

Spis treści

1	A : \	Vstep	12
	1.1	Cel	12
	1.2	Wybór	12
	1.3	Programowanie i dowodzenie	13
		1.3.1 Alan Turing i jego maszyna	13
		1.3.2 Alonzo Church i rachunek	13
		1.3.3 Martin-Löf, Coquand, CoC, CIC i Coq	14
	1.4	Filozofia i matematyka	14
		1.4.1 Konstruktywizm	14
		1.4.2 Praktyka	15
		1.4.3 Homofobia ekhm, homotopia, czyli quo vadimus?	16
	1.5	Literatura	16
		1.5.1 Książki	16
		1.5.2 Blogi	18
		1.5.3 Inne	19
	1.6	Sprawy techniczne	19
2		ogika	2 0
	2.1	Logika klasyczna i konstruktywna	20
	2.2	Dedukcja naturalna i taktyki	21
	2.3	Konstruktywny rachunek zdań	21
		2.3.1 Implikacja	22
		2.3.2 Falsz	25
		2.3.3 Prawda	26
		2.3.4 Negacja	27
		2.3.5 Koniunkcja	29
		2.3.6 Równoważność zdaniowa	30
		2.3.7 Dysjunkcja	31
	2.4	Konstruktywny rachunek kwantyfikatorów	32
		2.4.1 Kwantyfikacja uniwersalna	32
		2.4.2 Kwantyfikacja egzystencjalna	34
	2.5	Paradoks golibrody	35
	2.6	Paradoks pieniadza i kebaba	37

	2.7	Kombinatory taktyk
		2.7.1 ; (średnik)
		2.7.2 (alternatywa)
		2.7.3 idtac, do oraz repeat
		2.7.4 try i fail
	2.8	Zadania
		2.8.1 Konstruktywny rachunek zdań
		2.8.2 Konstruktywny rachunek kwantyfikatorów
		2.8.3 Klasyczny rachunek zdań (i kwantyfikatorów) 45
	2.9	Paradoks pijoka
	2.10	Ściąga
		Konkluzja
	2.11	Tronkrazja
3	C: F	Podstawy teorii typów - TODO 50
	3.1	Typy i termy
	3.2	Typy i termy, kanoniczność i uzasadnienie reguł eliminacji 51
	3.3	Typy a zbiory
	3.4	Uniwersa
	3.5	Pięć rodzajów reguł
	3.3	3.5.1 Reguly formacji
		3.5.2 Reguly wprowadzania
		3.5.3 Reguly eliminacji
		3.5.4 Reguly obliczania
		3.5.5 Reguły unikalności
		o.o.o Rogary amazamosor
4	D1:	Indukcja i rekursja 64
	4.1	Typy induktywne
		4.1.1 Enumeracje
		4.1.2 Konstruktory rekurencyjne
		4.1.3 Typy polimorficzne i właściwości konstruktorów
		4.1.4 Listy, czyli parametry + rekursja
		4.1.5 Przydatne komendy
		4.1.6 Ważne typy induktywne
		4.1.7 Kiedy typ induktywny jest pusty?
	4.2	Induktywne zdania i predykaty
		4.2.1 Induktywne zdania
		4.2.2 Induktywne predykaty
		4.2.3 Indukcja po dowodzie
		4.2.4 Definicje stałych i spójników logicznych
		4.2.5 Równość
		4.2.6 Indukcja wzajemna
	4.3	Różne
		4.3.1 Rodziny typów induktywnych

		4.3.2 Indukcja wzajemna a indeksowane rodziny typów	01
		4.3.3 Sumy zależne i podtypy	02
		4.3.4 Kwantyfikacja egzystencjalna	04
	4.4	Wyższe czary	05
		4.4.1 Przypomnienie	05
		4.4.2 Indukcja-indukcja	12
		4.4.3 Indukcja-rekursja	20
		4.4.4 Indeksowana indukcja-rekursja	26
		4.4.5 Indukcja-indukcja-rekursja	27
		4.4.6 Najstraszniejszy potfur	28
	4.5	Dobre, złe i podejrzane typy induktywne	28
			28
			30
		4.5.3 Twierdzenie Cantora jako młot na negatywność	34
		4.5.4 Poradnik rozpoznawania negatywnych typów induktywnych 1	37
			40
			41
			51
			60
	4.6		64
5		,	66
	5.1	U U	67
	5.2	3 0	68
	5.3	<i>3</i> 1 1	69
	5.4	o	72
	5.5	3	78
	5.6	v v	85
	5.7		89
	5.8	3	91
	5.9		98
	5.10	Metoda induktywnej dziedziny 2	03
	5.11	Plugin Equations	10
	5.12	Podsumowanie	10
c	D0:		
6	D2ip		$\frac{11}{11}$
	6.1	0 1 0 0 /	11
	6.2	9 ()	15
	6.3	,	15
	6.4	0 0 0 1	15
	6.5	3 ()	22
	6.6	Ü	22
	6.7	Rząd rżnie głupa, czyli o pierwszym i wyższym rzędzie	28

	6.8	Rekursja wyższego rzędu (TODO)	228
7	D3 : 7.1 7.2 7.3 7.4	Logika boolowska, rozstrzygalność i reflekcja [TODO] Logika boolowska	230 231 233
	7.5 7.6 7.7 7.8	Typy hybrydowe	$\frac{236}{237}$
8	D4:	Arytmetyka Peano	238
	8.1 8.2 8.3	$\begin{array}{llllllllllllllllllllllllllllllllllll$	238 238 239 239 240 240 241 242 243 243 246 246 246
	8.5	8.4.2 Rozstrzygalność równości	$\frac{247}{247}$
9	D5 : 9.1	Listy Proste funkcje 9.1.1 isEmpty 9.1.2 length 9.1.3 snoc 9.1.4 app 9.1.5 rev 9.1.6 map	249 249 250 250

	9.1.7	join	253
	9.1.8	bind	253
	9.1.9	replicate	254
	9.1.10	iterate i iter	
	9.1.11	head, tail i uncons	255
	9.1.12	last, init i unsnoc	258
	9.1.13	nth	262
	9.1.14	$take \dots \dots$	264
		drop	266
	9.1.16	cycle	269
	9.1.17	splitAt	270
	9.1.18	insert	274
	9.1.19	replace	277
		remove	281
	9.1.21	zip	284
	9.1.22	$\stackrel{ ext{-}}{unzip}$	287
	9.1.23	zip With	287
	9.1.24	unzipWith	289
9.2		je z predykatem boolowskim	289
	9.2.1	any	289
	9.2.2		292
	9.2.3	find i findLast	294
	9.2.4	removeFirst i removeLast	297
	9.2.5	findIndex	301
	9.2.6	count	305
	9.2.7	filter	307
	9.2.8	partition	310
	9.2.9	findIndices	311
	9.2.10	$takeWhile \ i \ dropWhile \ . \ . \ . \ . \ . \ . \ . \ . \ . \ $	
	9.2.11	span	316
9.3	Sekcja	mocno ad hoc	319
	9.3.1	pmap	319
9.4	Bardzi	ej skomplikowane funkcje	323
	9.4.1	intersperse	323
9.5	Proste	predykaty	326
	9.5.1	elem	326
	9.5.2	In	330
	9.5.3	NoDup	334
	9.5.4	Dup	336
	9.5.5	$\stackrel{-}{Rep}$	339
	9.5.6	Exists	341
	9.5.7	For all	344

		$9.5.8 At Least \dots \dots \dots$				 		 					 	٠	347
		9.5.9 Exactly				 		 					 		352
		$9.5.10 \ AtMost \dots \dots \dots$				 		 					 		354
	9.6	Relacje między listami				 		 					 		355
		9.6.1 Listy jako termy				 		 					 		355
		9.6.2 Prefiksy				 		 					 		359
		9.6.3 Sufiksy				 		 					 		364
		9.6.4 Listy jako ciągi													365
		9.6.5 Zawieranie													
		9.6.6 Listy jako zbiory													375
		9.6.7 Listy jako multizbiory													377
		9.6.8 Listy jako cykle													386
	9.7	Niestandardowe reguły indukcyjne .													
	9.,	9.7.1 Palindromy													392
		5.,	-	-	-	-	-		-		-	-		-	00-
10		Rekordy, klasy i moduły - TODO													395
	10.1	Rekordy (TODO)				 		 					 		395
	10.2	Klasy (TODO)				 		 					 		396
	10.3	Moduły (TODO)				 		 					 		400
	- 0														401
11		Funkcje													401
		Funkcje													
		Aksjomat ekstensjonalności													403
		Odwrotności i izomorfizmy (TODO)													404
		Skracalność (TODO)													
		Injekcje													
		Surjekcje													
		Bijekcje													411
		Inwolucje													
		Uogólnione inwolucje													
	11.10	Oldempotencja				 	•	 			•		 	•	415
10	⊏2.	Delegie													417
14		Relacje binarne													417
		·													417
		Identyczność relacji													
		Operacje na relacjach													419
		Rodzaje relacji heterogenicznych													421
		Rodzaje relacji heterogenicznych v2													425
	12.6	Rodzaje relacji homogenicznych													429
		12.6.1 Zwrotność													430
		12.6.2 Symetria													434
		12.6.3 Przechodniość													437
		12.6.4 Inne				 		 					 		437

	12.8 12.9 12.10	Złożone relacje homogeniczne	438 439 440 441 444
13	F1:	Koindukcja i korekursja	145
	13.1	9 (/	445
		13.1.1 Strumienie (TODO)	445
		13.1.2 Kolisty	449
		13.1.3 Drzewka	454
			455
	13.2	Ćwiczenia	460
14	F2:	Liczby konaturalne	162
15	F3:	Strumienie 4	170
			470
		•	471
16	F4:	Kolisty	178
17	G: Ir	nne spojrzenia na typy induktywne i koinduktywne	186
			486
		· · · · · · · · · · · · · · · · · · ·	492
			496
	17.4	Indeksowane M-typy?	499
	17.5	Kodowanie Churcha (TODO)	499
18	H1:	Uniwersa - pusty	501
19	H2:	Równość - pusty	502
20	l1: l	_tac — język taktyk	503
		• • • •	503
		· · · · · · · · · · · · · · · · · · ·	506
			508
			511
		-	519
			522
			522
			523
			524
	20.7		526

	20.8	Konkluzja	527
21	12: 5	Spis przydatnych taktyk 5	2 8
	21.1	refine — matka wszystkich taktyk	528
	21.2	Drobne taktyki	531
		21.2.1 clear 5	531
		21.2.2 fold	533
		21.2.3 move	533
		21.2.4 pose i remember	534
		21.2.5 rename	534
		21.2.6 admit	35
	21.3	Średnie taktyki	535
		21.3.1 case_eq	35
		21.3.2 contradiction	536
		21.3.3 constructor	37
		21.3.4 decompose	538
		21.3.5 intros	539
		21.3.6 fix	541
		21.3.7 functional induction i functional inversion	643
			643
	21.4		544
			544
			546
		21.4.3 rewrite	550
	21.5	Taktyki dla redukcji i obliczeń (TODO)	552
		· · · · · · · · · · · · · · · · · · ·	552
			552
		21.6.2 congruence	553
		21.6.3 decide equality	554
		21.6.4 omega	555
		21.6.5 Procedury decyzyjne dla logiki	556
	21.7		558
		21.7.1 auto i trivial	558
		21.7.2 autorewrite i $autounfold$	662
	21.8		665
	21.9	Zmienne egzystencjalne i ich taktyki (TODO)	665
			665
		v - v - v - v - v - v - v - v - v - v -	665
			666
			667

22	13: F	Reflekcja w dużej skali, czyli jak odbijać z rozmachem	568
	22.1	Ltac: manipulowanie termami	568
	22.2	Taktyki dla unifikacji	568
	22.3	Programowanie funkcyjne w Ltacu	569
	22.4	Big scale reflection	569
23	J: K	im jesteśmy i dokąd zmierzamy - pusty	570
24	W1:	Konstruktywny rachunek zdań [schowany na końcu dla niepoznaki]	571
	24.1	Zdania i spójniki logiczne	571
		24.1.1 Implikacja	571
		24.1.2 Koniunkcja	571
		24.1.3 Dysjunkcja	571
		24.1.4 Prawda i fałsz	571
		24.1.5 Równoważność	571
		24.1.6 Negacja	571
		24.1.7 Silna negacja	571
		24.1.8 Czy Bozia dała inne spójniki logiczne?	575
	24.2	Paradoks pieniądza i kebaba	575
	24.3	Zadania	576
	24.4	Ściąga	576
วะ	M/2 .	Kanstruktyvyny rachunak kwantyfikatorów Isahawany na końcu dla nie	
<i>∠</i> ე	vv∠. pozr	- Konstruktywny rachunek kwantyfikatorów [schowany na końcu dla nie nakil	:- 577
		Typy i ich elementy	
		Predykaty i relacje	
		Kwantyfikatory	
		25.3.1 Kwantyfikator uniwersalny	
		25.3.2 Kwantyfikator egzystencjalny	
	25.4	Zmienne związane	
		Predykatywizm	
		Paradoks golibrody	577
	25.7	Zadania	
		Ściąga	
26	W3-	Logika klasyczna [schowana na końcu dla niepoznaki]	580
		Prawa logiki klasycznej	580
		Logika klasyczna jako logika Boga	580
		Logika klasyczna jako logika materialnej implikacji i równoważności	581
		Logika klasyczna jako logika diabła	584
		Logika klasyczna jako logika kontrapozycji	585
		Logika klasyczna jako logika Peirce'a	

		26.6.2 Logika Peirce'a	586
	26.7	Paradoks pijoka	587
	26.8	Paradoks Curry'ego	589
	26.9	Zadania	589
	26.10	$0 m \acute{S}ciaga$	589
27	W4:	Inne logiki [schowane na końcu dla niepoznaki] 5	90
	27.1	Porównanie logiki konstruktywnej i klasycznej	590
	27.2	Pluralizm logiczny	590
	27.3	Inne logiki?	590
	27.4	Logika de Morgana	590
	27.5	Inne logiki - podsumowanie	590
	27.6	Kombinatory taktyk	590
		27.6.1 ; (średnik)	591
		27.6.2 (alternatywa)	592
		27.6.3 idtac, do oraz repeat	593
		27.6.4 try i fail	594
	27.7	Zadania	595
	27.8	Jakieś podsumowanie	595
28	Z: Z	Złożoność obliczeniowa 5	96
	28.1	Czas działania programu	596
	28.2	Złożoność obliczeniowa	597
	28.3	Złożoność asymptotyczna	598
	28.4	Duże O	599
		28.4.1 Definicja i intuicja	599
		28.4.2 Złożoność formalna i nieformalna	600
		28.4.3 Duże Omega	601
	28.5	Duże Theta	601
	28.6	Złożoność typowych funkcji na listach	502
		28.6.1 Analiza nieformalna	502
		28.6.2 Formalne sprawdzenie	502
	28.7		304
	28.8	Przyspieszanie funkcji rekurencyjnych	305
		28.8.1 Złożoność <i>rev</i>	305
		28.8.2 Pamięć	606
	28.9	Podsumowanie	507

Rozdział 1

A: 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 ^ b (a podniesione do potęgi b) jest liczbą wymierną.

Dowód: jeżeli $\sqrt{2}$ /2jestniewymierny, toniecha = $\sqrt{2}$ /2, $b = \sqrt{2}$. Wtedy $= (\sqrt{2})/2 = \sqrt{2}$ /2 * Fajny dowód, co? To teraz dam ci zagadkę: podaj mi dwie niewymierne liczby a i b takie, że a ^ 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. Co ciekawe, Brouwer był przeciwnikiem formalizacji, a jego idee sformalizował dopiero jego uczeń, Arend Heyting.

Ciekawostka: po polsku L. E. J. Brouwer można czytać jako "lej browar".

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 w swoim dowodzie używali 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, a potem w USA) był sobie matematyk nazwiskiem Voevodsky (czyt. "wojewódzki"). Zajmował się on takimi dziwnymi rzeczami, jak teoria homotopii czy kohomologia motywiczna (nie pytaj co to, bo nawet najstarsi górale tego nie wiedzą). Za swoje osiągnięcia w tych dziedzinach otrzymał medal Fieldsa, czyli najbardziej prestiżową nagrodę dla matematyków. Musiał być więc raczej zdolny.

Jego historia jest jednak historią popełniania błędu na błędzie błędem poprawianym. Dla przykładu, w jednej ze swoich prac popełnił błąd, którego znalezienie zajęło 7 lat, a poprawienie - kolejne 6 lat. W innym, nieco hardkorowszym przypadku, w jego pracy z 1989 roku inny ekspert błąd znalazł w roku 1998, ale Voevodsky nie wierzył, że faktycznie jest tam błąd - obu panom po prostu ciężko się było dogadać. Ostatecznie Voevodsky o swym błędzie przekonał się dopiero w roku 2013.

Czy powyższe perypetie świadczą o tym, że Voevodsky jest krętaczem (lub po prostu idiotą)? Oczywiście nie. Świadczą one o tym, że matematyka uprawiana na wysokim poziomie abstrakcji jest bardzo trudna do ogarnięcia przez ludzi. Ludzie, w tym matematycy, mają ograniczoną ilość pamięci oraz umiejętności rozumowania, a na dodatek ślepo ufają autorytetom - bardzo skomplikowanych i nudnych twierdzeń z dziedzin, którymi mało kto się zajmuje, po prostu (prawie) nikt nie sprawdza.

Powyższe skłoniło Voevodskyego do porzucenia swych dziwnych zainteresowań i zajęcia się czymś równie dziwnym (przynajmniej dla matematyków), czyli formalną weryfikacją rozumowań matematycznych przez komputery. Po długich przemyśleniach związał on swe nadzieje właśnie z Coqiem. Jednak duchy przeszłości nie przestawały go nawiedzać i to aż do tego stopnia, że wymyślił on (do spółki z takimi ludźmi jak Awodey, Warren czy van den Berg) homotopiczną interpretację teorii typów.

O co chodzi? W skrócie: typy zamiast programów reprezentują przestrzenie, zaś programy to punkty w tych przestrzeniach. Programy, które dają takie same wyniki są połączone ścieżkami. Programowanie (i robienie matematyki) staje się więc w takim układzie niczym innym jak rzeźbieniem figurek w bardzo abstrakcyjnych przestrzeniach.

Jakkolwiek powyższe brzmi dość groźnie, to jest bardzo użyteczne i pozwala zarówno załatać różne praktyczne braki teorii typów (np. brak typów ilorazowych, cokolwiek to jest) jak i ułatwia czysto teoretyczne rozumowania w wielu aspektach.

Homotopia to przyszłość!

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, ko-indukcji, zaawansowanej rekurencji i reflekcji. 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 dowodzenia zwaną small scale reflection (pol. 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 specy-fikacji)
- 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
- $\bullet \ \, http://coqhott.gforge.inria.fr/blog/$

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: https://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

B: 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 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 obliczenio-żerną, 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.

2.2 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!

2.3 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 i taktyki.

W razie problemów z Coq
IDE poszukaj pomocy w manualu: https://coq.inria.fr/refman/practicaltools/coqide.html

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. Używamy tej komendy, gdyż potrzebujemy jakichś zdań logicznych, na których będziemy operować.

2.3.1 Implikacja

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

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 → 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. $dowód_na_to_2e_P_zachodzi$, 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.

```
\label{eq:lemma_impl_refl'} \begin{split} \operatorname{Lemma} & impl\_refl': P \to P. \\ \operatorname{Proof.} & \text{intro. assumption.} \\ \operatorname{Qed.} & \end{split}
```

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 (pol. "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 *)
```

Uwaga: w komentarzach postaci (* ===> *) będę przedstawiać wyniki wypisywane

przez komendy, żeby leniwi czytacze nie musieli sami sprawdzać.

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 \Rightarrow ?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 $modus_ponens$:

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

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 $P \rightarrow Q$ 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 $(H:P \rightarrow Q)$ $(H0:P) \Rightarrow ?1$. 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 $impl_trans$:

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

Lemma $impl_permute$:

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

Lemma $impl_dist$:

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

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

 ${\tt Lemma}\ modus_ponens':$

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

2.3.2 Falsz

Lemma $ex_falso: False \rightarrow P$.

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.

2.3.3 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.

Komendy Check i Locate

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

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.

```
Check \neg P.

(* ===> ~ P : Prop *)
```

W Coqu negację zdania P oznaczamy przez $\neg P$. Symbol \neg nie jest jednak nazwą negacji — nazwy nie mogą zawierać symboli. Jest to jedynie notacja, która ma uczynić zapis krótszym i bardziej podobnym do tego używanego na co dzień. 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. Jak widać, negacja w Coqu nazywa się not.

2.3.4 Negacja

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:

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: \neg P \rightarrow P \rightarrow 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': \neg P \rightarrow P \rightarrow 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 $False \rightarrow 42 = 666$ 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 : $\neg False$.

Lemma $not_True : \neg True \rightarrow False$.

Ćwiczenie (podwójna negacja) Udowodnij poniższe twierdzenia. Zastanów się, czy można udowodnić $\tilde{} P \to P$.

Lemma $dbl_neq_intro: P \rightarrow {}^{\sim}P.$

 ${\tt Lemma}\ double_neg_elim_irrefutable\ :$

 $\sim (\sim P \rightarrow P).$

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

Lemma $triple_neg_rev: \ ^{\sim \sim}P \to \neg P.$

2.3.5 Koniunkcja

```
 \begin{tabular}{lll} {\tt Lemma} & and\_intro: P \to Q \to P \land Q. \\ {\tt Proof.} & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & &
```

Symbol \land oznacza koniunkcję dwóch zdań logicznych i podobnie jak \neg 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 \land Q \rightarrow 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 \land Q \to 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\_proj\mathcal{Z}: P \land Q \to Q. Lemma and\mathcal{Z}_intro: P \to Q \to R \to P \land Q \land R.
```

```
Lemma and3\_proj: P \land Q \land R \rightarrow Q.
Lemma noncontradiction: \ \ (P \land \neg P).
```

2.3.6 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 H\theta. assumption. Qed.
```

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

```
Lemma iff\_proj1: (P \leftrightarrow Q) \rightarrow (P \rightarrow Q). Proof. intros. destruct H as [HPQ\ HQP]. apply HPQ. assumption. Qed.
```

Równoważnosć 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.

```
\begin{array}{l} \operatorname{Lemma} \ iff\_refl: P \leftrightarrow P. \\ \\ \operatorname{Lemma} \ iff\_symm: (P \leftrightarrow Q) \rightarrow (Q \leftrightarrow P). \\ \\ \operatorname{Lemma} \ iff\_trans: (P \leftrightarrow Q) \rightarrow (Q \leftrightarrow R) \rightarrow (P \leftrightarrow R). \\ \\ \operatorname{Lemma} \ iff\_not: (P \leftrightarrow Q) \rightarrow (^{\sim}P \leftrightarrow \neg Q). \\ \\ \operatorname{Lemma} \ curry\_uncurry: (P \rightarrow Q \rightarrow R) \leftrightarrow (P \land Q \rightarrow R). \end{array}
```

2.3.7 Dysjunkcja

```
\label{eq:left} \begin{array}{l} \text{Lemma } \textit{or\_left}: P \rightarrow P \ \lor \ Q. \\ \text{Proof.} \\ \text{intro. left. assumption.} \\ \text{Qed.} \end{array}
```

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 \lor Q \to Q \lor P. Proof. intro. destruct H as [p \mid 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 $\lceil nazwa1 \rceil nazwa2 \rceil$.

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).
```

2.4 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.

2.4.1 Kwantyfikacja uniwersalna

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

```
Lemma impl\_refl'': \forall \ P: \texttt{Prop}, \ P \to 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.

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 sa 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.

```
\label{eq:lemma_general_to_particular} \begin{tabular}{ll} $\forall P:nat \to {\tt Prop}, \\ $(\forall n:nat, P:n) \to P$ 42. \\ \\ {\tt Proof.} \\ $\tt intros. apply $H$. \\ \\ {\tt Restart.} \\ $\tt intros. specialize $(H:42)$. assumption. \\ \\ {\tt Qed.} \end{tabular}
```

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.

```
\begin{array}{l} \operatorname{Lemma}\ and\_proj1\text{''}:\\ \forall\ (P\ Q:nat\to\operatorname{Prop}),\\ (\forall\ n:nat,\ P\ n\land\ Q\ n)\to(\forall\ n:nat,\ P\ n). \end{array} \operatorname{Proof.}\\ \operatorname{intros}\ P\ Q\ H\ k.\ \operatorname{destruct}\ (H\ k).\ \operatorname{assumption}. \\ \operatorname{Qed.} \end{array}
```

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: \forall (A: \mathtt{Type}) \ (P \ Q: A \to \mathtt{Prop}),
```

```
(\forall x : A, P x \land Q x) \leftrightarrow (\forall x : A, P x) \land (\forall x : A, Q x).
```

2.4.2 Kwantyfikacja egzystencjalna

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

```
Lemma ex\_example1:

\exists n: nat, n = 0.

Proof.

\exists 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 \exists odpowiada stwierdzenie: "liczbą posiadającą tę właściwość jest 0". Następnie pozostaje nam udowodnić, iż rzeczywiście 0=0, co jest trywialne.

