznie rozstrzygnąć. Udowodnienie zdania P to skonstruowanie obiektu $p : P$. W Coqu zdania służą nam do podawania specyfikacji programów. W celach klasyfikacyjnych możemy uznać, że są to funkcje biorące zero argumentów i zwracające `Prop`.

- predykaty to funkcje typu $A \rightarrow$ `Prop` dla jakiegoś A : `Type`. Można za ich pomocą przedstawiać stwierdzenia na temat właściwości obiektów: "liczba 5 jest parzysta", *odd* 5. Dla niektórych argumentów zwracane przez nie zdania mogą być prawdziwe, a dla innych już nie. Dla celów klasyfikacji uznajemy je za funkcje biorące jeden argument i zwracające `Prop`.

- relacje to funkcje biorące dwa lub więcej argumentów, niekoniecznie o takich samych typach, i zwracające `Prop`. Służą one do opisywania zależności między obiektami, np. "Grażyna jest matką Karyny", *Permutation* $(l \mathrel{++} l')$ $(l' \mathrel{++} ')$. Niektóre kombinacje obiektów mogą być ze sobą w relacji, tzn. zdanie zwracane dla nich przez relację może być prawdziwe, a dla innych nie.

Istnieje jednak zasadnicza różnica między definiowaniem "zwykłych" funkcji oraz definiowaniem relacji: zwykłe funkcje możemy definiować jedynie przez pattern matching i rekurencję, zaś relacje możemy poza tymi metodami definiować także przez indukcję, dzięki czemu możemy wyrazić więcej konceptów niż za pomocą rekursji.

`Definition` *hrel* $(A\ B$: `Type`$)$: `Type` $:= A \to B \to$ `Prop`.

Najważniejszym rodzajem relacji są relacje binarne, czyli relacje biorące dwa argumenty. To właśnie im poświęcimy ten rozdział, pominiemy zaś relacje biorące trzy i więcej argumentów. Określenia "relacja binarna" będę używał zarówno na określenie relacji binarnych heterogenicznych (czyli biorących dwa argumnty różnych typów) jak i na określenie relacji binarnych homogenicznych (czyli biorących dwa argumenty tego samego typu).

13.2 Identyczność relacji

`Definition` *subrelation* $\{A\ B$: `Type`$\}$ $(R\ S$: *hrel* $A\ B)$: `Prop` $:=$
  $\forall\ (a : A)\ (b : B),\ R\ a\ b \to S\ a\ b.$

`Notation` Ż –> S":= (*subrelation* $R\ S$) (`at level` 40).

`Definition` *same_hrel* $\{A\ B$: `Type`$\}$ $(R\ S$: *hrel* $A\ B)$: `Prop` $:=$
  subrelation $R\ S \land$ *subrelation* $S\ R.$

`Notation` Ż <–> S":= (*same_hrel* $R\ S$) (`at level` 40).

Zacznijmy od ustalenia, jakie relacje będziemy uznawać za "identyczne". Okazuje się, że używanie równości *eq* do porównywania zdań nie ma zbyt wiele sensu. Jest tak dlatego, że nie interesuje nas postać owych zdań, a jedynie ich logiczna zawartość.

Z tego powodu właściwym dla zdań pojęciem "identyczności" jest równoważność, czyli $\leftrightarrow$. Podobnie jest w przypadku relacji: uznamy dwie relacje za identyczne, gdy dla wszystkich argumentów zwracają one równoważne zdania.

Formalnie wyrazimy to nieco na około, za pomocą pojęcia subrelacji. R jest subrelacją S, jeżeli $R\ a\ b$ implikuje $S\ a\ b$ dla wszystkich $a : A$ i $b : B$. Możemy sobie wyobrażać, że jeżeli R jest subrelacją S, to w relacji R są ze sobą tylko niektóre pary argumentów, które są w relacji S, a inne nie.

Ćwiczenie `Inductive` *le'* : *nat* $\to$ *nat* $\to$ `Prop` $:=$
  | *le'_0* : $\forall\ n$: *nat*, *le'* $0\ n$
  | *le'_SS* : $\forall\ n\ m$: *nat*, *le'* $n\ m \to$ *le'* $(S\ n)\ (S\ m)$.

Udowodnij, że powyższa definicja *le'* porządku "mniejszy lub równy" na liczbach naturalnych jest tą samą relacją, co *le*. Być może przyda ci się kilka lematów pomocniczych.

```
Lemma le_le'_same : le <-> le'.
```

Uporawszy się z pojęciem "identyczności" relacji możemy przejść dalej, a mianowicie do operacji, jakie możemy wykonywać na relacjach.

13.3 Operacje na relacjach

```
Definition Rcomp
  {A B C : Type} (R : hrel A B) (S : hrel B C) : hrel A C :=
    fun (a : A) (c : C) => exists b : B, R a b /\ S b c.

Definition Rid {A : Type} : hrel A A := @eq A.
```

Podobnie jak w przypadku funkcji, najważniejszą operacją jest składanie relacji, a najważniejszą relacją — równość. Składanie jest łączne, zaś równość jest elementem neutralnym tego składania. Musimy jednak zauważyć, że mówiąc o łączności relacji mamy na myśli coś innego, niż w przypadku funkcji.

```
Lemma Rcomp_assoc :
  forall
    (A B C D : Type) (R : hrel A B) (S : hrel B C) (T : hrel C D),
      Rcomp R (Rcomp S T) <-> Rcomp (Rcomp R S) T.

Lemma Rid_left :
  forall (A B : Type) (R : hrel A B),
    Rcomp (@Rid A) R <-> R.

Lemma Rid_right :
  forall (A B : Type) (R : hrel A B),
    Rcomp R (@Rid B) <-> R.
```

Składanie funkcji jest łączne, gdyż złożenie trzech funkcji z dowolnie rozstawionymi nawiasami daje wynik identyczny w sensie *eq*. Składanie relacji jest łączne, gdyż złożenie trzech relacji z dowolnie rozstawionymi nawiasami daje wynik identyczny w sensie *same_hrel*.

Podobnie sprawa ma się w przypadku stwierdzenia, że *eq* jest elementem nautralnym składania relacji.

```
Definition Rinv {A B : Type} (R : hrel A B) : hrel B A :=
  fun (b : B) (a : A) => R a b.
```

Rinv to operacja, która zamienia miejscami argumenty relacji. Relację *Rinv R* będziemy nazywać relacją odwrotną do *R*.

```
Lemma Rinv_Rcomp :
  forall (A B C : Type) (R : hrel A B) (S : hrel B C),
    Rinv (Rcomp R S) <-> Rcomp (Rinv S) (Rinv R).
```

Lemma $Rinv\_Rid$:
  $\forall$ A : Type, $same\_hrel$ (@Rid A) ($Rinv$ (@Rid A)).

Złożenie dwóch relacji możemy odwrócić, składając ich odwrotności w odwrotnej kolejności. Odwrotnością relacji identycznościowej jest zaś ona sama.

Definition $Rnot$ $\{A\ B$: Type$\}$ (R : $hrel$ A B) : $hrel$ A B :=
  fun (a : A) (b : B) $\Rightarrow$ $\neg$ R a b.

Definition $Rand$ $\{A\ B$: Type$\}$ (R S : $hrel$ A B) : $hrel$ A B :=
  fun (a : A) (b : B) $\Rightarrow$ R a b $\land$ S a b.

Definition Ror $\{A\ B$: Type$\}$ (R S : $hrel$ A B) : $hrel$ A B :=
  fun (a : A) (b : B) $\Rightarrow$ R a b $\lor$ S a b.

Pozostałe trzy operacje na relacjach odpowiadają spójnikom logicznym — mamy więc negację relacji oraz koniunkcję i dysjunkcję dwóch relacji. Zauważ, że operacje te możemy wykonywać jedynie na relacjach o takich samych typach argumentów.

Sporą część naszego badania relacji przeznaczymy na sprawdzanie, jak powyższe operacj mają się do różnych specjalnych rodzajów relacji. Nim to się stanie, zbadajmy jednak właściwości samych operacji.

Definition $RTrue$ $\{A\ B$: Type$\}$: $hrel$ A B :=
  fun (a : A) (b : B) $\Rightarrow$ $True$.

Definition $RFalse$ $\{A\ B$: Type$\}$: $hrel$ A B :=
  fun (a : A) (b : B) $\Rightarrow$ $False$.

Zacznijmy od relacjowych odpowiedników *True* i *False*. Przydadzą się nam one do wyrażania właściwości *Rand* oraz *Ror*.

Lemma $Rnot\_double$:
  $\forall$ (A B : Type) (R : $hrel$ A B),
    R –> $Rnot$ ($Rnot$ R).

Lemma $Rand\_assoc$:
  $\forall$ (A B : Type) (R S T : $hrel$ A B),
    $Rand$ R ($Rand$ S T) <–> $Rand$ ($Rand$ R S) T.

Lemma $Rand\_comm$:
  $\forall$ (A B : Type) (R S : $hrel$ A B),
    $Rand$ R S <–> $Rand$ S R.

Lemma $Rand\_RTrue\_l$:
  $\forall$ (A B : Type) (R : $hrel$ A B),
    $Rand$ $RTrue$ R <–> R.

Lemma $Rand\_RTrue\_r$:
  $\forall$ (A B : Type) (R : $hrel$ A B),
    $Rand$ R $RTrue$ <–> R.

Lemma $Rand\_RFalse\_l$:

```
  ∀ (A B : Type) (R : hrel A B),
    Rand RFalse R <-> RFalse.

Lemma Rand_RFalse_r :
  ∀ (A B : Type) (R : hrel A B),
    Rand R RFalse <-> RFalse.

Lemma Ror_assoc :
  ∀ (A B : Type) (R S T : hrel A B),
    Ror R (Ror S T) <-> Ror (Ror R S) T.

Lemma Ror_comm :
  ∀ (A B : Type) (R S : hrel A B),
    Ror R S <-> Ror S R.

Lemma Ror_RTrue_l :
  ∀ (A B : Type) (R : hrel A B),
    Ror RTrue R <-> RTrue.

Lemma Ror_RTrue_r :
  ∀ (A B : Type) (R : hrel A B),
    Ror R RTrue <-> RTrue.

Lemma Ror_RFalse_l :
  ∀ (A B : Type) (R : hrel A B),
    Ror RFalse R <-> R.

Lemma Ror_RFalse_r :
  ∀ (A B : Type) (R : hrel A B),
    Ror R RFalse <-> R.
```

To nie wszystkie właściwości tych operacji, ale myślę, że widzisz już, dokąd to wszystko zmierza. Jako, że *Rnot*, *Rand* i *Ror* pochodzą bezpośrednio od spójników logicznych *not*, *and* i *or*, to dziedziczą one po nich wszystkie ich właściwości.

Fenomen ten nie jest w żaden sposób specyficzny dla relacji i operacji na nich. TODO: mam nadzieję, że w przyszłych rozdziałach jeszcze się z nim spotkamy. Tymczasem przyjrzyjmy się bliżej specjalnym rodzajom relacji.

13.4 Rodzaje relacji heterogenicznych

```
Class LeftUnique {A B : Type} (R : hrel A B) : Prop :=
{
    left_unique :
      ∀ (a a' : A) (b : B), R a b → R a' b → a = a'
}.

Class RightUnique {A B : Type} (R : hrel A B) : Prop :=
{
```

right_unique :
  $\forall$ (a : A) (b b' : B), R a b $\rightarrow$ R a b' $\rightarrow$ b = b'
}.

Dwoma podstawowymi rodzajami relacji są relacje unikalne z lewej i prawej strony. Relacja lewostronnie unikalna to taka, dla której każde b : B jest w relacji z co najwyżej jednym a : A. Analogicznie definiujemy relacje prawostronnie unikalne.

Instance *LeftUnique_eq* (A : Type) : *LeftUnique* (@*eq* A).

Instance *RightUnique_eq* (A : Type) : *RightUnique* (@*eq* A).

Najbardziej elementarną intuicję stojącą za tymi koncepcjami można przedstawić na przykładzie relacji równości: jeżeli dwa obiekty są równe jakiemuś trzeciemu obiektowi, to muszą być także równe sobie nawzajem.

Pojęcie to jest jednak bardziej ogólne i dotyczy także relacji, które nie są homogeniczne. W szczególności jest ono różne od pojęcia relacji przechodniej, które pojawi się już niedługo.

Instance *LeftUnique_Rcomp* :
  $\forall$ (A B C : Type) (R : *hrel* A B) (S : *hrel* B C),
    LeftUnique R $\rightarrow$ *LeftUnique* S $\rightarrow$ *LeftUnique* (*Rcomp* R S).

Instance *RightUnique_Rcomp* :
  $\forall$ (A B C : Type) (R : *hrel* A B) (S : *hrel* B C),
    RightUnique R $\rightarrow$ *RightUnique* S $\rightarrow$ *RightUnique* (*Rcomp* R S).

Składanie zachowuje oba rodzaje relacji unikalnych. Nie ma tu co za dużo filozofować — żeby się przekonać, narysuj obrazek. TODO.

Instance *LeftUnique_Rinv* :
  $\forall$ (A B : Type) (R : *hrel* A B),
    RightUnique R $\rightarrow$ *LeftUnique* (*Rinv* R).

Instance *RightUnique_Rinv* :
  $\forall$ (A B : Type) (R : *hrel* A B),
    LeftUnique R $\rightarrow$ *RightUnique* (*Rinv* R).

Już na pierwszy rzut oka widać, że pojęcia te są w pewien sposób symetryczne. Aby uchwycić tę symetrię, możemy posłużyć się operacją *Rinv*. Okazuje się, że zamiana miejscami argumentów relacji lewostronnie unikalnej daje relację prawostronnie unikalną i vice versa.

Instance *LeftUnique_Rand* :
  $\forall$ (A B : Type) (R S : *hrel* A B),
    LeftUnique R $\rightarrow$ *LeftUnique* (*Rand* R S).

Instance *RightUnique_Rand* :
  $\forall$ (A B : Type) (R S : *hrel* A B),
    RightUnique R $\rightarrow$ *RightUnique* (*Rand* R S).

Lemma *Ror_not_LeftUnique* :
  $\exists$ (A B : Type) (R S : *hrel* A B),
    LeftUnique R $\land$ *LeftUnique* S $\land$ $\neg$ *LeftUnique* (*Ror* R S).

Lemma $Ror\_not\_RightUnique$:
  $\exists$ (A B : Type) (R S : $hrel$ A B),
    $RightUnique$ R $\land$ $RightUnique$ S $\land$ $\neg$ $RightUnique$ (Ror R S).

Koniunkcja relacji unikalnej z inną daje relację unikalną, ale dysjunkcja nawet dwóch relacji unikalnych nie musi dawać w wyniku relacji unikalnej. Wynika to z interpretacji operacji na relacjach jako operacji na kolekcjach par.

Wyobraźmy sobie, że relacja R : $hrel$ A B to kolekcja par p : $A \times B$. Jeżeli para jest elementem kolekcji, to jej pierwszy komponent jest w relacji R z jej drugim komponentem. Dysjunkcję relacji R i S w takim układzie stanowi kolekcja, która zawiera zarówno pary z kolekcji odpowiadającej R, jak i te z kolekcji odpowiadającej S. Koniunkcja odpowiada kolekcji par, które są zarówno w kolekcji odpowiadającej R, jak i tej odpowiadającej S.

Tak więc dysjunkcja R i S może do R "dorzucić" jakieś pary, ale nie może ich zabrać. Analogicznie, koniunkcja R i S może zabrać pary z R, ale nie może ich dodać.

Teraz interpretacja naszego wyniku jest prosta. W przypadku relacji lewostronnie unikalnych jeżeli każde b : B jest w relacji z co najwyżej jednym a : A, to potencjalne zabranie jakichś par z tej relacji niczego nie zmieni. Z drugiej strony, nawet jeżeli obie relacje są lewostronnie unikalne, to dodanie do R par z S może spowodować wystąpienie powtórzeń, co niszczy unikalność.

Lemma $Rnot\_not\_LeftUnique$:
  $\exists$ (A B : Type) (R : $hrel$ A B),
    $LeftUnique$ R $\land$ $\neg$ $LeftUnique$ ($Rnot$ R).

Lemma $Rnot\_not\_RightUnique$:
  $\exists$ (A B : Type) (R : $hrel$ A B),
    $LeftUnique$ R $\land$ $\neg$ $LeftUnique$ ($Rnot$ R).

Negacja relacji unikalnej również nie musi być unikalna. Spróbuj podać interpretację tego wyniku z punktu widzenia operacji na kolekcjach par.

Ćwiczenie Znajdź przykład relacji, która:

- nie jest unikalna ani lewostronnie, ani prawostronnie
- jest unikalna lewostronnie, ale nie prawostronnie
- jest unikalna prawostronnie, ale nie nie lewostronnie
- jest obustronnie unikalna

Class $LeftTotal$ {A B : Type} (R : $hrel$ A B) : Prop :=
{
    $left\_total$: $\forall$ a : A, $\exists$ b : B, R a b
}.

Class $RightTotal$ {A B : Type} (R : $hrel$ A B) : Prop :=

{
    $right\_total : \forall\ b : B, \exists\ a : A, R\ a\ b$
}.

Kolejnymi dwoma rodzajami heterogenicznych relacji binarnych są relacje lewo- i prawostronnie totalne. Relacja lewostronnie totalna to taka, że każde $a : A$ jest w relacji z jakimś elementem B. Definicja relacji prawostronnie totalnej jest analogiczna.

Za pojęciem tym nie stoją jakieś wielkie intuicje: relacja lewostronnie totalna to po prostu taka, w której żaden element $a : A$ nie jest "osamotniony".

```
Instance LeftTotal_eq :
  ∀ A : Type, LeftTotal (@eq A).

Instance RightTotal_eq :
  ∀ A : Type, RightTotal (@eq A).
```

Równość jest relacją totalną, gdyż każdy term $x : A$ jest równy samemu sobie.

```
Instance LeftTotal_Rcomp :
  ∀ (A B C : Type) (R : hrel A B) (S : hrel B C),
    LeftTotal R → LeftTotal S → LeftTotal (Rcomp R S).

Instance RightTotal_Rcomp :
  ∀ (A B C : Type) (R : hrel A B) (S : hrel B C),
    RightTotal R → RightTotal S → RightTotal (Rcomp R S).

Instance RightTotal_Rinv :
  ∀ (A B : Type) (R : hrel A B),
    LeftTotal R → RightTotal (Rinv R).

Instance LeftTotal_Rinv :
  ∀ (A B : Type) (R : hrel A B),
    RightTotal R → LeftTotal (Rinv R).
```

Między lewo- i prawostronną totalnością występuje podobna symetria jak między dwoma formami unikalności: relacja odwrotna do lewostronnie totalnej jest prawostronnie totalna i vice versa. Totalność jest również zachowywana przez składanie.

```
Lemma Rand_not_LeftTotal :
  ∃ (A B : Type) (R S : hrel A B),
    LeftTotal R ∧ LeftTotal S ∧ ¬ LeftTotal (Rand R S).

Lemma Rand_not_RightTotal :
  ∃ (A B : Type) (R S : hrel A B),
    RightTotal R ∧ RightTotal S ∧ ¬ RightTotal (Rand R S).

Lemma LeftTotal_Ror :
  ∀ (A B : Type) (R S : hrel A B),
    LeftTotal R → LeftTotal (Ror R S).

Lemma RightTotal_Ror :
  ∀ (A B : Type) (R S : hrel A B),
```

$RightTotal\ R \to RightTotal\ (Ror\ R\ S)$.

Związki totalności z koniunkcją i dysjunkcją relacji są podobne jak w przypadku unikalności, lecz tym razem to dysjunkcja zachowuje właściwość, a koniunkcja ją niszczy. Wynika to z tego, że dysjunkcja nie zabiera żadnych par z relacji, więc nie może uszkodzić totalności. Z drugiej strony koniunkcja może zabrać jakąś parę, a wtedy relacja przestaje być totalna.

```
Lemma Rnot_not_LeftTotal :
  ∃ (A B : Type) (R : hrel A B),
    RightTotal R ∧ ¬ RightTotal (Rnot R).

Lemma Rnot_not_RightTotal :
  ∃ (A B : Type) (R : hrel A B),
    RightTotal R ∧ ¬ RightTotal (Rnot R).
```

Negacja relacji totalnej nie może być totalna. Nie ma się co dziwić — negacja wyrzuca z relacji wszystkie pary, których w niej nie było, a więc pozbywa się czynnika odpowiedzialnego za totalność.

Ćwiczenie Znajdź przykład relacji, która:

- nie jest totalna ani lewostronnie, ani prawostronnie
- jest totalna lewostronnie, ale nie prawostronnie
- jest totalna prawostronnie, ale nie nie lewostronnie
- jest obustronnie totalna

Bonusowe punkty za relację, która jest "naturalna", tzn. nie została wymyślona na chama specjalnie na potrzeby zadania.

13.5 Rodzaje relacji heterogenicznych v2

Poznawszy cztery właściwości, jakie relacje mogą posiadać, rozważymy teraz relacje, które posiadają dwie lub więcej z tych właściwości.

```
Class Functional {A B : Type} (R : hrel A B) : Prop :=
{
    F_LT :> LeftTotal R;
    F_RU :> RightUnique R;
}.
```

Lewostronną totalność i prawostronną unikalność możemy połączyć, by uzyskać pojęcie relacji funkcyjnej. Relacja funkcyjna to relacja, która ma właściwości takie, jak funkcje — każdy lewostronny argument $a : A$ jest w relacji z dokładnie jednym $b : B$ po prawej stronie.

```
Instance fun_to_Functional {A B : Type} (f : A → B)
```

: *Functional* (`fun` $(a : A)\ (b : B) \Rightarrow f\ a = b$).

Z każdej funkcji można w prosty sposób zrobić relację funkcyjną, ale bez dodatkowych aksjomatów nie jesteśmy w stanie z relacji funkcyjnej zrobić funkcji. Przemilczając kwestie aksjomatów możemy powiedzieć więc, że relacje funkcyjne odpowiadają funkcjom.

`Instance` *Functional_eq* :
  $\forall$ A : `Type`, *Functional* (@*eq* A).

Równość jest rzecz jasna relacją funkcyjną. Funkcją odpowiadającą relacji @*eq* A jest funkcja identycznościowa @*id* A.

`Instance` *Functional_Rcomp* :
  $\forall$ (A B C : `Type`) (R : *hrel* A B) (S : *hrel* B C),
    Functional R $\rightarrow$ *Functional* S $\rightarrow$ *Functional* (*Rcomp* R S).

Złożenie relacji funkcyjnych również jest relacją funkcyjną. Nie powinno nas to dziwić — wszakże relacje funkcyjne odpowiadają funkcjom, a złożenie funkcji jest przecież funkcją. Jeżeli lepiej mu się przyjrzeć, to okazuje się, że składanie funkcji odpowiada składaniu relacji, a stąd już prosta droga do wniosku, że złożenie relacji funkcyjnych jest relacją funkcyjną.

`Lemma` *Rinv_not_Functional* :
  $\exists$ (A B : `Type`) (R : *hrel* A B),
    Functional R $\land$ $\neg$ *Functional* (*Rinv* R).

Odwrotność relacji funkcyjnej nie musi być funkcyjna. Dobrą wizualicją tego faktu może być np. funkcja f(x) = x^2 na liczbach rzeczywistych. Z pewnością jest to funkcja, a zatem relacja funkcyjna. Widać to na jej wykresie — każdemu punktowi dziedziny odpowiada dokładnie jeden punkt przeciwdziedziny. Jednak po wykonaniu operacji *Rinv*, której odpowiada tutaj obrócenie układu współrzędnych o 90 stopni, nie otrzymujemy wcale wykresu funkcji. Wprost przeciwnie — niektórym punktom z osi X na takim wykresie odpowiadają dwa punkty na osi Y (np. punktowi 4 odpowiadają 2 i -2). Stąd wnioskujemy, że odwrócenie relacji funkcyjnej f nie daje w wyniku relacji funkcyjnej.

`Lemma` *Rand_not_Functional* :
  $\exists$ (A B : `Type`) (R S : *hrel* A B),
    Functional R $\land$ *Functional* S $\land$ $\neg$ *Functional* (*Rand* R S).

`Lemma` *Ror_not_Functional* :
  $\exists$ (A B : `Type`) (R S : *hrel* A B),
    Functional R $\land$ *Functional* S $\land$ $\neg$ *Functional* (*Ror* R S).

`Lemma` *Rnot_not_Functional* :
  $\exists$ (A B : `Type`) (R : *hrel* A B),
    Functional R $\land$ $\neg$ *Functional* (*Rnot* R).

Ani koniunkcje, ani dysjunkcje, ani negacje relacji funkcyjnych nie muszą być wcale relacjami funkcyjnymi. Jest to po części konsekwencją właściwości relacji lewostronnie totalnych i prawostronnie unikalnych: pierwsze mają problem z *Rand*, a drugie z *Ror*, oba zaś z *Rnot*.

Ćwiczenie Możesz zadawać sobie pytanie: po co nam w ogóle pojęcie relacji funkcyjnej, skoro mamy funkcje? Funkcje muszą być obliczalne (ang. computable) i to na mocy definicji, zaś relacje — niekonieczne. Czasem prościej może być najpierw zdefiniować relację, a dopiero później pokazać, że jest funkcyjna. Czasem zdefiniowanie danego bytu jako funkcji może być niemożliwe.

Funkcję Collatza klasycznie definiuje się w ten sposób: jeżeli n jest parzyste, to f(n) = n/2. W przeciwnym przypadku f(n) = 3n + 1.

Zaimplementuj tę funkcję w Coqu. Spróbuj zaimplementować ją zarówno jako funkcję rekurencyjną, jak i relację. Czy twoja funkcja dokładnie odpowiada powyższej specyfikacji? Czy jesteś w stanie pokazać, że twoja relacja jest funkcyjna?

Udowodnij, że f(42) = 1.

```
Class Injective {A B : Type} (R : hrel A B) : Prop :=
{
    I_Fun :> Functional R;
    I_LU :> LeftUnique R;
}.

Instance inj_to_Injective :
  ∀ (A B : Type) (f : A → B),
    injective f → Injective (fun (a : A) (b : B) ⇒ f a = b).
```

Relacje funkcyjne, które są lewostronnie unikalne, odpowiadają funkcjom injektywnym.

```
Instance Injective_eq :
  ∀ A : Type, Injective (@eq A).

Instance Injective_Rcomp :
  ∀ (A B C : Type) (R : hrel A B) (S : hrel B C),
    Injective R → Injective S → Injective (Rcomp R S).

Lemma Rinv_not_Injective :
  ∃ (A B : Type) (R : hrel A B),
    Injective R ∧ ¬ Injective (Rinv R).

Lemma Rand_not_Injective :
  ∃ (A B : Type) (R S : hrel A B),
    Injective R ∧ Injective S ∧ ¬ Injective (Rand R S).

Lemma Ror_not_Injective :
  ∃ (A B : Type) (R S : hrel A B),
    Injective R ∧ Injective S ∧ ¬ Injective (Ror R S).

Lemma Rnot_not_Injective :
  ∃ (A B : Type) (R : hrel A B),
    Injective R ∧ ¬ Injective (Rnot R).
```

Właściwości relacji injektywnych są takie, jak funkcji injektywnych, gdyż te pojęcia ściśle sobie odpowiadają.

Ćwiczenie Udowodnij, że powyższe zdanie nie kłamie.

```
Lemma injective_Injective :
  ∀ (A B : Type) (f : A → B),
    injective f ↔ Injective (fun (a : A) (b : B) ⇒ f a = b).
Proof.
  split.
    compute; intros. repeat split; intros.
      ∃ (f a). reflexivity.
      rewrite ← H0, ← H1. reflexivity.
      apply H. rewrite H0, H1. reflexivity.
    destruct 1 as [[[] []] []]. red. intros.
      apply left_unique0 with (f x').
        assumption.
        reflexivity.
Qed.
```

```
(*  end hide *)
```

```
Class Surjective {A B : Type} (R : hrel A B) : Prop :=
{
    S_Fun :> Functional R;
    S_RT :> RightTotal R;
}.
```

```
Instance sur_to_Surjective :
  ∀ (A B : Type) (f : A → B),
    surjective f → Surjective (fun (a : A) (b : B) ⇒ f a = b).
```

Relacje funkcyjne, które są prawostronnie totalne, odpowiadają funkcjom surjektywnym.

```
Instance Surjective_eq :
  ∀ A : Type, Surjective (@eq A).
```

```
Instance Surjective_Rcomp :
  ∀ (A B C : Type) (R : hrel A B) (S : hrel B C),
    Surjective R → Surjective S → Surjective (Rcomp R S).
```

```
Lemma Rinv_not_Surjective :
  ∃ (A B : Type) (R : hrel A B),
    Surjective R ∧ ¬ Surjective (Rinv R).
```

```
Lemma Rand_not_Surjective :
  ∃ (A B : Type) (R S : hrel A B),
    Surjective R ∧ Surjective S ∧ ¬ Surjective (Rand R S).
```

```
Lemma Ror_not_Surjective :
  ∃ (A B : Type) (R S : hrel A B),
    Surjective R ∧ Surjective S ∧ ¬ Surjective (Ror R S).
```

```
Lemma Rnot_not_Surjective :
```

```
  ∃ (A B : Type) (R : hrel A B),
    Surjective R ∧ ¬ Surjective (Rnot R).
```

Właściwości relacji surjektywnych także są podobne do tych, jakie są udziałem relacji funkcyjnych.

```
Class Bijective {A B : Type} (R : hrel A B) : Prop :=
{
    B_Fun :> Functional R;
    B_LU :> LeftUnique R;
    B_RT :> RightTotal R;
}.

Instance bij_to_Bijective :
  ∀ (A B : Type) (f : A → B),
    bijective f → Bijective (fun (a : A) (b : B) ⇒ f a = b).
```

Relacje funkcyjne, które są lewostronnie totalne (czyli injektywne) oraz prawostronnie totalne (czyli surjektywne), odpowiadają bijekcjom.

```
Instance Bijective_eq :
  ∀ A : Type, Bijective (@eq A).

Instance Bijective_Rcomp :
  ∀ (A B C : Type) (R : hrel A B) (S : hrel B C),
    Bijective R → Bijective S → Bijective (Rcomp R S).

Instance Bijective_Rinv :
  ∀ (A B : Type) (R : hrel A B),
    Bijective R → Bijective (Rinv R).

Lemma Rand_not_Bijective :
  ∃ (A B : Type) (R S : hrel A B),
    Bijective R ∧ Bijective S ∧ ¬ Bijective (Rand R S).

Lemma Ror_not_Bijective :
  ∃ (A B : Type) (R S : hrel A B),
    Bijective R ∧ Bijective S ∧ ¬ Bijective (Ror R S).

Lemma Rnot_not_Bijective :
  ∃ (A B : Type) (R : hrel A B),
    Bijective R ∧ ¬ Bijective (Rnot R).
```

Właściwości relacji bijektywnych różnią się jednym szalenie istotnym detalem od właściwości relacji funkcyjnych, injektywnych i surjektywnych: odwrotność relacji bijektywnej jest relacją bijektywną.

13.6 Rodzaje relacji homogenicznych

```
Definition rel (A : Type) : Type := hrel A A.
```

Relacje homogeniczne to takie, których wszystkie argumenty są tego samego typu. Warunek ten pozwala nam na wyrażenie całej gamy nowych właściwości, które relacje takie mogą posiadać.

Uwaga terminologiczna: w innych pracach to, co nazwałem *Antireflexive* bywa zazwyczaj nazywane *Irreflexive*. Ja przyjąłem następujące reguły tworzenia nazw różnych rodzajów relacji:

- "podstawowa" własność nie ma przedrostka, np. "zwrotna", "reflexive"
- zanegowana własność ma przedrostek "nie" (lub podobny w nazwach angielskich), np. "niezwrotny", "irreflexive"
- przeciwieństwo tej właściwości ma przedrostek "anty-" (po angielsku "anti-"), np. "antyzwrotna", "antireflexive"

13.6.1 Zwrotność

```
Class Reflexive {A : Type} (R : rel A) : Prop :=
{
    reflexive : ∀ x : A, R x x
}.
Class Irreflexive {A : Type} (R : rel A) : Prop :=
{
    irreflexive : ∃ x : A, ¬ R x x
}.
Class Antireflexive {A : Type} (R : rel A) : Prop :=
{
    antireflexive : ∀ x : A, ¬ R x x
}.
```

Relacja R jest zwrotna (ang. reflexive), jeżeli każdy $x : A$ jest w relacji sam ze sobą. Przykładem ze świata rzeczywistego może być relacja "x jest blisko y". Jest oczywiste, że każdy jest blisko samego siebie.

```
Instance Reflexive_empty :
  ∀ R : rel Empty_set, Reflexive R.
```

Okazuje się, że wszystkie relacje na *Empty_set* (a więc także na wszystkich innych typach pustych) są zwrotne. Nie powinno cię to w żaden sposób zaskakiwać — jest to tzw. pusta prawda (ang. vacuous truth), zgodnie z którą wszystkie zdania kwantyfikowane uniwersalnie po typie pustym są prawdziwe. Wszyscy w pustym pokoju są debilami.

```
Instance Reflexive_eq {A : Type} : Reflexive (@eq A).
```

```
Instance Reflexive_RTrue :
  ∀ A : Type, Reflexive (@RTrue A A).
```

```
Lemma RFalse_nonempty_not_Reflexive :
  ∀ A : Type, A → ¬ Reflexive (@RFalse A A).
```

Najważniejszym przykładem relacji zwrotnej jest równość. *eq* jest relacją zwrotną, gdyż ma konstruktor *eq_refl*, który głosi, że każdy obiekt jest równy samemu sobie. Zwrotna jest też relacja *RTrue*, gdyż każdy obiekt jest w jej przypadku w relacji z każdym, a więc także z samym sobą. Zwrotna nie jest za to relacja *RFalse* na typie niepustym, gdyż tam żaden obiekt nie jest w relacji z żadnym, a więc nie może także być w relacji z samym sobą.

```
Lemma eq_subrelation_Reflexive :
  ∀ (A : Type) (R : rel A), Reflexive R →
    subrelation (@eq A) R.
```

Równość jest "najmniejszą" relacją zwrotną w tym sensie, że jest ona subrelacją każdej relacji zwrotnej. Intuicyjnym uzasadnieniem jest fakt, że w definicji *eq* poza konstruktorem *eq_refl*, który daje zwrotność, nie ma niczego innego.

```
Instance Reflexive_Rcomp :
  ∀ (A : Type) (R S : rel A),
    Reflexive R → Reflexive S → Reflexive (Rcomp R S).

Instance Reflexive_Rinv :
  ∀ (A : Type) (R : rel A),
    Reflexive R → Reflexive (Rinv R).

Instance Reflexive_Rand :
  ∀ (A : Type) (R S : rel A),
    Reflexive R → Reflexive S → Reflexive (Rand R S).

Instance Reflexive_Ror :
  ∀ (A : Type) (R S : rel A),
    Reflexive R → Reflexive (Ror R S).
```

Jak widać, złożenie, odwrotność i koniunkcja relacji zwrotnych są zwrotne. Dysjunkcja posiada natomiast dużo mocniejszą właściwość: dysjunkcja dowolnej relacji z relacją zwrotną daje relację zwrotną. Tak więc dysjunkcja R z *eq* pozwala nam łatwo "dodać" zwrotność do R. Słownie dysjunkcja z *eq* odpowiada zwrotowi "lub równy", który możemy spotkać np. w wyrażeniach "mniejszy lub równy", "większy lub równy".