```
Lemma ex\_example2: \neg \exists \ 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:

\forall (A: \mathtt{Type}) (P \ Q: A \rightarrow \mathtt{Prop}),

(\exists \ x: A, P \ x \lor Q \ x) \leftrightarrow

(\exists \ x: A, P \ x) \lor (\exists \ x: A, Q \ x).
```

2.5 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:

```
\forall \ (man: \mathsf{Type}) \ (barber: man) \ (shaves: man \to man \to \mathsf{Prop}), \ (\forall \ x: man, \ shaves \ barber \ x \leftrightarrow \neg \ shaves \ x \ x) \to \mathit{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 $\forall x : man$, shaves barber $x \leftrightarrow \neg$ shaves 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 $(\forall\ x:man)$, 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:

Lemma $barbers_paradox'$:

```
\forall \ (A: \mathtt{Type}) \ (x:A) \ (P:A \to A \to \mathtt{Prop}), \\ (\forall \ y:A, \ P \ x \ y \leftrightarrow \neg \ P \ y \ y) \to \mathit{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.

2.6 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".

```
Lemma copy: \forall P : \texttt{Prop}, P \rightarrow 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 $andn: nat \to \mathsf{Prop} \to \mathsf{Prop},$ 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!

2.7 Kombinatory taktyk

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

```
 \begin{array}{c} \mathsf{Lemma} \ iff\_intro': \\ \forall \ P \ Q: \mathsf{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \mathsf{Proof}. \\ \mathsf{intros}. \ \mathsf{split}. \end{array}
```

```
intro. apply H. assumption. intro. apply H\theta. 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 różnia 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.

2.7.1; (średnik)

```
 \begin{array}{l} \operatorname{Lemma} \ iff\_intro'': \\ \forall \ P \ Q : \operatorname{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \\ \operatorname{Proof}. \\ \operatorname{split}; \operatorname{intros}; [\operatorname{apply} H \mid \operatorname{apply} H\theta]; \operatorname{assumption}. \\ \\ \operatorname{Qed}. \end{array}
```

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 splitem, 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; $[t1 \mid t2 \mid ... \mid tn]$ 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; $[t2 \mid t2 \mid ... \mid t2]$.

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.

```
 \begin{array}{l} \operatorname{Lemma} \ or\_comm\_ex: \\ \forall \ P \ Q: \operatorname{Prop}, \ P \lor \ Q \to Q \lor P. \\ \\ \operatorname{Lemma} \ diamond: \\ \forall \ P \ Q \ R \ S: \operatorname{Prop}, \\ (P \to Q) \lor (P \to R) \to (Q \to S) \to (R \to S) \to P \to S. \end{array}
```

2.7.2 || (alternatywa)

```
\label{eq:lemma_lemma} \begin{array}{l} \operatorname{Lemma} \ iff\_intro''': \\ \forall \ P \ Q : \operatorname{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \\ \operatorname{Proof}. \\ \operatorname{split}; \operatorname{intros}; \operatorname{apply} \ H\theta \ || \operatorname{apply} \ H; \operatorname{assumption}. \\ \\ \operatorname{Qed}. \end{array}
```

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 $t1 \mid\mid t2$ używa na obecnym celu t1. Jeżeli t1 dokona postępu, to $t1 \mid\mid t2$ 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 $t1 \mid\mid t2$ będzie miało taki efekt jak t2 i skończy się sukcesem. Jeżeli t2 nie dokona postępu, to $t1 \mid\mid t2$ 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.

2.7.3 idtac, do oraz repeat

```
\label{eq:lemma_def} \begin{array}{l} \operatorname{Lemma}\ idtac\_do\_example: \\ \forall\ P\ Q\ R\ S: \operatorname{Prop}, \\ P \to S \lor R \lor Q \lor P. \\ \\ \operatorname{Proof.} \\ \operatorname{idtac.\ intros.\ do\ 3\ right.\ assumption.} \\ \operatorname{Qed.} \end{array}
```

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.

```
\label{eq:lemma_repeat_example} \begin{split} &\forall~P~A~B~C~D~E~F: \texttt{Prop}, \\ &P \to A \lor B \lor C \lor D \lor E \lor F \lor P. \end{split} \label{eq:lemma_repeat_example} \\ &P \to A \lor B \lor C \lor D \lor E \lor F \lor P. \end{split} \label{eq:lemma_repeat_example} \\ &\text{Proof.} \\ &\text{intros. repeat right. assumption.} \\ \\ &\text{Qed.} \end{split}
```

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.

2.7.4 try i fail

```
\label{eq:lemma_lemma} \begin{array}{l} \operatorname{Lemma} \ iff\_intro4 : \\ \forall \ P \ Q : \operatorname{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \\ \operatorname{Proof.} \\ \operatorname{split}; \operatorname{intros}; \operatorname{try} \ (\operatorname{apply} \ H\theta; \ \operatorname{assumption}; \ \operatorname{fail}); \\ \operatorname{try} \ (\operatorname{apply} \ H; \ \operatorname{assumption}; \ \operatorname{fail}). \\ \\ \operatorname{Qed.} \end{array}
```

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; $t1 \mid ... \mid tn$, 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 kombina-

torów z biblioteki standardowej znajdziesz tu: https://coq.inria.fr/refman/coq-tacindex.html

2.8 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.

2.8.1 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 :

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

Kuryfikacja i dekuryfikacja

Lemma curry:

$$(P \land Q \rightarrow R) \rightarrow (P \rightarrow Q \rightarrow R).$$

Lemma uncurry:

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

Prawa de Morgana

Lemma $deMorgan_{-}1$:

$$(P \lor Q) \leftrightarrow \neg P \land \neg Q.$$

Lemma $deMorgan_{-}2$:

$$\neg P \lor \neg Q \to \tilde{\ } (P \land Q).$$

Niesprzeczność i zasada wyłączonego środka

 ${\tt Lemma}\ noncontradiction':$

$$(P \wedge \neg P)$$
.

 ${\tt Lemma}\ noncontradiction_v2\ :$

$$\neg (P \leftrightarrow \neg P).$$

 ${\tt Lemma}\ em_irrefutable\ :$

$$\sim (P \vee \neg P)$$
.

Elementy neutralne i anihilujące

 ${\tt Lemma}\ and_false_annihilation:$

$$P \wedge False \leftrightarrow False.$$

 ${\tt Lemma} \ or_false_neutral :$

$$P \vee False \leftrightarrow P$$
.

Lemma $and_true_neutral$:

$$P \wedge True \leftrightarrow P$$
.

Lemma $or_true_annihilation$:

$$P \vee True \leftrightarrow True$$
.

Inne

Lemma or_imp_and : $(P \lor Q \to R) \leftrightarrow (P \to R) \land (Q \to R).$ Lemma and_not_imp : $P \land \neg Q \to \neg (P \to Q).$ Lemma or_not_imp : $\neg P \lor Q \to (P \to Q).$ Lemma contraposition: $(P \to Q) \to (^{\sim}Q \to \neg P).$ 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 . $(* ===> forall P Q : Prop, P / Q \to Q / P *)$

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.

2.8.2 Konstruktywny rachunek kwantyfikatorów

Section Quantifiers Exercises.

Variable A: Type.

 $\text{Hypotheses } P\ Q: A \to \operatorname{Prop}.$

Projekcje

Lemma $forall_and_proj1$: $(\forall x: A, P \ x \land Q \ x) \rightarrow (\forall x: A, P \ x).$ Lemma $forall_and_proj2$: $(\forall x: A, P \ x \land Q \ x) \rightarrow (\forall x: A, Q \ x).$

Rozdzielność

Lemma $forall_dist_and$:

```
 \begin{array}{l} (\forall \; x \; \colon A, \; P \; x \; \land \; Q \; x) \; \leftrightarrow \\ (\forall \; x \; \colon A, \; P \; x) \; \land \; (\forall \; x \; \colon A, \; Q \; x). \\ \\ \text{Lemma } \; exists\_dist\_or \; \colon \\ (\exists \; x \; \colon A, \; P \; x \; \lor \; Q \; x) \; \leftrightarrow \\ (\exists \; x \; \colon A, \; P \; x) \; \lor \; (\exists \; x \; \colon A, \; Q \; x). \\ \\ \text{Lemma } \; ex\_dist\_and \; \colon \\ (\exists \; x \; \colon A, \; P \; x \; \land \; Q \; x) \; \rightarrow \\ (\exists \; y \; \colon A, \; P \; y) \; \land \; (\exists \; z \; \colon A, \; Q \; z). \end{array}
```

Inne

```
 \begin{array}{l} \text{Lemma } forall\_or\_imp: \\ (\forall \ x:\ A,\ P\ x) \lor (\forall \ x:\ A,\ Q\ x) \to \\ \forall \ x:\ A,\ P\ x \lor Q\ x. \\ \\ \text{Lemma } forall\_imp\_imp: \\ (\forall \ x:\ A,\ P\ x \to Q\ x) \to \\ (\forall \ x:\ A,\ P\ x) \to (\forall \ x:\ A,\ Q\ x). \\ \\ \text{Lemma } forall\_inhabited\_nondep: \\ \forall \ R:\ \text{Prop},\ A \to ((\forall \ x:\ A,\ R) \leftrightarrow R). \\ \\ \text{Lemma } forall\_or\_nondep: \\ \forall \ R:\ \text{Prop}, \\ (\forall \ x:\ A,\ P\ x) \lor R \to (\forall \ x:\ A,\ P\ x \lor R). \\ \\ \text{Lemma } forall\_nondep\_impl: \\ \forall \ R:\ \text{Prop}, \\ (\forall \ x:\ A,\ R \to P\ x) \leftrightarrow (R \to \forall \ x:\ A,\ P\ x). \\ \\ \text{End } Quantifiers Exercises. \\ \end{array}
```

2.8.3 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.

```
 \begin{array}{lll} \text{(* } &===> \text{ forall P : Prop, P } \text{/~~} \text{P *)} \\ \text{Lemma } imp\_and\_or: (P \rightarrow Q \vee R) \rightarrow ((P \rightarrow Q) \vee (P \rightarrow R)). \\ \text{Lemma } deMorgan\_2\_conv: \neg (P \wedge Q) \rightarrow \neg P \vee \neg Q. \\ \text{Lemma } not\_impl: \neg (P \rightarrow Q) \rightarrow P \wedge \neg Q. \\ \text{Lemma } impl\_not\_or: (P \rightarrow Q) \rightarrow (^{\sim} P \vee Q). \\ \text{Lemma } material\_implication: (P \rightarrow Q) \leftrightarrow (^{\sim} P \vee Q). \\ \text{Lemma } contraposition\_conv: (^{\sim} Q \rightarrow \neg P) \rightarrow (P \rightarrow Q). \\ \text{Lemma } excluded\_middle: P \vee \neg P. \\ \text{Lemma } peirce: ((P \rightarrow Q) \rightarrow P) \rightarrow P. \\ \text{End } ClassicalExercises. \\ \end{array}
```

2.9 Paradoks pijoka

```
Theorem drinkers\_paradox:
\forall \; (man: \mathsf{Type}) \; (drinks: man \to \mathsf{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 $P \to Q \leftrightarrow \neg P \lor Q$, 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.

Čwiczenie (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 wymagaja logiki intuicjonistycznej, a które klasycznej?

Lemma $dp_nonempty$:

2.10 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.
- $P \wedge Q$ 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 $P \wedge Q$, to za pomocą taktyki destruct możemy z niego wyciągnąć dowody na P i na Q.
- P ∨ 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 ∨ 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 $P \rightarrow False$. 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.

2.11 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

C: Podstawy teorii typów - TODO

Uwaga: ten rozdział jest póki co posklejany z fragmentów innych rozdziałów. Czytając go, weź na to poprawkę. W szczególności zawiera on zadania, których nie będziesz w stanie zrobić, bo niezbędny do tego materiał jest póki co w kolejnym rozdziale. Możesz więc przeczytać część teoretyczną, a zadania pominąć (albo w ogóle pominąć cały ten rozdział).

3.1 Typy i termy

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 sa 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.

3.2 Typy i termy, kanoniczność i uzasadnienie reguł eliminacji

Co to są termy? Po polsku: wyrażenia. Są to napisy zbudowane według pewnych reguł (które będziemy chcieli poznać), które mogą oznaczać przeróżne rzeczy: zdania logiczne i ich dowody, programy i ich specyfikacje, obiekty matematyczne takie jak liczby czy funkcje, struktury danych takie jak napisy czy listy.

Najważniejszym, co wiemy o każdym termie, jest jego typ. Co to jest typ? To taki klasyfikator, który mówi, czego możemy się po termie spodziewać - można liczyć za pomocą liczb, ale nie za pomocą wartości logicznych. Można dowodzić zdań, ale nie napisów. Można skleić ze sobą dwa napisy, ale nie napis i funkcję etc.

Każdy term ma tylko jeden typ, więc każdy typ możemy sobie wyobrazić jako wielki worek z termami. Dla przykładu, typ *nat*, czyli typ liczb naturalnych, to worek, w którym są takie wyrażenia, jak:

- 42
- 2 + 2
- 10×10
- jeżeli słowo "dupa" zawiera "i", to 123, a w przeciwnym wypadku 765
- długość listy [a, b, c, d, e]

Najważniejsze termy są nazywane elementami. Dla *nat* są to 0, 1, 2, 3, 4, 5 i tak dalej. Elementy wyróżnia to, że są w postaci normalnej (zwanej też postacią kanoniczną). Znaczy to intuicyjnie, że są one ostatecznymi wynikami obliczeń, np.:

• obliczenie 42 daje 42

- \bullet obliczenie 2 + 2 daje 4
- obliczenie 10×10 daje 100
- obliczenie ... daje 765
- obliczenie długości listy daje 5

Czym dokładnie są obliczenia, dowiemy się później. Na razie wystarczy nam wiedzieć, że każdy term zamknięty, czyli taki, o którym wiadomo wystarczająco dużo, oblicza się do postaci normalnej, np. 5+1 oblicza się do 6. Jeżeli jednak czegoś nie wiadomo, to term się nie oblicza, np. n+1 nie wiadomo ile wynosi, jeżeli nie wiadomo, co oznacza n.

Podsumowując, każdy element jest termem, a każdy term oblicza się do postaci normalnej, czyli do elementu.

3.3 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 " $\mathbf{x} \in A$ " jestsemantyczna, tzn. jestzdaniem logicznymi wymagadowodu. Relacja " $\mathbf{x} : A$ " jestsyntaktyczna, awicnie jestzdaniem logicznyminie wymagadowodu "Coqjestwstanie sprawdzie" logicznyminiew y magadowodu.
- 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

3.4 Uniwersa

Jeżeli przeczytałeś uważnie sekcję "Typy i termy" z rozdziału o logice, 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 uniwersami. Czasami używa się też nazwy "sort", bo określenie "jest sortu" jest znacznie krótsze, niż "należy do uniwersum" albo "żyje w uniwersum".

Prop, jak już wiesz, jest uniwersum 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 uniwersum, 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 \Rightarrow 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.

Jeżeli czytasz uważnie, to pewnie wciąż czujesz niedosyt — wszakże uniwersa, jako typy, także są termami. Jakiego zatem typu są uniwersa? Przekonajmy się.

```
Check Prop.
```

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

Prop oraz Set są sortu Type. 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 żyje w uniwersum Type(0)
- Type(0) żyje w uniwersum Type(1)
- w ogólności, Type(i) żyje w uniwersum 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.

3.5 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.

3.5.1 Reguly 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:

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 $prod\ (A\ B: {\tt Type}): {\tt Type}.$ To, co w regule jest nad kreską $(A: {\tt Type}\ i\ B: {\tt Type}),$ 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**.

3.5.2 Reguly 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:

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:

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:

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 manualu: https://coq.inria.fr/refman/language/cic.html

Ćwiczenie Napisz (bez podglądania) jak wyglądają reguły wprowadzania dla *option*, *nat* oraz *list*. Następnie zweryfikuj swoje odpowiedzi za pomocą komendy **Print**.

3.5.3 Reguly 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.

Regula 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 ifa musza być tego samego typu. Przyjrzyjmy się nieco bardziej ogólnej regule.

Regula ta mówi nam, że jeżeli mamy rodzinę typów $P:bool \to \mathsf{Type}$ 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 druge służą jedynie do przeprowadzania rozumowań przez przypadki oraz definiowania funkcji przez dopasowanie do wzorca, 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 zaprezentowana klasyfikacja, pierwsza z naszych reguł jest:

- \bullet niezależna, gdyż obie gałęzie **if**a są tego samego typu. Innymi słowy, definiujemy term typu A, który nie jest zależny
- $\bullet\,$ nierekurencyjna, gdyż typboolnie 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):

Regula 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 dopasowania do wzorca 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: \mathsf{Type})\ (P: list\ A \to \mathsf{Prop})

(H0: P\ [])\ (H1: \forall\ x: A,\ P\ [x])

(H2: \forall\ (x\ y: A)\ (l: list\ A),\ P\ l \to 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.

3.5.4 Reguly 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.

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, wktrymwmiejscezmiennej xpodstawionoterm <math>t.

Zauważ, że zarówno symbol $\equiv jakinotacjae\{x/t\}stylkonie formalnymizapisamiiniemaj adnegoznacze Nie jest tak, że dla każdego typu jest tylko jedna reguła obliczania. Jako, że reguły obliczania pokazują związek między regułami eliminacji i wprowadzania, ich ilość można przybliżyć prostym wzorem:$

```
\# regul obliczania = \# regul eliminacji * \# regul 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:

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  = x  if true then x else y  \equiv x  A : Type, x : A, y : A  = x  if false then x else y  \equiv x
```

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 dopasowania do wzorca i rekursji, tak reguły obliczania pochdzą od opisanych już wcześniej reguł redukcji (beta, delta, jota i zeta), a także konwersji alfa.

Čwiczenie Napisz reguły obliczania dla liczb naturalnych oraz list (dla reguł eliminacji nat_ind oraz $list_ind$).

3.5.5 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.

Powyższa reguła unikalności dla funkcji jest nazywana "redukcją eta". Stwierdza ona, że funkcja stworzona za pomocą abstrakcji $\mathbf{fun}\ x:A$, 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 $\equiv mogczasemuywa\ standardowej, zdaniowejrwnociCoqa, czyli = .Niedokocapasujete dor$

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: \forall (A \ B : \texttt{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 (\forall) , sum zależnych (sigT) i unit (zapisz je w notacji z poziomą kreską). Zdecyduj, gdzie w powyższej klasyfikacji mieszcza się te reguły. Jeżeli to możliwe, wyraź je i udowodnij w Coqu.

Rozdział 4

D1: 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, jak definiować nowe typy przy pomocy indukcji oraz jak użyć rekursji do definiowania funkcji, które operują na tych typach.

4.1 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.

4.1.1 Enumeracje

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

```
\begin{array}{c|c} \textbf{Inductive} \ bool : \texttt{Set} := \\ & | \ true : \ bool \\ & | \ false : \ bool. \end{array}
```

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  \mid true \Rightarrow false \\ \mid false \Rightarrow 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 dopasowanie do wzorca (ang. pattern matching). Pozwala ono sprawdzić, którego konstruktora użyto do zrobienia dopasowywanej wartości. Podobnym tworem występującym w językach imperatywnych jest instrukcja switch, ale dopasowanie do wzorca jest od niej dużo potężniejsze.

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.

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

Eval cbv delta beta in negb true.

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 cbw — w taki wypadku zostaną zaaplikowane wszystkie możliwe redukcje, czyli program zostanie wykonany. Do jego wykonania możemy też użyć komend Eval cbn oraz Eval compute (a także Eval simpl, ale taktyka simpl jest przestarzała, więc nie polecam).

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 interesująca nas własność, a następnie spróbować je udowodnić. Jeżeli nam się powiedzie, mamy całkowita pewność, że funkcja rzeczywiście posiada żądaną własność.

```
Theorem neqb\_involutive:
  \forall b : bool, negb (negb b) = b.
Proof.
  intros. destruct b.
     cbn. reflexivity.
     cbn. reflexivity.
Qed.
```

Nasze twierdzenie głosi, że funkcja negb jest inwolucja, 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 = truefalse. Potem przy pomocy taktyki cbn 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 dopasowania do wzorca 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 cbn, Eval cbv, Eval compute, Eval simpl). Niestety, będziesz musiał się do tego przyzwyczaić — język taktyka Coqa jest strasznie zabałaganiony i niesie ze soba 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/coq-tacindex.html
- https://coq.inria.fr/refman/proof-engine/ltac.html

```
Theorem negb\_involutive': \forall \ b: bool, \ negb\ (negb\ b) = b. Proof. destruct b; \ cbn; 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:
\forall b1\ b2\ b3:bool,
andb\ b1\ (andb\ b2\ b3) = andb\ (andb\ b1\ b2)\ b3.

Theorem andb\_comm:
\forall\ b1\ b2:bool,
andb\ b1\ b2 = andb\ b2\ b1.

Theorem orb\_assoc:
\forall\ b1\ b2\ b3:bool,
orb\ b1\ (orb\ b2\ b3) = orb\ (orb\ b1\ b2)\ b3.

Theorem orb\_comm:
\forall\ b1\ b2:bool,
orb\ b1\ b2 = orb\ b2\ b1.

Theorem andb\_true\_elim:
\forall\ b1\ b2:bool,
andb\ b1\ b2 = true \rightarrow b1 = true \land\ 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 podrozdziałe o polimorfizmie.

```
Lemma turnL\_inj:
\forall \ x \ y: \ D, \ turnL \ x = turnL \ y \rightarrow x = y.

Lemma turnR\_inj:
\forall \ x \ y: \ D, \ turnR \ x = turnR \ y \rightarrow x = y.

Lemma opposite\_inj:
\forall \ x \ y: \ D, \ opposite \ x = opposite \ y \rightarrow 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) Zdefiniuj typy induktywne reprezentujące:

- dni tygodnia
- miesiace
- kolory podstawowe systemu RGB

Wymyśl do nich jakieś ciekawe funkcje i twierdzenia.

4.1.2 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 := \mid O: nat \mid S: nat \rightarrow 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.

```
 \begin{split} & \text{Check } (S \ (S \ (S \ 0))). \\ & (* ===> S \ (S \ (S \ 0)) \ : \ \text{nat *}) \\ & \text{End } NatDef. \\ & \text{Check } S \ (S \ (S \ 0)). \\ & (* ===> 3 \ : \ \text{nat *}) \end{split}
```

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\Rightarrow m |\ S\ n'\Rightarrow 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 \Rightarrow).

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: \forall \ n: nat, \ plus \ 0 \ n=n. Proof. intro. cbn. 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: \forall n: nat, plus \ n \ 0 = n. Proof. intro. destruct n. trivial. cbn. 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: \forall \ n: nat, \ plus \ n \ 0 = n. Proof. intro. induction n. trivial. cbn. 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: \forall n \ m : nat, \ plus \ n \ m = plus \ m \ n. Proof. induction n as [\mid n' \mid]; \ cbn; intros. rewrite plus\_n\_O. trivial. induction m as [\mid m' \mid]. cbn. rewrite plus\_n\_O. trivial. cbn. rewrite IHn'. rewrite \leftarrow IHm'. cbn. rewrite IHn'. 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 $[a11 \dots a1n \mid \dots \mid am1 \dots amk]$. Pionowa kreska oddziela argumenty poszczególnych konstruktorów: $a11 \dots a1n$ to argumenty pierwszego konstruktora, zaś $am1 \dots amk$ to argumenty m-tego konstruktora. nat ma dwa konstruktory, z czego pierwszy nie bierze argumentów, a drugi bierze jeden, więc nasz wzorzec ma postać $[\mid n']$. 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 \leftarrow . 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: \forall n: nat, mult \ 0 \ n = 0. Theorem mult_- 0_- r: \forall n: nat, mult \ n \ 0 = 0. Theorem mult_- 1_- l: \forall n: nat, mult \ 1 \ n = n. Theorem mult_- 1_- r: \forall 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
      \mid 0 \Rightarrow n
      \mid S \mid m' \Rightarrow plus' (S \mid n) \mid m'
end.
Theorem plus'_{-}n_{-}\theta:
  \forall n : nat, plus' n 0 = n.
Theorem plus'\_S:
  \forall n \ m : nat, plus' (S \ n) \ m = S (plus' \ n \ m).
Theorem plus'_-\theta_-n:
  \forall n : nat, plus' \mid 0 \mid 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.
```

4.1.3 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.

```
\begin{array}{l} \texttt{Inductive} \ option \ (A : \texttt{Type}) : \ \texttt{Type} := \\ \mid Some : A \rightarrow option \ A \\ \mid None : option \ A. \end{array}
```

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 którch warto wiedzieć. Po pierwsze, wartości zrobione za pomocą różnych konstruktorów są różne. Jest to konieczne, gdyż za pomocą dopasowania do wzorca możemy rozróżnić różne konstruktory — gdyby były one równe, uzyskalibyśmy sprzeczność.

```
Definition isSome \ \{A: \mathtt{Type}\}\ (a:option\ A): \mathtt{Prop}:= \mathtt{match}\ a \ \mathtt{with} \mid Some\ \_ \Rightarrow True \\ \mid None \Rightarrow False end.
```

Pomocnicza funkcja isSome ma za zadanie sprawdzić, którym konstruktorem zrobiono wartość typu $option\ A$. Zapis $\{A: {\tt Type}\}$ 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: \forall (A: \mathsf{Type})\ (a:A),\ Some\ a \neq None. Proof. unfold not; intros. change False with (isSome\ (@None\ A)). rewrite \leftarrow H.\ cbn. 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': \forall (A: Type) (a: A), Some \ a \neq 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: \forall \ (A: \texttt{Type}) \ (x \ y: A), Some \ x = Some \ y \to 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': \forall (A: \texttt{Type}) \ (x \ y: A), \ Some \ x = Some \ y \to 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'': \forall (A: \texttt{Type}) \ (x \ y: A), \ Some \ x = Some \ y \rightarrow 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 "sprzata".

W naszym przypadku hipoteza $H\theta: x = y$ 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 \neq 1.
```

Dwie opisane właściwości, choć pozornie niewinne, a nawet przydatne, mają bardzo istotne i daleko idące konsekwencje. Powoduję 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 := \mid mk\_rational: \forall (sign:bool) (numerator\ denominator:nat), denominator \neq 0 \rightarrow rational.

Axiom rational\_eq: \forall (s\ s':bool)\ (p\ p'\ q\ q':nat) (H:q\neq 0)\ (H':q'\neq 0),\ p\times q'=p'\times q \rightarrow 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.

4.1.4 Listy, czyli parametry + rekursja

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.

```
\begin{array}{c|c} \textbf{Inductive } list \ (A: \texttt{Type}) : \texttt{Type} := \\ & \mid nil : list \ A \\ & \mid cons : A \rightarrow list \ A \rightarrow list \ A. \end{array}
```

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 : \mathtt{Type}\}\ (l1\ l2 : list\ A) : list\ A := \mathtt{match}\ l1\ \mathtt{with} \mid [] \Rightarrow l2 \quad \quad \mid h :: t \Rightarrow h :: app\ t\ l2\ \mathtt{end}. Notation \{1 + + 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: \forall (A: {\tt Type}) \ (l: list\ A), \ [] ++ l = l. Proof. intros. cbn. reflexivity. Qed. Theorem app\_nil\_r: \forall (A: {\tt Type}) \ (l: list\ A), \ l ++ [] = l. Proof. induction l as [|\ h\ t]. cbn. reflexivity. cbn. 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 cbn. 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).

Čwiczenie (właściwości funkcji app) Udowodnij poniższe właściwości funkcji app. Wska-zówka: może ci się przydać taktyka specialize.

```
Theorem app\_assoc: \forall (A: {\tt Type}) \ (l1\ l2\ l3: list\ A), \ l1\ ++\ (l2\ ++\ l3) = (l1\ ++\ l2)\ ++\ l3. Theorem app\_not\_comm: \neg \ \forall \ (A: {\tt 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: \forall A: {\tt Type}, \ length \ (@nil\ A) = 0. Theorem length\_cons: \forall \ (A: {\tt Type})\ (h:A)\ (t: list\ A), \ length\ (h::t) \neq 0. Theorem length\_app: \forall \ (A: {\tt Type})\ (l1\ l2: list\ A), \ length\ (l1\ ++\ l2) = length\ l1\ +\ length\ l2. End MyList.
```

4.1.5 Przydatne komendy

Czas, aby opisać kilka przydatnych komend.

Check 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.

```
SearchPattern (_+ + _- = _-).
```

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/coq-cmdindex.html

4.1.6 Ważne typy induktywne

Module Important Types.

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 : \mathtt{Type}\}\ (x : Empty\_set) : A := \mathtt{match}\ x \ \mathtt{with}\ \mathtt{end}.
```

Jeżeli mamy term typu $Empty_set$, to możemy w sposób niemal magiczny wyczarować term dowolnego typu A, używając dopasowania do wzorca z pustym wzorcem.

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

Theorem create_unique:

```
\forall (A: \mathtt{Type}) (f: Empty\_set \rightarrow A), \ (\forall 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: \forall (A: \texttt{Type}) \ (a: A) \ (f: A \rightarrow 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: \mathtt{Type}) (f: A \rightarrow unit), \ (\forall x: A, delete \ x = f \ x).
```

Produkt

```
Inductive prod\ (A\ B: {\tt Type}): {\tt Type}:= \\ \mid pair: A \rightarrow B \rightarrow 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:

- $fst: \forall A B: \mathsf{Type}, prod A B \to A$ wyciąga z pary jej pierwszy element
- $\bullet \ snd : \forall \ A \ B : {\tt Type}, \ prod \ A \ B \rightarrow B$ wyciąga z pary jej drugi element

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

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

Suma

```
\begin{array}{l} \texttt{Inductive} \ sum \ (A \ B : \texttt{Type}) : \ \texttt{Type} := \\ | \ inl : \ A \rightarrow sum \ A \ B \\ | \ inr : \ B \rightarrow sum \ A \ B. \\ \\ Arguments \ inl \ [A] \ [B] \ \_. \\ \\ Arguments \ inr \ [A] \ [B] \ \_. \end{array}
```

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: \forall \ (proj: \forall \ A \ B: \texttt{Type}, \ sum \ A \ B \rightarrow A), \ False. Theorem sum\_no\_snd: \forall \ (proj: \forall \ A \ B: \texttt{Type}, \ sum \ A \ B \rightarrow B), \ False. End ImportantTypes.
```

4.1.7 Kiedy typ induktywny jest pusty?

4.1.7 Kledy typ maaktywny jest pasty:

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 \rightarrow Empty.

Theorem Empty\_is\_empty: \forall \ 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: {\tt Type}): {\tt Type} := \\ \mid InfiniteCons: A \rightarrow InfiniteList\ A \rightarrow 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: \forall \ A: {\tt Type}, \ InfiniteList\ A \to 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: \forall (A: \texttt{Type}), \ list \ A \rightarrow 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 $\mathit{IHt}: False$, 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.

4.2 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.

4.2.1 Induktywne zdania

```
\begin{array}{l} \textbf{Inductive} \ false\_prop : \texttt{Prop} := . \\ \textbf{Inductive} \ true\_prop : \texttt{Prop} := \\ \mid obvious\_proof : true\_prop \\ \mid tricky\_proof : true\_prop \\ \mid weird\_proof : true\_prop \end{array}
```

```
\mid 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 $true_prop_iff_True: true_prop \leftrightarrow True$.

4.2.2 Induktywne predykaty

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

```
\begin{array}{l} \textbf{Inductive} \ even: \ nat \rightarrow \texttt{Prop} := \\ \mid \ even 0 : \ even \ 0 \\ \mid \ even SS : \forall \ n : \ nat, \ even \ n \rightarrow \ even \ (S \ (S \ n)). \end{array}
```

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 even 0. Qed.
```

Jak możemy udowodnić, że 0 jest liczbą parzystą? Posłuży nam do tego konstruktor $even\theta$, który wprost głosi, że $even\ 0$. Nie daj się zwieść: $even\theta$, 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 even\theta. Qed.
```

Jak możemy udowodnić, że 2 jest parzyste? Konstruktor $even\theta$ 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 (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: \neg even 1. Proof.

unfold not. intro. destruct H.

Abort.

Theorem one\_not\_even: \neg 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 $even\theta$ 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: \neg 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:\neg odd 0.

Theorem two\_not\_odd:\neg odd 2.
```

4.2.3 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: \forall n \ m : nat, even \ n \rightarrow even \ m \rightarrow even \ (n + m).
```

```
Proof. induction n as [\mid n' \mid]; cbn; intros. trivial. induction m as [\mid m' \mid]; rewrite plus\_comm; cbn; 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 $(P \ n \rightarrow P \ (S \ n))$, podczas gdy w definicji even mamy konstruktor głoszący, że $(even \ n \rightarrow even \ (S \ (S \ n)))$.

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: \forall \ n \ m : nat, \ even \ n \to even \ m \to even \ (n+m). Proof. intros n \ m \ Hn \ Hm. destruct Hn, \ Hm; \ cbn. constructor. constructor. assumption. rewrite plus\_comm. \ cbn. constructor. assumption. rewrite plus\_comm. \ cbn. 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: \forall \ n \ m: nat, \ even \ n \rightarrow even \ m \rightarrow even \ (n+m). Proof. intros n \ m \ Hn \ Hm. induction Hn as [|\ n' \ Hn']. cbn. assumption. cbn. 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 $even\theta$, 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:
\forall \ 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:
\forall \ n: nat, \ n+0+0=n.

Proof.

intro. assert (H: n+0=n).

apply plus\_\theta\_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:
\forall n: nat, even (2 \times n).
Theorem even\_is\_double:
\forall n: nat, even n \rightarrow \exists k: nat, n = 2 \times k.
```

4.2.4 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.

```
\begin{array}{c} \texttt{Inductive} \ True : \texttt{Prop} := \\ \mid I : True. \end{array}
```

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 \rightarrow Q \rightarrow 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: \texttt{Prop}): \texttt{Prop} := \\ \mid or\_introl: P \rightarrow or\ P\ Q \\ \mid or\_intror: Q \rightarrow 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.

4.2.5 Równość

Module MyEq.

Czym jest równość? To pytanie stawiało sobie wielu filozofów, szczególnie politycznych, zaś wyjątkowo rzadko nad tą sprawą zastanawiali się sami bojownicy o równość, tak jakby wszystko dokładnie wiedzieli. 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 : \mathtt{Type}\}\ (x : A) : A \to \mathtt{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: \forall A: \mathsf{Type}, \ eq\ (\mathsf{fun}\ x: A \Rightarrow x)\ (\mathsf{fun}\ y: A \Rightarrow y). Proof. intro. change (\mathsf{fun}\ x: A \Rightarrow x) with (\mathsf{fun}\ y: A \Rightarrow y). apply eq\_refl. Qed. Theorem eq\_refl\_beta: \forall \ m: nat, \ eq\ ((\mathsf{fun}\ n: nat \Rightarrow n+n)\ m)\ (m+m). Proof.
```

```
intro. cbn. apply eq_{-}refl.
Qed.
Definition ultimate\_answer: nat := 42.
Theorem eq\_refl\_delta: eq ultimate\_answer 42.
Proof.
  unfold ultimate_answer. apply eq_reft.
Qed.
Theorem eq_refl_iota:
  eq 42 \text{ (match 0 with } | 0 \Rightarrow 42 | \_ \Rightarrow 13 \text{ end)}.
  cbn. apply eq_refl.
Qed.
Theorem eq_refl_zeta:
  let n := 42 in eq n 42.
Proof.
  reflexivity.
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 dopasowanie do wzorca. W naszym przypadku 0 jest termem, który postać jest znana (został on skonstruowany konstruktorem 0) i który pasuje do wzorca | 0 ⇒ 42, a zatem redukcja jota zamienia całe wyrażenie od match aż do end na 42.
- Redukcja zeta odwija lokalną definicję poczynioną za pomocą leta

Termy x i y są konwertowalne, gdy za pomocą serii konwersji alfa oraz redukcji beta, delta, jota i zeta oba redukują się do tego samego termu (który dzięki silnej normalizacji istnieje i jest w postaci kanonicznej).

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

Trzeba tutaj dokonać pewnego doprecyzowania. Termy 0 + n i n + 0 są konwertowalne dla każdego konkretnego n, np. 0 + 42 i 42 + 0 są konwertowalne. Konwertowalne nie są natomiast, gdy n jest zmienną - jest tak dlatego, że nie możemy wykonać redukcji iota, bo nie wiemy, czy n jest zerem czy następnikiem.

Odpowiedzią na pytanie są reguły eliminacji, głównie dla typów induktywnych. Reguły te mają konkluzje postaci $\forall x: I, P x$, więc w szczególności możemy użyć ich dla P x:=x=y dla jakiegoś y:A. Dzięki nim przeprowadzaliśmy już wielokrotnie mniej więcej takie rozumowania: n jest wprawdzie nie wiadomo czym, ale przez indukcję może to być albo 0, albo S n, gdzie dla n zachodzi odpowiednia hipoteza indukcyjna.

End MyEq.

4.2.6 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.

```
\mid oddS : \forall n : nat, even n \rightarrow 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_-\theta: even\ 0.

Lemma odd_-1: odd\ 1.

Lemma even_-2: even\ 2.

Lemma even_-42: even\ 42.

Lemma not_-odd_-\theta: \neg\ odd\ 0.

Lemma not_-even_-1: \neg\ even\ 1.
```

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

```
Lemma even\_SS: \forall n: nat, even \ n \rightarrow even \ (S \ (S \ n)). Lemma odd\_SS: \forall n: nat, odd \ n \rightarrow odd \ (S \ (S \ n)). Lemma even\_plus: \forall n \ m: nat, even \ n \rightarrow even \ m \rightarrow 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: \forall \ n \ m: nat, \ even \ n \to even \ m \to even \ (n+m). Proof. induction n; intros. assumption. cbn. 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:
\forall \ n \ m: nat, \ even \ n \to even \ m \to even \ (n+m).
Proof.
induction 1; cbn; 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:
\forall \ n \ m: nat, \ odd \ n \rightarrow even \ m \rightarrow odd \ (n+m).
Proof.
induction n; intros.
inversion H.
cbn. 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: \forall \ n \ m: nat, \ even \ n \rightarrow even \ m \rightarrow even \ (n+m) with odd\_even\_plus: \forall \ n \ m: nat, \ odd \ n \rightarrow even \ m \rightarrow odd \ (n+m). Proof. assumption. assumption. Fail Qed.
```

```
Restart.

destruct n as [\mid n' \mid]; cbn; intros.

assumption.

constructor. apply odd\_even\_plus.

inversion H. assumption.

destruct n as [\mid n' \mid]; cbn; 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, jaka 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: \forall n: nat, even\ (2\times n). Proof. induction n as [\mid n']; cbn in ^*; constructor. rewrite \leftarrow plus\_n\_O in ^*. rewrite plus\_comm.\ cbn. constructor. assumption. Qed. End MutInd.
```

4.3 Różne

4.3.1 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*:

```
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 $\forall (x1:A1) \dots (xN:AN)$ jest dodawana na początek każdego konstruktora
- \bullet w konkluzji konstruktora Tmusi wystąpić za
aplikowany do tych argumentów, czyli jako T
x1 ... xN wstawienie innych argumentów jest błędem

```
Fail Inductive option' (A: Type): Type := |Some': A \rightarrow option' A |None': <math>\forall B: Type, option' B.
```

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

```
(* Error: Last occurrence of óption'" must have A" 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 \rightarrow Type.

```
Inductive option': Type \rightarrow Type := \mid Some': \forall A: Type, A \rightarrow option' A \mid 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.

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.

```
\begin{array}{l} \texttt{Inductive} \ \textit{vec} \ (A : \texttt{Type}) : \textit{nat} \rightarrow \texttt{Type} := \\ \mid \textit{vnil} : \textit{vec} \ A \ 0 \\ \mid \textit{vcons} : \forall \ n : \textit{nat}, \ A \rightarrow \textit{vec} \ A \ n \rightarrow \textit{vec} \ A \ (S \ n). \end{array}
```

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
- ullet 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

4.3.2 Indukcja wzajemna a indeksowane rodziny typów

Module $MutualIndution_vs_InductiveFamilies$.

Indukcja wzajemna nie jest zbyt użyteczna. Pierwszym, praktycznym, powodem jest to, że, jak pewnie zdążyłeś się już na własnej skórze przekonać, jej używanie jest dość upierdliwe. Drugi, teoretyczny, powód jest taki, że definicje przez indukcję wzajemną możemy łatwo zasymulować za pomocą indeksowanych rodzin typów.

Rzućmy jeszcze raz okiem na znaną nam już definicję predykatów even i odd przez indukcję wzajemną. Nie dzieje się tu nic niezwykłego, a najważniejszym spostrzeżeniem, jakie możemy poczynić, jest to, że even i odd to dwa byty - nie trzy, nie pięć, ale dwa.

```
Inductive even\_odd:bool \rightarrow nat \rightarrow \texttt{Prop}:= | even\theta': even\_odd true 0
```

```
\mid evenS':
\forall n: nat, even\_odd \ false \ n \rightarrow even\_odd \ true \ (S \ n)
\mid oddS':
\forall n: nat, even\_odd \ true \ n \rightarrow even\_odd \ false \ (S \ n).
Definition even':= even\_odd \ true.
Definition odd':= even\_odd \ false.
```

Co z tego wynika? Ano, zamiast definiować przez indukcję wzajemną dwóch predykatów even i odd możemy za jednym zamachem zdefiniować relację even_odd, która jednocześnie odpowiada obu tym predykatom. Kluczem w tej sztuczce jest dodatkowy indeks, którym jest dwuelementowy typ bool. Dzięki niemu możemy zakodować definicję even za pomocą even_odd true, zaś odd jako even_odd false.

```
Lemma even\_even':
\forall n: nat, even \ n \rightarrow even' \ n
with odd\_odd':
\forall n: nat, odd \ n \rightarrow odd' \ n.
Lemma even'\_even:
\forall n: nat, even' \ n \rightarrow even \ n
with odd'\_odd:
\forall n: nat, odd' \ n \rightarrow odd \ n.
```

Obie definicje są, jak widać (ćwiczenie!), równoważne, choć pod względem estetycznym oczywiście dużo lepiej wypada indukcja wzajemna.

End $MutualIndution_vs_InductiveFamilies$.

Na koniec wypada jeszcze powiedzieć, że indeksowane typy induktywne są potężniejsze od typów wzajemnie induktywnych. Wynika to z tego prostego faktu, że przez wzajemną indukcję możemy zdefiniować na raz jedynie skończenie wiele typów, zaś indeksowane typy induktywne indeksowane mogą być typami nieskończonymi.

4.3.3 Sumy zależne i podtypy

W Coqu, w przeciwieństwie do wielu języków imperatywnych, nie ma mechanizmu podtypowania, a każde dwa typy są ze sobą rozłączne. Nie jest to problemem, gdyż podtypowanie możemy zasymulować za pomocą sum zależnych, a te zdefiniować możemy induktywnie.

Module sigma.

```
Inductive sigT (A : \mathsf{Type}) (P : A \to \mathsf{Type}) : \mathsf{Type} := | existT : \forall x : A, P x \to 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 $A \to B$. Jego elementami są termy postaci fun $x:A \Rightarrow \dots$ 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 $x: A \Rightarrow \dots$ 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: {\tt Type})\ (P: A \to {\tt Prop}): {\tt Type} := |\ exist: \forall\ x: A,\ P\ x \to 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 jęzku Ltac), które piszą programy (w jęzku 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 $n \rightarrow sig$ nat even. Wobec tego exist 42 jest typu even 42 \rightarrow 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.

4.3.4 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: \mathsf{Type})\ (P: A \to \mathsf{Prop}): \mathsf{Prop} := |\ ex\_intro: \forall\ x: A, P\ x \to ex\ A\ P.
```

ex to kolejne wcielenie 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 ∃.

```
Theorem exists\_greater: \forall \ n: \ nat, \ ex \ nat \ (\texttt{fun} \ k: \ nat \Rightarrow n < k). End ex.
```

4.4 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 tu na myśli mechanizmy takie jak indukcja-indukcja, indukcja-rekursja oraz indukcja-indukcja-rekursja (jak widać, w świecie poważnych uczonych, podobnie jak świecie Goebbelsa, im więcej razy powtórzy się dane słowo, tym więcej płynie z niego mocy).

4.4.1 Przypomnienie

Zanim jednak wyjaśnimy, co to za stwory, przypomnijmy sobie różne, coraz bardziej innowacyjne sposoby definiowania przez indukcję oraz dowiedzmy się, jak sformułować i udowodnić wynikające z nich reguły rekursji oraz indukcji.

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.

Enumeracje

```
{\tt Module}\ enum.
```

```
\begin{array}{c|c} \textbf{Inductive } I : \texttt{Type} := \\ & \mid c\theta : I \\ & \mid c1 : I \\ & \mid c2 : I. \end{array}
```

Najprymitywniejszymi z typów induktywnych są enumeracje. Definiując je, wymieniamy po prostu wszystkie ich elementy.

```
Definition I\_case\_nondep\_type: Type := \forall P : \text{Type}, P \to P \to P \to I \to 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 $I \to P$.

```
 \textbf{Definition} \ \textit{I\_case\_nondep} : \textit{I\_case\_nondep\_type} := \\
```

```
\begin{array}{l} \texttt{fun} \ (P: \texttt{Type}) \ (c\theta' \ c1' \ c2': P) \ (i: I) \Rightarrow \\ \texttt{match} \ i \ \texttt{with} \\ & \mid c\theta \Rightarrow c\theta' \\ & \mid c1 \Rightarrow c1' \\ & \mid c2 \Rightarrow c2' \\ \texttt{end.} \end{array}
```

Regułę zdefiniować możemy za pomocą dopasowania do wzorca.

```
Definition I\_case\_dep\_type: Type := \forall (P:I \rightarrow \texttt{Type}) (c\theta':Pc\theta) (c1':Pc1) (c2':Pc2), \\ \forall i:I,Pi.
```

Zależną regułę definiowania przez przypadki możemy uzyskać z poprzedniej uzależniając przeciwdziedzinę P od dziedziny.

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.

```
\begin{array}{c} \text{Inductive } I: \text{Type} := \\ \mid x:I \to I \\ \mid D:I \to I. \end{array}
```

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 := \forall P : \texttt{Type}, (P \to P) \to (P \to P) \to I \to 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: \mathsf{Type}) (x': P \to P) (D': P \to P) (i: I): P:=
```

```
\begin{array}{c|c} \mathtt{match}\ i\ \mathtt{with} \\ & \mid x\ i' \Rightarrow x'\ (I\_\mathit{rec}\ P\ x'\ D'\ i') \\ & \mid D\ i' \Rightarrow D'\ (I\_\mathit{rec}\ P\ x'\ D'\ i') \\ \mathtt{end.} \end{array}
```

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.

```
\begin{array}{l} \texttt{Definition} \ I\_ind\_type : \texttt{Type} := \\ \forall \ (P:I \to \texttt{Type}) \\ (x': \forall \ i:I, \ P \ i \to P \ (x \ i)) \\ (D': \forall \ i:I, \ P \ i \to P \ (D \ i)), \\ \forall \ i:I, \ P \ i. \end{array}
```

Reguła indukcji (czyli induktor - cóż za piękna nazwa!) powstaje z reguły rekursji przez uzależnienie przeciwdziedziny P od dziedziny I.

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:

- ullet reguly moga być zależne lub nie, w zależności od tego czy P zależy od I
- reguly moga być rekurencyjne lub nie
- reguly mogą być dla sortu Type, Prop albo nawet Set

End rec.

Parametry

```
Module param.
```

```
Inductive I(A B : Type) : Type :=
```

```
 \begin{array}{c|c} \mid c\theta:A\rightarrow I \ A \ B \\ \mid c1:B\rightarrow I \ A \ B \\ \mid c2:A\rightarrow B\rightarrow I \ A \ B. \end{array}   Arguments \ c\theta \ \{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 := \forall (A \ B \ P : \texttt{Type}) \ (c\theta' : A \to P) \ (c1' : B \to P) \ (c2' : A \to B \to P), I \ A \ B \to 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
```

```
 \begin{array}{l} (A \ B \ P : {\tt Type}) \ (c0' : A \to P) \ (c1' : B \to P) \ (c2' : A \to B \to P) \\ (i : I \ A \ B) : P := \\ {\tt match} \ i \ {\tt with} \\ \mid c0 \ a \Rightarrow c0' \ a \\ \mid c1 \ b \Rightarrow c1' \ b \\ \mid c2 \ a \ b \Rightarrow c2' \ a \ b \\ \end{array}
```

end.

Implementacja jest banalna.