Właściwością odwrotną do zwrotności jest antyzwrotność. Relacja antyzwrotna to taka, że żaden $x : A$ nie jest w relacji sam ze sobą.

```
Instance Antireflexive_neq :
  ∀ (A : Type), Antireflexive (fun x y : A ⇒ x ≠ y).

Instance Antireflexive_lt : Antireflexive lt.
```

Typowymi przykładami relacji antyzwrotnych są nierówność $\neq$ oraz porządek "mniejszy niż" ($<$) na liczbach naturalnych. Ze względu na sposób działania ludzkiego mózgu antyzwrotna jest cała masa relacji znanych nam z codziennego życia: "x jest matką y", "x jest ojcem y", "x jest synem y", "x jest córką y", "x jest nad y", "x jest pod y", "x jest za y", "x jest przed y", etc.

```
Lemma Antireflexive_empty :
  ∀ R : rel Empty_set, Antireflexive R.
```

```
Lemma eq_nonempty_not_Antireflexive :
  ∀ A : Type, A → ¬ Antireflexive (@eq A).
```

```
Lemma RTrue_nonempty_not_Antireflexive :
  ∀ A : Type, A → ¬ Antireflexive (@RTrue A A).
```

```
Instance Antireflexive_RFalse :
  ∀ A : Type, Antireflexive (@RFalse A A).
```

Równość na typie niepustym nie jest antyzwrotna, gdyż jest zwrotna (wzajemne związki między tymi dwoma pojęciami zbadamy już niedługo). Antyzwrotna nie jest także relacja *RTrue* na typie niepustym, gdyż co najmniej jeden element jest w relacji z samym sobą. Antyzwrotna jest za to relacja pusta (*RFalse*).

```
Lemma Rcomp_not_Antireflexive :
  ∃ (A : Type) (R S : rel A),
    Antireflexive R ∧ Antireflexive S ∧
    ¬ Antireflexive (Rcomp R S).
```

```
Instance Antireflexive_Rinv :
  ∀ (A : Type) (R : rel A),
    Antireflexive R → Antireflexive (Rinv R).
```

```
Instance Antireflexive_Rand :
  ∀ (A : Type) (R S : rel A),
    Antireflexive R → Antireflexive (Rand R S).
```

```
Instance Antireflexive_Ror :
  ∀ (A : Type) (R S : rel A),
    Antireflexive R → Antireflexive S → Antireflexive (Ror R S).
```

Złożenie relacji antyzwrotnych nie musi być antyzwrotne, ale odwrotność i dysjunkcja już tak, zaś koniunkcja dowolnej relacji z relacją antyzwrotną daje nam relację antyzwrotną. Dzięki temu możemy dowolnej relacji R "zabrać" zwrotność koniunkcjując ją z $\neq$.

Kolejną właściwością jest niezwrotność. Relacja niezwrotna to taka, która nie jest zwrotna. Zauważ, że pojęcie to zasadniczo różni się od pojęcia relacji antyzwrotnej: tutaj mamy kwantyfikator $\exists$, tam zaś $\forall$.

```
Instance Irreflexive_neq_nonempty :
  ∀ A : Type, A → Irreflexive (Rnot (@eq A)).
```

```
Instance Irreflexive_gt : Irreflexive gt.
```

Typowym przykładem relacji niezwrotnej jest nierówność $x \neq y$. Jako, że każdy obiekt jest równy samemu sobie, to żaden obiekt nie może być nierówny samemu sobie. Zauważ jednak, że typ A musi być niepusty, gdyż w przeciwnym wypadku nie mamy czego dać kwantyfikatorowi $\exists$.

Innym przykładem relacji niezwrotnej jest porządek "większy niż" na liczbach naturalnych. Porządkami zajmiemy się już niedługo.

```
Lemma empty_not_Irreflexive :
  ∀ R : rel Empty_set, ¬ Irreflexive R.

Lemma eq_empty_not_Irreflexive :
  ¬ Irreflexive (@eq Empty_set).

Lemma eq_nonempty_not_Irreflexive :
  ∀ A : Type, A → ¬ Irreflexive (@eq A).
```

Równość jest zwrotna, a więc nie może być niezwrotna. Zauważ jednak, że musimy podać aż dwa osobne dowody tego faktu: jeden dla typu pustego *Empty_set*, a drugi dla dowolnego typu niepustego. Wynika to z tego, że nie możemy sprawdzić, czy dowolny typ A jest pusty, czy też nie.

```
Lemma RTrue_empty_not_Irreflexive :
  ¬ Irreflexive (@RTrue Empty_set Empty_set).

Lemma RTrue_nonempty_not_Irreflexive :
  ∀ A : Type, A → ¬ Irreflexive (@RTrue A A).

Lemma RFalse_empty_not_Irreflexive :
  ¬ Irreflexive (@RFalse Empty_set Empty_set).

Instance Irreflexive_RFalse_nonempty :
  ∀ A : Type, A → Irreflexive (@RFalse A A).
```

Podobnej techniki możemy użyć, aby pokazać, że relacja pełna (*RTrue*) nie jest niezwrotna. Inaczej jest jednak w przypadku *RFalse* — na typie pustym nie jest ona niezwrotna, ale na dowolnym typie niepustym już owszem.

```
Lemma Rcomp_not_Irreflexive :
  ∃ (A : Type) (R S : rel A),
    Irreflexive R ∧ Irreflexive S ∧ ¬ Irreflexive (Rcomp R S).
```

Złożenie relacji niezwrotnych nie musi być niezwrotne. Przyjrzyj się uważnie definicji *Rcomp*, a z pewnością uda ci się znaleźć jakiś kontrprzykład.

```
Instance Irreflexive_Rinv :
  ∀ (A : Type) (R : rel A),
    Irreflexive R → Irreflexive (Rinv R).

Instance Irreflexive_Rand :
  ∀ (A : Type) (R S : rel A),
    Irreflexive R → Irreflexive (Rand R S).

Lemma Ror_not_Irreflexive :
  ∃ (A : Type) (R S : rel A),
    Irreflexive R ∧ Irreflexive S ∧ ¬ Irreflexive (Ror R S).
```

Odwrotność relacji niezwrotnej jest niezwrotna. Koniunkcja dowolnej relacji z relacją niezwrotną daje relację niezwrotną. Tak więc za pomocą koniunkcji i dysjunkcji możemy łatwo dawać i zabierać zwrotność różnym relacjom. Okazuje się też, że dysjunkcja nie zachowuje niezwrotności.

Na zakończenie zbadajmy jeszcze, jakie związki zachodzą pomiędzy zwrotnością, antyzwrotnością i niezwrotnością.

```
Instance Reflexive_Rnot :
  ∀ (A : Type) (R : rel A),
    Antireflexive R → Reflexive (Rnot R).
```

```
Instance Antireflexive_Rnot :
  ∀ (A : Type) (R : rel A),
    Reflexive R → Antireflexive (Rnot R).
```

Podstawowa zależność między nimi jest taka, że negacja relacji zwrotnej jest antyzwrotna, zaś negacja relacji antyzwrotnej jest zwrotna.

```
Lemma Reflexive_Antireflexive_empty :
  ∀ R : rel Empty_set, Reflexive R ∧ Antireflexive R.
```

```
Lemma Reflexive_Antireflexive_nonempty :
  ∀ (A : Type) (R : rel A),
    A → Reflexive R → Antireflexive R → False.
```

Każda relacja na typie pustym jest jednocześnie zwrotna i antyzwrotna, ale nie może taka być żadna relacja na typie niepustym.

```
Instance Irreflexive_nonempty_Antireflexive :
  ∀ (A : Type) (R : rel A),
    A → Antireflexive R → Irreflexive R.
```

Związek między niezwrotnością i antyzwrotnością jest nadzwyczaj prosty: każda relacja antyzwrotna na typie niepustym jest też niezwrotna.

13.6.2 Symetria

```
Class Symmetric {A : Type} (R : rel A) : Prop :=
{
    symmetric : ∀ x y : A, R x y → R y x
}.
Class Antisymmetric {A : Type} (R : rel A) : Prop :=
{
    antisymmetric : ∀ x y : A, R x y → ¬ R y x
}.
Class Asymmetric {A : Type} (R : rel A) : Prop :=
{
    asymmetric : ∃ x y : A, R x y ∧ ¬ R y x
}.
```

Relacja jest symetryczna, jeżeli kolejność podawania argumentów nie ma znaczenia. Przykładami ze świata rzeczywistego mogą być np. relacje "jest blisko", "jest obok", "jest naprzeciwko".

```
Lemma Symmetric_char :
  ∀ (A : Type) (R : rel A),
    Symmetric R ↔ same_hrel (Rinv R) R.
```

Alterntywną charakteryzacją symetrii może być stwierdzenie, że relacja symetryczna to taka, która jest swoją własną odwrotnością.

```
Instance Symmetric_eq :
  ∀ A : Type, Symmetric (@eq A).

Instance Symmetric_RTrue :
  ∀ A : Type, Symmetric (@RTrue A A).

Instance Symmetric_RFalse :
  ∀ A : Type, Symmetric (@RFalse A A).
```

Równość, relacja pełna i pusta są symetryczne.

```
Lemma Rcomp_not_Symmetric :
  ∃ (A : Type) (R S : rel A),
    Symmetric R ∧ Symmetric S ∧ ¬ Symmetric (Rcomp R S).
```

Złożenie relacji symetrycznych nie musi być symetryczne.

```
Instance Symmetric_Rinv :
  ∀ (A : Type) (R : rel A),
    Symmetric R → Symmetric (Rinv R).

Instance Symmetric_Rand :
  ∀ (A : Type) (R S : rel A ),
    Symmetric R → Symmetric S → Symmetric (Rand R S).

Instance Symmetric_Ror :
  ∀ (A : Type) (R S : rel A ),
    Symmetric R → Symmetric S → Symmetric (Ror R S).

Instance Symmetric_Rnot :
  ∀ (A : Type) (R : rel A ),
    Symmetric R → Symmetric (Rnot R).
```

Pozostałe operacje (odwracanie, koniunkcja, dysjunkcja, negacja) zachowują symetrię.

Relacja antysymetryczna to przeciwieństwo relacji symetrycznej — jeżeli x jest w relacji z y, to y nie może być w relacji z x. Sporą klasę przykładów stanowią różne relacje służące do porównywania: "x jest wyższy od y", "x jest silniejszy od y", "x jest bogatszy od y".

```
Lemma Antisymmetric_empty :
  ∀ R : rel Empty_set, Antisymmetric R.

Lemma eq_nonempty_not_Antisymmetric :
```

$\forall\ A$: Type, $A \to \neg$ *Antisymmetric* (@*eq* A).

Lemma *RTrue_nonempty_not_Antisymmetric* :
  $\forall\ A$: Type, $A \to \neg$ *Antisymmetric* (@*RTrue* A A).

Instance *RFalse_Antiymmetric* :
  $\forall\ A$: Type, *Antisymmetric* (@*RFalse* A A).

Każda relacja na typie pustym jest antysymetryczna. Równość nie jest antysymetryczna, podobnie jak relacja pełna (ale tylko na typie niepustym). Relacja pusta jest antysymetryczna, gdyż przesłanka $R\ x\ y$ występująca w definicji antysymetrii jest zawsze fałszywa.

Lemma *Rcomp_not_Antisymmetric* :
  $\exists$ (A : Type) (R S : *rel* A),
    Antisymmetric R $\wedge$ *Antisymmetric* S $\wedge$
    $\neg$ *Antisymmetric* (*Rcomp* R S).

Instance *Antisymmetric_Rinv* :
  $\forall$ (A : Type) (R : *rel* A),
    Antisymmetric $R \to$ *Antisymmetric* (*Rinv* R).

Instance *Antisymmetric_Rand* :
  $\forall$ (A : Type) (R S : *rel* A),
    Antisymmetric $R \to$ *Antisymmetric* (*Rand* R S).

Lemma *Ror_not_Antisymmetric* :
  $\exists$ (A : Type) (R S : *rel* A),
    Antisymmetric R $\wedge$ *Antisymmetric* S $\wedge$
    $\neg$ *Antisymmetric* (*Ror* R S).

Lemma *Rnot_not_Antisymmetric* :
  $\exists$ (A : Type) (R : *rel* A),
    Antisymmetric R $\wedge$ $\neg$ *Antisymmetric* (*Rnot* R).

Lemma *empty_not_Asymmetric* :
  $\forall$ R : *rel Empty_set*, $\neg$ *Asymmetric* R.

Lemma *Rcomp_not_Asymmetric* :
  $\exists$ (A : Type) (R S : *rel* A),
    Asymmetric R $\wedge$ *Asymmetric* S $\wedge$ $\neg$ *Asymmetric* (*Rcomp* R S).

Instance *Asymmetric_Rinv* :
  $\forall$ (A : Type) (R : *rel* A),
    Asymmetric $R \to$ *Asymmetric* (*Rinv* R).

Lemma *Rand_not_Asymmetric* :
  $\exists$ (A : Type) (R S : *rel* A),
    Asymmetric R $\wedge$ *Asymmetric* S $\wedge$ $\neg$ *Asymmetric* (*Rand* R S).

Lemma *Ror_not_Asymmetric* :
  $\exists$ (A : Type) (R S : *rel* A),
    Asymmetric R $\wedge$ *Asymmetric* S $\wedge$ $\neg$ *Asymmetric* (*Ror* R S).

13.6.3 Przechodniość

Class *Transitive* $\{A : \texttt{Type}\}$ $(R : rel\ A)$: Prop :=
{
  transitive : $\forall\ x\ y\ z : A,\ R\ x\ y \to R\ y\ z \to R\ x\ z$
}.

Instance *Transitive_eq* :
  $\forall\ A : \texttt{Type},\ Transitive\ (@eq\ A)$.

Lemma *Rcomp_not_Transitive* :
  $\exists\ (A : \texttt{Type})\ (R\ S : rel\ A),$
    $Transitive\ R \land Transitive\ S \land \neg\ Transitive\ (Rcomp\ R\ S)$.

Instance *Transitive_Rinv* :
  $\forall\ (A : \texttt{Type})\ (R : rel\ A),$
    $Transitive\ R \to Transitive\ (Rinv\ R)$.

Instance *Transitive_Rand* :
  $\forall\ (A : \texttt{Type})\ (R\ S : rel\ A),$
    $Transitive\ R \to Transitive\ S \to Transitive\ (Rand\ R\ S)$.

Lemma *Ror_not_Transitive* :
  $\exists\ (A : \texttt{Type})\ (R\ S : rel\ A),$
    $Transitive\ R \land Transitive\ S \land \neg\ Transitive\ (Ror\ R\ S)$.

Lemma *Rnot_not_Transitive* :
  $\exists\ (A : \texttt{Type})\ (R : rel\ A),$
    $Transitive\ R \land \neg\ Transitive\ (Rnot\ R)$.

13.6.4 Inne

Class *Total* $\{A : \texttt{Type}\}$ $(R : rel\ A)$: Prop :=
{
  total : $\forall\ x\ y : A,\ R\ x\ y \lor R\ y\ x$
}.

Instance *Total_Rinv* :
  $\forall\ (A : \texttt{Type})\ (R : rel\ A),$
    $Total\ R \to Total\ (Rinv\ R)$.

Instance *Total_Ror* :
  $\forall\ (A : \texttt{Type})\ (R\ S : rel\ A),$
    $Total\ R \to Total\ S \to Total\ (Ror\ R\ S)$.

Lemma *Rnot_not_Total* :
  $\exists\ (A : \texttt{Type})\ (R : rel\ A),$
    $Total\ R \land \neg\ Total\ (Rnot\ R)$.

```
Instance Total_Reflexive :
  ∀ (A : Type) (R : rel A),
    Total R → Reflexive R.

Class Trichotomous {A : Type} (R : rel A) : Prop :=
{
    trichotomous : ∀ x y : A, R x y ∨ x = y ∨ R y x
}.

Instance Trichotomous_empty :
  ∀ R : rel Empty_set, Trichotomous R.

Instance Trichotomous_eq_singleton :
  ∀ A : Type, (∀ x y : A, x = y) → Trichotomous (@eq A).

Instance Total_Trichotomous :
  ∀ (A : Type) (R : rel A),
    Total R → Trichotomous R.

Lemma eq_not_Trichotomous :
  ∃ A : Type, ¬ Trichotomous (@eq A).

Instance Trichotomous_Rinv :
  ∀ (A : Type) (R : rel A),
    Trichotomous R → Trichotomous (Rinv R).

Lemma Rnot_not_Trichotomous :
  ∃ (A : Type) (R : rel A),
    Trichotomous R ∧ ¬ Trichotomous (Rnot R).

Class Dense {A : Type} (R : rel A) : Prop :=
{
    dense : ∀ x y : A, R x y → ∃ z : A, R x z ∧ R z y
}.

Instance Dense_eq :
  ∀ A : Type, Dense (@eq A).

Instance Dense_Rinv :
  ∀ (A : Type) (R : rel A),
    Dense R → Dense (Rinv R).

Instance Dense_Ror :
  ∀ (A : Type) (R S : rel A),
    Dense R → Dense S → Dense (Ror R S).
```

13.7 Relacje równoważności

```
Class Equivalence {A : Type} (R : rel A) : Prop :=
```

```
{
    Equivalence_Reflexive :> Reflexive R;
    Equivalence_Symmetric :> Symmetric R;
    Equivalence_Transitive :> Transitive R;
}.
Instance Equivalence_eq :
  ∀ A : Type, Equivalence (@eq A).

Instance Equivalence_Rinv :
  ∀ (A : Type) (R : rel A),
    Equivalence R → Equivalence (Rinv R).
Instance Equivalence_Rand :
  ∀ (A : Type) (R S : rel A),
    Equivalence R → Equivalence S → Equivalence (Rand R S).
```

13.8 Słabe relacje homogeniczne

```
Class WeakAntisymmetric {A : Type} (R : rel A) : Prop :=
{
    wantisymmetric : ∀ x y : A, R x y → R y x → x = y
}.
Instance WeakAntisymmetric_eq :
  ∀ A : Type, WeakAntisymmetric (@eq A).
Lemma Rcomp_not_WeakAntisymmetric :
  ∃ (A : Type) (R S : rel A),
    WeakAntisymmetric R ∧ WeakAntisymmetric S ∧
      ¬ WeakAntisymmetric (Rcomp R S).
Instance WeakAntisymmetric_Rinv :
  ∀ (A : Type) (R : rel A),
    WeakAntisymmetric R → WeakAntisymmetric (Rinv R).
Instance WeakAntisymmetric_Rand :
  ∀ (A : Type) (R S : rel A),
    WeakAntisymmetric R → WeakAntisymmetric S →
      WeakAntisymmetric (Rand R S).
Lemma Ror_not_WeakAntisymmetric :
  ∃ (A : Type) (R S : rel A),
    WeakAntisymmetric R ∧ WeakAntisymmetric S ∧
      ¬ WeakAntisymmetric (Ror R S).
Lemma Rnot_not_WeakAntisymmetric :
```

```
  ∃ (A : Type) (R : rel A),
    WeakAntisymmetric R ∧ ¬ WeakAntisymmetric (Rnot R).

Class WeakAntisymmetric' {A : Type} {E : rel A}
  (H : Equivalence E) (R : rel A) : Prop :=
{
    wasym : ∀ x y : A, R x y → R y x → E x y
}.

Instance WeakAntisymmetric_equiv :
  ∀ (A : Type) (E : rel A) (H : Equivalence E),
    WeakAntisymmetric' H E.

Lemma Rcomp_not_WeakAntisymmetric' :
  ∃ (A : Type) (E R S : rel A), ∀ H : Equivalence E,
    WeakAntisymmetric' H R ∧ WeakAntisymmetric' H S ∧
      ¬ WeakAntisymmetric' H (Rcomp R S).

Instance WeakAntisymmetric'_Rinv :
  ∀ (A : Type) (E : rel A) (H : Equivalence E) (R : rel A),
    WeakAntisymmetric' H R → WeakAntisymmetric' H (Rinv R).

Instance WeakAntisymmetric'_Rand :
  ∀ (A : Type) (E : rel A) (H : Equivalence E) (R S : rel A),
    WeakAntisymmetric' H R → WeakAntisymmetric' H S →
      WeakAntisymmetric' H (Rand R S).

Lemma Ror_not_WeakAntisymmetric' :
  ∃ (A : Type) (E R S : rel A), ∀ H : Equivalence E,
    WeakAntisymmetric' H R ∧ WeakAntisymmetric' H S ∧
      ¬ WeakAntisymmetric' H (Ror R S).

Lemma Rnot_not_WeakAntisymmetric' :
  ∃ (A : Type) (E R : rel A), ∀ H : Equivalence E,
    WeakAntisymmetric' H R ∧ ¬ WeakAntisymmetric' H (Rnot R).
```

13.9 Złożone relacje homogeniczne

```
Class Preorder {A : Type} (R : rel A) : Prop :=
{
    Preorder_refl :> Reflexive R;
    Preorder_trans :> Transitive R;
}.

Class PartialOrder {A : Type} (R : rel A) : Prop :=
{
    PartialOrder_Preorder :> Preorder R;
```

```
    PartialOrder_WeakAntisymmetric :> WeakAntisymmetric R;
}.
Class TotalOrder {A : Type} (R : rel A) : Prop :=
{
    TotalOrder_PartialOrder :> PartialOrder R;
    TotalOrder_Total : Total R;
}.
Class StrictPreorder {A : Type} (R : rel A) : Prop :=
{
    StrictPreorder_Antireflexive :> Antireflexive R;
    StrictPreorder_Transitive :> Transitive R;
}.
Class StrictPartialOrder {A : Type} (R : rel A) : Prop :=
{
    StrictPartialOrder_Preorder :> StrictPreorder R;
    StrictPartialOrder_WeakAntisymmetric :> Antisymmetric R;
}.
Class StrictTotalOrder {A : Type} (R : rel A) : Prop :=
{
    StrictTotalOrder_PartialOrder :> StrictPartialOrder R;
    StrictTotalOrder_Total : Total R;
}.
```

13.10 Domknięcia

```
Inductive refl_clos {A : Type} (R : rel A) : rel A :=
    | rc_step : ∀ x y : A, R x y → refl_clos R x y
    | rc_refl : ∀ x : A, refl_clos R x x.

Instance Reflexive_rc :
  ∀ (A : Type) (R : rel A), Reflexive (refl_clos R).

Lemma subrelation_rc :
  ∀ (A : Type) (R : rel A), subrelation R (refl_clos R).

Lemma rc_smallest :
  ∀ (A : Type) (R S : rel A),
    subrelation R S → Reflexive S → subrelation (refl_clos R) S.

Lemma rc_idempotent :
  ∀ (A : Type) (R : rel A),
    refl_clos (refl_clos R) <-> refl_clos R.

Lemma rc_Rinv :
```

```
  ∀ (A : Type) (R : rel A),
    Rinv (refl_clos (Rinv R)) <-> refl_clos R.
Inductive symm_clos {A : Type} (R : rel A) : rel A :=
    | sc_step :
        ∀ x y : A, R x y → symm_clos R x y
    | sc_symm :
        ∀ x y : A, symm_clos R x y → symm_clos R y x.

Instance Symmetric_sc :
  ∀ (A : Type) (R : rel A), Symmetric (symm_clos R).
Lemma subrelation_sc :
  ∀ (A : Type) (R : rel A), subrelation R (symm_clos R).
Lemma sc_smallest :
  ∀ (A : Type) (R S : rel A),
    subrelation R S → Symmetric S → subrelation (symm_clos R) S.
Lemma sc_idempotent :
  ∀ (A : Type) (R : rel A),
    symm_clos (symm_clos R) <-> symm_clos R.
Lemma sc_Rinv :
  ∀ (A : Type) (R : rel A),
    Rinv (symm_clos (Rinv R)) <-> symm_clos R.
Inductive trans_clos {A : Type} (R : rel A) : rel A :=
    | tc_step :
        ∀ x y : A, R x y → trans_clos R x y
    | tc_trans :
        ∀ x y z : A,
          trans_clos R x y → trans_clos R y z → trans_clos R x z.

Instance Transitive_tc :
  ∀ (A : Type) (R : rel A), Transitive (trans_clos R).
Lemma subrelation_tc :
  ∀ (A : Type) (R : rel A), subrelation R (trans_clos R).
Lemma tc_smallest :
  ∀ (A : Type) (R S : rel A),
    subrelation R S → Transitive S →
      subrelation (trans_clos R) S.
Lemma tc_idempotent :
  ∀ (A : Type) (R : rel A),
    trans_clos (trans_clos R) <-> trans_clos R.
Lemma tc_Rinv :
```

$\forall$ (A : `Type`) (R : *rel* A),
  Rinv (*trans_clos* (*Rinv* R)) <-> *trans_clos* R.

`Inductive` *equiv_clos* {A : `Type`} (R : *rel* A) : *rel* A :=
| *ec_step* :
    $\forall$ x y : A, R x y $\rightarrow$ *equiv_clos* R x y
| *ec_refl* :
    $\forall$ x : A, *equiv_clos* R x x
| *ec_symm* :
    $\forall$ x y : A, *equiv_clos* R x y $\rightarrow$ *equiv_clos* R y x
| *ec_trans* :
    $\forall$ x y z : A,
      equiv_clos R x y $\rightarrow$ *equiv_clos* R y z $\rightarrow$ *equiv_clos* R x z.

`Instance` *Equivalence_ec* :
  $\forall$ (A : `Type`) (R : *rel* A), *Equivalence* (*equiv_clos* R).

`Lemma` *subrelation_ec* :
  $\forall$ (A : `Type`) (R : *rel* A), *subrelation* R (*equiv_clos* R).

`Lemma` *ec_smallest* :
  $\forall$ (A : `Type`) (R S : *rel* A),
    subrelation R S $\rightarrow$ *Equivalence* S $\rightarrow$
      subrelation (*equiv_clos* R) S.

`Lemma` *ec_idempotent* :
  $\forall$ (A : `Type`) (R : *rel* A),
    equiv_clos (*equiv_clos* R) <-> *equiv_clos* R.

`Lemma` *ec_Rinv* :
  $\forall$ (A : `Type`) (R : *rel* A),
    Rinv (*equiv_clos* (*Rinv* R)) <-> *equiv_clos* R.

Domknięcie zwrotnosymetryczne

`Definition` *rsc* {A : `Type`} (R : *rel* A) : *rel* A :=
  refl_clos (*symm_clos* R).

`Instance` *Reflexive_rsc* :
  $\forall$ (A : `Type`) (R : *rel* A), *Reflexive* (*rsc* R).

`Instance` *Symmetric_rsc* :
  $\forall$ (A : `Type`) (R : *rel* A), *Symmetric* (*rsc* R).

`Lemma` *subrelation_rsc* :
  $\forall$ (A : `Type`) (R : *rel* A), *subrelation* R (*rsc* R).

`Lemma` *rsc_smallest* :
  $\forall$ (A : `Type`) (R S : *rel* A),

subrelation R S → *Reflexive S* → *Symmetric S* →
subrelation (*rsc R*) *S*.

`Lemma` *rsc_idempotent* :
∀ (*A* : `Type`) (*R* : *rel A*),
rsc (*rsc R*) <–> *rsc R*.

`Lemma` *rsc_Rinv* :
∀ (*A* : `Type`) (*R* : *rel A*),
Rinv (*rsc* (*Rinv R*)) <–> *rsc R*.

Domknięcie równoważnościowe v2

`Definition` *rstc* {*A* : `Type`} (*R* : *rel A*) : *rel A* :=
trans_clos (*symm_clos* (*refl_clos R*)).

`Instance` *Equivalence_rstc* :
∀ (*A* : `Type`) (*R* : *rel A*),
Equivalence (*trans_clos* (*symm_clos* (*refl_clos R*))).

`Lemma` *subrelation_rstc* :
∀ (*A* : `Type`) (*R* : *rel A*), *subrelation R* (*rstc R*).

`Lemma` *rstc_smallest* :
∀ (*A* : `Type`) (*R S* : *rel A*),
subrelation R S → *Equivalence S* → *subrelation* (*rstc R*) *S*.

13.11 Redukcje

`Definition` *rr* {*A* : `Type`} (*R* : *rel A*) : *rel A* :=
fun *x y* : *A* ⇒ *R x y* ∧ *x* ≠ *y*.

`Instance` *Antireflexive_rr* :
∀ (*A* : `Type`) (*R* : *rel A*), *Antireflexive* (*rr R*).

`Lemma` *rr_rc* :
∀ (*A* : `Type`) (*R* : *rel A*),
Reflexive R → *refl_clos* (*rr R*) <–> *R*.

Rozdział 14

X6: Rozdział z odpadami z R2

14.1 Parametryczność

UWAGA TODO: ten podrozdział zawiera do pewnego stopnia kłamstwa (tzn. dość uproszczony punkt widzenia).

Niech A B : `Set`. Zadajmy sobie następujące pytanie: ile jest funkcji typu $A \to B$? Żeby ułatwić sobie zadanie, ograniczmy się jedynie do typów, które mają skończoną ilość elementów.

Nietrudno przekonać się, że ich ilość to |B|^|A|, gdzie ^ oznacza potęgowanie, zaś |T| to ilość elementów typu T (ta notacja nie ma nic wspólnego z Coqiem — zaadaptowałem ją z teorii zbiorów jedynie na potrzeby tego podrozdziału).

Udowodnić ten fakt możesz (choć póki co nie w Coqu) posługując się indukcją po ilości elementów typu A. Jeżeli A jest pusty, to jest tylko jedna taka funkcja, o czym przekonałeś się już podczas ćwiczeń w podrozdziale o typie *Empty_set*.

Ćwiczenie Udowodnij (nieformalnie, na papierze), że w powyższym akapicie nie okłamałem cię.

Ćwiczenie Zdefiniuj wszystkie możliwe funkcje typu *unit* → *unit*, *unit* → *bool* i *bool* → *bool*.

Postawmy sobie teraz trudniejsze pytanie: ile jest funkcji typu ∀ A : `Set`, $A \to A$? W udzieleniu odpowiedzi pomoże nam parametryczność — jedna z właściwości Coqowego polimorfizmu.

Stwierdzenie, że polimorfizm w Coqu jest parametryczny, oznacza, że funkcja biorąca typ jako jeden z argumentów działa w taki sam sposób niezależnie od tego, jaki typ przekażemy jej jako argument.

Konsekwencją tego jest, że funkcje polimorficzne nie wiedzą (i nie mogą wiedzieć), na wartościach jakiego typu operują. Wobec tego elementem typu ∀ A : `Set`, $A \to A$ nie może być funkcja, która np. dla typu *nat* stale zwraca 42, a dla innych typów po prostu zwraca przekazany jej argument.

Stąd konkludujemy, że typ $\forall\ A$: `Set`, $A \rightarrow A$ ma tylko jeden element, a mianowicie polimorficzną funkcję identycznościową.

```
Definition id' : ∀ A : Set, A → A :=
  fun (A : Set) (x : A) ⇒ x.
```

Ćwiczenie Zdefiniuj wszystkie elementy następujących typów lub udowodnij, że istnienie choć jednego elementu prowadzi do sprzeczności:

- $\forall\ A$: `Set`, $A \rightarrow A \rightarrow A$
- $\forall\ A$: `Set`, $A \rightarrow A \rightarrow A \rightarrow A$
- $\forall\ A\ B$: `Set`, $A \rightarrow B$
- $\forall\ A\ B$: `Set`, $A \rightarrow B \rightarrow A$
- $\forall\ A\ B$: `Set`, $A \rightarrow B \rightarrow B$
- $\forall\ A\ B$: `Set`, $A \rightarrow B \rightarrow A \times B$
- $\forall\ A\ B$: `Set`, $A \rightarrow B \rightarrow sum\ A\ B$
- $\forall\ A\ B\ C$: `Set`, $A \rightarrow B \rightarrow C$
- $\forall\ A$: `Set`, $option\ A \rightarrow A$
- $\forall\ A$: `Set`, $list\ A \rightarrow A$

TODO: tu opisać kłamstwo

```
Inductive path {A : Type} (x : A) : A → Type :=
    | idpath : path x x.

Arguments idpath {A} _.

Axiom LEM : ∀ (A : Type), A + (A → Empty_set).

Open Scope type_scope.

Definition bad' (A : Type) :
  {f : A → A &
    (@path Type bool A × ∀ x : A, f x ≠ x) +
    ((@path Type bool A → Empty_set) × ∀ x : A, f x = x)}.
Proof.
  destruct (LEM (@path Type bool A)).
    destruct p. esplit with negb. left. split.
      exact (@idpath Type bool).
      destruct x; cbn; inversion 1.
```

```
    esplit with (fun x : A ⇒ x). right. split.
      assumption.
      reflexivity.
Defined.

Definition bad (A : Type) : A → A := projT1 (bad' A).

Lemma bad_is_bad :
  ∀ b : bool, bad bool b ≠ b.
Proof.
  intros. unfold bad. destruct bad'. cbn. destruct s as [[p H] | [p H]].
    apply H.
    destruct (p (idpath _)).
Defined.

Lemma bad_ist_gut :
  ∀ (A : Type) (x : A),
    (@path Type bool A → Empty_set) → bad A x = x.
Proof.
  unfold bad. intros A x p. destruct bad' as [f [[q H] | [q H]]]; cbn.
    destruct (p q).
    apply H.
Defined.
```

14.2 Rozstrzygalność

```
Theorem excluded_middle :
  ∀ P : Prop, P ∨ ¬ P.
Proof.
  intro. left.
Restart.
  intro. right. intro.
Abort.
```

Próba udowodnienia tego twierdzenia pokazuje nam zasadniczą różnicę między logiką konstruktywną, która jest domyślną logiką Coqa, oraz logiką klasyczną, najpowszechniej znanym i używanym rodzajem logiki.

Każde zdanie jest, w pewnym "filozoficznym" sensie, prawdziwe lub fałszywe i to właśnie powyższe zdanie oznacza w logice klasycznej. Logika konstruktywna jednak, jak już wiemy, nie jest logiką prawdy, lecz logiką udowadnialności i ma swoją interpretację obliczeniową. Powyższe zdanie w logice konstruktywnej oznacza: program komputerowy *exluded_middle* rozstrzyga, czy dowolne zdanie jest prawdziwe, czy fałszywe.

Skonstruowanie programu o takim typie jest w ogólności niemożliwe, gdyż dysponujemy zbyt małą ilością informacji: nie wiemy czym jest zdanie P, a nie posiadamy żadnego ogólnego sposobu dowodzenia lub obalania zdań o nieznanej nam postaci. Nie możemy np. użyć indukcji, gdyż nie wiemy, czy zdanie P zostało zdefiniowane induktywnie, czy też nie. W Coqu jedynym sposobem uzyskania termu o typie $\forall$ P : Prop, $P \lor \neg P$ jest przyjęcie go jako aksjomat.

```
Theorem True_dec : True ∨ ¬ True.
Proof.
  left. trivial.
Qed.
```

Powyższe dywagacje nie przeszkadzają nam jednak w udowadnianiu, że reguła wyłączonego środka zachodzi dla pewnych konkretnych zdań. Zdanie takie będziemy nazywać zdaniami rozstrzygalnymi (ang. decidable). O pozostałych zdaniach będziemy mówić, że są nierozstrzygalne (ang. undecidable). Ponieważ w Coqu wszystkie funkcje są rekurencyjne, a dowody to programy, to możemy powyższą definicję rozumieć tak: zdanie jest rozstrzygalne, jeżeli istnieje funkcja rekurencyjna o przeciwdzidzinie *bool*, która sprawdza, czy jest ono prawdziwe, czy fałszywe.