```
\begin{array}{l} \texttt{Definition} \ I\_{case\_dep\_type} : \texttt{Type} := \\ \forall \ (A \ B : \texttt{Type}) \ (P : I \ A \ B \rightarrow \texttt{Type}) \\ (c0' : \forall \ a : A, \ P \ (c0 \ a)) \\ (c1' : \forall \ b : B, \ P \ (c1 \ b)) \\ (c2' : \forall \ (a : A) \ (b : B), \ P \ (c2 \ a \ b)), \\ \forall \ i : I \ A \ B, \ P \ i. \end{array}
```

A regułę indukcję uzyskujemy przez uzależnienie P od I.

```
Definition I\_case\_dep
```

```
\mid c2 \ a \ b \Rightarrow c2' \ a \ b
end.
```

End param.

Indukcja wzajemna

```
Module mutual.
Inductive Smok: Type :=
    |Wysuszony:Zmok \rightarrow Smok|
with Zmok: Type :=
```

 $| Zmoczony : Smok \rightarrow Zmok.$

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 :=
  \forall S : \mathsf{Type}, (Zmok \to S) \to Smok \to S.
Definition Zmok\_case\_nondep\_type : Type :=
  \forall Z : \mathsf{Type}, (Smok \to Z) \to Zmok \to Z.
```

Reguła niezależnej analizy przypadków dla Smoka wygląda banalnie: jeżeli ze Zmokapotrafimy wyprodukować S, to ze Smoka też. Dla Zmoka jest analogicznie.

```
Definition Smok\_case\_nondep
  (S: \mathsf{Type}) \; (\mathit{Wy}: \mathit{Zmok} \to S) \; (\mathit{smok}: \mathit{Smok}) : S :=
match smok with
     | Wysuszony zmok \Rightarrow Wy zmok
end.
Definition Zmok\_case\_nondep
  (Z: \mathsf{Type}) \; (Zm: Smok 	o Z) \; (zmok: Zmok): Z:=
match zmok with
     |Zmoczony smok \Rightarrow Zm smok|
end.
   Implementacja jest banalna.
```

```
Definition Smok\_rec\_type : Type :=
  \forall~S~Z: {\tt Type},~(Z 	o S) 	o (S 	o Z) 	o Smok 	o S.
Definition Zmok\_rec\_type : Type :=
  \forall \ S \ Z : \texttt{Type}, \ (Z \to S) \to (S \to Z) \to Zmok \to Z.
```

Typ rekursora jest jednak nieco bardziej zaawansowany. Zeby zdefiniować funkcję typu $Smok \rightarrow S$, musimy mieć nie tylko rzeczy w kształcie konstruktorów Smoka, ale także w kształcie konstruktorów Zmoka, gdyż rekurencyjna struktura obu typów jest ze sobą nierozerwalnie związana.

```
Fixpoint Smok\_rec
  (S Z : \mathsf{Type}) (Wy : Z \to S) (Zm : S \to Z) (smok : Smok) : S :=
match smok with
     |Wysuszony|zmok \Rightarrow Wy (Zmok\_rec S Z Wy Zm zmok)|
end
with Zmok\_rec
  (S\ Z: \mathsf{Type})\ (Wy: Z \to S)\ (Zm: S \to Z)\ (zmok: Zmok): Z:=
match zmok with
     |Zmoczony \ smok \Rightarrow 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 \rightarrow \mathsf{Type}) (Z: Zmok \rightarrow \mathsf{Type})
     (Wy: \forall zmok: Zmok, Zzmok \rightarrow S(Wysuszonyzmok))
     (Zm: \forall smok: Smok, S smok \rightarrow Z (Zmoczony smok)),
       \forall smok : Smok, S smok.
Definition Zmok\_ind\_type : Type :=
  \forall (S: Smok \rightarrow \mathsf{Type}) (Z: Zmok \rightarrow \mathsf{Type})
     (Wy: \forall zmok: Zmok, Zzmok \rightarrow S(Wysuszonyzmok))
     (Zm: \forall smok: Smok, S smok \rightarrow Z (Zmoczony smok)),
       \forall zmok : Zmok, Z zmok.
Fixpoint Smok\_ind
  (S:Smok \rightarrow \mathsf{Type}) \ (Z:Zmok \rightarrow \mathsf{Type})
  (Wy: \forall zmok: Zmok, Z zmok \rightarrow S (Wysuszony zmok))
  (Zm: \forall smok: Smok, S smok \rightarrow Z (Zmoczony smok))
  (smok : Smok) : S \ smok :=
match smok with
     |Wysuszony|zmok \Rightarrow Wy|zmok (Zmok_ind S|Z|Wy|Zm|zmok)
end
with Zmok\_ind
  (S:Smok \rightarrow \mathsf{Type}) \ (Z:Zmok \rightarrow \mathsf{Type})
  (Wy: \forall zmok: Zmok, Z zmok \rightarrow S (Wysuszony zmok))
  (Zm: \forall smok: Smok, S smok \rightarrow Z (Zmoczony smok))
  (zmok : Zmok) : Z zmok :=
match zmok with
     |Zmoczony \ smok \Rightarrow 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.

```
\begin{array}{c} \texttt{Inductive} \ I: nat \to \texttt{Type} := \\ \mid c\theta: bool \to I \ 0 \\ \mid c42: nat \to I \ 42. \end{array}
```

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 $A_{-}1 \rightarrow ... \rightarrow A_{-}n \rightarrow Type/Prop$, gdzie $A_{-}i$ to indeksy.

```
Definition I\_case\_very\_nondep\_type: Type := \forall \ (P: {\tt Type}) \ (c\theta': bool \to P) \ (c42': nat \to P), \ \forall \ n: nat, \ I \ n \to P. Definition I\_case\_very\_nondep  (P: {\tt Type}) \ (c\theta': bool \to P) \ (c42': nat \to P) \ \{n: nat\} \ (i: I \ n): P:=  match i with | \ c\theta \ b \Rightarrow c\theta' \ b \ | \ c42 \ n \Rightarrow 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 := \forall \ (P:nat \to \texttt{Type}) \ (c\theta':bool \to P\ 0) \ (c42':nat \to P\ 42), \forall \ n:nat, \ I\ n \to P\ n. Definition I\_case\_nondep (P:nat \to \texttt{Type}) \ (c\theta':bool \to P\ 0) \ (c42':nat \to P\ 42) \{n:nat\} \ (i:I\ n):P\ n:= match i with |\ c\theta\ b\Rightarrow c\theta'\ b |\ c42\ n\Rightarrow 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 := \forall (P : \forall n : nat, I \ n \rightarrow \texttt{Type}) (c0' : \forall b : bool, P \ 0 \ (c0 \ b))
```

```
 \begin{array}{c} (\mathit{c42'} : \forall \; n \; : \; \mathit{nat}, \; P \; 42 \; (\mathit{c42} \; \; n)), \\ \forall \; (n \; : \; \mathit{nat}) \; (i \; : \; I \; n), \; P \; n \; i. \\ \\ \text{Definition} \; I\_\mathit{case\_dep} \\ (P : \forall \; n \; : \; \mathit{nat}, \; I \; n \to \mathsf{Type}) \\ (\mathit{c0'} : \forall \; b \; : \; \mathit{bool}, \; P \; 0 \; (\mathit{c0} \; b)) \\ (\mathit{c42'} : \forall \; n \; : \; \mathit{nat}, \; P \; 42 \; (\mathit{c42} \; n)) \\ (n \; : \; \mathit{nat}) \; (i \; : \; I \; n) \; : \; P \; n \; i \; := \\ \\ \mathsf{match} \; i \; \mathsf{with} \\ \mid \; \mathit{c0} \; b \; \Rightarrow \; \mathit{c0'} \; b \\ \mid \; \mathit{c42} \; n \; \Rightarrow \; \mathit{c42'} \; n \\ \\ \mathsf{end}. \\ \end{array}  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.

Nie zapomnijmy ponownie nakazać Coqowi generowania reguł indukcji. Set *Elimination Schemes*.

4.4.2 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 $B:A\to {\sf Type}$. 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: {\tt Type}\}~(R:A\to A\to {\tt Prop}): {\tt Type}:=|snil:slist~R
```

```
 \mid scons : \forall \ (h:A) \ (t:slist\ A), \ ok\ h\ t \rightarrow slist\ A  with ok\ \{A: \mathsf{Type}\}\ \{R:A\rightarrow A\rightarrow \mathsf{Prop}\}: A\rightarrow slist\ R\rightarrow \mathsf{Prop}:= \\ \mid ok\_snil: \forall\ x:A, \ ok\ x\ snil \\ \mid ok\_scons: \\ \forall\ (h:A)\ (t:slist\ A)\ (p:ok\ h\ t)\ (x:A), \\ R\ x\ h\rightarrow 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 $ok: A \to slist R \to \mathsf{Prop}$, 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
- ullet 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.

Regula 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 $\forall (x : A) (l : slist R)$, $ok x l \to Q$.

Regułe indukcji można uzyskać dodając tyle zależności, ile tylko zdołamy unieść. Zobaczmy więc, jak zrealizować to wszystko za pomocą aksjomatów.

Axioms

```
 \begin{array}{l} (slist: \forall \ \{A: \mathtt{Type}\}, \ (A \rightarrow A \rightarrow \mathtt{Prop}) \rightarrow \mathtt{Type}) \\ (ok: \forall \ \{A: \mathtt{Type}\} \ \{R: A \rightarrow A \rightarrow \mathtt{Prop}\}, \ A \rightarrow slist \ R \rightarrow \mathtt{Prop}). \end{array}
```

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

```
 \begin{array}{l} (snil: \forall \; \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow \mathtt{Prop}\}, \; slist \; R) \\ (scons: \\ \forall \; \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow \mathtt{Prop}\} \; (h: A) \; (t: slist \; R), \\ ok \; h \; t \rightarrow slist \; R) \\ (ok\_snil: \\ \forall \; \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow \mathtt{Prop}\} \; (x: A), \; ok \; x \; (@snil\_R)) \\ (ok\_scons: \\ \forall \\ \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow \mathtt{Prop}\} \\ (h: A) \; (t: slist \; R) \; (p: ok \; h \; t) \\ (x: A), \; R \; x \; h \rightarrow ok \; x \; (scons \; h \; t \; p)). \end{array}
```

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

```
 \begin{array}{l} (ind:\forall\\ (A:\mathsf{Type})\;(R:A\to A\to \mathsf{Prop})\\ (P:\mathit{slist}\;R\to \mathsf{Type})\\ (Q:\forall\;(h:A)\;(t:\mathit{slist}\;R),\;ok\;h\;t\to \mathsf{Type})\\ (Psnil:P\;snil)\\ (Pscons:\\ \forall\;(h:A)\;(t:\mathit{slist}\;R)\;(p:ok\;h\;t),\\ P\;t\to Q\;h\;t\;p\to P\;(scons\;h\;t\;p))\\ (Qok\_snil:\forall\;x:A,\;Q\;x\;snil\;(ok\_snil\;x))\\ (Qok\_scons:\\ \forall\\ (h:A)\;(t:\mathit{slist}\;R)\;(p:ok\;h\;t)\\ (x:A)\;(H:R\;x\;h),\\ P\;t\to Q\;h\;t\;p\to Q\;x\;(scons\;h\;t\;p)\;(ok\_scons\;h\;t\;p\;x\;H)),\\ \{f:(\forall\;l:\mathit{slist}\;R,\;P\;l)\;\& \end{array}
```

```
 \{g: (\forall \ (h:A) \ (t:slist \ R) \ (p:ok \ h \ t), \ Q \ h \ t \ p) \mid \\ f \ snil = Psnil \ \land \\ (\forall \ (h:A) \ (t:slist \ R) \ (p:ok \ h \ t), \\ f \ (scons \ h \ t \ p) = Pscons \ h \ t \ p \ (f \ t) \ (g \ h \ t \ p)) \ \land \\ (\forall \ x:A, \\ g \ x \ snil \ (ok\_snil \ x) = Qok\_snil \ x) \ \land \\ (\forall \\ (h:A) \ (t:slist \ R) \ (p:ok \ h \ t) \\ (x:A) \ (H:R \ x \ h), \\ g \ x \ (scons \ h \ t \ p) \ (ok\_scons \ h \ t \ p \ x \ H) = \\ Qok\_scons \ h \ t \ p \ x \ H \ (f \ t) \ (g \ h \ t \ p))  \}\}
```

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 Smoka i Zmoka) - 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: $\forall l: slist R, P l \text{ oraz } g: \forall (h:A) (t:slist R) (p:okht), Q h t p.$ Tak więc P to kodziedzina f, a Q - g.

Teraz potrzebujemy rozważyć wszystkie możliwe przypadki - tak jak przy dopasowaniu do wzorca. Przypadek snil jest dość banalny. Przypadek scons jest trochę cięższy. Przede wszystkim chcemy, żeby konkluzja była postaci P (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 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,
```

```
 \begin{array}{l} \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow \mathtt{Prop}\} \\ (P: \mathtt{Type}) \; (snil': P) \; (scons': A \rightarrow P \rightarrow P) : \\ \{f: slist \; R \rightarrow P \mid \\ f \; snil = snil' \; \land \end{array}
```

```
\forall (h:A) (t:slist R) (p:ok h t),
       f(scons \ h \ t \ p) = scons' \ h \ (f \ t)
  }.
Proof.
  destruct
     ind
     A R
    (fun \rightarrow P) (fun \rightarrow True)
    snil' (fun h - t = scons' h t)
    (fun \rightarrow I) (fun \rightarrow I)
  )
  as (f \& g \& H1 \& H2 \& H3 \& H4).
  \exists f. split.
    exact H1.
    exact H2.
Defined.
```

Możemy na przykład dość łatwo zdefiniować niezależny 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.

Używanie takiego 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
```

```
 \begin{array}{c} (scons\ 1\\ snil\\ (ok\_snil\ 1))\\ (ok\_scons\ 1\ snil\ (ok\_snil\ 1)\ 0\ (le\_S\ 0\ 0\ (le\_n\ 0))). \end{array}  Niezbyt piękna, prawda?
```

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, nie obliczają się.

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);\ cbn. rewrite\ 2!H2,\ H1.\ reflexivity. Qed.
```

Compute $toList\ slist_01$.

Najlepsze, co możemy osiągnąć, mając taką definicję, to udowodnienie, że jej wynik faktycznie jest taki, jak się spodziewamy.

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 $blist: (A \to A \to \mathsf{Prop}) \to A \to \mathsf{Type}$, 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 $ind_{-}ind$.

Na koniec wypadałoby jeszcze wspomnieć, do czego tak naprawdę można w praktyce użyć indukcji-indukcji (definiowanie list posortowanych nie jest jedną z tych rzeczy, o czym przekonałeś się w ćwiczeniach). Otóż najciekawszym przykładem wydaje się być formalizacja teorii typów, czyli, parafrazując, implementacja Coqa w Coqu.

Zeby się za to zabrać, musimy zdefiniować konteksty, typy i termy, a także relacje konwertowalności dla typów i termów. Są tutaj możliwe dwa podejścia:

- Curry'ego (ang. Curry style lub mądrzej extrinsic style) staramy się definiować wszystko osobno, a potem zdefiniować relacje "term x jest typu A w kontekście", "typ A jest poprawnie sformowany w kontekście" etc. Najważniejszą cechą tego sposobu jest to, że możemy tworzyć termy, którym nie da się przypisać żadnego typu oraz typy, które nie są poprawnie sformowane w żadnym kontekście.
- Churcha (ang. Church style lub mądrzej intrinsic style) definiujemy wszystko na raz w jednej wielkiej wzajemnej indukcji. Zamiastów typów definiujemy od razu predykat "typ A jest poprawnie sformowany w kontekście", a zamiast termów definiujemy od razu relację "term x ma typ A w kontekście". Parafrazując wszystkie termy, które jesteśmy w stanie skonstruować, są poprawnie typowane (a wszystkie typy poprawnie sformowane w swoich kontekstach).

Zamiast tyle gadać zobaczmy, jak mogłoby to wyglądać w Coqu. Oczywiście będą to same nagłówki, bo podanie tutaj pełnej definicji byłoby mocno zaciemniającym przegięciem.

Nagłówki w tej definicji powinniśmy interpretować tak:

- Ctx to typ reprezentujący konteksty.
- Ty ma reprezentować typy, ale nie jest to typ, lecz rodzina typów indeksowana kontekstami każdy typ jest typem w jakimś kontekście, np. $list\ A$ jest typem w kontekście zawierającym A: Type, ale nie jest typem w pustym kontekście.
- Term ma reprezentować termy, ale nie jest to typ, lecz rodzina typów indeksowana kontekstami i typami każdy term ma jakiś typ, a typy, jak już się rzekło, zawsze są typami w jakimś kontekście. Przykład: jeżeli x jest zmienną, to cons x nil jest typu list A w kontekście, w którym x jest typu A, ale nie ma żadnego typu (i nie jest nawet poprawnym termem) w kontekście pustym ani w żadnym, w którym nie występuje x.

- TyConv A B zachodzi, gdy typy A i B są konwertowalne, czyli obliczają się do tego samego (relacja taka jest potrzebna, gdyż w Coqu i ogólnie w teorii typów występować mogą takie typy jak if true then nat else bool, który jest konwertowalny z nat). Jako się rzekło, typy zawsze występują w kontekście, więc konwertowalne mogą być też tylko w kontekście.
- $TermConv\ A\ x\ y$ znaczy, że termy x i y są konwertowalne, np. if true then 42 else 0 jest konwertowalne z 42. Ponieważ każdy term ciągnie za sobą swój typ, TermConv ma jako indeks typ A, a ponieważ typ ciągnie za sobą kontekst, indeksem TermConv jest także .

Jak widać, indukcji jest w powyższym przykładzie na pęczki - jest ona wręcz teleskopowa, gdyż Ctx jest indeksem Ty, Ctx i Ty są indeksami Term, a Ctx, Ty i Term są indeksami TermConv.

Cóż, to by było na tyle w tym temacie. Ława oburzonych wyraża w tym momencie swoje najwyższe oburzenie na brak indukcji-indukcji w Coqu: https://www.sadistic.pl/lawa-oburzonych-vt22270.htm

Jednak uszy do góry - istnieją już języki, które jakoś sobie radzą z indukcją-indukcją. Jednym z nich jest wspomniana we wstępie Agda, którą można znaleźć tu: https://agda.readthedocs.io/en/late

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 $h: BHeap\ R$. Wskazówka: prawdopodobnie nie uda ci się zdefiniować mirror. Zastanów się, dlaczego jest tak trudno.

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

 \bullet jest jedna relacja, $\mathit{ok},$ gdzie $\mathit{ok}\ v\ t$ oznacza, że vjest R-mniejszeod wszystkich elementów t

Najpierw napisz pseudodefinicję, a potem przetłumacz ją na odpowiedni zestaw aksjomatów.

Następnie użyj swojej aksjomatycznej definicji, aby zdefiniować funkcję mirror, która tworzy lustrzane odbicie drzewa $t:BST\ R$. Wskazówka: dość możliwe, że ci się nie uda.

4.4.3 Indukcja-rekursja

Module ind_rec .

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.

end. *)

Coq niestety nie wspiera indukcji-rekursji, a próba napisania powyższej definicji kończy się błędem składni, przy którym nie pomaga nawet komenda *Fail*. Podobnie jak poprzednio, pomożemy sobie za pomocą aksjomatów, jednak najpierw prześledźmy definicję.

Typ slist działa następująco:

- ullet R to jakiś porządek. Zauważ, że tym razem $R:A\to A\to bool$, a więc porządek jest reprezentowany przez funkcję, która go rozstrzyga
- 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:

- \bullet dla *snil* zwróć *true* każde x:A można dostawić do listy pustej
- ullet 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 rekursję strukturalną po argumencie typu slist R.

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 powyższa definicję za pomocą aksjomatów.

Axioms

```
 \begin{array}{l} (slist: \forall \; \{A: \mathtt{Type}\}, \; (A \rightarrow A \rightarrow bool) \rightarrow \mathtt{Type}) \\ (ok: \\ \forall \; \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow bool\}, \; A \rightarrow slist \; R \rightarrow bool) \\ (snil: \\ \forall \; \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow bool\}, \; slist \; R) \\ (scons: \\ \forall \\ \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow bool\} \\ (h: A) \; (t: slist \; R), \\ ok \; h \; t = true \rightarrow slist \; R) \\ (ok\_snil: \\ \end{array}
```

```
\forall \; \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow bool\} \; (x:A), ok \; x \; (@snil \; \_R) = true) (ok\_scons: \forall \forall \{A: \mathtt{Type}\} \; \{R: A \rightarrow A \rightarrow 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. Następnym krokiem jest zadeklarowanie konstruktorów slist, a później równań definiujących funkcję ok - koniecznie w tej kolejności, gdyż równania zależą od konstruktorów.

Jak widać, aksjomaty są bardzo proste i sprowadzają się do przepisania powyższej definicji odrzuconej przez Coqa.

Axiom

```
 \begin{split} &ind: \forall\\ &(A: \texttt{Type}) \; (R: A \rightarrow A \rightarrow bool)\\ &(P: slist \; R \rightarrow \texttt{Type})\\ &(Psnil: \; P\; snil)\\ &(Pscons: \\ &\forall \; (h: A) \; (t: slist \; R) \; (p: ok \; h \; t = true),\\ &P\; t \rightarrow P \; (scons \; h \; t \; p)),\\ &\{f: \; \forall \; l: \; slist \; R, \; P \; l \; |\\ &f\; snil = Psnil \; \land\\ &(\forall \; (h: A) \; (t: slist \; R) \; (p: ok \; h \; t = true),\\ &f\; (scons \; h \; t \; p) = Pscons \; h \; t \; p \; (f \; t))\}. \end{split}
```

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 spodziewamy się jej po definicji slist, ale nie robi niczego ponad to. Może to sugerować, że zamiast indukcji-indukcji, o ile to możliwe, lepiej jest używać indukcji-rekursji, a predykaty i relacje definiować przez rekursję.

Powyższą regułę możemy odczytać następująco:

- $A: \mathsf{Type} \ \mathrm{i} \ R: A \to A \to bool \ \mathrm{to} \ \mathrm{parametry} \ slist, \ \mathrm{więc} \ \mathrm{musz}$ ą się pojawić
- $P: slist \ R \to \mathsf{Type}$ to przeciwdziedzina funkcji definiowanej za pomocą reguly
- Psnil to wynik funkcji dla snil
- \bullet Pscons produkuje wynik funkcji dla scons h t pz hipotezy indukcyjnej/wywołania rekurencyjnego dla t
- $f: \forall l: slist\ R,\ P\ l$ to funkcja zdefiniowana przez regułę, zaś równania formalizują to, co zostało napisane powyżej o Psnil i Pscons

Termy induktywno-rekurencyjnego $slist\ R$ wyglądają następująco (najpierw definiujemy sobie funkcję rozstrzygającą standardowy porządek na liczbach naturalnych):

```
Fixpoint leb (n\ m:nat):bool:= match n,\ m with |\ 0,\ \_\Rightarrow true |\ \_,\ 0\Rightarrow false |\ S\ n',\ S\ m'\Rightarrow leb\ n'\ m' end. Definition slist\_01:slist\ leb:= scons\ 0 (scons\ 1 snil (ok\_snil\ 1)) (ok\_scons\ 0\ 1\ snil\ (ok\_snil\ 1)).
```

Nie wygląda wiele lepiej od poprzedniej, induktywno-induktywnej wersji, prawda? Ta rażąca kiepskość nie jest jednak zasługą indukcji-rekursji, lecz kodowania za pomocą aksjomatów - funkcja ok się nie oblicza, więc zamiast eq_refl musimy używać aksjomatów ok_snil i ok_scons .

W tym momencie znów wkracza ława oburzonych i wyraża swoje oburzenie na fakt, że Coq nie wspiera indukcji-rekursji (ale Agda już tak). Gdyby *Coq* wspierał indukcję-rekursję, to ten term wyglądałby tak:

```
Fail Definition slist\_01 : slist \ leb := scons \ 0 \ (scons \ 1 \ snil \ eq\_reft) \ eq\_reft.
```

Dużo lepiej, prawda? Na koniec zobaczmy, jak zdefiniować funkcję zapominającą o fakcie, że lista jest posortowana.

```
Require Import List.

Import ListNotations.

Definition toList'
\{A: \mathsf{Type}\}\ \{R: A \to A \to bool\}: \{f: slist\ R \to list\ A \mid f\ snil = [] \land \forall (h:A)\ (t: slist\ R)\ (p: ok\ h\ t = true), f\ (scons\ h\ t\ p) = h::f\ t
\}.

Proof.

exact (ind\ A\ R\ (\mathsf{fun}\ \_ \Rightarrow list\ A)\ []\ (\mathsf{fun}\ h\ \_ \_\ r \Rightarrow h::\ r)).

Defined.

Definition toList
\{A: \mathsf{Type}\}\ \{R: A \to A \to bool\}: slist\ R \to list\ A:= proj1\_sig\ toList'.
```

Ponownie jest to dużo prostsze, niż w przypadku indukcji-indukcji - wprawdzie wciąż musimy ręcznie wpisywać termy do reguły indukcji, ale dzięki prostocie reguły jest to znacznie łatwiejsze. Alternatywnie: udało nam się zaoszczędzić trochę czasu na definiowaniu reguły rekursji, co w przypadku indukcji-indukcji było niemal konieczne, żeby nie zwariować.

```
Lemma toList\_slist\_01\_result: toList\ slist\_01 = [0;\ 1]. Proof. unfold toList,\ slist\_01. destruct toList' as (f\ \&\ H1\ \&\ H2). cbn. rewrite 2!H2,\ H1. reflexivity. Qed.
```

Udowodnienie, że nasza funkcja daje taki wynik jak chcieliśmy, jest równie proste jak poprzednio.

Ćwiczenie Zdefiniuj dla list posortowanych funkcję slen, która liczy ich długość. Udowodnij oczywiste twierdzenie wiążące ze sobą slen, toList oraz length. Czy było łatwiej, niż w przypadku indukcji-indukcji?

End ind_rec .

Čwiczenie No cóż, jeszcze raz to samo. Zdefiniuj za pomocą indukcji-rekursji jednocześnie typ stert binarnych $BHeap\ R$, gdzie $R:A\to A\to bool$ rozstrzyga porządek, i funkcję $ok:A\to BHeap\ R\to bool$, gdzie $ok\ x\ h=true$, gdy stertę h można podczepić pod element x.

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 $h: BHeap\ R$. Wskazówka: tym razem powinno ci się udać.

Porównaj induktywno-rekurencyjną implementację $BHeap\ R$ i ok z implementacją przez indukcję-indukcję. Która jest bardziej ogólna? Która jest "lżejsza"? Która jest lepsza?

Ćwiczenie No cóż, jeszcze raz to samo. Zdefiniuj za pomocą indukcji-rekursji jednocześnie typ drzew wyszukiwań binarnych BST R, gdzie $R: A \rightarrow A \rightarrow bool$ rozstrzyga porządek, oraz funkcje oklr/okl i okr/ok, które dbają o odpowiednie warunki (wybierz tylko jeden wariant z trzech, które testowałeś w tamtym zadaniu).

Najpierw napisz pseudodefinicję, a potem przetłumacz ją na odpowiedni zestaw aksjomatów.

Następnie użyj swojej aksjomatycznej definicji, aby zdefiniować funkcję mirror, która tworzy lustrzane odbicie drzewa $t:BST\ R$. Wskazówka: tym razem powinno ci się udać.

Porównaj induktywno-rekurencyjną implementację $BST\ R$ z implementacją przez indukcję-indukcję. Która jest bardziej ogólna? Która jest "lżejsza"? Która jest lepsza?

Podobnie jak poprzednio, pojawia się pytanie: do czego w praktyce można użyć indukcjirekursji (poza rzecz jasna głupimi strukturami danych, jak listy posortowane)? W świerszczykach dla bystrzaków (czyli tzw. "literaturze naukowej") przewija się głównie jeden (ale jakże użyteczny) pomysł: uniwersa.

Czym są uniwersa i co mają wspólnego z indukcją-rekursją? Najlepiej będzie przekonać się na przykładzie programowania generycznego:

Ćwiczenie (zdecydowanie za trudne) Zaimplementuj generyczną funkcję *flatten*, która spłaszcza dowolnie zagnieżdżone listy list do jednej, płaskiej listy.

```
flatten 5 = [5]

flatten [1; 2; 3] = [1; 2; 3]

flatten [[1]; [2]; [3]] = [1; 2; 3]

flatten [[[1; 2]]; [[3]]; [[4; 5]; [6]]] = [1; 2; 3; 4; 5; 6]

Trudne, prawda? Ale robialne, a robi się to tak.
```

W typach argumentów *flatten* na powyższym przykładzie widać pewien wzorzec: są to kolejno *nat*, *list nat*, *list (list nat)*, *list (list nat)*) i tak dalej. Możemy ten "wzorzec" bez problemu opisać za pomocą następującego typu:

```
Inductive FlattenType: Type := | Nat: FlattenType | List: FlattenType \rightarrow FlattenType.
```

Żeby było śmieszniej, FlattenType to dokładnie to samo co nat, ale przemilczmy to. Co dalej? Możemy myśleć o elementach FlattenType jak o kodach prawdziwych typów, a skoro są to kody, to można też napisać funkcję dekodującą:

```
Fixpoint decode\ (t:FlattenType): Type := match t with |Nat\Rightarrow nat\ |List\ t'\Rightarrow list\ (decode\ t') end.
```

decode każdemu kodowi przyporządkowuje odpowiadający mu typ. O kodach możemy myśleć jak o nazwach - Nat to nazwa nat, zaś $List\ t$ ' to nazwa typu $list\ (decode\ t$ '), np. $List\ (List\ Nat)$ to nazwa typu $list\ (list\ nat)$.

Para (Flatten Type, decode) jest przykładem uniwersum.

Uniwersum to, najprościej pisząc, worek, który zawiera jakieś typy. Formalnie uniwersum składa się z typu kodów (czyli "nazw" typów) oraz funkcji dekodującej, która przyporządkowuje kodom prawdziwe typy.

Programowanie generyczne to programowanie funkcji, które operują na kolekcjach typów o dowolnych kształtach, czyli na uniwersach właśnie. Generyczność od polimorfizmu różni się tym, że funkcja polimorficzna działa dla dowolnego typu, zaś generyczna - tylko dla typu o pasującym kształcie.

Jak dokończyć implementację funkcji *flatten*? Kluczowe jest zauważenie, że możemy zdefiniować *flatten* przez rekursję strutkuralną po argumencie domyślnym typu *FlattenType*.

Ostatni problem to jak zrobić, żeby Coq sam zgadywał kod danego typu - dowiemy się tego w rozdziale o klasach.

Co to wszystko ma wspólnego z uniwersami? Ano, jeżeli chcemy definiować bardzo za-awansowane funkcje generyczne, musimy mieć do dyspozycji bardzo potężne uniwersa i to właśnie je zapewnia nam indukcja-rekursja. Ponieważ w powyższym przykładzie generyczność nie była zbyt wyrafinowana, nie było potrzeby używania indukcji-rekursji, jednak uszy do góry: przykład nieco bardziej skomplikowanego uniwersum pojawi się jeszcze w tym rozdziale.

Ćwiczenia Nieco podchwytliwe zadanie: zdefiniuj uniwersum funkcji $nat \to nat$, $nat \to (nat \to nat)$, $(nat \to nat) \to nat$, $(nat \to nat) \to (nat \to nat)$ i tak dalej, dowolnie zagnieżdżonych.

Zagadka: czy potrzebna jest nam indukcja-rekursja?

4.4.4 Indeksowana indukcja-rekursja

Za siedmioma górami, za siedmioma lasami, za siedmioma rzekami, za siedmioma budkami telefonicznymi, nawet za indukcją-rekursją (choć tylko o kroczek) leży indeksowana indukcja-rekursja, czyli połączenie indukcji-rekursji oraz indeksowanych rodzin typów.

Jako, że w porównaniu do zwykłej indukcji-rekursji nie ma tu za wiele innowacyjności, przejdźmy od razu do przykładu przydatnej techniki, którą nasza tytułowa bohaterka umożliwia, a zwie się on metodą induktywnej dziedziny.

Pod tą nazwą kryje się sposób definiowania funkcji, pozwalający oddzielić samą definicję od dowodu jej terminacji. Jeżeli ten opis nic ci nie mówi, nie martw się: dotychczas definiowaliśmy tylko tak prymitywne funkcje, że tego typu fikołki nie były nam potrzebne.

Metoda induktywnej dziedziny polega na tym, żeby zamiast funkcji $f:A\to B$, która nie jest strukturalnie rekurencyjna (na co Coq nie pozwala) napisać funkcję $f:\forall x:A$, $Dx\to B$, gdzie $D:A\to {\rm Type}$ jest "predykatem dziedziny", który sprawia, że dziwna rekursja z oryginalnej definicji f staje się rekursją strukturalną po dowodzie Dx. Żeby zdefiniować oryginalne $f:A\to B$ wystarczy udowodnić, że każde x:A spełnia predykat dziedziny.

Co to wszystko ma wspólnego z indeksowaną indukcją-rekursją? Już piszę. Otóż metoda ta nie wymaga w ogólności indukcji-rekursji - ta staje się potrzebna dopiero, gdy walczymy z bardzo złośliwymi funkcjami, czyli takimi, w których rekursja jest zagnieżdżona, tzn. robimy wywołanie rekurencyjne na wyniku poprzedniego wywołania rekurencyjnego.

Predykat dziedziny dla takiej funkcji musi zawierać konstruktor w stylu "jeżeli wynik wywołania rekurencyjnego na x należy do dziedziny, to x też należy do dziedziny". To właśnie tu ujawnia się indukcja-rekursja: żeby zdefiniować predykat dziedziny, musimy odwołać się do funkcji (żeby móc powiedzieć coś o wyniku wywołania rekurencyjnego), a żeby zdefiniować funkcję, musimy mieć predykat dziedziny.

Brzmi skomplikowanie? Jeżeli czegoś nie rozumiesz, to jesteś debi... a nie, czekaj. Jeżeli czegoś nie rozumiesz, to nie martw się: powyższy przykład miał na celu jedynie zilustrować jakieś praktyczne zastosowanie indeksowanej indukcji-rekursji. Do metody induktywnej

dziedziny powrócimy w kolejnym rozdziale. Pokażemy, jak wyeliminować z niej indukcjęrekursję, tak żeby uzyskane za jej pomocą definicje można było odpalać w Coqu. Zobaczymy też, jakimi sposobami dowodzić, że każdy element dziedziny spełnia predykat dziedziny, co pozwoli nam odzyskać oryginalną definicję funkcji, a także dowiemy się, jak z "predykatu" o typie $D:A\to T$ ype zrobić prawdziwy predykat $D:A\to P$ rop.

4.4.5 Indukcja-indukcja-rekursja

Ufff... przebrnęliśmy przez indukcję-indukcję i (indeksowaną) indukcję-rekursję. Czy mogą istnieć jeszcze potężniejsze i bardziej innowacyjne sposoby definiowania typów przez indukcję?

Ależ oczywiście. Jest nim... uwaga uwaga, niespodzianka... indukcja-indukcja-rekursja, która jest nie tylko strasznym potfurem, ale też powinna dostać Oskara za najlepszą nazwę.

Chodzi tu oczywiście o połączenie indukcji-indukcji i indukcji-rekursji: możemy jednocześnie zdefiniować jakiś typ A, rodzinę typów B indeksowaną przez A oraz operujące na nich funkcje, do których konstruktory A i B mogą się odwoływać.

Nie ma tu jakiejś wielkiej filozofii: wszystkiego, co powinieneś wiedzieć o indukcji-indukcji-rekursji, dowiedziałeś się już z dwóch poprzednich podrozdziałów. Nie muszę chyba dodawać, że ława oburzonych jest oburzona faktem, że Coq nie wspiera indukcji-indukcji-rekursji.

Rodzi się jednak to samo super poważne pytanie co zawsze, czyli do czego można tego tałatajstwa użyć? Przez całkiem długi czas nie miałem pomysłu, ale okazuje się, że jest jedno takie zastosowanie i w sumie narzuca się ono samo.

Przypomnij sobie metodę induktywno-rekurencyjnej dziedziny, czyli jedno ze sztandarowych zastosowań indeksowanej indukcji-rekursji. Zaczynamy od typu I: Type, na którym chcemy zdefiniować funkcję o niestandardowym kształcie rekursji. W tym celu definiujemy dziedzinę $D:I\to T$ ype wraz z funkcją $f:\forall i:I,D:i\to R$.

Zauważmy, jaki jest związek typu I z funkcją f: najpierw jest typ, potem funkcja. Co jednak, gdy musimy I oraz f zdefiniować razem za pomocą indukcji-rekursji? Wtedy f może być zdefiniowane jedynie za pomocą rekursji strukturalnej po I, co wyklucza rekursję o fikuśnym kształcie...

I tu wchodzi indukcja-indukcja-rekursja, cała na biało. Możemy użyć jej w taki sposób, że definiujemy jednocześnie:

- typ I, który odnosi się do funkcji f
- predykat dziedziny $D: I \to \mathsf{Type}$, który jest indeksowany przez I
- \bullet funkcję f, która zdefiniowana jest przez rekursję strukturalną po dowodzie należenia do dziedziny

Jak widać, typ zależy od funkcji, funkcja od predykatu, a predykat od typu i koło się zamyka.

Następuje jednak skądinąd uzasadnione pytanie: czy faktycznie istnieje jakaś sytuacja, w której powyższy schemat działania jest tym słusznym? Odpowiedź póki co może być tylko jedna: nie wiem, ale się domyślam.

4.4.6 Najstraszniejszy potfur

Na koniec dodam jeszcze na zachętę (albo zniechętę, zależy jakie kto ma podejście), że istnieje jeszcze jeden potfur, straszniejszy nawet niż indukcja-indukcja-rekursja, ale jest on zbyt straszny jak na ten rozdział i być może w ogóle zbyt straszny jak na tę książkę - panie boże, daj odwagę na omówienie go!

4.5 Dobre, złe i podejrzane typy induktywne

Poznana przez nas dotychczas definicja typów induktywnych (oraz wszelkich ich ulepszeń, jak indukcja-indukcja, indukcja-rekursja etc.) jest niepełna. Tak jak świat pełen jest złoczyńców oszukujących starszych ludzi metodą "na wnuczka", tak nie każdy typ podający się za induktywny faktycznie jest praworządnym obywatelem krainy typów induktywnych.

Na szczęście typy induktywne to istoty bardzo prostolinijne, zaś te złe można odróżnić od tych dobrych gołym okiem, za pomocą bardzo prostego kryterium: złe typy induktywne to te, które nie są ściśle pozytywne. Zanim jednak dowiemy się, jak rozpoznawać złe typy, poznajmy najpierw dwa powody, przez które złe typy induktywne są złe.

4.5.1 Nieterminacja jako źródło zła na świecie

Przyjrzyjmy się poniższemu typowemu przypadkowi negatywnego typu induktywnego (czyli takiego, który wygląda na induktywny, ale ma konstruktory z negatywnymi wystąpieniami argumentu indukcyjnego):

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.

Pierwszym powodem nielegalności nie-ściśle-pozytywnych typów induktywnych jest to, że unieważniają one filozoficzną interpretację teorii typów i ogólnie wszystkiego, co robimy w Coqu. W praktyce problemy filozoficzne mogą też prowadzić do sprzeczności.

Załóżmy, że Coq akceptuje powyższą definicję wut. Możemy wtedy napisać następujący program:

```
Fail Definition loop (A: Type): A:= let f (w: wut \ A): A:= match w with | \ C \ g \Rightarrow g \ w end in f (C \ f).
```

Już sam typ tego programu wygląda podejrzanie: dla każdego typu A zwraca on element typu A. Nie dziwota więc, że możemy uzyskać z niego dowód fałszu.

```
Fail Definition santa\_is\_a\_pedophile: False:=loop\ False.
```

Paradoksalnie jednak to nie ta rażąca sprzeczność jest naszym największym problemem - nie z każdego złego typu induktywnego da się tak łatwo dostać sprzeczność (a przynajmniej ja nie umiem; systematyczny sposób dostawania sprzeczności z istnienia takich typów zobaczymy później). W rzeczywistości jest nim nieterminacja.

Nieterminacja (ang. nontermination, divergence) lub kolokwialniej "zapętlenie" to sytuacja, w której program nigdy nie skończy się wykonywać. Ta właśnie bolączka jest przypadłością *loop*, czego nie trudno domyślić się z nazwy.

Dlaczego tak jest? Przyjrzyjmy się definicji loop. Za pomocą leta definiujemy funkcję f: $wut\ A \to A$, która odpakowuje swój argument w, wyciąga z niego funkcję g: $wut\ A \to A$ i aplikuje g do w. Wynikiem programu jest f zaaplikowane do siebie samego zawiniętego w konstruktor C.

Przyjrzyjmy się, jak przebiegają próby wykonania tego nieszczęsnego programu:

```
\begin{array}{l} loop \ A = \\ \texttt{let} \ f := \dots \ \texttt{in} \ f \ (C \ f) = \\ \texttt{let} \ f := \dots \ \texttt{in} \ \texttt{match} \ C \ f \ \texttt{with} \ | \ C \ g \Rightarrow g \ (C \ f) \ \texttt{end} = \\ \texttt{let} \ f := \dots \ \texttt{in} \ f \ (C \ f) \\ \texttt{i} \ \texttt{tak} \ \texttt{dalej}. \end{array}
```

Nie powinno nas to dziwić - praktycznie rzecz biorąc aplikujemy f samo do siebie, zaś konstruktor C jest tylko pośrednikiem sprawiającym, że typy się zgadzają. Ogólniej sytuacja, w której coś odnosi się samo do siebie, nazywa się autoreferencją i często prowadzi do różnych wesołych paradoksów.

Ćwiczenie Poniższą zagadkę pozwolę sobie wesoło nazwać "paradoks hetero". Zagadka brzmi tak:

Niektóre słowa opisują same siebie, np. słowo "krótki" jest krótkie, a niektóre inne nie, np. słowo "długi" nie jest długie. Podobnie słowo "polski" jest słowem polskim, ale słowo "niemiecki" nie jest słowem niemieckim. Słowa, które nie opisują samych siebie będziemy nazywać słowami heterologicznymi. Pytanie: czy słowo "heterologiczny" jest heterologiczne?

Czujesz sprzeczność? Innym przyk... dobra, wystarczy tych głupot.

Przyjrzyjmy się teraz problemom filozoficznym powodowanym przez nieterminację. W skrócie: zmienia ona fundamentalne właściwości obliczeń, co prowadzi do zmiany interpretacji pojęcia typu, zaś to pociąga za sobą kolejne przykre skutki, takie jak np. to, że reguły eliminacji tracą swoje uzasadnienie.

Brzmi mega groźnie, prawda? Kiedy wspomniałem o tym Sokratesowi, to sturlał się z podłogi na sufit bez pośrednictwa ściany.

Na szczęście tak naprawdę, to sprawa jest prosta. W Coqu wymagamy, aby każde obliczenie się kończyło. Wartości, czyli końcowe wyniki obliczeń (które są też nazywane postaciami kanonicznymi albo normalnymi), możemy utożsamiać z elementami danego typu. Dla przykładu wynikami obliczania termów typu bool są true i false, więc możemy myśleć, że są to elementy typu bool i bool składa się tylko z nich. To z kolei daje nam uzasadnienie reguły eliminacji (czyli indukcji) dla typu bool: żeby udowodnić $P:bool \rightarrow Prop$ dla każdego b:bool, wystarczy udowodnić P true i P false, gdyż true i false są jedynymi elementami typu bool.

Nieterminacja obraca tę jakże piękną filozoficzną wizję w perzynę: nie każde obliczenie się kończy, a przez to powstają nowe, "dziwne" elementy różnych typów. loop bool nigdy nie obliczy się do true ani do false, więc możemy traktować je jako nowy element typu bool. To sprawia, że bool, typ z założenia dwuelementowy, ma teraz co najmniej trzy elementy - true, false i loop bool. Z tego też powodu reguła eliminacji przestaje obowiązywać, bo wymaga ona dowodów jedynie dla true i false, ale milczy na temat loop bool. Moglibyśmy próbować naiwnie ją załatać, uwzględniając ten dodatkowy przypadek, ale tak po prawdzie, to nie wiadomo nawet za bardzo jakie jeszcze paskudztwa rozpanoszyły się w typie bool z powodu nieterminacji.

Morał jest prosty: nieterminacja to wynalazek szatana, a negatywne typy induktywne to też wynalazek szatana.

4.5.2 Twierdzenie Cantora

Zanim zaczniemy ten rozdział na poważnie, mam dla ciebie wesoły łamaniec językowy:

Cantor - kanciarz, który skradł zza kurtyny kantoru z Kantonu kontury kartonu Koranicznemu kanarowi, który czasem karał karczystych kafarów czarami za karę za kantowanie i za zakatowanie zza kontuaru konarem kontrkulturowych kuluarowych karłów.

Dobra, wystarczy. Reszta tego podrozdziału będzie śmiertelnie poważna, a przyjrzymy się w niej jednemu z mega klasycznych twierdzeń z końca XIX w. głoszącemu mniej więcej, że "zbiór potęgowy zbioru liczb naturalnych ma większą moc od zbioru liczb naturalnych".

Co za bełkot, pomyślisz zapewne. Ten podrozdział poświęcimy właśnie temu, żeby ów bełkot nieco wyklarować. Jeżeli zaś zastanawiasz się, po co nam to, to odpowiedź jest prosta - na (uogólnionym) twierdzeniu Cantora opierać się będzie nasza systematyczna metoda dowodzenia nielegalności negatywnych typów induktywnych.

Oczywiście oryginalne sformułowanie twierdzenia powstało na długo przed powstaniem teorii typów czy Coqa, co objawia się np. w tym, że mówi ono o *zbiorze* liczb naturalnych, podczas gdy my dysponujemy *typem* liczb naturalnych. Musimy więc oryginalne sformułowanie lekko przeformułować, a także wprowadzić wszystkie niezbędne nam pojęcia.

```
Definition surjective \{A \ B : \mathtt{Type}\}\ (f : A \to B) : \mathtt{Prop} := \forall \ b : B, \exists \ a : A, f \ a = b.
```

Pierwszym niezbędnym pojęciem jest pojęcie surjekcji. Powyższa definicja głosi, że funkcja jest surjektywna, gdy każdy element przeciwdziedziny jest wynikiem funkcji dla pewnego elementu dziedziny. Surjekcja to funkcja, która jest surjektywna.

O co chodzi w tej definicji? Samo słowo "surjekcja" pochodzi od fr. "sur" - "na" oraz łac. "iacere" - "rzucać". Ubogim tłumaczeniem na polski może być słowo "narzut".

Idea jest taka, że surjekcja $f:A\to B$ "narzuca" swoją dziedzinę A na przeciwdziedzinę B tak, że A "pokrywa" całe B. Innymi słowy, każdy element b:B jest pokryty przez jakiś element a:A, co wyraża równość f a=b. Oczywiście to a nie musi być unikalne - b może być pokrywane przez dużo różnych a. Jak widać, dokładnie to jest napisane w powyższej definicji.

Mówiąc jeszcze prościej: jeżeli $f:A\to B$ jest surjekcją, to typ A jest większy (a precyzyjniej mówiący, nie mniejszy) niż typ B. Oczywiście nie znaczy to, że jeżeli f nie jest surjekcją, to typ A jest mniejszy niż B - mogą przecież istnieć inne surjekcje. Powiemy, że A jest mniejszy od B, jeżeli nie istnieje żadna surjekcja między nimi.

Spróbujmy rozrysować to niczym Jacek Gmoch... albo nie, bo nie umiem jeszcze rysować, więc zamiast tego będzie przykład i ćwiczenie.

Funkcja *isZero*, która sprawdza, czy jej argument jest zerem, jest surjekcją, bo każdy element typu *bool* może być jej wynikiem - *true* jest wynikiem dla 0, zaś *false* jest jej wynikiem dla wszystkich innych argumentów. Wobec tego możemy skonkludować, że typ *nat* jest większy niż typ *bool* i w rzeczywistości faktycznie tak jest: *bool* ma dwa elementy, a *nat* nieskończenie wiele.

Do kwestii tego, który typ ma ile elementów wrócimy jeszcze w rozdziale o typach i funkcjach. Tam też zapoznamy się lepiej z surjekcjami i innymi rodzajami funkcji. Tymczasem ćwiczenie:

Ćwiczenie Czy funkcja plus 5 jest surjekcją? A funkcja fun n: $nat \Rightarrow minus$ n 5? Sprawdź swoje odpowiedzi w Coqu. Na koniec filozoficznie zinterpretuj otrzymany wynik.

Wskazówka: minus to funkcja z biblioteki standardowej, która implementuje odejmowanie liczb naturalnych z obcięciem, tzn. np. 2 - 5 = 0. Użyj Printa, żeby dokładnie zbadać jej definicję.