Przykładami zdań, predykatów czy problemów rozstrzygalnych są:

- sprawdzanie, czy lista jest niepusta
- sprawdzanie, czy liczba naturalna jest parzysta
- sprawdzanie, czy dwie liczby naturalne są równe

Przykładem problemów nierozstrzygalnych są:

- dla funkcji f g : $nat \to nat$ sprawdzenie, czy $\forall$ n : nat, f $n = g$ n — jest to w ogólności niemożliwe, gdyż wymaga wykonania nieskończonej ilości porównań (co nie znaczy, że nie da się rozwiązać tego problemu dla niektórych funkcji)
- sprawdzenie, czy słowo o nieskończonej długości jest palindromem

Ćwiczenie

```
Theorem eq_nat_dec :
  ∀ n m : nat, n = m ∨ ¬ n = m.
```

14.2.1 Techniczne apekty rozstrzygalności

Podsumowując powyższe rozważania, moglibyśmy stwierdzić: zdanie P jest rozstrzygalne, jeżeli istnieje term typu $P \lor \neg P$. Stwierdzenie takie nie zamyka jednak sprawy, gdyż bywa czasem mocno bezużyteczne.

Żeby to zobrazować, spróbujmy użyć twierdzenia *eq_nat_dec* do napisania funkcji, która sprawdza, czy liczna naturalna n występuje na liście liczb naturalnych l:

```
Fail Fixpoint inb_nat (n : nat) (l : list nat) : bool :=
match l with
    | nil ⇒ false
```

```
  | cons h t ⇒
      match eq_nat_dec n h with
          | or_introl _ ⇒ true
          | _ ⇒ inb_nat n t
      end
end.
```

Coq nie akceptuje powyższego kodu, racząc nas informacją o błędzie:

```
(*  Error:
   Incorrect elimination of ęq_nat_dec n h0" in the inductive type ór":
   the return type has sort Śet" while it should be "Prop".
   Elimination of an inductive object of sort Prop
   is not allowed on a predicate in sort Set
   because proofs can be eliminated only to build proofs. *)
```

Nasza porażka wynika z faktu, że do zdefiniowania funkcji, która jest programem (jej dziedzina i przeciwdziedzina są sortu `Set`) próbowaliśmy użyć termu *eq_nat_dec n h*, który jest dowodem (konkluzją *eq_nat_dec* jest równość, która jest sortu `Prop`).

Mimo korespondencji Curry'ego-Howarda, która odpowiada za olbrzymie podobieństwo specyfikacji i zdań, programów i dowodów, sortu `Set` i sortu `Prop`, są one rozróżniane i niesie to za sobą konsekwencje: podczas gdy programów możemy używać wszędzie, dowodów możemy używać jedynie do konstruowania innych dowodów.

Praktycznie oznacza to, że mimo iż równość liczb naturalnych jest rozstrzygalna, pisząc program nie mamy możliwości jej rozstrzygania za pomocą *eq_nat_dec*. To właśnie miałem na myśli pisząc, że termy typu $P \lor \neg P$ są mocno bezużyteczne.

Uszy do góry: nie wszystko stracone! Jest to tylko drobna przeszkoda, którą bardzo łatwo ominąć:

```
Inductive sumbool (A B : Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B.
```

Typ *sumbool* jest niemal dokładną kopią *or*, jednak nie żyje on w `Prop`, lecz w `Set`. Ta drobna sztuczka, że termy typu *sumbool A B* formalnie są programami, mimo że ich naturalna interpretacja jest taka sama jak *or*, a więc jako dowodu dysjunkcji.

Ćwiczenie Udowodnij twierdzenie *eq_nat_dec'* o rozstrzygalności = na liczbach naturalnych. Użyj typu *sumbool*. Następnie napisz funkcję *inb_nat*, która sprawdza, czy liczba naturalna n jest obecna na liście l.

14.3 Pięć rodzajów reguł

Być może jeszcze tego nie zauważyłeś, ale większość logiki konstruktywnej, programowania funkcyjnego, a przede wszystkim teorii typów kręci się wokół pięciu rodzajów reguł. Są to reguły:

- formacji (ang. formation rules)
- wprowadzania (ang. introduction rules)
- eliminacji (ang. elimination rules)
- obliczania (ang. computation rules)
- unikalności (ang. uniqueness principles)

W tym podrozdziale przyjrzymy się wszystkim pięciu typom reguł. Zobaczymy jak wyglądają, skąd się biorą i do czego służą. Podrozdział będzie miał charakter mocno teoretyczny.

14.3.1 Reguły formacji

Reguły formacji mówią nam, jak tworzyć typy (termy sortów `Set` i `Type`) oraz zdania (termy sortu `Prop`). Większość z nich pochodzi z nagłówków definicji induktywnych. Reguła dla typu *bool* wygląda tak:

```
(*
    -----
    bool : Set
  *)
```

Ten mistyczny zapis pochodzi z publikacji dotyczących teorii typów. Nad kreską znajdują się przesłanki reguły, a pod kreską znajduje się konkluzja reguły.

Regułę tę możemy odczytać: *bool* jest typem sortu `Set`. Postać tej reguły wynika wprost z definicji typu *bool*.

```
Print bool.
```

```
(*  ===> Inductive bool : Set :=  true : bool | false : bool *)
```

Powyższej regule formacji odpowiada tutaj fragment `Inductive` *bool* : `Set`, który stwierdza po prostu, że *bool* jest typem sortu `Set`.

Nie zawsze jednak reguły formacji są aż tak proste. Reguła dla produktu wygląda tak:

```
(*
    A : Type, B : Type
    ---------
    prod A B : Type
  *)
```

Reguła formacji dla *prod* głosi: jeżeli A jest typem sortu `Type` oraz B jest typem sortu `Type`, to *prod* A B jest typem sortu `Type`. Jest ona rzecz jasna konsekwencją definicji produktu.

```
Print prod.
```

```
(*  ===> Inductive prod (A B : Type) : Type :=
```

```
        pair : A -> B -> A * B *)
```

Regule odpowiada fragment `Inductive` *prod* (A B : `Type`) : `Type`. To, co w regule jest nad kreską (A : `Type` i B : `Type`), tutaj występuje przed dwukropkiem, po prostu jako argumentu typu *prod*. Jak widać, nagłówek typu induktywnego jest po prostu skompresowaną formą reguły formacji.

Należy zauważyć, że nie wszystkie reguły formacji pochodzą z definicji induktywnych. Tak wygląda reguła formacji dla funkcji (między typami sortu `Type`):

```
(*
    A : Type, B : Type
    ---------
    A -> B : Type
  *)
```

Reguła nie pochodzi z definicji induktywnej, gdyż typ funkcji $A \to B$ jest typem wbudowanym i nie jest zdefiniowany indukcyjnie.

Ćwiczenie Napisz, bez podglądania, jak wyglądają reguły formacji dla *option*, *nat* oraz *list*. Następnie zweryfikuj swoje odpowiedzi za pomocą komendy `Print`.

14.3.2 Reguły wprowadzania

Reguły wprowadzania mówią nam, w jaki sposób formować termy danego typu. Większość z nich pochodzi od konstruktorów typów induktywnych. Dla typu bool reguły wprowadzania wyglądają tak:

```
(*
    ------
    true : bool

    ------
    false : bool
  *)
```

Reguły te stwierdzają po prostu, że *true* jest termem typu *bool* oraz że *false* jest termem typu *bool*. Wynikają one wprost z definicji typu *bool* — każda z nich odpowiada jednemu konstruktorowi.

Wobec powyższego nie powinna zaskoczyć cię reguła wprowadzania dla produktu:

```
(*
    A : Type, B : Type, a : A, b : B
    ----------------
    pair A B a b : prod A B
  *)
```

Jeżeli jednak zaskoczyła cię obecność w regule A : `Type` i B : `Type`, przyjrzyj się dokładnie typowi konstruktora *pair*:

```
Check @pair.
(*  ===> pair : forall A B : Type, A -> B -> A * B *)
```

Widać tutaj jak na dłoni, że *pair* jest funkcją zależną biorącą cztery argumenty i zwracają wynik, którego typ jest produktem jej dwóch pierwszych argumentów.

Podobnie jak w przypadku reguł formacji, nie wszystkie reguły wprowadzania pochodzą od konstruktorów typów induktywnych. W przypadku funkcji reguła wygląda mniej więcej tak:

```
(*
     |- A -> B : Type, ; x : T |- y : B
    -------------------
     |- fun x => y : A -> B
   *)
```

Pojawiło się tu kilka nowych rzeczy: litera oznacza kontekst, zaś zapis |- j, że osąd j zachodzi w kontekście . Zapis ; j oznacza rozszerzenie kontekstu poprzez dodanie do niego osądu j.

Regułę możemy odczytać tak: jeżeli $A \to B$ jest typem sortu `Type` w kontekście i y jest termem typu B w kontekście rozszerzonym o osąd x : T, to `fun` $x \Rightarrow y$ jest termem typu A $\to B$ w kontekście .

Powyższa reguła nazywana jest "lambda abstrakcją" (gdyż zazwyczaj jest zapisywana przy użyciu symbolu zamiast słowa kluczowego `fun`, jak w Coqu). Nie przejmuj się, jeżeli jej. Znajomość reguł wprowadzania nie jest nam potrzebna, by skutecznie posługiwać się Coqiem.

Należy też dodać, że reguła ta jest nieco uproszczona. Pełniejszy opis teoretyczny induktywnego rachunku konstrukcji można znaleźć w rozdziałach 4 i 5 manuala: https://coq.inria.fr/refman/toc.h

Ćwiczenie Napisz (bez podglądania) jak wyglądają reguły wprowadzania dla *option*, *nat* oraz *list*. Następnie zweryfikuj swoje odpowiedzi za pomocą komendy `Print`.

14.3.3 Reguły eliminacji

Reguły eliminacji są w pewien sposób dualne do reguł wprowadzania. Tak jak reguły wprowadzania dla typu T służą do konstruowania termów typu T z innych termów, tak reguły eliminacji dla typu T mówią nam, jak z termów typu T skonstruować termy innych typów.

Zobaczmy, jak wygląda jedna z reguł eliminacji dla typu *bool*.

```
(*
    A : Type, x : A, y : A, b : bool
    ----------------
    if b then x else y : A
   *)
```

Reguła ta mówi nam, że jeżeli mamy typ A oraz dwie wartości x i y typu A, a także term b typu *bool*, to możemy skonstruować inną wartość typu A, mianowicie `if` b `then` x `else` y.

Reguła ta jest dość prosta. W szczególności nie jest ona zależna, tzn. obie gałęzie `if`a muszą być tego samego typu. Przyjrzyjmy się nieco bardziej ogólnej regule.

```
(*
   P : bool -> Type, x : P true, y : P false, b : bool
   -------------------------
   bool_rect P x y b : P b
  *)
```

Reguła ta mówi nam, że jeżeli mamy rodzinę typów $P : bool \to$ `Type` oraz termy x typu P *true* i y typu P *false*, a także term b typu *bool*, to możemy skonstruować term *bool_rect* P x y b typu P b.

Spójrzmy na tę regułę z nieco innej strony:

```
(*
   P : bool -> Type, x : P true, y : P false
   -------------------------
   bool_rect P x y : forall b : bool, P b
  *)
```

Widzimy, że reguły eliminacji dla typu induktywnego T służą do konstruowania funkcji, których dziedziną jest T, a więc mówią nam, jak "wyeliminować" term typu T, aby uzyskać term innego typu.

Reguły eliminacji występują w wielu wariantach:

- zależnym i niezależnym — w zależności od tego, czy służą do definiowania funkcji zależnych, czy nie.
- rekurencyjnym i nierekurencyjnym — te drugie służą jedynie do przeprowadzania rozumowań przez przypadki oraz definiowania funkcji przez pattern matching, ale bez rekurencji. Niektóre typy nie mają rekurencyjnych reguł eliminacji.
- pierwotne i wtórne — dla typu induktywnego T Coq generuje regułę *T_rect*, którą będziemy zwać regułą pierwotną. Jej postać wynika wprost z definicji typu T. Reguły dla typów nieinduktywnych (np. funkcji) również będziemy uważać za pierwotne. Jednak nie wszystkie reguły są pierwotne — przekonamy się o tym w przyszłości, tworząc własne reguły indukcyjne.

Zgodnie z zaprezentowaną klasyfikacją, pierwsza z naszych reguł jest:

- niezależna, gdyż obie gałęzie `if`a są tego samego typu. Innymi słowy, definiujemy term typu A, który nie jest zależny
- nierekurencyjna, gdyż typ *bool* nie jest rekurencyjny i wobec tego może posiadać jedynie reguły nierekurencyjne

- wtórna — regułą pierwotną dla *bool* jest *bool_rect*

Druga z naszych reguł jest:

- zależna, gdyż definiujemy term typu zależnego $P\ b$
- nierekurencyjna z tych samych powodów, co reguła pierwsza
- pierwotna — Coq wygenerował ją dla nas automatycznie

W zależności od kombinacji powyższych cech reguły eliminacji mogą występować pod różnymi nazwami:

- reguły indukcyjne są zależne i rekurencyjne. Służą do definiowania funkcji, których przeciwdziedzina jest sortu `Prop`, a więc do dowodzenia zdań przez indukcję
- rekursory to rekurencyjne reguły eliminacji, które służą do definiowania funkcji, których przeciwdziedzina jest sortu `Set` lub `Type`

Nie przejmuj się natłokiem nazw ani rozróżnień. Powyższą klasyfikację wymyśliłem na poczekaniu i nie ma ona w praktyce żadnego znaczenia.

Zauważmy, że podobnie jak nie wszystkie reguły formacji i wprowadzania pochodzą od typów induktywnych, tak i nie wszystkie reguły eliminacji od nich pochodzą. Kontrprzykładem niech będzie reguła eliminacji dla funkcji (niezależnych):

```
(*
    A : Type, B : Type, f : A -> B, x : A
    -------------------
    f x : B
  *)
```

Reguła ta mówi nam, że jeżeli mamy funkcję f typu $A \to B$ oraz argument x typu A, to aplikacja funkcji f do argumentu x jest typu B.

Zauważmy też, że mimo iż reguły wprowadzania i eliminacji są w pewien sposób dualne, to istnieją między nimi różnice.

Przede wszystkim, poza regułami wbudowanymi, obowiązuje prosta zasada: jeden konstruktor typu induktywnego — jedna reguła wprowadzania. Innymi słowy, reguły wprowadzania dla typów induktywnych pochodzą bezpośrednio od konstruktorów i nie możemy w żaden sposób dodać nowych. Są one w pewien sposób pierwotne i nie mamy nad nimi (bezpośredniej) kontroli.

Jeżeli chodzi o reguły eliminacji, to są one, poza niewielką ilością reguł pierwotnych, w pewnym sensie wtórne — możemy budować je z pattern matchingu i rekursji strukturalnej i to właśnie te dwie ostatnie idee są w Coqu ideami pierwotnymi. Jeżeli chodzi o kontrolę, to możemy swobodnie dodawać nowe reguły eliminacji za pomocą twierdzeń lub definiując je bezpośrednio.

Działanie takie jest, w przypadku nieco bardziej zaawansowanych twierdzeń niż dotychczas widzieliśmy, bardzo częste. Ba! Częste jest także tworzenie reguł eliminacji dla każdej funkcji z osobna, perfekcyjnie dopasowanych do kształtu jej rekursji. Jest to nawet bardzo wygodne, gdyż Coq potrafi automatycznie wygenerować dla nas takie reguły.

Przykładem niestandardowej reguły może być reguła eliminacji dla list działająca "od tyłu":

```
(*
    A : Type, P : list A -> Prop,
    H : P [],
    H' : forall (h : A) (t : list A), P t -> P (t ++ [h])
    -------------------------------
    forall l : list A, P l
  *)
```

Póki co wydaje mi się, że udowodnienie słuszności tej reguły będzie dla nas za trudne. W przyszłości na pewno napiszę coś więcej na temat reguł eliminacji, gdyż ze względu na swój "otwarty" charakter są one z punktu widzenia praktyki najważniejsze.

Tymczasem na otarcie łez zajmijmy się inną, niestandardową regułą dla list.

Ćwiczenie Udowodnij, że reguła dla list "co dwa" jest słuszna. Zauważ, że komenda `Fixpoint` może służyć do podawania definicji rekurencyjnych nie tylko "ręcznie", ale także za pomocą taktyk.

Wskazówka: użycie hipotezy indukcyjnej *list_ind_2* zbyt wcześnie ma podobne skutki co wywołanie rekurencyjne na argumencie, który nie jest strukturalnie mniejszy.

```
Module EliminationRules.

Require Import List.
Import ListNotations.

Fixpoint list_ind_2
  (A : Type) (P : list A → Prop)
  (H0 : P []) (H1 : ∀ x : A, P [x])
  (H2 : ∀ (x y : A) (l : list A), P l → P (x :: y :: l))
    (l : list A) : P l.
```

Ćwiczenie Napisz funkcję `apply`, odpowiadającą regule eliminacji dla funkcji (niezależnych). Udowodnij jej specyfikację.

Uwaga: notacja "$" na oznaczenie aplikacji funkcji pochodzi z języka Haskell i jest tam bardzo często stosowana, gdyż pozwala zaoszczędzić stawiania zbędnych nawiasów.

```
Notation "f $ x":= (apply f x) (at level 5).

Theorem apply_spec :
  ∀ (A B : Type) (f : A → B) (x : A), f $ x = f x.

End EliminationRules.
```

14.3.4 Reguły obliczania

Poznawszy reguły wprowadzania i eliminacji możemy zadać sobie pytanie: jakie są między nimi związki? Jedną z odpowiedzi na to pytanie dają reguły obliczania, które określają, w jaki sposób reguły eliminacji działają na obiekty stworzone za pomocą reguł wprowadzania. Zobaczmy o co chodzi na przykładzie.

```
(*
    A : Type, B : Type, x : A |- e : B, t : A
    --------------------
    (fun x : A => e) t ≡  e{x/t}
  *)
```

Powyższa reguła nazywa się "redukcja beta". Mówi ona, jaki efekt ma aplikacja funkcji zrobionej za pomocą lambda abstrakcji do argumentu, przy czym aplikacja jest regułą eliminacji dla funkcji, a lambda abstrakcja — regułą wprowadzania.

Możemy odczytać ją tak: jeżeli A i B są typami, zaś e termem typu B, w którym występuje zmienna wolna x typu A, to wyrażenie (`fun` x : $A \Rightarrow e$) t redukuje się (symbol $\equiv) doe, wktrymwmiejscezmiennejxpodstawionotermt.$

Zauważ, że zarówno symbol $\equiv$ $jakinotacjae\{x/t\}stylkonieformalnymizapisamiiniemaj adnegoznacze$

Nie jest tak, że dla każdego typu jest tylko jedna reguła obliczania. Jako, że reguły obliczania pokazują związek między regułami eliminacji i wprowadzania, ich ilość można przybliżyć prostym wzorem:

\# reguł obliczania = # reguł eliminacji * # reguł wprowadzania,

gdzie # to nieformalny symbol oznaczający "ilość". W Coqowej praktyce zazwyczaj oznacza to, że reguł obliczania jest nieskończenie wiele, gdyż możemy wymyślić sobie nieskończenie wiele reguł eliminacji. Przykładem typu, który ma więcej niż jedną regułę obliczania dla danej reguły eliminacji, jest *bool*:

```
(*
    P : bool -> Type, x : P true, y : P false
    --------------------
    bool_rect P x y true ≡  x

    P : bool -> Type, x : P true, y : P false
    --------------------
    bool_rect P x y false ≡  y
  *)
```

Typ *bool* ma dwie reguły wprowadzania pochodzące od dwóch konstruktorów, a zatem ich związki z regułą eliminacji *bool_rect* będą opisywać dwie reguły obliczania. Pierwsza z nich mówi, że *bool_rect P x y true* redukuje się do x, a druga, że *bool_rect P x y false* redukuje się do y.

Gdyby zastąpić w nich regułe *bool_rect* przez nieco prostszą regułę, w której nie występują typy zależne, to można by powyższe reguły zapisać tak:

```
(*
```

```
   A : Type, x : A, y : A
   ---------------------
   if true then x else y ≡ x

   A : Type, x : A, y : A
   ---------------------
   if false then x else y ≡ y
 *)
```

Wygląda dużo bardziej znajomo, prawda?

Na zakończenie wypadałoby napisać, skąd biorą się reguły obliczania. W nieco mniej formalnych pracach teoretycznych na temat teorii typów są one zazwyczaj uznawane za byty podstawowe, z których następnie wywodzi się reguły obliczania takich konstrukcji, jak np. `match`.

W Coqu jest na odwrót. Tak jak reguły eliminacji pochodzą od pattern matchingu i rekursji, tak reguły obliczania pochdzą od opisanych już wcześniej reguł redukcji (beta, delta, jota i zeta), a także konwersji alfa.

Ćwiczenie Napisz reguły obliczania dla liczb naturalnych oraz list (dla reguł eliminacji *nat_ind* oraz *list_ind*).

14.3.5 Reguły unikalności

Kolejną odpowiedzią na pytanie o związki między regułami wprowadzania i eliminacji są reguły unikalności. Są one dualne do reguł obliczania i określają, w jaki sposób reguły wprowadzania działają na obiekty pochodzące od reguł eliminacji. Przyjrzyjmy się przykładowi.

```
(*
   A : Type, B : Type, f : A -> B
   ---------------
   (fun x : A => f x) ≡ f
 *)
```

Powyższa reguła unikalności dla funkcji jest nazywana "redukcją eta". Stwierdza ona, że funkcja stworzona za pomocą abstrakcji `fun` x : A, której ciałem jest aplikacja f x jest definicyjnie równa funkcji f. Regułą wprowadzania dla funkcji jest oczywiście abstrakcja, a regułą eliminacji — aplikacja.

Reguły unikalności różnią się jednak dość mocno od reguł obliczania, gdyż zamiast równości definicyjnej $\equiv mogczasemuywa\ standardowej, zdaniowejrwnociCoqa, czyli = .Niedokocapasujete\ do$

```
(*
   A : Type, B : Type, p : A * B
   ----------------
   (fst p, snd p) = p
 *)
```

Powyższa reguła głosi, że para, której pierwszym elementem jest pierwszy element pary p, a drugim elementem — drugi element pary p, jest w istocie równa parze p. W Coqu możemy ją wyrazić (i udowodnić) tak:

Theorem *prod_uniq* :
  ∀ (A B : Type) (p : $A \times B$),
    (*fst* p, *snd* p) = p.
Proof.
  destruct p. *cbn*. trivial.
Qed.

Podsumowując, reguły unikalności występują w dwóch rodzajach:

- dane nam z góry, niemożliwe do wyrażenia bezpośrednio w Coqu i używające równości definicyjnej, jak w przypadku redukcji eta dla funkcji
- możliwe do wyrażenia i udowodnienia w Coqu, używające zwykłej równości, jak dla produktów i w ogólności dla typów induktywnych

Ćwiczenie Sformułuj reguły unikalności dla funkcji zależnych (∀), sum zależnych (*sigT*) i *unit* (zapisz je w notacji z poziomą kreską). Zdecyduj, gdzie w powyższej klasyfikacji mieszczą się te reguły. Jeżeli to możliwe, wyraź je i udowodnij w Coqu.

14.4 Typy hybrydowe

Ostatnim z typów istotnych z punktu widzenia silnych specyfikacji jest typ o wdzięcznej nazwie *sumor*.

Module *sumor*.

Inductive *sumor* (A : Type) (B : Prop) : Type :=
    | *inleft* : $A \to$ *sumor* A B
    | *inright* : $B \to$ *sumor* A B.

Jak sama nazwa wskazuje, *sumor* jest hybrydą sumy rozłącznej *sum* oraz dysjunkcji *or*. Możemy go interpretować jako typ, którego elementami są elementy A albo wymówki w stylu "nie mam elementu A, ponieważ zachodzi zdanie B". B nie zależy od A, a więc jest to zwykła suma (a nie suma zależna, czyli uogólnienie produktu). *sumor* żyje w Type, a więc jest to specyfikacja i liczy się konkretna postać jego termów, a nie jedynie fakt ich istnienia.

Ćwiczenie (*pred'*) Zdefiniuj funkcję *pred'*, która przypisuje liczbie naturalnej jej poprzednik. Poprzednikiem 0 nie powinno być 0. Mogą przydać ci się typ *sumor* oraz sposób definiowania za pomocą taktyk, omówiony w podrozdziale dotyczącym sum zależnych.

End *sumor*.

14.5 Small scale reflection

Rozdział 15

X7: Liczby konaturalne

TODO: coś tu napisać.

Zdefiniuj liczby konaturalne oraz ich relację bipodobieństwa. Pokaż, że jest to relacja równoważności.

```
Lemma sim_refl :
  ∀ n : conat, sim n n.

Lemma sim_sym :
  ∀ n m : conat, sim n m → sim m n.

Lemma sim_trans :
  ∀ a b c : conat, sim a b → sim b c → sim a c.
```

Dzięki poniższemu będziemy mogli używac taktyki `rewrite` do przepisywania *sim* tak samo jak =.

```
Require Import Setoid.

Instance Equivalence_sim : Equivalence sim.
Proof.
  esplit; red.
    apply sim_refl.
    apply sim_sym.
    apply sim_trans.
Defined.
```

Zdefiniuj zero, następnik oraz liczbę omega - jest to nieskończona liczba konaturalna, która jest sama swoim poprzednikiem. Udowodnij ich kilka podstawowych właściwości.

```
Lemma succ_pred :
  ∀ n m : conat,
    n = succ m ↔ pred n = Some m.

Lemma zero_not_omega :
  ¬ sim zero omega.

Lemma sim_succ_omega :
```

$\forall$ *n* : *conat*, *sim n* (*succ n*) $\to$ *sim n* `omega`.

`Lemma` *succ_omega* :
`omega` = *succ* `omega`.

`Lemma` *sim_succ* :
$\forall$ *n m* : *conat*, *sim n m* $\to$ *sim* (*succ n*) (*succ m*).

`Lemma` *sim_succ_inv* :
$\forall$ *n m* : *conat*, *sim* (*succ n*) (*succ m*) $\to$ *sim n m*.

Zdefiniuj dodawanie liczb konaturalnych i udowodnij jego podstawowe właściwości.

`Lemma` *add_zero_l* :
$\forall$ *n* : *conat*, *sim* (*add zero n*) *n*.

`Lemma` *add_zero_r* :
$\forall$ *n* : *conat*, *sim* (*add n zero*) *n*.

`Lemma` *add_omega_l* :
$\forall$ *n* : *conat*, *sim* (*add* `omega` *n*) `omega`.

`Lemma` *add_omega_r* :
$\forall$ *n* : *conat*, *sim* (*add n* `omega`) `omega`.

`Lemma` *add_succ_l* :
$\forall$ *n m* : *conat*, *sim* (*add* (*succ n*) *m*) (*add n* (*succ m*)).

`Lemma` *add_succ_r* :
$\forall$ *n m* : *conat*, *sim* (*add n* (*succ m*)) (*add* (*succ n*) *m*).

`Lemma` *add_succ_l'* :
$\forall$ *n m* : *conat*, *sim* (*add* (*succ n*) *m*) (*succ* (*add n m*)).

`Lemma` *add_succ_r'* :
$\forall$ *n m* : *conat*, *sim* (*add n* (*succ m*)) (*succ* (*add n m*)).

`Lemma` *add_assoc* :
$\forall$ *a b c* : *conat*, *sim* (*add* (*add a b*) *c*) (*add a* (*add b c*)).

`Lemma` *add_comm* :
$\forall$ *n m* : *conat*, *sim* (*add n m*) (*add m n*).

`Lemma` *sim_add_zero_l* :
$\forall$ *n m* : *conat*,
sim (*add n m*) *zero* $\to$ *sim n zero*.

`Lemma` *sim_add_zero_r* :
$\forall$ *n m* : *conat*,
sim (*add n m*) *zero* $\to$ *sim m zero*.

Zdefiniuj relację $\leq$ na liczbach konaturalnych i udowodnij jej podstawowe właściwości.

`Lemma` *le_refl* :
$\forall$ *n* : *conat*, *le n n*.

`Lemma` *le_trans* :
  ∀ *a b c* : *conat*, *le a b* → *le b c* → *le a c*.

`Lemma` *le_sim* :
  ∀ *n1 n2 m1 m2* : *conat*,
    sim n1 n2 → *sim m1 m2* → *le n1 m1* → *le n2 m2*.

`Lemma` *le_0_l* :
  ∀ *n* : *conat*, *le zero n*.

`Lemma` *le_0_r* :
  ∀ *n* : *conat*, *le n zero* → *n* = *zero*.

`Lemma` *le_omega_r* :
  ∀ *n* : *conat*, *le n* `omega`.

`Lemma` *le_omega_l* :
  ∀ *n* : *conat*, *le* `omega` *n* → *sim n* `omega`.

`Lemma` *le_succ_r* :
  ∀ *n m* : *conat*, *le n m* → *le n* (*succ m*).

`Lemma` *le_succ* :
  ∀ *n m* : *conat*, *le n m* → *le* (*succ n*) (*succ m*).

`Lemma` *le_add_l* :
  ∀ *a b c* : *conat*,
    le a b → *le a* (*add b c*).

`Lemma` *le_add_r* :
  ∀ *a b c* : *conat*,
    le a c → *le a* (*add b c*).

`Lemma` *le_add* :
  ∀ *n1 n2 m1 m2* : *conat*,
    le n1 n2 → *le m1 m2* → *le* (*add n1 m1*) (*add n2 m2*).

`Lemma` *le_add_l'* :
  ∀ *n m* : *conat*, *le n* (*add n m*).

`Lemma` *le_add_r'* :
  ∀ *n m* : *conat*, *le m* (*add n m*).

`Lemma` *le_add_l''* :
  ∀ *n n' m* : *conat*,
    le n n' → *le* (*add n m*) (*add n' m*).

`Lemma` *le_add_r''* :
  ∀ *n m m'* : *conat*,
    le m m' → *le* (*add n m*) (*add n m'*).

Zdefiniuj funkcje *min* i *max* i udowodnij ich właściwości.

`Lemma` *min_zero_l* :

∀ n : conat, min zero n = zero.

Lemma min_zero_r :
∀ n : conat, min n zero = zero.

Lemma min_omega_l :
∀ n : conat, sim (min omega n) n.

Lemma min_omega_r :
∀ n : conat, sim (min n omega) n.

Lemma min_succ :
∀ n m : conat,
sim (min (succ n) (succ m)) (succ (min n m)).

Lemma max_zero_l :
∀ n : conat, sim (max zero n) n.

Lemma max_zero_r :
∀ n : conat, sim (max n zero) n.

Lemma max_omega_l :
∀ n : conat, sim (max omega n) omega.

Lemma max_omega_r :
∀ n : conat, sim (max n omega) omega.

Lemma max_succ :
∀ n m : conat,
sim (max (succ n) (succ m)) (succ (max n m)).

Lemma min_assoc :
∀ a b c : conat, sim (min (min a b) c) (min a (min b c)).

Lemma max_assoc :
∀ a b c : conat, sim (max (max a b) c) (max a (max b c)).

Lemma min_comm :
∀ n m : conat, sim (min n m) (min m n).

Lemma max_comm :
∀ n m : conat, sim (max n m) (max m n).

Lemma min_add_l :
∀ a b c : conat,
sim (min (add a b) (add a c)) (add a (min b c)).

Lemma min_add_r :
∀ a b c : conat,
sim (min (add a c) (add b c)) (add (min a b) c).

Lemma max_add_l :
∀ a b c : conat,
sim (max (add a b) (add a c)) (add a (max b c)).

Lemma max_add_r :
$\forall\ a\ b\ c :\ conat,$
$sim\ (max\ (add\ a\ c)\ (add\ b\ c))\ (add\ (max\ a\ b)\ c).$

Lemma min_le :
$\forall\ n\ m :\ conat,\ le\ n\ m \rightarrow sim\ (min\ n\ m)\ n.$

Lemma max_le :
$\forall\ n\ m :\ conat,\ le\ n\ m \rightarrow sim\ (max\ n\ m)\ m.$

Lemma min_refl :
$\forall\ n :\ conat,\ sim\ (min\ n\ n)\ n.$

Lemma max_refl :
$\forall\ n :\ conat,\ sim\ (max\ n\ n)\ n.$

Lemma sim_min_max :
$\forall\ n\ m :\ conat,$
$sim\ (min\ n\ m)\ (max\ n\ m) \rightarrow sim\ n\ m.$

Lemma min_max :
$\forall\ a\ b :\ conat,\ sim\ (min\ a\ (max\ a\ b))\ a.$

Lemma max_min :
$\forall\ a\ b :\ conat,\ sim\ (max\ a\ (min\ a\ b))\ a.$

Lemma min_max_distr :
$\forall\ a\ b\ c :\ conat,$
$sim\ (min\ a\ (max\ b\ c))\ (max\ (min\ a\ b)\ (min\ a\ c)).$

Lemma max_min_distr :
$\forall\ a\ b\ c :\ conat,$
$sim\ (max\ a\ (min\ b\ c))\ (min\ (max\ a\ b)\ (max\ a\ c)).$

Zdefiniuj funkcję *div2*, która dzieli liczbę konaturalną przez 2 (cokolwiek to znaczy). Udowodnij jej właściwości.

Lemma $div2\_zero$:
$sim\ (div2\ zero)\ zero.$

Lemma $div2\_omega$:
$sim\ (div2$ omega) omega.

Lemma $div2\_succ$:
$\forall\ n :\ conat,\ sim\ (div2\ (succ\ (succ\ n)))\ (succ\ (div2\ n)).$

Lemma $div2\_add$:
$\forall\ n :\ conat,\ sim\ (div2\ (add\ n\ n))\ n.$

Lemma $le\_div2\_l\_aux$:
$\forall\ n\ m :\ conat,\ le\ n\ m \rightarrow le\ (div2\ n)\ m.$

Lemma le_div2_l :

$\forall\ n : conat,\ le\ (div2\ n)\ n.$

Lemma *le_div2* :
$\forall\ n\ m : conat,\ le\ n\ m \to le\ (div2\ n)\ (div2\ m).$

Zdefiniuj predykaty *Finite* i *Infinite*, które wiadomo co znaczą. Pokaż, że omega jest liczbą nieskończoną i że nie jest skończona, oraz że każda liczba nieskończona jest bipodobna do omegi. Pokaż też, że żadna liczba nie może być jednocześnie skończona i nieskończona.

Lemma *omega_not_Finite* :
$\neg$ *Finite* omega.

Lemma *Infinite_omega* :
Infinite omega.

Lemma *Infinite_omega'* :
$\forall\ n : conat,\ Infinite\ n \to sim\ n$ omega.

Lemma *Finite_Infinite* :
$\forall\ n : conat,\ Finite\ n \to Infinite\ n \to False.$

Zdefiniuj predykaty *Even* i *Odd*. Pokaż, że omega jest jednocześnie parzysta i nieparzysta. Pokaż, że jeżeli liczba jest jednocześnie parzysta i nieparzysta, to jest nieskończona. Wyciągnij z tego oczywisty wniosek. Pokaż, że każda liczba skończona jest parzysta albo nieparzysta.

Lemma *Even_zero* :
Even zero.

Lemma *Odd_zero* :
$\neg$ *Odd zero.*

Lemma *Even_Omega* :
Even omega.

Lemma *Odd_Omega* :
Odd omega.