Pozostaje jeszcze kwestia tego, czym jest "zbiór potęgowy zbioru liczb naturalnych". Mimo groźnej nazwy sprawa jest prosta: jest to archaiczne określenie na typ funkcji $nat \rightarrow bool$. Każdą funkcję $f: nat \rightarrow bool$ możemy interpretować jako pewną kolekcję (czyli właśnie zbiór) elementów typu nat, zaś f n, czyli wynik f na konkretnym n, mówi nam, czy n znajduje się w tej kolekcji, czy nie.

To w zasadzie wyczerpuje zestaw pojęć potrzebnych nam do sformułowania twierdzenia. Pojawiająca się w oryginalnej wersji "większa moc" to po prostu synonim określenia "większy", które potrafimy już wyrażać za pomocą pojęcia surjekcji. Tak więc nowszą (czyli bardziej postępową) wersję twierdzenia Cantora możemy sformułować następująco: nie istnieje surjekcja z nat w $nat \rightarrow bool$. Lub jeszcze bardziej obrazowo: nat jest mniejsze niż $nat \rightarrow bool$.

```
Theorem Cantor: \forall f: nat \rightarrow (nat \rightarrow bool), \neg surjective f. Proof. unfold surjective. intros f Hf. pose (diagonal:= fun n: nat \Rightarrow negb \ (f \ n \ n)). destruct (Hf \ diagonal) as [n \ Hn]. apply (\texttt{f\_equal} \ (\texttt{fun} \ h: nat \rightarrow bool \Rightarrow h \ n)) in Hn. unfold diagonal in Hn. destruct (f \ n \ n); inversion Hn. Qed.
```

Dowód twierdzenia jest równie legendarny jak samo twierdzenie, a na dodatek bajecznie prosty i niesamowicie użyteczny - jeżeli będziesz zajmował się w życiu matematyką i informatyką, spotkasz go w jeszcze wielu odsłonach. Metoda stojąca za dowodem nazywana bywa argumentem przekątniowym - choć nazwa ta może się wydawać dziwna, to za chwilę stanie się zupełnia jasna.

O co więc chodzi w powyższym dowodzie? Po pierwsze zauważmy, że mamy do czynienia z funkcją $f: nat \to (nat \to bool)$, czyli funkcją, która bierze liczbę naturalną i zwraca funkcję z liczb naturalnych w bool. Pamiętajmy jednak, że \to łączy w prawo i wobec tego typ f możemy zapisać też jako $nat \to nat \to bool$. Tak więc f jest funkcją, która bierze dwie liczby naturalne i zwraca element typu bool.

Dzięki temu zabiegowi możemy wyobrażać sobie f jako dwuwymiarową tabelkę, której wiersze i kolumny są indeksowane liczbami naturalnymi, a komórki w tabelce wypełnione są wartościami typu bool. Przyjmijmy, że pierwszy argument f to indeks wiersza, zaś drugi to indeks kolumny. W takim układzie f n m to wartość n-tej funkcji na argumencie m. Wobec tego twierdzenie możemy sparafrazować mówiąc, że każda funkcja $nat \rightarrow bool$ znajduje się w którymś wierszu tabelki.

To tyle wyobraźni - możemy już udowodnić twierdzenie. Na początku oczywiście bierzemy dowolne f oraz zakładamy, że jest surjekcją, uprzednio odwijając definicję bycia surjekcją.

Teraz musimy jakoś wyciągnąć sprzeczność z hipotezy Hf, czyli, używając naszej tabelkowej parafrazy, znaleźć funkcję z nat w bool, która nie znajduje się w tabelce. A nie znajdować się w tabelce, panie Ferdku, to znaczy: różnić się od każdej funkcji z tabelki na jakimś argumencie.

Zamiast jednak szukać takiej funkcji po omacku, skonstruujmy ją z tego, co mamy pod ręką - czyli z naszej tabelki. Jak przerobić funkcje z tabelki na nową, której w nie ma w tabelce? Tu właśnie ujawnia się geniuszalność Cantora: użyjemy metody przekątniowej, czyli spojrzymy na przekątną naszej tabelki.

Definiujemy więc nową funkcję $diagonal: nat \rightarrow bool$ następująco: dla argumentu n: nat bierzemy funkcję z n-tego wiersza w tabelce, patrzymy na n-tą kolumnę, czyli na wartość funkcji na argumencie n, i zwracamy negację tego, co tam znajdziemy. Czujesz sprzeczność?

Nasze założenie mówi, że diagonal znajduje się w którymś wierszu tabelki - niech ma on numer n. Wiemy jednak, że g różni się od n-tej funkcji z tabelki na argumencie n, gdyż zdefiniowaliśmy ją jako negację tej właśnie komórki w tabelce. Dostajemy stąd równość f n = diagonal n = negb (f n n), co po analizie przypadków daje ostatecznie true = false lub false = true.

Voilà! Sprzeczność osiągnięta, a zatem początkowe założenie było błędne i nie istnieje żadna surjekcja z $nat \le nat \le nat$

Ćwiczenie Udowodnij, że nie ma surjekcji z nat w $nat \rightarrow nat$. Czy jest surjekcja z $nat \rightarrow bool$ w $(nat \rightarrow bool) \rightarrow bool$? A w $nat \rightarrow bool \rightarrow bool$?

Poznawszy twierdzenie Cantora, możemy powrócić do ścisłej pozytywności, czyż nie? Otóż nie, bo twierdzenie Cantora jest biedne. Żeby czerpać garściami niebotyczne profity, musimy najpierw uogólnić je na dowolne dwa typy A i B znajdując kryterium mówiące, kiedy nie istnieje surjekcja z A w $A \rightarrow B$.

```
Theorem Cantor': \forall \{A \ B : \mathsf{Type}\}\ (f : A \to (A \to B))\ (modify : B \to B), \ (\forall \ b : B, modify \ b \neq b) \to \neg \ surjective \ f. Proof. unfold surjective. intros A \ B \ f \ modify \ H \ Hf. pose (g := \mathsf{fun} \ x : A \Rightarrow modify \ (f \ x \ x)). destruct (Hf \ g) as [x \ Hx]. apply (\mathsf{f\_equal}\ (\mathsf{fun}\ h \Rightarrow h \ x)) in Hx. unfold g in Hx. apply (H \ (f \ x \ x)). symmetry. assumption. Qed.
```

Uogólnienie jest dość banalne. Najpierw zastępujemy nat i bool przez dowolne typy A i B. W oryginalnym twierdzeniu nie użyliśmy żadnej właściwości liczb naturalnych, więc nie musimy szukać żadnych kryteriów dla typu A. Nasza tabelka może równie dobrze być indeksowana elementami dowolnego typu - dalej jest to tabelka i dalej ma przekątną.

Twierdzenie było jednak zależne od pewnej właściwości bool, mianowicie funkcja diagonal była zdefiniowana jako negacja przekątnej. Było nam jednak potrzeba po prostu funkcji,

która dla każdego elementu z przekątnej zwraca element bool od niego różny. Ponieważ bool ma dokładnie dwa elementy, to negacja jest jedyną taką funkcją.

Jednak w ogólnym przypadku dobra będzie dowolna B-endofunkcja bez punktów stałych. Ha! Nagły atak żargonu bezzębnych ryb, co? Zróbmy krótką przerwę, żeby zbadać sposób komunikacji tych czarodziejskich zwierząt pływających po uczelnianych korytarzach.

Endofunkcja to funkcja, która ma taką samą dziedzinę i przeciwdziedzinę. Można się zatem domyślać, że B-endofunkcja to funkcja o typie $B \to B$. Punkt stały zaś to takie x, że f x = x. Jest to więc dokładnie ta własność, której chcemy, żeby pożądana przez nas funkcja nie miała dla żadnego x. Jak widać, żargon bezzębnych ryb jest równie zwięzły jak niepenetrowalny dla zwykłych śmiertelników.

Podsumowując: w uogólnionym twierdzeniu Cantora nie wymagamy niczego od A, zaś od B wymagamy tylko, żeby istniała funkcja $modify: B \to B$, która spełnia $\forall b: B, modify$ $b \neq b$. Dowód twierdzenia jest taki jak poprzednio, przy czym zastępujemy użycie negb przez modify.

Ćwiczenie Znajdź jedyny słuszny typ B, dla którego nie istnieje żadna B-endofunkcja bez punktów stałych.

Podpowiedź: to zadanie jest naprawdę proste i naprawdę istnieje jedyny słuszny typ o tej właściwości.

Pytanie (bardzo trudne): czy da się udowodnić w Coqu, że istnieje dokładnie jeden taki typ? Jeżeli tak, to w jakim sensie typ ten jest unikalny i jakich aksjomatów potrzeba do przepchnięcia dowodu?

4.5.3 Twierdzenie Cantora jako młot na negatywność

Z Cantorem po naszej stronie możemy wreszcie kupić ruble... ekhem, możemy wreszcie zaprezentować ogólną metodę dowodzenia, że negatywne typy induktywne prowadzą do sprzeczności. Mimo szumnej nazwy ogólna metoda nie jest aż taka ogólna i często będziemy musieli się bonusowo napracować.

Module Example 1.

Otworzymy sobie nowy moduł, żeby nie zaśmiecać globalnej przestrzeni nazw - wszystkie nasze złe typy będą się nazywały wut. Przy okazji, zdecydowanie powinieneś nabrać podejrzeń do tej nazwy - jeżeli coś w tej książce nazywa się wut, to musi to być złowrogie, podejrzane paskudztwo.

Axioms

```
(wut: \texttt{Type} \to \texttt{Type})

(C: \forall \{A: \texttt{Type}\}, (wut \ A \to A) \to wut \ A)

(dcase: \forall (A: \texttt{Type}))

(P: wut \ A \to \texttt{Type})

(PC: \forall f: wut \ A \to A, P(Cf)),
```

```
 \{f: \forall w: wut \ A, P \ w \mid \\ \forall g: wut \ A \rightarrow A, f \ (C \ g) = PC \ g\} \}.
```

wut to aksjomatyczne kodowanie tego samego typu, o którym poprzednio tylko udawaliśmy, że istnieje. Zauważmy, że nie jest nam potrzebna reguła indukcji - wystarczy jeden z prostszych eliminatorów, mianowicie dcase, czyli zależna reguła analizy przypadków.

```
Definition bad (A: Type): wut A \rightarrow (wut A \rightarrow A). Proof. apply (dcase\ A\ (fun\ \_\Rightarrow wut\ A \rightarrow A)). intro f. exact f. Defined.
```

Dlaczego typ wut A jest nielegalny, a jego definicja za pomocą komendy Inductive jest odrzucana przez Coqa? Poza wspomnianymi w poprzednim podrozdziale problemami filozoficznymi wynikającymi z nieterminacji, jest też drugi, bardziej namacalny powód: istnienie typu wut A jest sprzeczne z (uogólnionym) twierdzeniem Cantora.

Powód tej sprzeczności jest dość prozaiczny: za pomocą konstruktora C możemy z dowolnej funkcji $wut\ A \to A$ zrobić element $wut\ A$, a skoro tak, to dowolne $w: wut\ A$ możemy odpakować i wyjąć z niego funkcję $f: wut\ A \to A$.

```
Lemma bad\_sur:
\forall \ A : \texttt{Type}, \ surjective \ (bad \ A).
Proof.
\texttt{unfold} \ surjective. \ \texttt{intros} \ A \ f.
\texttt{unfold} \ bad. \ \texttt{destruct} \ (dcase \ \_) \ \texttt{as} \ [g \ H].
\exists \ (C \ f). \ \texttt{apply} \ H.
```

Skoro możemy włożyć dowolną funkcję, to możemy także wyjąć dowolną funkcję, a zatem mamy do czynienia z surjekcją.

```
 \begin{array}{l} \texttt{Lemma} \ worst : False. \\ \texttt{Proof.} \\ \texttt{apply} \ (Cantor' \ (bad \ bool) \ negb). \\ \texttt{destruct} \ b; \ \texttt{inversion} \ 1. \\ \texttt{apply} \ bad\_sur. \\ \\ \texttt{Qed.} \end{array}
```

W połączeniu zaś z twierdzeniem Cantora surjekcja $wut\ A \to (wut\ A \to A)$ prowadzi do sprzeczności - wystarczy za A wstawić bool.

End Example 1.

Qed.

Przykład może ci się jednak wydać niezadowalający - typ wut jest przecież dość nietypowy, bo ma tylko jeden konstruktor. A co, gdy konstruktorów jest więcej?

Początkowo miałem opisać kilka przypadków z większą liczbą konstruktorów, ale stwierdziłem, że jednak mi się nie chce. W ćwiczeniach zobaczymy, czy będziesz w stanie sam wykombinować, jak się z nimi uporać.

Ćwiczenie Poniższe paskudztwo łamie prawo ścisłej pozytywności nie jednym, lecz aż dwoma swoimi konstruktorami.

Zakoduj ten typ aksjomatycznie i udowodnij, że jego istnienie prowadzi do sprzeczności. Metoda jest podobna jak w naszym przykładzie, ale trzeba ją troszkę dostosować do zastanej sytuacji.

Module Exercise1.

```
 \begin{array}{c|c} Fail \ \ Inductive \ wut: \ \ Type := \\ | \ C0: (wut \rightarrow bool) \rightarrow wut \\ | \ C1: (wut \rightarrow nat) \rightarrow wut. \end{array}
```

Lemma $wut_illegal$: False.

End Exercise1.

Ćwiczenie Poniższe paskudztwo ma jeden konstruktor negatywny, a drugi pozytywny, niczym typowa panienka z borderlinem...

Polecenie jak w poprzednim ćwiczeniu.

Module Exercise 2.

```
Fail Inductive wut: Type := |C0:(wut \rightarrow wut) \rightarrow wut
|C1:nat \rightarrow wut.
```

Lemma $wut_illegal$: False.

End Exercise 2.

 $\overset{\bullet}{\mathbf{C}}$ wiczenie Poniższy typ reprezentuje termy beztypowego rachunku lambda, gdzie V to typ reprezentujący zmienne. Co to za zwierzątko ten rachunek lambda to my się jeszcze przekonamy... chyba, oby.

Taki sposób reprezentowania rachunku lambda (i ogólnie składni języków programowania) nazywa się HOAS, co jest skrótem od ang. Higher Order Abstract Syntax. W wielu językach funkcyjnych jest to popularna technika, ale w Coqu, jak zaraz udowodnisz, jest ona nielegalna. Ława oburzonych jest rzecz jasna oburzona!

Module Exercise3.

```
 \begin{array}{ll} Fail \ \  \mbox{Inductive} \ \ Term \ (V: \mbox{Type}) : \mbox{Type} := \\ & | \ \ Var : \ V \rightarrow \ Term \ \ V \\ & | \ \  Lam : (Term \ \ V \rightarrow \ Term \ \ V) \rightarrow \ Term \ \ V \\ & | \ \  App : \ Term \ \ V \rightarrow \ Term \ \ V \rightarrow \ Term \ \ V. \end{array}
```

Lemma $Term_illegal$: False.

End Exercise3.

Ćwiczenie Poniższe bydle jest najgorsze z możliwych - póki co nie wiem, jak to udowodnić. Powodzenia!

```
Module Exercise4.

Fail Inductive wut: Type := \mid C: (wut \rightarrow wut) \rightarrow wut.

Lemma wut\_illegal: False.

End Exercise4.
```

4.5.4 Poradnik rozpoznawania negatywnych typów induktywnych

Skoro już wiemy, że negatywne typy induktywne są wynalazkiem szatana, to czas podać proste kryterium na ich rozpoznawanie. Jeżeli jesteś sprytny, to pewnie sam zdążyłeś już zauważyć ogólną regułę. Jednak aby nie dyskryminować osób mało sprytnych, trzeba napisać ją wprost.

Kryterium jest banalne. Mając dany typ T musimy rzucić okiem na jego konstruktory, a konkretniej na ich argumenty. Argumenty nieindukcyjne (czyli o typach, w których nie występuje T) są zupełnie niegroźnie. Interesować nas powinny jedynie argumenty indukcyjne, czyli takie, w których występuje typ T.

Niektóre typy argumentów indukcyjnych są niegroźne, np. $T \times T$, T + T, list T albo $\{t : T \mid P \mid t\}$, ale powodują one, że Coq nie potrafi wygenerować odpowiedniej reguły indukcji i zadowala się jedynie regułą analizy przypadków. Nie prowadzą one do sprzeczności, ale powinniśmy ich unikać.

Argument typu $T \times T$ można zastąpić dwoma argumentami typu T i podobnie dla $\{t: T \mid P \mid t\}$. Konstruktor z argumentem typu T + T możemy rozbić na dwa konstruktory (i powinniśmy, bo jest to bardziej czytelne). Konstruktor z wystąpieniem list T możemy przerobić na definicję przez indukcję wzajemną (ćwiczenie: sprawdź jak), ale lepiej chyba po prostu zaimplementować regułę indukcji ręcznie.

Rodzaje nieszkodliwych typów argumentów widać na powyższym przykładzie. Konstruktory $c\theta$ i c1 są nieindukcyjne, więc są ok. Konstruktor c2 jest indukcyjny - jest jeden argument typu $T\theta$. Zauważ, że typem konstruktora c2 jest $T\theta \to T\theta$, ale nie oznacza to, że $T\theta$ występuje po lewej stronie strzałki!

Jest tak, gdyż ostatnie wystąpienie $T\theta$ jest konkluzją konstruktora c2. Ważne są tylko wystąpienia po lewej stronie strzałki w argumentach (gdyby konstruktor c2 nie był legalny, to jedynymi legalnymi typami induktywnymi byłyby enumeracje).

Konstruktory c3, c4, c5 i c6 są induktywne i również w pełni legalne, ale są one powodem tego, że Coq nie generuje dla $T\theta$ reguły indukcji, a jedynie regułę analizy przypadków (choć nazwa się ona $T\theta_-ind$).

Ćwiczenie Zrefaktoryzuj powyższy upośledzony typ.

Problem pojawia się dopiero, gdy typ T występuje po lewej stronie strzałki, np. $T \to bool, T \to T, (T \to T) \to T$, lub gdy jest skwantyfikowany uniwersalnie, np. $\forall t: T, P t, \forall f: (\forall t: T, P t), Q f$.

W trzech poprzednich podrozdziałach mierzyliśmy się z sytuacjami, gdy typ T występował bezpośrednio na lewo od strzałki, ale oczywiście może on być dowolnie zagnieżdżony. Dla każdego wystąpienia T w argumentach możemy policzyć, na lewo od ilu strzałek (albo jako jak mocno zagnieżdżona dziedzina kwantyfikacji) się ono znajduje. Liczbę tę nazywać będziemy niedobrością. W zależności od niedobrości, wystąpienie nazywamy:

- 0 wystąpienie ściśle pozytywne
- liczba nieparzysta wystąpienie negatywne
- liczba parzysta (poza 0) wystąpienie pozytywne

Jeżeli w definicji mamy wystąpienie negatywne, to typ możemy nazywać negatywnym typem induktywnym (choć oczywiście nie jest to typ induktywny). Jeżeli nie ma wystąpień negatywnych, ale są wystąpienia pozytywne, to typ nazywamy pozytywnym typem induktywnym (lub nie ściśle pozytywnym typem induktywnym), choć oczywiście również nie jest to typ induktywny. Jeżeli wszystkiego wystąpienia są ściśle pozytywne, to mamy do czynienia po prostu z typem induktywnym.

Podobne nazewnictwo możemy sobie wprowadzić dla konstruktorów (konstruktory negatywne, pozytywne i ściśle pozytywne), ale nie ma sensu, bo za tydzień i zapomnisz o tych mało istotnych detalach. Ważne, żebyś zapamiętał najważniejsze, czyli ideę.

W powyższym przykładzie wystąpienie T1 w pierwszym argumencie $T1_0$ jest ściśle pozytywne (na lewo od 0 strzałek). Pierwsze wystąpienie T1 w $T1_1$ jest negatywne (na lewo od 1 strzałki), a drugie ściśle pozytywne (na lewo od 0 strzałek). Pierwsze wystąpienie T1 w $T1_2$ jest pozytywne (na lewo od 2 strzałek), drugie negatywne (na lewo od 1 strzałki), trzecie zaś ściśle pozytywne (na lewo od 0 strzałek). Pierwsze wystąpienie T1 w $T1_3$ jest

negatywne (dziedzina kwantyfikacji), drugie zaś pozytywne (na lewo od jednej strzałki, ale ta strzałka jest w typie, po którym kwantyfikujemy).

Konstruktor $T1_0$ jest ściśle pozytywne, zaś konstruktory $T1_1$, $T1_2$ oraz $T1_3$ są negatywne. Wobec tego typ T1 jest negatywnym typem induktywnym (czyli wynalazkiem szatana, którego zaakceptowanie prowadzi do sprzeczności).

```
 \begin{split}  & \acute{\mathbf{C}}\mathbf{wiczenie} \quad \mathit{Fail} \; \mathsf{Inductive} \; \mathit{T2} : \; \mathsf{Type} := \\ & \mid \mathit{T2\_0} : \forall \; f : \; (\forall \; g : \; (\forall \; t : \; \mathit{T2}, \; \mathit{nat}), \; \mathsf{Prop}), \; \mathit{T2} \\ & \mid \mathit{T2\_1} : \; (((((\mathit{T2} \rightarrow \mathit{T2}) \rightarrow \mathit{T2} \\ & \mid \mathit{T2\_2} : \\ & \quad ((\forall \; (\mathit{n} : \; \mathit{nat}) \; (\mathit{P} : \; \mathit{T2} \rightarrow \mathsf{Prop}), \\ & \quad (\forall \; t : \; \mathit{T2}, \; \mathit{nat})) \rightarrow \mathit{T2}) \rightarrow \mathit{T2} \rightarrow \mathit{T2} \rightarrow \mathit{T2}. \end{split}
```

Policz niedobrość każdego wstąpienia T2 w powyższej definicji. Sklasyfikuj konstruktory jako negatywne, pozytywne lub ściśle pozytywne. Następnie sklasyfikuj sam typ jako negatywny, pozytywny lub ściśle pozytywny.

Ćwiczenie (* Inductive T : Type := *)

Rozstrzygnij, czy następujące konstruktory spełniają kryterium ścisłej pozytywności. Jeżeli tak, narysuj wesołego jeża. Jeżeli nie, napisz zapętlającą się funkcję podobną do loop (zakładamy, że typ T ma tylko ten jeden konstruktor). Następnie sprawdź w Coqu, czy udzieliłeś poprawnej odpowiedzi.

- $\bullet \mid C1:T$
- $\bullet \mid C2 : bool \rightarrow T$
- $\bullet \mid C3: T \rightarrow T$
- \bullet | C4 : $T \rightarrow nat \rightarrow T$
- | $C5: \forall A: Type, T \rightarrow A \rightarrow T$
- | $C6: \forall A: Type, A \rightarrow T \rightarrow T$
- | $C7: \forall A: Type, (A \rightarrow T) \rightarrow T$
- | $C8 : \forall A : Type, (T \rightarrow A) \rightarrow T$
- $\bullet \mid C9 : (\forall x : T, T) \rightarrow T$
- | $C10: (\forall (A: \mathsf{Type}) (x:T), A) \to T$
- | $C11: \forall A B C: Type, A \rightarrow (\forall x: T, B) \rightarrow (C \rightarrow T) \rightarrow T$

4.5.5 Kilka bonusowych pułapek

Wiemy już, że niektóre typy argumentów indukcyjnych są ok $(T \times T, list\ T)$, a niektóre inne nie $(T \to T, \forall\ t:\ T, ...)$. Uważny i żądny wiedzy czytelnik (daj boże, żeby tacy istnieli) zeche zapewne postawić sobie pytanie: które dokładnie typy argumentów indukcyjnych są ok, a które są wynalazkiem szatana?

Najprościej będzie sprawę zbadać empirycznie, czyli na przykładzie. Żeby zaś przykład był reprezentatywny, niech parametrem definicji będzie dowolna funkcja $F: Type \rightarrow Type$.

```
Fail Inductive wut\ (F: {\tt Type} \to {\tt Type}): {\tt Type} := |\ wut\_\theta: F\ (wut\ F) \to wut\ F. (* ===> The command has indeed failed with message: Non strictly positive occurrence of "wut" in "F (wut F) -> wut F". *)
```

Jak widać, jeżeli zaaplikujemy F do argumentu indukcyjnego, to Coq krzyczy, że to wystąpienie nie jest ściśle pozytywne. Dlaczego tak jest, skoro F nie jest ani strzałką, ani kwantyfikatorem uniwersalnym? Dlatego, że choć nie jest nimi, to może nimi zostać. Jeżeli zdefiniujemy sobie gdzieś na boku $F := \operatorname{fun} A : \operatorname{Type} \Rightarrow A \to bool$, to wtedy $wut_0 F := (wut_0 F \to bool) \to wut_0 F$, a z takim diabelstwem już się mierzyliśmy i wiemy, że nie wróży ono niczego dobrego.