Lemma *Even_Odd_Infinite* :
$\forall\ n : conat,\ Even\ n \to Odd\ n \to Infinite\ n.$

Lemma *Even_succ* :
$\forall\ n : conat,\ Odd\ n \to Even\ (succ\ n).$

Lemma *Odd_succ* :
$\forall\ n : conat,\ Even\ n \to Odd\ (succ\ n).$

Lemma *Even_succ_inv* :
$\forall\ n : conat,\ Odd\ (succ\ n) \to Even\ n.$

Lemma *Odd_succ_inv* :
$\forall\ n : conat,\ Even\ (succ\ n) \to Odd\ n.$

Lemma *Finite_Even_Odd* :

$\forall\ n : conat, Finite\ n \to Even\ n \lor Odd\ n.$

`Lemma` $Finite\_not\_both\_Even\_Odd$:
$\forall\ n : conat, Finite\ n \to \neg\ (Even\ n \land Odd\ n).$

`Lemma` $Even\_add\_Odd$:
$\forall\ n\ m : conat, Odd\ n \to Odd\ m \to Even\ (add\ n\ m).$

`Lemma` $Even\_add\_Even$:
$\forall\ n\ m : conat, Even\ n \to Even\ m \to Even\ (add\ n\ m).$

`Lemma` $Odd\_add\_Even\_Odd$:
$\forall\ n\ m : conat, Even\ n \to Odd\ m \to Odd\ (add\ n\ m).$

Było już o dodawaniu, przydałoby się powiedzieć też coś o odejmowaniu. Niestety, ale odejmowania liczb konaturalnych nie da się zdefiniować (a przynajmniej tak mi się wydaje). Nie jest to również coś, co można bezpośrednio udowodnić. Jest to fakt żyjący na metapoziomie, czyli mówiący coś o Coqu, a nie mówiący coś w Coqu. Jest jednak pewien wewnętrzny sposób by upewnić się, że odejmowanie faktycznie nie jest koszerne.

`Definition` $Sub\ (n\ m\ r : conat)$: `Prop` :=
$sim\ (add\ r\ m)\ n.$

W tym celu definiujemy relację *Sub*, która ma reprezentować wykres funkcji odejmującej, tzn. specyfikować, jaki jest związek argumentów funkcji z jej wynikiem.

`Definition` sub : `Type` :=
$\{f : conat \to conat \to conat\ |$
$\forall\ n\ m\ r : conat, f\ n\ m = r \leftrightarrow Sub\ n\ m\ r\}.$

Dzięki temu możemy napisać precyzyjny typ, który powinna mieć nasza funkcja - jest to funkcja biorąca dwie liczby konaturalne i zwracająca liczbę konaturalną, która jest poprawna i pełna względem wykresu.

`Lemma` Sub_nondet :
$\forall\ r : conat$, *Sub* `omega` `omega` r.

Niestety mimo, że definicja relacji *Sub* jest tak oczywista, jak to tylko możliwe, relacja ta nie jest wykresem żadnej funkcji, gdyż jest niedeterministyczna.

`Lemma` sub_cant_exist :
$sub \to False.$

Problem polega na tym, że `omega - omega` może być dowolną liczbą konaturalną. Bardziej obrazowo:

- Chcielibyśmy, żeby n - $n = 0$
- Chcielibyśmy, żeby $(n + 1)$ - $n = 1$
- Jednak dla $n =$ `omega` daje to `omega - omega` $= 0$ oraz `omega - omega` $= 1$, co prowadzi do sprzeczności

Dzięki temu możemy skonkludować, że typ *sub* jest pusty, a zatem pożądana przez nas funkcją odejmująca nie może istnieć.

Najbliższą odejmowaniu operacją, jaką możemy wykonać na liczbach konaturalnych, jest odejmowanie liczby naturalnej od liczby konaturalnej.

```
CoInductive Mul (n m r : conat) : Prop :=
{
    Mul' :
      (n = zero ∧ r = zero) ∨
      (m = zero ∧ r = zero) ∨
      ∃ n' m' r' : conat,
        n = succ n' ∧ m = succ m' ∧ Mul n' m r' ∧ sim r (add r' m);
}.

Ltac unmul H :=
  destruct H as [H]; decompose [or and ex] H; clear H; subst.

Lemma Mul_zero_l :
  ∀ n r : conat, Mul zero n r → sim r zero.

Lemma Mul_zero_r:
  ∀ n r : conat, Mul n zero r → sim r zero.

Lemma Mul_one_l :
  ∀ n r : conat, Mul (succ zero) n r → sim n r.

Lemma Mul_one_r :
  ∀ n r : conat, Mul n (succ zero) r → sim n r.

Lemma Mul_det :
  ∀ n m r1 r2 : conat, Mul n m r1 → Mul n m r2 → sim r1 r2.

Lemma Mul_comm :
  ∀ n m r : conat, Mul n m r → Mul m n r.

Lemma Mul_assoc :
  ∀ a b c d r1 r2 s1 s2 : conat,
    Mul a b r1 → Mul r1 c r2 →
    Mul b c s1 → Mul a s1 s2 → sim r2 s2.
```

Rozdział 16

X8: Strumienie

TODO: w tym rozdziale będą ćwiczenia dotyczące strumieni, czyli ogólnie wesołe koinduktywne zabawy, o których jeszcze nic nie napisałem.

```
CoInductive Stream (A : Type) : Type :=
{
    hd : A;
    tl : Stream A;
}.

Arguments hd {A}.
Arguments tl {A}.
```

16.1 Bipodobieństwo

```
CoInductive sim {A : Type} (s1 s2 : Stream A) : Prop :=
{
    hds : hd s1 = hd s2;
    tls : sim (tl s1) (tl s2);
}.

Lemma sim_refl :
  ∀ (A : Type) (s : Stream A), sim s s.

Lemma sim_sym :
  ∀ (A : Type) (s1 s2 : Stream A),
    sim s1 s2 → sim s2 s1.

Lemma sim_trans :
  ∀ (A : Type) (s1 s2 s3 : Stream A),
    sim s1 s2 → sim s2 s3 → sim s1 s3.
```

16.2 *sapp*

Zdefiniuj funkcję *sapp*, która konkatenuje dwa strumienie. Czy taka funkcja w ogóle ma sens?

```
CoFixpoint sapp {A : Type} (s1 s2 : Stream A) : Stream A :=
{|
    hd := hd s1;
    tl := sapp (tl s1) s2;
|}.

Lemma sapp_pointless :
  ∀ (A : Type) (s1 s2 : Stream A),
    sim (sapp s1 s2) s1.

Lemma map_id :
  ∀ (A : Type) (s : Stream A), sim (map (@id A) s) s.

Lemma map_compose :
  ∀ (A B C : Type) (f : A → B) (g : B → C) (s : Stream A),
    sim (map g (map f s)) (map (fun x ⇒ g (f x)) s).

(*
   CoFixpoint unzipWith
  {A B C : Type} (f : C -> A * B) (s : Stream C) : Stream A * Stream B
   *)
```

TODO: join : Stream (Stream A) -> Stream A, unzis

```
Lemma intersperse_merge_repeat :
  ∀ (A : Type) (x : A) (s : Stream A),
    sim (intersperse x s) (merge s (repeat x)).

(* Dlaczego s nie musi tu być indeksem? *)
Inductive Elem {A : Type} (x : A) (s : Stream A) : Prop :=
    | Elem_hd : x = hd s → Elem x s
    | Elem_tl : Elem x (tl s) → Elem x s.

Hint Constructors Elem.

Inductive Dup {A : Type} (s : Stream A) : Prop :=
    | Dup_hd : Elem (hd s) (tl s) → Dup s
    | Dup_tl : Dup (tl s) → Dup s.

Ltac inv H := inversion H; subst; clear H.

Require Import Arith.

Lemma Elem_nth :
  ∀ (A : Type) (x : A) (s : Stream A),
    Elem x s ↔ ∃ n : nat, nth n s = x.
```

```
Lemma nth_from :
  ∀ n m : nat,
    nth n (from m) = n + m.

Lemma Elem_from_add :
  ∀ n m : nat, Elem n (from m) →
    ∀ k : nat, Elem (k + n) (from m).

Lemma Elem_from :
  ∀ n m : nat, Elem n (from m) ↔ m ≤ n.

Lemma Dup_spec :
  ∀ (A : Type) (s : Stream A),
    Dup s ↔ ∃ n m : nat, n ≠ m ∧ nth n s = nth m s.

Lemma NoDup_from :
  ∀ n : nat, ¬ Dup (from n).

(*  To samo: dlaczego s nie musi być indeksem? *)
Inductive Exists {A : Type} (P : A → Prop) (s : Stream A) : Prop :=
    | Exists_hd : P (hd s) → Exists P s
    | Exists_tl : Exists P (tl s) → Exists P s.

Lemma Exists_spec :
  ∀ (A : Type) (P : A → Prop) (s : Stream A),
    Exists P s ↔ ∃ n : nat, P (nth n s).

CoInductive Forall {A : Type} (s : Stream A) (P : A → Prop) : Prop :=
{
    Forall_hd : P (hd s);
    Forall_tl : Forall (tl s) P;
}.

Lemma Forall_spec :
  ∀ (A : Type) (s : Stream A) (P : A → Prop),
    Forall s P ↔ ∀ n : nat, P (nth n s).

Lemma Forall_spec' :
  ∀ (A : Type) (s : Stream A) (P : A → Prop),
    Forall s P ↔ ∀ x : A, Elem x s → P x.

Lemma Forall_Exists :
  ∀ (A : Type) (P : A → Prop) (s : Stream A),
    Forall s P → Exists P s.

CoInductive Substream {A : Type} (s1 s2 : Stream A) : Prop :=
{
    n : nat;
    p : hd s1 = nth n s2;
    Substream' : Substream (tl s1) (drop (S n) s2);
```

}.

Lemma *drop_tl* :
  ∀ (*A* : Type) (*n* : *nat*) (*s* : *Stream A*),
    drop n (*tl s*) = *drop* (*S n*) *s*.

Lemma *tl_drop* :
  ∀ (*A* : Type) (*n* : *nat*) (*s* : *Stream A*),
    tl (*drop n s*) = *drop n* (*tl s*).

Lemma *nth_drop* :
  ∀ (*A* : Type) (*n m* : *nat*) (*s* : *Stream A*),
    nth n (*drop m s*) = *nth* (*n* + *m*) *s*.

Lemma *drop_drop* :
  ∀ (*A* : Type) (*n m* : *nat*) (*s* : *Stream A*),
    drop m (*drop n s*) = *drop* (*n* + *m*) *s*.

Require Import *Arith*.

Lemma *Substream_tl* :
  ∀ (*A* : Type) (*s1 s2* : *Stream A*),
    Substream s1 s2 → *Substream* (*tl s1*) (*tl s2*).

Lemma *Substream_drop* :
  ∀ (*A* : Type) (*n* : *nat*) (*s1 s2* : *Stream A*),
    Substream s1 s2 → *Substream* (*drop n s1*) (*drop n s2*).

Lemma *hd_drop* :
  ∀ (*A* : Type) (*n* : *nat*) (*s* : *Stream A*),
    hd (*drop n s*) = *nth n s*.

Lemma *Substream_drop_add* :
  ∀ (*A* : Type) (*n m* : *nat*) (*s1 s2* : *Stream A*),
    Substream s1 (*drop n s2*) → *Substream s1* (*drop* (*n* + *m*) *s2*).

Lemma *Substream_trans* :
  ∀ (*A* : Type) (*s1 s2 s3* : *Stream A*),
    Substream s1 s2 → *Substream s2 s3* → *Substream s1 s3*.

Lemma *Substream_not_antisymm* :
  ∃ (*A* : Type) (*s1 s2* : *Stream A*),
    Substream s1 s2 ∧ *Substream s2 s1* ∧ ¬ *sim s1 s2*.

Inductive *Suffix* {*A* : Type} : *Stream A* → *Stream A* → Prop :=
    | *Suffix_refl* :
        ∀ *s* : *Stream A*, *Suffix s s*
    | *Suffix_tl* :
        ∀ *s1 s2* : *Stream A*,
          Suffix (*tl s1*) *s2* → *Suffix s1 s2*.

Fixpoint *snoc* {*A* : Type} (*x* : *A*) (*l* : *list A*) : *list A* :=

```
match l with
      | nil ⇒ cons x nil
      | cons h t ⇒ cons h (snoc x t)
end.

Lemma Suffix_spec :
  ∀ (A : Type) (s1 s2 : Stream A),
    Suffix s1 s2 ↔ ∃ l : list A, s1 = lsapp l s2.

Definition Incl {A : Type} (s1 s2 : Stream A) : Prop :=
  ∀ x : A, Elem x s1 → Elem x s2.

Definition SetEquiv {A : Type} (s1 s2 : Stream A) : Prop :=
  Incl s1 s2 ∧ Incl s2 s1.

Lemma sim_Elem :
  ∀ (A : Type) (x : A) (s1 s2 : Stream A),
    sim s1 s2 → Elem x s1 → Elem x s2.

Lemma sim_Incl :
  ∀ (A : Type) (s1 s1' s2 s2' : Stream A),
    sim s1 s1' → sim s2 s2' → Incl s1 s2 → Incl s1' s2'.

Lemma sim_SetEquiv :
  ∀ (A : Type) (s1 s1' s2 s2' : Stream A),
    sim s1 s1' → sim s2 s2' → SetEquiv s1 s2 → SetEquiv s1' s2'.

Definition scons {A : Type} (x : A) (s : Stream A) : Stream A :=
{|
    hd := x;
    tl := s;
|}.

Inductive SPermutation {A : Type} : Stream A → Stream A → Prop :=
    | SPerm_refl :
        ∀ s : Stream A, SPermutation s s
    | SPerm_skip :
        ∀ (x : A) (s1 s2 : Stream A),
          SPermutation s1 s2 → SPermutation (scons x s1) (scons x s2)
    | SPerm_swap :
        ∀ (x y : A) (s1 s2 : Stream A),
          SPermutation s1 s2 →
            SPermutation (scons x (scons y s1)) (scons y (scons x s2))
    | SPerm_trans :
        ∀ s1 s2 s3 : Stream A,
          SPermutation s1 s2 → SPermutation s2 s3 → SPermutation s1 s3.

Hint Constructors SPermutation.

(*  TODO *)
```

```
Require Import Permutation.

Lemma lsapp_scons :
  ∀ (A : Type) (l : list A) (x : A) (s : Stream A),
    lsapp l (scons x s) = lsapp (snoc x l) s.

Lemma SPermutation_Permutation_lsapp :
  ∀ (A : Type) (l1 l2 : list A) (s1 s2 : Stream A),
    Permutation l1 l2 → SPermutation s1 s2 →
      SPermutation (lsapp l1 s1) (lsapp l2 s2).

Lemma take_drop :
  ∀ (A : Type) (n : nat) (s : Stream A),
    s = lsapp (take n s) (drop n s).

Lemma take_add :
  ∀ (A : Type) (n m : nat) (s : Stream A),
    take (n + m) s = List.app (take n s) (take m (drop n s)).

Lemma SPermutation_spec :
  ∀ (A : Type) (s1 s2 : Stream A),
    SPermutation s1 s2 ↔
    ∃ n : nat,
      Permutation (take n s1) (take n s2) ∧
      drop n s1 = drop n s2.
```

Strumienie za pomocą przybliżeń.

```
Print take.

Inductive Vec (A : Type) : nat → Type :=
    | vnil : Vec A 0
    | vcons : ∀ n : nat, A → Vec A n → Vec A (S n).

Arguments vnil {A}.
Arguments vcons {A} {n}.

Definition vhd {A : Type} {n : nat} (v : Vec A (S n)) : A :=
match v with
    | vcons h _ ⇒ h
end.

Definition vtl {A : Type} {n : nat} (v : Vec A (S n)) : Vec A n :=
match v with
    | vcons _ t ⇒ t
end.

Require Import Program.Equality.

Lemma vhd_vtl :
  ∀ (A : Type) (n : nat) (v : Vec A (S n)),
    v = vcons (vhd v) (vtl v).
```

```
Fixpoint vtake {A : Type} (s : Stream A) (n : nat) : Vec A n :=
match n with
    | 0 ⇒ vnil
    | S n' ⇒ vcons (hd s) (vtake (tl s) n')
end.

Fixpoint vtake' {A : Type} (s : Stream A) (n : nat) : Vec A (S n) :=
match n with
    | 0 ⇒ vcons (hd s) vnil
    | S n' ⇒ vcons (hd s) (vtake' (tl s) n')
end.

CoFixpoint unvtake {A : Type} (f : ∀ n : nat, Vec A (S n)) : Stream A :=
{|
    hd := vhd (f 0);
    tl :=
      unvtake (fun n : nat ⇒ vtl (f (S n)))
|}.

Fixpoint vnth {A : Type} {n : nat} (v : Vec A n) (k : nat) : option A :=
match v, k with
    | vnil, _ ⇒ None
    | vcons h t, 0 ⇒ Some h
    | vcons h t, S k' ⇒ vnth t k'
end.

Ltac depdestr x :=
  let x' := fresh "x"in remember x as x'; dependent destruction x'.

Lemma unvtake_vtake' :
  ∀ (A : Type) (n : nat) (f : ∀ n : nat, Vec A (S n)),
    (∀ m1 m2 k : nat, k ≤ m1 → k ≤ m2 →
      vnth (f m1) k = vnth (f m2) k) →
        vtake' (unvtake f) n = f n.

Lemma vtake_unvtake :
  ∀ (A : Type) (s : Stream A),
    sim (unvtake (vtake' s)) s.
```

Pomysł dawno zapomniany: induktywne specyfikacje funkcji.

```
Inductive Filter {A : Type} (f : A → bool) : Stream A → Stream A → Prop :=
    | Filter_true :
        ∀ s r r' : Stream A,
          f (hd s) = true → Filter f (tl s) r →
            hd r' = hd s → tl r' = r → Filter f s r'
    | Filter_false :
        ∀ s r : Stream A,
```

f $(hd\ s) = false \rightarrow Filter\ f\ (tl\ s)\ r \rightarrow Filter\ f\ s\ r.$

`Lemma` *Filter_bad* :
  $\forall$ $(A$: `Type`$)$ $(f : A \rightarrow bool)$ $(s\ r : Stream\ A)$,
    $Filter\ f\ s\ r \rightarrow (\forall\ x : A, f\ x = false) \rightarrow False.$

`CoInductive` *Filter'* $\{A$: `Type`$\}$ $(f : A \rightarrow bool)$ $(s\ r : Stream\ A)$: `Prop` :=
{
    Filter'_true :
      $f\ (hd\ s) = true \rightarrow hd\ s = hd\ r \wedge Filter'\ f\ (tl\ s)\ (tl\ r);$
    Filter'_false :
      $f\ (hd\ s) = false \rightarrow Filter'\ f\ (tl\ s)\ r;$
}.

`Lemma` *Filter'_const_false* :
  $\forall$ $(A$: `Type`$)$ $(s\ r : Stream\ A)$,
    Filter' (`fun` _ $\Rightarrow$ *false*) $s\ r.$

`Lemma` *Filter'_const_true* :
  $\forall$ $(A$: `Type`$)$ $(s\ r : Stream\ A)$,
    Filter' (`fun` _ $\Rightarrow$ *true*) $s\ r \rightarrow sim\ s\ r.$

Rozdział 17

X9: Kolisty

```
Require Import book.X3.
```

Ten rozdział będzie o kolistach, czyli koinduktywnych odpowiednikach list różniących się od nich tym, że mogą być potencjalnie nieskończone.

```
CoInductive coList (A : Type) : Type :=
{
    uncons : option (A × coList A);
}.
Arguments uncons {A}.
```

Przydatny będzie następujący, dość oczywisty fakt dotyczący równości kolist.

```
Lemma eq_uncons :
  ∀ (A : Type) (l1 l2 : coList A),
    uncons l1 = uncons l2 → l1 = l2.
```

Zdefiniuj relację bipodobieństwa dla kolist. Udowodnij, że jest ona relacją równoważności. Z powodu konfliktu nazw bipodobieństwo póki co nazywać się będzie *lsim*.

```
Lemma lsim_refl :
  ∀ (A : Type) (l : coList A), lsim l l.

Lemma lsim_symm :
  ∀ (A : Type) (l1 l2 : coList A),
    lsim l1 l2 → lsim l2 l1.

Lemma lsim_trans :
  ∀ (A : Type) (l1 l2 l3 : coList A),
    lsim l1 l2 → lsim l2 l3 → lsim l1 l3.
```

Przyda się też instancja klasy *Equivalence*, żebyśmy przy dowodzeniu o *lsim* mogli używać taktyk `reflexivity`, `symmetry` oraz `rewrite`.

```
Instance Equivalence_lsim (A : Type) : Equivalence (@lsim A).
Proof.
```

```
  esplit; red.
    apply lsim_refl.
    apply lsim_symm.
    apply lsim_trans.
Defined.
```

Zdefiniuj *conil*, czyli kolistę pustą, oraz *cocons*, czyli funkcję, która dokleja do kolisty nową głowę. Udowodnij, że *cocons* zachowuje i odbija bipodobieństwo.

```
Lemma lsim_cocons :
  ∀ (A : Type) (x y : A) (l1 l2 : coList A),
    x = y → lsim l1 l2 → lsim (cocons x l1) (cocons y l2).

Lemma lsim_cocons_inv :
  ∀ (A : Type) (x y : A) (l1 l2 : coList A),
    lsim (cocons x l1) (cocons y l2) → x = y ∧ lsim l1 l2.
```

Przygodę z funkcjami na kolistach zaczniemy od długości. Tak jak zwykła, induktywna lista ma długość wyrażającą się liczbą naturalną, tak też i długość kolisty można wyrazić za pomocą liczby konaturalnej.

Napisz funkcję *len*, która oblicza długość kolisty. Pokaż, że bipodobne kolisty mają tę samą długość. Długość kolisty pustej oraz *cocons*a powinny być oczywiste.

```
Require Import X7.

Lemma sim_len :
  ∀ (A : Type) (l1 l2 : coList A),
    lsim l1 l2 → sim (len l1) (len l2).

Lemma len_conil :
  ∀ A : Type,
    len (@conil A) = zero.

Lemma len_cocons :
  ∀ (A : Type) (x : A) (l : coList A),
    len (cocons x l) = succ (len l).
```

Zdefiniuj funkcję *snoc*, która dostawia element na koniec kolisty.

```
Lemma snoc_cocons :
  ∀ (A : Type) (l : coList A) (x y : A),
    lsim (snoc (cocons x l) y) (cocons x (snoc l y)).

Lemma len_snoc :
  ∀ (A : Type) (l : coList A) (x : A),
    sim (len (snoc l x)) (succ (len l)).
```

Zdefiniuj funkcję *app*, która skleja dwie kolisty. Czy jest to w ogóle możliwe? Czy taka funkcja ma sens? Porównaj z przypadkiem sklejania strumieni.

```
Lemma app_conil_l :
```

$\forall$ (A : Type) (l : $coList$ A),
  app $conil$ l = l.

Lemma app_conil_r :
 $\forall$ (A : Type) (l : $coList$ A),
  $lsim$ (app l $conil$) l.

Lemma app_cocons_l :
 $\forall$ (A : Type) (x : A) ($l1$ $l2$: $coList$ A),
  $lsim$ (app ($cocons$ x $l1$) $l2$) ($cocons$ x (app $l1$ $l2$)).

Lemma len_app :
 $\forall$ (A : Type) ($l1$ $l2$: $coList$ A),
  sim (len (app $l1$ $l2$)) (add (len $l1$) (len $l2$)).

Lemma $snoc\_app$:
 $\forall$ (A : Type) ($l1$ $l2$: $coList$ A) (x : A),
  $lsim$ ($snoc$ (app $l1$ $l2$) x) (app $l1$ ($snoc$ $l2$ x)).

Lemma app_snoc_l :
 $\forall$ (A : Type) ($l1$ $l2$: $coList$ A) (x : A),
  $lsim$ (app ($snoc$ $l1$ x) $l2$) (app $l1$ ($cocons$ x $l2$)).

Lemma app_assoc :
 $\forall$ (A : Type) ($l1$ $l2$ $l3$: $coList$ A),
  $lsim$ (app (app $l1$ $l2$) $l3$) (app $l1$ (app $l2$ $l3$)).

Zdefiniuj funkcję $lmap$, która aplikuje funkcję $f : A \to B$ do każdego elementu kolisty.
TODO: wyklarować, dlaczego niektóre rzeczy mają "l" na początku nazwy

Lemma $lmap\_conil$:
 $\forall$ (A B : Type) (f : $A \to B$),
  $lmap$ f $conil$ = $conil$.

Lemma $lmap\_cocons$:
 $\forall$ (A B : Type) (f : $A \to B$) (x : A) (l : $coList$ A),
  $lsim$ ($lmap$ f ($cocons$ x l)) ($cocons$ (f x) ($lmap$ f l)).

Lemma len_lmap :
 $\forall$ (A B : Type) (f : $A \to B$) (l : $coList$ A),
  sim (len ($lmap$ f l)) (len l).

Lemma $lmap\_snoc$:
 $\forall$ (A B : Type) (f : $A \to B$) (l : $coList$ A) (x : A),
  $lsim$ ($lmap$ f ($snoc$ l x)) ($snoc$ ($lmap$ f l) (f x)).

Lemma $lmap\_app$:
 $\forall$ (A B : Type) (f : $A \to B$) ($l1$ $l2$: $coList$ A),
  $lsim$ ($lmap$ f (app $l1$ $l2$)) (app ($lmap$ f $l1$) ($lmap$ f $l2$)).

Lemma $lmap\_id$:

$\forall$ (A : `Type`) (l : *coList* A),
lsim (*lmap* *id* l) l.

`Lemma` *lmap_compose* :
$\forall$ (A B C : `Type`) (f : $A \rightarrow B$) (g : $B \rightarrow C$) (l : *coList* A),
lsim (*lmap* g (*lmap* f l)) (*lmap* (`fun` $x \Rightarrow g$ (f x)) l).

`Lemma` *lmap_ext* :
$\forall$ (A B : `Type`) (f g : $A \rightarrow B$) (l : *coList* A),
($\forall$ x : A, f x = g x) $\rightarrow$ *lsim* (*lmap* f l) (*lmap* g l).

Zdefiniuj funkcję *iterate*, która tworzy nieskończoną kolistę przez iterowanie funkcji f poczynając od pewnego ustalonego elementu.

`Lemma` *len_iterate* :
$\forall$ (A : `Type`) (f : $A \rightarrow A$) (x : A),
sim (*len* (*iterate* f x)) `omega`.

Zdefiniuj funkcję *piterate*, która tworzy kolistę przez iterowanie funkcji częściowej f : A $\rightarrow$ *option* B poczynając od pewnego ustalonego elementu.

Zdefiniuj funkcję *zipW*, która bierze funkcję f : $A \rightarrow B \rightarrow C$ oraz dwie kolisty *l1* i *l2* i zwraca kolistę, której elementy powstają z połączenia odpowiadających sobie elementów *l1* i *l2* za pomocą funkcji f.

`Lemma` *zipW_conil_l* :
$\forall$ (A B C : `Type`) (f : $A \rightarrow B \rightarrow C$) ($l$: *coList* B),
lsim (*zipW* f *conil* l) *conil*.

`Lemma` *zipW_conil_r* :
$\forall$ (A B C : `Type`) (f : $A \rightarrow B \rightarrow C$) (*l1* : *coList* A) (*l2* : *coList* B),
sim (*len* (*zipW* f *l1* *l2*)) (*min* (*len* *l1*) (*len* *l2*)).

`Lemma` *len_zipW* :
$\forall$ (A B C : `Type`) (f : $A \rightarrow B \rightarrow C$) (*l1* : *coList* A) (*l2* : *coList* B),
sim (*len* (*zipW* f *l1* *l2*)) (*min* (*len* *l1*) (*len* *l2*)).

Napisz funkcję *scan*, która przekształca l : *coList* A w kolistę sum częściowych działania f : $B \rightarrow A \rightarrow B$.

`Lemma` *scan_conil* :
$\forall$ (A B : `Type`) (f : $B \rightarrow A \rightarrow B$) ($b$: B),
lsim (*scan* *conil* f b) *conil*.

`Lemma` *scan_cocons* :
$\forall$ (A B : `Type`) (x : A) (l : *coList* A) (f : $B \rightarrow A \rightarrow B$) ($b$: B),
lsim (*scan* (*cocons* x l) f b) (*cocons* b (*scan* l f (f b x))).

`Lemma` *len_scan* :
$\forall$ (A B : `Type`) (l : *coList* A) (f : $B \rightarrow A \rightarrow B$) ($b$: B),

$sim\ (len\ (scan\ l\ f\ b))\ (len\ l)$.

TODO: snoc, app, map, iterate, piterate.

Napisz funkcję *intersperse*, która działa analogicznie jak dla list.

```
Lemma intersperse_conil :
  ∀ (A : Type) (x : A),
    lsim (intersperse x conil) conil.
```

Pułapka: czy poniższe twierdzenie jest prawdziwe?

```
Lemma len_intersperse :
  ∀ (A : Type) (x : A) (l : coList A),
    sim (len (intersperse x l)) (succ (add (len l) (len l))).
```

Napisz rekurencyjną funkcję *splitAt*. *splitAt l n* zwraca *Some* (*begin*, *x*, *rest*), gdzie *begin* jest listą reprezentującą początkowy fragment kolisty l o długości n, x to element l znajdujący się na pozycji n, zaś *rest* to kolista będącą tym, co z kolisty l pozostanie po zabraniu z niej l oraz x. Jeżeli l nie ma fragmentu początkowego o długości n, funkcja *splitAt* zwraca *None*.

Funkcji *splitAt* można użyć do zdefiniowania całej gamy funkcji działających na kolistach - rozbierających ją na kawałki, wstawiających, zamieniających i usuwających elementy, etc.

```
Definition nth {A : Type} (l : coList A) (n : nat) : option A :=
match splitAt l n with
    | None ⇒ None
    | Some (_, x, _) ⇒ Some x
end.

Definition take {A : Type} (l : coList A) (n : nat) : option (list A) :=
match splitAt l n with
    | None ⇒ None
    | Some (l, _, _) ⇒ Some l
end.

Definition drop {A : Type} (l : coList A) (n : nat) : option (coList A) :=
match splitAt l n with
    | None ⇒ None
    | Some (_, _, l) ⇒ Some l
end.

Fixpoint fromList {A : Type} (l : list A) : coList A :=
match l with
    | [] ⇒ conil
    | h :: t ⇒ cocons h (fromList t)
end.

Definition insert {A : Type} (l : coList A) (n : nat) (x : A)
  : option (coList A) :=
match splitAt l n with
```

```
    | None ⇒ None
    | Some (start, mid, rest) ⇒
        Some (app (fromList start) (cocons x (cocons mid rest)))
end.
```

```
Definition remove {A : Type} (l : coList A) (n : nat)
  : option (coList A) :=
match splitAt l n with
    | None ⇒ None
    | Some (start, _, rest) ⇒ Some (app (fromList start) rest)
end.
```

Zdefiniuj predykaty *Finite* oraz *Infinite*, które są spełnione, odpowiednio, przez skończone i nieskończone kolisty. Zastanów się dobrze, czy definicje powinny być induktywne, czy koinduktywne.

Udowodnij własności tych predykatów oraz sprawdź, które kolisty i operacje je spełniają.

```
Lemma Finite_not_Infinite :
  ∀ (A : Type) (l : coList A),
    Finite l → Infinite l → False.

Lemma sim_Infinite :
  ∀ (A : Type) (l1 l2 : coList A),
    lsim l1 l2 → Infinite l1 → Infinite l2.

Lemma len_Finite :
  ∀ (A : Type) (l : coList A),
    Finite l → len l ≠ omega.

Lemma len_Infinite :
  ∀ (A : Type) (l : coList A),
    len l = omega → Infinite l.

Lemma Finite_snoc :
  ∀ (A : Type) (l : coList A) (x : A),
    Finite l → Finite (snoc l x).

Lemma Infinite_snoc :
  ∀ (A : Type) (l : coList A) (x : A),
    Infinite l → lsim (snoc l x) l.

Lemma Infinite_app_l :
  ∀ (A : Type) (l1 l2 : coList A),
    Infinite l1 → Infinite (app l1 l2).

Lemma Infinite_app_r :
  ∀ (A : Type) (l1 l2 : coList A),
    Infinite l2 → Infinite (app l1 l2).

Lemma Finite_app :
```

$\forall$ (A : Type) ($l1$ $l2$: $coList$ A),
  $Finite$ $l1$ $\to$ $Finite$ $l2$ $\to$ $Finite$ (app $l1$ $l2$).

Lemma $Finite\_app\_conv$:
  $\forall$ (A : Type) ($l1$ $l2$: $coList$ A),
    $Finite$ (app $l1$ $l2$) $\to$ $Finite$ $l1$ $\lor$ $Finite$ $l2$.

Lemma $Finite\_lmap$:
  $\forall$ (A B : Type) (f : $A \to B$) (l : $coList$ A),
    $Finite$ l $\to$ $Finite$ ($lmap$ f l).

Lemma $Infinite\_lmap$:
  $\forall$ (A B : Type) (f : $A \to B$) (l : $coList$ A),
    $Infinite$ l $\to$ $Infinite$ ($lmap$ f l).

Lemma $Infinite\_iterate$:
  $\forall$ (A : Type) (f : $A \to A$) (x : A),
    $Infinite$ ($iterate$ f x).

Lemma $piterate\_Finite$:
  $\forall$ (A : Type) (f : $A \to option$ A) (x : A),
    $Finite$ ($piterate$ f x) $\to \exists$ x : A, f x = $None$.

Lemma $Finite\_zipW\_l$:
  $\forall$ (A B C : Type) (f : $A \to B \to C$) ($l1$: $coList$ A) ($l2$: $coList$ B),
    $Finite$ $l1$ $\to$ $Finite$ ($zipW$ f $l1$ $l2$).

Lemma $Finite\_zipW\_r$:
  $\forall$ (A B C : Type) (f : $A \to B \to C$) ($l1$: $coList$ A) ($l2$: $coList$ B),
    $Finite$ $l2$ $\to$ $Finite$ ($zipW$ f $l1$ $l2$).

Lemma $Infinite\_zipW\_l$:
  $\forall$ (A B C : Type) (f : $A \to B \to C$) ($l1$: $coList$ A) ($l2$: $coList$ B),
    $Infinite$ ($zipW$ f $l1$ $l2$) $\to$ $Infinite$ $l1$.

Lemma $Infinite\_zipW\_r$:
  $\forall$ (A B C : Type) (f : $A \to B \to C$) ($l1$: $coList$ A) ($l2$: $coList$ B),
    $Infinite$ ($zipW$ f $l1$ $l2$) $\to$ $Infinite$ $l1$.

Lemma $Infinite\_splitAt$:
  $\forall$ (A : Type) (n : nat) (l : $coList$ A),
    $Infinite$ l $\to$
      $\exists$ ($start$: $list$ A) (x : A) ($rest$: $coList$ A),
        $splitAt$ l n = $Some$ ($start$, x, $rest$).

Zdefiniuj predykaty $Exists$ P oraz $Forall$ P, które są spełnione przez kolisty, których odpowiednio jakiś/wszystkie elementy spełniają predykat P. Zastanów się dobrze, czy definicje powinny być induktywne, czy koinduktywne.

Sprawdź, które z praw de Morgana zachodzą.

Inductive $Exists$ {A : Type} (P : $A \to$ Prop) : $coList$ A $\to$ Prop :=

```
  | Exists_hd :
      ∀ (l : coList A) (h : A) (t : coList A),
        uncons l = Some (h, t) → P h → Exists P l
  | Exists_tl :
      ∀ (l : coList A) (h : A) (t : coList A),
        uncons l = Some (h, t) → Exists P t → Exists P l.

CoInductive All {A : Type} (P : A → Prop) (l : coList A) : Prop :=
{
  All' :
    uncons l = None ∨
    ∃ (h : A) (t : coList A),
      uncons l = Some (h, t) ∧ P h ∧ All P t;
}.

Lemma Exists_not_All :
  ∀ (A : Type) (P : A → Prop) (l : coList A),
    Exists P l → ¬ All (fun x : A ⇒ ¬ P x) l.
```