Morał z tej historii jest dość banalny: gdy definiujemy typ induktywny T, jedynymi prawilnymi typami dla argumentu indukcyjnego są T oraz typy funkcji, które mają T jako konkluzję $(A \to T, A \to B \to T \text{ etc.})$. Wszystkie inne albo rodzą problemy z automatyczną generacją reguł indukcji $(T \times T, list\ T)$, albo prowadzą do sprzeczności $(T \to T, \forall t: T, ...)$, albo mogą prowadzić do sprzeczności, jeżeli wstawi się za nie coś niedobrego (T, T).

 $\hat{\mathbf{C}}$ wiczenie Module wutF.

```
Definition F(A: \mathsf{Type}): \mathsf{Type} := A \to bool.
```

Zakoduj aksjomatycznie rodzinę typów wut z powyższego przykładu. Następnie wstaw za parametr zdefiniowane powyżej F i udowodnij, że typ wut F prowadzi do sprzeczności.

Lemma $wut_illegal$: False.

End wutF.

To jeszcze nie koniec wrażeń na dziś - póki co omówiliśmy wszakże kryterium ścisłej pozytywności jedynie dla bardzo prostych typów induktywnych. Słowem nie zająknęliśmy się nawet na temat typów wzajemnie induktywnych czy indeksowanych typów induktywnych. Nie trudno będzie nam jednak uzupełnić naszą wiedzę, gdyż w przypadku oby tych mechanizmów kryterium ścisłej pozytywności wygląda podobnie jak w znanych nam już przypadkach.

```
Fail \ \mathtt{Inductive} \ X : \mathtt{Type} := \ \mid X\theta : X
```

```
\mid X1: (Y \to X) \to X with Y: \text{Type} := \mid Y0: Y \\ \mid Y1: X \to Y. (* ===> The command has indeed failed with message: Non strictly positive occurrence of "Y" in "(Y -> X) -> X". *)
```

Jak widać, definicja X i Y przez wzajemną indukcję jest nielegalna, gdyż jedyny argument konstruktora X1 ma typ $Y \to X$. Mogłoby wydawać się, że wszystko jest w porządku, wszakże X występuje tutaj na pozycji ściśle pozytywnej. Jednak ponieważ jest to definicja przez indukcję wzajemną, kryterium ścisłej pozytywności stosuje się nie tylko do wystąpień X, ale także do wystąpień Y - wszystkie wystąpienia X oraz Y muszą być ściśle pozytywne zarówno w konstruktorach typu X, jak i w konstruktorach typu Y.

Čwiczenie Zakoduj aksjomatycznie definicję typów X i Y. Spróbuj napisać zapętlającą się funkcję loop (czy raczej dwie wzajemnie rekurencyjne zapętlające się funkcje loopx i loopy), a następnie udowodnij za pomocą twierdzenia Cantora, że typy X i Y są nielegalne.

Module XY.

4.5.6 Jeszcze więcej pułapek

To już prawie koniec naszej wędrówki przez świat nielegalnych typów "induktywnych". Dowiedzieliśmy się, że negatywne typy induktywne prowadzą do nieterminacji i nauczyliśmy się wykorzystywać twierdzenie Cantora do dowodzenia nielegalności takich typów.

Poznaliśmy też jednak klasyfikację typów wyglądających na induktywne (ściśle pozytywne, pozytywne, negatywne), a w szczególności pojęcie "niedobrości" indukcyjnego wystąpienia definiowanego typu w konstruktorze (upraszczając, na lewo od ilu strzałek znajduje się to wystąpienie).

Piszę "jednak", gdyż z jej powodu możemy czuć pewien niedosyt - wszystkie dotychczasowe przykłady były typami negatywnymi o niedobrości równej 1. Podczas naszej intelektualnej wędrówki zwiedziliśmy mniej miejscówek, niż moglibyśmy chcieć. W tym podrozdziale spróbujemy ten przykry niedosyt załatać, rozważając (nie ściśle) pozytywne typy induktywne. Zobaczymy formalny dowód na to, że nie są one legalne (lub, precyzyjniej pisząc, dowód na to, że conajmniej jeden z nich nie jest legalny). Zanim jednak to się stanie, zobaczmy, czy wypracowane przez nas techniki działają na negatywne typy induktywne o niedobrości innej niż 1.

Module T3.

```
Fail Inductive T3: Type := \mid T3\_0 : (((T3 \rightarrow bool) \rightarrow bool) \rightarrow bool) \rightarrow T3.
```

Przyjrzyjmy się powyższej definicji. Występienie indukcyjne typu T3 ma współczynnik niedobrości równy 3, gdyż znajduje się na lewo od 3 strzałek. Prawe strony wszystkich z nich to bool.

Axioms

```
 \begin{array}{l} (T3: {\tt Type}) \\ (T3\_\theta: (((T3 \to bool) \to bool) \to bool) \to T3) \\ (T3\_case: \\ \forall \\ (P: T3 \to {\tt Type}) \\ (PT3\_\theta: \forall f: ((T3 \to bool) \to bool) \to bool, P \ (T3\_\theta \ f)), \\ \{f: \forall x: T3, P \ x \mid \\ \forall \ g: ((T3 \to bool) \to bool) \to bool, \\ f \ (T3\_\theta \ g) = PT3\_\theta \ g\}). \end{array}
```

Po ciężkich bojach, przez które przeszedłeś, aksjomatyczne kodowanie tego typu nie powinno cię dziwić. Warto zauważyć jedynie, że do naszej dyspozycji mamy jedynie regułę zależnej analizy przypadków, gdyż nie wiadomo, jak miałyby wyglądać wywołania indukcyjne.

Zanim zobaczymy, jak pokazać nielegalność tego typu metodą Cantora, przypomnijmy sobie pewien kluczowy fakt dotyczący negacji i jego banalne uogólnienie.

```
Ćwiczenie Lemma triple_negation:
```

```
\begin{array}{l} \forall~P: \texttt{Prop}, \, \neg \, \neg \, \neg \, P \to \neg \, P. \\ \\ \texttt{Lemma}~triple\_arrow: \\ \forall~A~B: \texttt{Type}, \, (((A \to B) \to B) \to B) \to (A \to B). \end{array}
```

Ćwiczenie to przypomina nam, że jeżeli chodzi o spamowanie negacją, to są w zasadzie tylko trzy sytuacje:

- brak negacji
- pojedyncza negacja
- podwójna negacja

Jeżeli mamy do czynienia z większą liczbą negacji, to możemy zdejmować po dwie aż dojdziemy do któregoś z powyższych przypadków. Ponieważ negacja to tylko implikacja, której kodziedziną jest False, a nie korzystamy w dowodzie z żadnych specjalnych właściwości False, analogiczna właściwość zachodzi także dla dowolnego innego B: Type.

```
Definition bad: T3 \rightarrow (T3 \rightarrow bool). Proof.
```

```
apply (T3\_case \ (\text{fun } \_ \Rightarrow T3 \to bool)). intros f x. apply f. intro g. apply g. exact x. Defined.
```

Wobec powyższych rozważań definicja funkcji bad zupełnie nie powinna cię zaskakiwać. Szczerze pisząc, reszta dowodu również nie jest jakoś specjalnie wymagająca czy oświecająca.

Ćwiczenie Dokończ dowód.

Theorem $T3_illegal$: False.

Ćwiczenie Napisanie zapętlającej się funkcji $loop: T3 \rightarrow bool$ też nie jest jakoś wybitnie trudne. Napisz ją i udowodnij (nieformlanie), że istnieje takie x: T3, że loop x się zapętla.

End T3.

Morał z powyższych rozważań jest prosty: nasze techniki działają także na negatywne typy induktywne o niedobrości równej 3. Myślę, że jesteś całkiem skłonny uwierzyć też, że zadziałają na te o niedobrości równej 5, 7 i tak dalej.

To wszystko jest prawdą jednak tylko wtedy, gdy wszystkie typy po prawych stronach strzałek będą takie same. A co, gdy będą różne?

Module T_4 .

```
 \begin{array}{l} Fail \  \, \text{Inductive} \  \, T \not : \  \, \text{Type} := \\ \quad \mid c\theta : (((T \not 4 \rightarrow bool) \rightarrow nat) \rightarrow Color) \rightarrow T \not 4. \\ \\ \text{Axioms} \\ (T \not 4 : \text{Type}) \\ (c\theta : (((T \not 4 \rightarrow bool) \rightarrow nat) \rightarrow Color) \rightarrow T \not 4) \\ (dcase : \\ \forall \\ \quad (P: T \not 4 \rightarrow \text{Type}) \\ (Pc\theta : \forall f : ((T \not 4 \rightarrow bool) \rightarrow nat) \rightarrow Color, \ P \ (c\theta \ f)), \\ \{f : \forall x : T \not 4, \ P \ x \mid \\ \quad \forall \ g : ((T \not 4 \rightarrow bool) \rightarrow nat) \rightarrow Color, \\ \quad f \ (c\theta \ g) = Pc\theta \ g \}). \end{array}
```

Powyższy przykład jest podobny do poprzedniego, ale tym razem zamiast trzech wystąpień bool mamy bool, nat oraz Color (to typ, który zdefiniowaliśmy na samym początku tego rozdziału, gdy uczyliśmy się o enumeracjach).

```
Definition bad: T4 \rightarrow (T4 \rightarrow bool). Proof. apply (dcase \ (fun \ \_ \Rightarrow T4 \rightarrow \_)). intros f \ x.
```

```
\begin{array}{l} \operatorname{apply} \; (\\ \quad \text{fun} \; c : Color \Rightarrow \\ \quad \operatorname{match} \; c \; \operatorname{with} \\ \quad \mid R \Rightarrow true \\ \quad \mid \_ \Rightarrow false \\ \quad \operatorname{end}). \\ \quad \operatorname{apply} \; f. \; \operatorname{intro} \; g. \\ \quad \operatorname{apply} \; (\operatorname{fun} \; b : bool \Rightarrow \operatorname{if} \; b \; \operatorname{then} \; 0 \; \operatorname{else} \; 1). \\ \quad \operatorname{exact} \; (g \; x). \\ \quad \operatorname{Defined}. \end{array}
```

Nasz modus operandi będzie taki jak poprzednio: spróbujemy wyjąć z elementu T4 funkcję typu $T4 \to bool$. W tym celu używamy zależnej reguły analizy przypadków i wprowadzamy rzeczy do kontekstu.

Tym razem nie możemy jednak bezpośrednio zaaplikować f, gdyż jej kodziedziną jest Color, a my musimy skonstruować coś typu bool. Możemy temu jednak zaradzić aplikując do celu skonstruowaną naprędce funkcję typu $Color \rightarrow bool$. Ta funkcja powinna być surjekcją (jeśli nie wierzysz, sprawdź, co się stanie, jeżeli zamienimy ją na funckję stałą).

Możemy już zaaplikować f i wprowadzić g do kontekstu. Chcielibyśmy teraz zaaplikować g, ale nie możemy, bo typy się nie zgadzają - g zwraca bool, a my musimy skonstruować liczbę naturalną. Robimy tutaj to samo co poprzednio - aplikujemy do celu jakąś funkcję $bool \rightarrow nat$. Tym razem nie musi ona być surjekcją (nie jest to nawet możliwe, gdyż nie ma surjekcji z $bool \le nat$). Dzięki temu możemy zaaplikować g i zakończyć, używając g x.

Require Import Functional Extensionality.

Żeby pokazać, że bad jest surjekcją, będziemy potrzebować aksjomatu ekstensjonalności dla funkcji (ang. functional extensionality axiom, w skrócie funext). Głosi on, że dwie funkcje $f, g: A \to B$ są równe, jeżeli uda nam się pokazać, że dają równe wyniki dla każdego argumentu (czyli $\forall x: A, f x = g x$).

Importując powyższy moduł zakładamy prawdziwość tego aksjomatu oraz uzyskujemy dostęp do taktyki extensionality, która ułatwia dowody wymagające użycia ekstensjonalności.

```
Lemma bad\_sur: surjective\ bad. Proof. unfold surjective. intro f. unfold bad. destruct (dcase\ \_) as [bad\ eq]. \exists\ (c\theta\ ( fun g:(T4\rightarrow bool)\rightarrow nat\Rightarrow match g\ f with |\ 0\Rightarrow R |\ \_\Rightarrow G end)).
```

```
rewrite eq. extensionality t. destruct (f\ t); reflexivity. Qed.
```

Dowód jest prawie taki jak zawsze: odwijamy definicję surjektywności i wprowadzamy hipotezy do kontekstu, a następnie odwijamy definicję bad i rozbijamy ją dla czytelności na właściwą funkcję bad oraz równanie eq.

Następnie musimy znaleźć a:T4, które bad mapuje na f. Zaczynamy od $c\theta$, bo jest to jedyny konstruktor T4. Bierze on jako argument funkcję typu $((T4 \rightarrow bool) \rightarrow nat) \rightarrow Color$. Żeby ją wyprodukować, bierzemy na wejściu funkcję $g:(T4 \rightarrow bool) \rightarrow nat$ i musimy zrobić coś typu Color.

Nie może to być jednak byle co - musimy użyć f, a jedynym sensownym sposobem, żeby to zrobić, jest zaaplikować g do f. Musimy zadbać też o to, żeby odwrócić funkcje konwertujące $Color \to bool$ oraz $bool \to nat$, których użyliśmy w definicji bad. Pierwsza z nich konwertowała R (czyli kolor czerwony) na true, a inne kolory na false, zaś druga konwertowała true na 0, a false na 1. Wobec tego dopasowując g f: nat musimy przekonwertować 0 na R, zaś 1 na coś innego niż R, np. na G (czyli kolor zielony).

Znalaziszy odpowiedni argument, możemy przepisać równanie definiujące bad. To już prawie koniec, ale próba użycia taktyki **reflexivity** w tym momencie skończyłaby się porażką. Na ratunek przychodzi nam aksjomat ekstensjonalności, którego używamy pisząc **extensionality** t. Dzięki temu pozostaje nam pokazać jedynie, że f t jest równe tej drugie funkcji dla argumentu t. W tym celu rozbijamy f t, a oba wyrażenia okazują się być konwertowalne.

```
Theorem T4\_illegal: False. Proof.

apply (Cantor'\ bad\ negb).

destruct b; inversion 1.

apply bad\_sur.

Qed.
```

Skoro mamy surjekcję z T4 w $T4 \rightarrow bool$, katastrofy nie da się uniknąć.

Moglibyśmy się też zastanowić nad napisaniem zapętlającej się funkcji *loop*, ale coś czuję, że ty coś czujesz, że byłoby to babranie się w niepotrzebnym problemie. Wobec tego (oczywiście o ile dotychczas się nie skapnąłeś) poczuj się oświecony!

```
Definition loop (x : T4) : bool := bad x x.
```

Ha! Tak tak, loop nie jest niczym innym niż lekko rozmnożoną wersją bad.

Lemma $loop_nontermination$:

```
true = loop\ (c0\ ( fun g: (T4 \rightarrow bool) \rightarrow nat \Rightarrow match g\ loop\ with |\ 0 \Rightarrow R |\ _- \Rightarrow G end)).
```

```
Proof. unfold loop,\ bad. destruct (dcase\ \_) as [bad\ eq]. rewrite 5!eq. Abort.
```

A skoro loop to tylko inne bad, to nie powinno cię też wcale a wcale zdziwić, że najbardziej oczywisty argument, dla którego loop się zapętla, jest żywcem wzięty z dowodu bad_sur (choć oczywiście musimy zastąpić f przez loop).

Oczywiście niemożliwe jest, żeby formalnie udowodnić w Coqu, że coś się zapętla. Powyższy lemat ma być jedynie demonstracją - ręczne rozpisanie tego przykładu byłoby zbyt karkołomne. Jak widać z dowodu, przepisywanie równania definiującego bad tworzy wesołą piramidkę zrobioną z matchy i ifów. Jeżeli chcesz poczuć pełnię zapętlenia, wypbróuj taktykę rewrite !eq - zapętli się ona, gdyż równanie eq można przepisywać w nieskończoność. End T4.

Mogłoby się wydawać, że teraz to już na pewno nasze metody działają na wszystkie możliwe negatywne typy induktywne. Cytując Tadeusza Sznuka: "Nic bardziej mylnego!".

Module T5.

```
Axioms
```

```
 \begin{array}{l} (T5: \mathtt{Type}) \\ (c\theta: (((T5 \rightarrow nat) \rightarrow bool) \rightarrow Color) \rightarrow T5) \\ (dcase: \\ \forall \\ (P: T5 \rightarrow \mathtt{Type}) \\ (Pc\theta: \forall f: ((T5 \rightarrow nat) \rightarrow bool) \rightarrow Color, \ P\ (c\theta\ f)), \\ \{f: \forall x: T5, P\ x \mid \\ \forall\ g: ((T5 \rightarrow nat) \rightarrow bool) \rightarrow Color, \\ f\ (c\theta\ g) = Pc\theta\ g\}). \end{array}
```

Rzućmy okiem na powyższy typ. Wygląda podobnie do poprzedniego, ale jest nieco inny - typy nat i bool zamieniły się miejscami. Jakie rodzi to konsekwencje? Sprawdźmy.

```
Definition bad: T5 \rightarrow (T5 \rightarrow nat). Proof.

apply (dcase \ (fun \ \_ \Rightarrow T5 \rightarrow \_)). intros f \ x.

apply (fun \ c : Color \Rightarrow fun \ c : Col
```

Defined.

Definicja bad jest podobna jak poprzednio, ale tym razem konwertujemy Color na nat za pomocą funkcji, która nie jest surjekcją.

```
{\tt Require\ Import\ } \textit{FunctionalExtensionality}.
```

```
Lemma bad\_sur: surjective\ bad.

Proof.

unfold surjective. intro f.

unfold bad. destruct (dcase\ \_) as [bad\ eq].

\exists\ (c\theta\ (fun\ g:(T5\to nat)\to bool\Rightarrow match\ g\ f\ with | true\Rightarrow R | false\Rightarrow B | end)).

rewrite eq. extensionality t. destruct (f\ t); cbn.

reflexivity.

Abort.
```

Dowód również przebiega podobnie jak poprzednio. Załamuje się on dopiero, gdy na samym końcu rozbijamy wyrażenie f t i upraszczamy używając cbn. W pierwszym podcelu 0=0 jeszcze jakoś udaje się nam udowodnić, ale w drugim naszym oczom ukazuje się cel 2=S n.

Problem polega na tym, że f t może być dowolną liczbą naturalną, ale zastosowana przez nas funkcja konwertująca $Color \rightarrow nat$ może zwracać jedynie 0, 1 lub 2. Teraz widzimy jak na dłoni, skąd wziął się wymóg, by funkcja konwertująca była surjekcją.

```
Definition loop\ (x:T5):nat:=bad\ x\ x. Lemma loop\_nontermination: 42 = loop\ (c0\ (fun\ g:(T5\to nat)\to bool\Rightarrow match\ g\ loop\ with |\ true\Rightarrow R\ |\ false\Rightarrow G\ end)). Proof. unfold loop,\ bad.\ destruct\ (dcase\ \_)\ as\ [bad\ eq]. rewrite 15!eq. Abort.
```

Co ciekawe, mimo że nie jesteśmy w stanie pokazać surjektywności bad, to wciąż możemy użyć tej funkcji do zdefiniowania zapętlającej się funkcji loop, zupełnie jak w poprzednim przykładzie.

Niesmak jednak pozostaje, gdyż szczytem naszych ambicji nie powinno być ograniczanie się do zdefiniowania loop, lecz do formalnego udowodnienia nielegalności T5. Czy wszystko stracone? Czy umrzemy? Tu dramatyczna pauza.

Nie.

Okazuje się, że jest pewien trikowy sposób na rozwiązanie tego problemu, a mianowicie: zamiast próbować wyjąć z T5 funkcję $T5 \rightarrow nat$, wyjmiemy stamtąd po prostu funckję $T5 \rightarrow bool$ i to mimo tego, że jej tam nie ma!

```
Definition bad': T5 \rightarrow (T5 \rightarrow bool). Proof.

apply (dcase \ (fun \ \_ \Rightarrow T5 \rightarrow \_)).

intros f x.

apply (fun \ c : Color \Rightarrow fun \ c : Color \Rightarrow fun \ c : Color \Rightarrow false

end).

apply f intro f f intro f f apply f intro f f apply f intro f f apply f intro f apply f apply f intro f apply f
```

W kluczowych momentach najpierw konwertujemy *Color* na *bool* tak jak w jednym z poprzednich przykładów, a potem konwertujemy *nat* na *bool* za pomocą funkcji *isZero*.

Require Import Functional Extensionality.

```
Lemma bad'_sur: surjective\ bad'. Proof. unfold surjective. intro f. unfold bad'. destruct (dcase\ \_) as [bad'\ eq]. \exists\ (c\theta\ (fun\ g:(T5\to nat)\to bool\Rightarrow if\ g\ (fun\ t:T5\Rightarrow if\ f\ t\ then\ 0\ else\ 1)\ then\ R\ else\ G)). rewrite eq. extensionality t. destruct (f\ t); cbn; reflexivity. Qed.
```

Ponieważ obydwie nasze funkcję konwertujące były surjekcjami, możemy je teraz odwrócić i wykazać ponad wszelką wątpliwość, że bad' faktycznie jest surjekcją.

```
Theorem T5\_illegal : False.

Proof.

apply (Cantor'\ bad'\ negb).

destruct b; inversion 1.
```

```
\label{eq:apply_bad'sur} \texttt{Qed}. 
 Qed.
```

Spróbujmy podsumować, co tak naprawdę stało się w tym przykładzie.

Tym razem, mimo że do T5 możemy włożyć dowolną funkcję $T5 \rightarrow nat$, to nie możemy jej potem wyjąć, uzyskując surjekcję, gdyż zawadzają nam w tym typy po prawych stronach strzałek (bool i Color), które mają za mało elementów, żeby móc surjektywnie przekonwertować je na typ nat.

Jednak jeżeli mamy wszystkie możliwe funkcje typu $T5 \rightarrow nat$, to możemy przerobić je (w locie, podczas "wyciągania") na wszystkie możliwe funkcje typu $T5 \rightarrow bool$, składając je z odpowiednią surjekcją (np. isZero). Ponieważ typ bool i Color jesteśmy w stanie surjektywnie przekonwertować na bool, reszta procesu działa podobnie jak w poprzednich przykładach.

```
Definition loop' (x:T5):bool:=bad' x x.

Lemma loop\_nontermination:
true = loop' (c0) (
fun g: (T5 \to nat) \to bool \Rightarrow
match g (fun t:T5 \Rightarrow if loop' t then 0 else 1) with
| true \Rightarrow R
| false \Rightarrow G
end)).

Proof.
unfold loop', bad'. destruct (dcase\_) as [bad' eq].
rewrite 15!eq.
Abort.
```

Takie trikowe bad' wciąż pozwala nam bez większych przeszkód zdefiniować zapętlającą się funkcję loop'. Osiągnęliśmy więc pełen sukces.

W ogólności nasz trik możnaby sformułować tak: jeżeli mamy konstruktor negatywny typu T, to możemy wyjąć z niego funkcję $T \to A$, gdzie A jest najmniejszym z typów występujących po prawych stronach strzałek.

No, teraz to już na pewno mamy obcykane wszystkie przypadki, prawda? Tadeuszu Sznuku przybywaj: "Otóż nie tym razem!".

```
End T5.
```

Module T6.

```
Axioms
```

```
 \begin{array}{l} (\mathit{T6}: \mathsf{Type}) \\ (\mathit{c0}: (((\mathit{T6} \rightarrow \mathit{unit}) \rightarrow \mathit{bool}) \rightarrow \mathit{Color}) \rightarrow \mathit{T6}) \\ (\mathit{dcase}: \\ \forall \\ (\mathit{P}: \mathit{T6} \rightarrow \mathsf{Type}) \\ (\mathit{Pc0}: \forall \mathit{f}: ((\mathit{T6} \rightarrow \mathit{unit}) \rightarrow \mathit{bool}) \rightarrow \mathit{Color}, \mathit{P} \; (\mathit{c0} \; \mathit{f})), \\ \{\mathit{f}: \forall \mathit{x}: \mathit{T6}, \mathit{P} \; \mathit{x} \; | \end{array} \right.
```

$$\forall g: ((T6 \rightarrow unit) \rightarrow bool) \rightarrow Color, f(c0 g) = Pc0 g\}).$$

Kolejnym upierdliwym przypadkiem, burzącym nawet nasz ostateczny trik, jest sytuacja, w której po prawej stronie strzałki wystąpi typ unit. Oczywiście zgodnie z trikiem możemy z T6 wyciągnąć surjekcję $T6 \rightarrow unit$, ale jest ona oczywiście bezużyteczna, bo taką samą możemy zrobić za darmo, stale zwracając po prostu tt. Surjekcja ta nie wystarcza rzecz jasna, żeby odpalić twierdzenie Cantora.

Tym razem jednak nie powinniśmy spodziewać się, że upierdliwość tę będzie dało się jakoś obejść. Typ $T6 \rightarrow unit$ jest jednoelementowy (jedynym elementem jest fun $_\Rightarrow tt$) podobnie jak unit. Bardziej poetycko możemy powiedzieć, że $T6 \rightarrow unit$ i unit są izomorficzne, czyli prawie równe - różnią się tylko nazwami elementów ("nazwa" jedynego elementu unita to tt).

Skoro tak, to typ konstruktora $c\theta$, czyli $(((T6 \rightarrow unit) \rightarrow bool) \rightarrow Color) \rightarrow T6)$, możemy równie dobrze zapisać jako $((unit \rightarrow bool) \rightarrow Color) \rightarrow T6)$. Zauważmy teraz, że $unit \rightarrow bool$ jest izomorficzne z bool, gdyż ma tylko dwa elementy, a mianowicie $fun = \Rightarrow true$ oraz $fun = \Rightarrow false$. Tak więc typ $c\theta$ możemy jeszcze prościej zapisać jako $(bool \rightarrow Color) \rightarrow T6$, a to oznacza, że typ T6 jest jedynie owijką na funkcje typu $bool \rightarrow Color$. Twierdzenie Cantora nie pozwala tutaj uzyskać sprzeczności.

Czy zatem takie typy sa legalne? Syntaktycznie nie - Coq odrzuca je podobnie jak wszystkie inne negatywne typy induktywne. Semantycznie również nie - o ile nie możemy uzyskać jawnej sprzeczności, to nasze rozważania o nieterminacji wciąż są w mocy.

Przypomnij sobie poprzedni przykład i nieudaną próbę wyłuskania z T5 surjekcji $T5 \rightarrow nat$. Udało nam się zaimplementować funkcję bad, której surjektywności nie potrafiliśmy pokazać, ale pomimo tego bez problemu udało nam się użyć jej do napisania funkcji loop. W obecnym przykładzie jest podobnie i nieterminacja to najlepsze, na co możemy liczyć.

Ćwiczenie Zdefiniuj funkcję *bad*, a następnie użyj jej do zdefiniowania funkcji *loop*. Zademonstruj w sposób podobny jak poprzednio, że *loop* się zapętla.

End T6.

No, teraz to już na pewno wiemy wszystko...

Ćwiczenie Otóż nie do końca. Ostatnim hamulcowym, groźniejszym nawet niż *unit*, jest wystąpienie po prawej stronie strzałki typu (czy raczej zdania) False. W tym przypadku nie tylko nie pomaga nam Cantor, ale nie pomaga też nieterminacja, gdyż najzwyczajniej w świecie nie da się zdefiniować żadnej funkcji.

Jako, że za cholerę nie wiem, co z tym fantem zrobić, zostawiam go tobie jako ćwiczenie: wymyśl metodę pokazywania nielegalności negatywnych typów induktywnych, w których po prawej stronie strzałki jest co najmniej jedno wystapienie False.

Module T8.

Axioms

```
 \begin{array}{l} (T8: \mathtt{Type}) \\ (c0: (((T8 \rightarrow bool) \rightarrow False) \rightarrow Color) \rightarrow T8) \\ (dcase: \\ \forall \\ (P: T8 \rightarrow \mathtt{Type}) \\ (Pc0: \forall f: ((T8 \rightarrow bool) \rightarrow False) \rightarrow Color, P \ (c0 \ f)), \\ \{f: \forall x: T8, P \ x \mid \\ \forall g: ((T8 \rightarrow bool) \rightarrow False) \rightarrow Color, \\ f \ (c0 \ g) = Pc0 \ g\}). \end{array}
```

End T8.

4.5.7 Promocja 2 w 1 czyli paradoksy Russella i Girarda

Istnieje teoria, że jeśli kiedyś ktoś się dowie, dlaczego powstało i czemu służy uniwersum, to zniknie ono i zostanie zastąpione czymś znacznie dziwaczniejszym i jeszcze bardziej pozbawionym sensu.

Istnieje także teoria, że dawno już tak się stało.

Douglas Adams, Restauracja na końcu wszechświata

W poprzednich podrozdziałach poznaliśmy twierdzenie Cantora oraz nauczyliśmy się używać go jako młota na negatywne typy induktywne.

W tym podrozdziale zapoznamy się z dwoma paradoksami (a precyzyjniej pisząc, z dwoma wersjami tego samego paradoksu), które okażą się być ściśle powiązane z twierdzeniem Cantora, a które będą służyć nam gdy staniemy w szranki z negatwynymi typami induktywno-rekurencyjnymi (czyli tymi, które definiuje się przez indukcję-rekursję). O tak: w tym podrozdziale, niczym Thanos, staniemy do walki przeciw uniwersum!

Zacznijmy od paradoksu Russella. Jest to bardzo stary paradoks, odkryty w roku 1901 przez... zgadnij kogo... gdy ów człek szukał dziury w całym w naiwnej teorii zbiorów (która to teoria jest już od dawna martwa).

Sformułowanie paradoksu brzmi następująco: niech V będzie zbiorem wszystkich zbiorów, które nie należą same do siebie. Pytanie: czy V należy do V?

Gdzie tu paradoks? Otóż jeżeli V należy do V, to na mocy definicji V, V nie należy do V. Jeżeli zaś V nie należy do V, to na mocy definicji V, V należy do V. Nie trzeba chyba dodawać, że jednoczesne należenie i nienależenie prowadzi do sprzeczności.

Cwiczenie To genialne ćwiczenie wymyśliłem dzięki zabłądzeniu na esperanckiej Wikipedii (ha! nikt nie spodziewał się esperanckiej Wikipedii w ćwiczeniu dotyczącym paradoksu Russella). Ćwiczenie brzmi tak:

W Wikipedii niektóre artykuły są listami (nie, nie w sensie typu induktywnego :)), np. lista krajów według PKB per capita. Pytanie: czy można stworzyć w Wikipedii listę wszystkich list? Czy na liście wszystkich list ona sama jest wymieniona? Czy można w Wikipedii stworzyć listę wszystkich list, które nie wymieniają same siebie?

Na czym tak naprawdę polega paradoks? Jakiś mądry (czyli przemądrzały) filozof mógłby rzec, że na nadużyciu pojęcia zbioru... albo czymś równie absurdalnym. Otóż nie! Paradoks Russella polega na tym samym, co cała masa innych paradoksów, czyli na autoreferencji.

Z autoreferencją spotkaliśmy się już co najmniej raz, w rozdziale pierwszym. Przypomnij sobie, że golibroda goli tych i tylko tych, którzy sami siebie nie golą. Czy golibroda goli sam siebie?

Takie postawienie sprawy daje paradoks. Podobnie z Russellem: V zawiera te i tylko te zbiory, które nie zawierają same siebie. Czy V zawiera V? Wot, paradoks. Żeby lepiej wczuć się w ten klimat, czas na więcej ćwiczeń.

Ćwiczenie A jak jest z poniższym paradoksem wujka Janusza?

Wujek Janusz lubi tych (i tylko tych) członków rodziny, którzy sami siebie nie lubią oraz nie lubi tych (i tylko tych), którzy sami siebie lubią. Czy wujek Janusz lubi sam siebie?

Ćwiczenie Powyższe ćwiczenie miało być ostatnim, ale co tam, dam jeszcze trochę. Co czuje serce twoje (ewentualnie: co widzisz przed oczyma duszy swojej) na widok poniższych wesołych zdań?

"To zdanie jest fałszywe."

"Zdanie po prawej jest fałszywe. Zdanie po lewej jest prawdziwe."

"Zdanie po prawej jest prawdziwe. Zdanie po lewej jest fałszywe."

Dobra, wystarczy już tych paradoksów... a nie, czekaj. Przecież został nam do omówienia jeszcze paradoks Girarda. Jednak poznawszy już tajniki autoreferencji, powinno pójść jak z płatka.

Paradoks Girarda to paradoks, który może zaistnieć w wielu systemach formalnych, takich jak teorie typów, języki programowania, logiki i inne takie. Źródłem całego zła jest zazwyczaj stwierdzenie w stylu Type: Type.

Check Type.

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

O nie! Czyżbyśmy właśnie zostali zaatakowani przez paradoks Girarda? W tym miejscu należy przypomnieć (albo obwieścić - niestety nie pamiętam, czy już o tym wspominałem), że Type jest w Coqu jedynie synonimem dla czegoś w stylu Type(i), gdzie i jest "poziomem" sortu Type, zaś każde Type(i) żyje tak naprawdę w jakimś Type(j), gdzie j jest większe od i - typy niższego poziomu żyją w typach wyższego poziomu. Będziesz mógł ów fakt ujrzeć na własne oczy, gdy w CoqIDE zaznaczysz opcję View > Display universe levels.

Check Type.

```
(* ===> Type0{Top.590} : Type0{Top.590+1} *)
```

Jak widać, jest mniej więcej tak jak napisałem wyżej. Nie przejmuj się tym tajemniczym *Top* - to tylko nie nieznaczący bibelocik. W twoim przypadku również poziom uniwersum może być inny niż 590. Co więcej, poziom ten będzie się zwiększał wraz z każdym odpaleniem komendy Check Type (czyżbyś pomyślał właśnie o doliczeniu w ten sposób do zyliona?).

Skoro już wiemy, że NIE zostaliśmy zaatakowani przez paradoks Girarda, to w czym problem z tym całym Type: Type? Jakiś przemądrzały (czyli mądry) adept informatyki teoretycznej mógłby odpowiedzieć, że to zależy od konkretnego systemu formalnego albo coś w tym stylu. Otóż niet! Jak zawsze, chodzi oczywiście o autoreferencję.

Gdyby ktoś był zainteresowany, to najlepsze dotychczas sformułowanie paradoksu znalazłem (zupełnie przez przypadek, wcale nie szukając) w pracy "An intuitionistic theory of types" Martina-Löfa (swoją drogą, ten koleś wymyślił podstawy dużej części wszystkiego, czym się tutaj zajmujemy). Można ją przeczytać tu (paradoks Girarda jest pod koniec pierwszej sekcji): archive-pml.github.io/martin-lof/pdfs /An-Intuitionistic-Theory-of-Types-1972.pdf

Nasze sformułowanie paradoksu będzie w sumie podobne do tego z powyższej pracy (co jest w sumie ciekawe, bo wymyśliłem je samodzielnie i to przez przypadek), ale dowód sprzeczności będzie inny - na szczęście dużo prostszy (albo i nie...).

Dobra, koniec tego ględzenia. Czas na konkrety.

Powyższa induktywno-rekurencyjna definicja typu U (i interpretującej go funkcji El), którą Coq rzecz jasna odrzuca (uczcijmy ławę oburzonych minutą oburzenia) to definicja pewnego uniwersum.

W tym miejscu wypadałoby wytłumaczyć, czym są uniwersa. Otóż odpowiedź jest dość prosta: uniwersum składa się z typu U: Type oraz funkcji $El:U\to T$ ype. Intuicja w tym wszystkim jest taka, że elementami typu U są nazwy typów (czyli bytów sortu Type), zaś fukncja El zwraca typ, którego nazwę dostanie.

Choć z definicji widać to na pierwszy rzut oka, to zaskakujący może wydać ci się fakt, że w zasadzie każdy typ można zinterpretować jako uniwersum i to zazwyczaj na bardzo wiele różnych sposobów (tyle ile różnych interpretacji *El* jesteśmy w stanie wymyślić). Najlepiej będzie, jeżeli przemyślisz to wszystko w ramach ćwiczenia.

Ćwiczenie Ćwiczenie będzie konceptualne, a składa się na nie kilka łamigłówek:

- zinterpretuj False jako uniwersum
- zinterpretuj *unit* jako uniwersum (ile jest możliwych sposobów?)

- czy istnieje uniwersum, które zawiera nazwę samego siebie? Uwaga: to nie jest tak proste, jak może się wydawać na pierwszy rzut oka.
- wymyśl ideologicznie słuszną interpretację typu *nat* jako uniwersum (tak, jest taka). Następnie wymyśl jakąś głupią interpretację *nat* jako uniwersum. Dlaczego ta interpretacja jest głupia?
- zdefiniuj uniwersum, którego elementami są nazwy typów funkcji z n-krotek liczb naturalnych w liczby naturalne. Uwaga: rozwiązanie jest bardzo eleganckie i możesz się go nie spodziewać.
- czy istnieje uniwersum, którego interpretacja jest surjekcją? Czy da się w Coqu udowodnić, że tak jest albo nie jest? Uwaga: tak bardzo podchwytliwe, że aż sam się złapałem.

Skoro wiemy już, czym są uniwersa, przyjrzyjmy się temu, które właśnie zdefiniowaliśmy. Żebyś nie musiał w rozpaczy przewijać do góry, tak wygląda aksjomatyczne kodowanie tego uniwersum:

Module PoorUniverse.

```
Axioms
   (U: {\tt Type})
   (El: U \to \mathsf{Type})
   (Pi: \forall (A:U) (B:El A \rightarrow U), U)
   (UU:U)
   (El_{-}Pi:
     \forall (A:U) (B:El A \rightarrow U),
         El(Pi \ A \ B) = \forall (x : El \ A), El(B \ x))
   (El_{-}UU : El \ UU = U)
   (ind: \forall
     (P:U\to \mathsf{Type})
     (PPi:
        \forall (A:U) (B:El A \rightarrow U),
           P A \rightarrow (\forall x : El A, P (B x)) \rightarrow P (Pi A B))
     (PUU: P\ UU),
        \{f: \forall u: U, Pu \mid
           (\forall (A:U) (B:El A \rightarrow U),
              f(Pi \ A \ B) =
              PPi \ A \ B \ (f \ A) \ (f un \ x : El \ A \Rightarrow f \ (B \ x))) \land
           (f \ UU = PUU)
```

).

U to typ, którego elementami są nazwy typów, zaś El jest jego interpretacją. Nazwy możemy tworzyć tylko na dwa sposoby: jeżeli A:U jest nazwą typu, zaś B:El $A\to U$ jest rodziną nazw typów indeksowaną przez elementy typu A, to Pi A B jest nazwą typu \forall x:El A, El (B x). Drugim konstruktorem jest UU, które oznacza nazwę samego uniwersum, tzn. El UU = U.

Reguła indukcji jest dość prosta: jeżeli $P:U\to \mathsf{Type}$ jest rodziną typów (tych prawdziwych) indeksowaną przez U (czyli nazwy typów), to żeby zdefiniować funkcję $f:\forall u:U$, P u musimy mieć dwie rzeczy: po pierwsze, musimy pokazać, że P (Pi A B) zachodzi, gdy zachodzi P A oraz P (B x) dla każdego x: El A. Po drugie, musi zachodzić P UU.

Mimo, że uniwersum wydaje się biedne, jest ono śmiertelnie sprzeczne, gdyż zawiera nazwę samego siebie. Jeżeli rozwiązałeś (poprawnie, a nie na odwal!) ostatnie ćwiczenie, to powinieneś wiedzieć, że niektóre uniwersa mogą zawierać nazwy samego siebie i wcale to a wcale nie daje to żadnych problemów.

Dlaczego więc w tym przypadku jest inaczej? Skoro UU nie jest złe samo w sobie, to problem musi leżeć w Pi, bo niby gdzie indziej? Zobaczmy więc, gdzie kryje się sprzeczność. W tym celu posłużymy się twierdzeniem Cantora: najpierw pokażemy surjekcję $U \to (U \to U)$, a potem, za pomocą metody przekątniowej, że taka surjekcja nie może istnieć.

```
(*
   Definition bad (u : U) : U -> U :=
   match u with
   | Pi UU B => B
   | _ => fun u : U => U
   end.
   *)
```

Jeżeli dostajemy $Pi\ A\ B$, gdzie A to UU, to wtedy $B:El\ A\to U$ tak naprawdę jest typu $U\to U$ (bo $El\ UU=U$). W innych przypadkach wystarczy po prostu zwrócić funkcję identycznościową. Niestety Coq nie wspiera indukcji-rekursji (ława oburzonych), więc funkcję bad musimy zdefiniować ręcznie:

```
Definition bad: U \to (U \to U).

Proof.

apply (ind \text{ (fun } \_ \Rightarrow U \to U)).

Focus 2. exact (fun u: U \Rightarrow u).

intros A \ B \ \_. revert A \ B.

apply (ind \text{ (fun } A: U \Rightarrow (El \ A \to U) \to (U \to U))).

intros; assumption.

intro B. rewrite El\_UU in B. exact B.

Defined.
```

Powyższa definicja za pomocą taktyk działa dokładnie tak samo jak nieformalna definicja bad za pomocą dopasowania do wzorca. Jedyna różnica jest taka, że $El\ UU$ nie jest definicyjnie równe U, lecz są one jedynie zdaniowo równe na mocy aksjomatu El_UU : El

UU = U. Musimy więc przepisać go w B, żeby typy się zgadzały.

Zanim będziemy mogli pokazać, że bad jest surjekcją, czeka nas kilka niemiłych detali technicznych (gdyby $El\ UU$ i U były definicyjnie równe, wszystkie te problemy by zniknęły).

Check eq_rect .

 eq_rect oraz eq_rect_r to groźnie wyglądające lematy, ale sprawa tak na prawdę jest dość prosta: to one wykonują całą pracę za każdym razem, kiedy używasz taktyki rewrite. Jeżeli cel jest postaci P x i użyjemy na nim rewrite H, gdzie H: x = y, to rewrite zamienia cel na $eq_rect____cel__H$, które jest już typu P y. eq_rect_r działa podobnie, ale tym razem równość jest postaci y = x (czyli obrócona).

Ponieważ w definicji bad używaliśmy rewrite'a, to przy dowodzeniu, że bad jest surjekcją, będziemy musieli zmierzyć się właśnie z eq_rect i eq_rect_r. Stąd poniższy lemat, który mówi mniej więcej, że jeżeli przepiszemy z prawa na lewo, a potem z lewa na prawo, to tak, jakby nic się nie stało.

```
 \begin{array}{c} {\sf Lemma}\ right\_to\_left\_to\_right: \\ &\forall \\ &(A: {\sf Type})\ (P: A \to {\sf Type})\ (x\ y: A)\ (p: x=y)\ (u: P\ y), \\ &eq\_rect\ x\ P\ (@eq\_rect\_r\ A\ y\ P\ u\ x\ p)\ y\ p=u. \\ \\ {\sf Proof.} \\ &{\sf destruct}\ p.\ cbn.\ {\sf reflexivity}. \\ {\sf Qed.} \end{array}
```

Dowód jest banalny. Ponieważ eq_rect i eq_rect_r są zdefiniowane przez dopasowanie do wzorca p:x=y, to wystarczy p potraktować destructem, a dalej wszystko już ładnie się oblicza.

```
Lemma bad\_sur: surjective\ bad. Proof.

unfold surjective\ bad\ intro\ f. destruct (ind\ \_) as [bad\ [bad\_Pi\ bad\_UU]]. destruct (ind\ \_) as [bad'\ [bad'\_Pi\ bad'\_UU]]. pose (f':=eq\_rect\_r\ (fun\ T: Type \Rightarrow T \to U)\ f\ El\_UU). \exists\ (Pi\ UU\ f'). unfold f'. rewrite bad\_Pi\ bad'\_UU\ right\_to\_left\_to\_right. reflexivity. Qed.
```

Dlaczego bad jest surjekcją? Intuicyjnie pisząc, każdą funkcję $U \to U$ możemy włożyć do konstruktora Pi jako jego drugi argument, jeżeli tylko zamienimy pierwsze U na $El\ UU$. Skoro każdą możemy tam włożyć, to każdą możemy wyjąć. Ot i cały sekret.

Technicznie dowód realizujemy tak: odwijamy definicje i wprowadzamy do kontekstu funkcję f. Następnie rozbijamy ind – pochodzące z definicji bad, rozkładając w ten sposób definicję bad na właściwe bad (sama funkcja), bad' (wewnętrzna funkcja pomocnicza) oraz równania dla bad i bad' dla poszczególnych przypadków.

Następnie musimy znaleźć takie a:U, że bad~a=f. Robimy to, co zasugerowałem wyżej, czyli w $f:U\to U$ pierwsze U zamieniamy na El~UU, uzyskując w ten sposób f'. Temu właśnie służy użycie eq_rect_r (nie używamy rewrite, bo potrzeba nam większej precyzji).

Wobec tego szukanym przez nas elementem U, któremu bad przyporządkuje f, jest $Pi\ UU$ f'. Możemy w tym miejscu odwinąć definicję f'. Gdyby Coq wspierał indukcję-rekursję, to w tym miejscu wystarczyłoby użyć tylko reflexivity - $bad\ (Pi\ UU\ f')$ obliczyłoby się do f na mocy definicji bad oraz dzięki temu, że $El\ UU$ obliczyłoby się do U. Niestety Coq nie wspiera indukcji rekursji (ława oburzonych), więc musimy wszystkie te trzy kroki obliczeń wykonać ręcznie za pomoca taktyki rewrite.

Ufff, udało się! Jeżeli przeraża cię ten dowód - nie martw się. Chodzi w nim o to samo, o co chodziło w poprzednich dowodach bycia surjekcją. Ten jest po prostu trochę bardziej skomplikowany, bo indukcja-rekursja jest nieco bardziej skomplikowana do użycia w Coqu niż prymitywniejsze formy indukcji.

```
Definition change : U \to U. Proof. apply ind. intros. exact UU. exact (Pi\ UU\ (\text{fun }\_\Rightarrow UU)). Defined.
```

Teraz czas udowodnić, że bad nie jest surjekcją. Zrobimy to metodą przekątniową, a w tym celu potrzebować będziemy funkcji $U \to U$, która dla każdego argumentu zwraca coś, co jest od niego różne.

Na szczęście sprawa jest prosta: jeżeli argumentem jest Pi A B, to zwracamy UU, zaś jeżeli UU, to zwracamy Pi UU (fun $_{-} \Rightarrow UU$).

```
\begin{array}{l} {\tt Definition} \ discern: \ U \to bool. \\ {\tt Proof.} \\ {\tt apply} \ ind. \\ {\tt intros.} \ {\tt exact} \ true. \\ {\tt exact} \ false. \\ {\tt Defined.} \end{array}
```

Przydałaby się też funkcja, która pozwoli nam rozróżnić konstruktory typu U. Normalnie użylibyśmy do tego taktyki inversion, ale używamy kodowania aksjomatycznego, więc inversion nie zadziała i musimy ręcznie zaimplementować sobie coś w jej stylu.

Nasza funkcja dla Pi zwraca true, a dla UU daje false.

```
apply ind.

intros A B H1 H2 eq.

apply (f_equal discern) in eq.

unfold change, discern in eq.

destruct (ind_-) as [d_-Pi_-d_-UU]],

(ind_-) as [ch_-Pi_-ch_-UU]].

rewrite d_-Pi, ch_-Pi, d_-UU in eq. inversion eq.

intro eq.

apply (f_equal discern) in eq.

unfold change, discern in eq.

destruct (ind_-) as [d_-Pi_-d_-UU]],

(ind_-) as [ch_-Pi_-ch_-UU]].

rewrite ch_-UU, d_-Pi, d_-UU in eq. inversion eq.

Qed.
```

Wypadałoby też pokazać, ża nasza funkcja działa tak, jak sobie tego życzymy. Dowód jest bardzo prosty, ale aksjomatyczne kodowanie znacznie go zaciemnia.

Zaczynamy od indukcji po u: U. W pierwszym przypadku mamy hipotezę eq: change $(Pi \ A \ B) = Pi \ A \ B$, a skoro tak, to po zaaplikowaniu discern musi być także discern (change $(Pi \ A \ B)) = discern \ (Pi \ A \ B)$.

Następnie rozkładamy definicje change i discern na atomy (change nazywa się teraz ch, a discern nazywa się d). Przepisujemy odpowiednie równania w hipotezie eq, dzięki czemu uzyskujemy false = true, co jest sprzeczne. Drugi przypadek jest analogiczny.

```
Lemma bad\_not\_sur:

\neg surjective \ bad.

Proof.

unfold surjective. intro.

destruct (H \ (\text{fun} \ u : U \Rightarrow \text{change} \ (bad \ u \ u))) as [u \ eq].

apply (f\_equal \ (\text{fun} \ f \Rightarrow f \ u)) in eq.

apply (change\_neq \ (bad \ u \ u)). symmetry. assumption.

Qed.
```

Teraz możemy już pokazać, że bad nie jest surjekcją. W tym celu wyobraźmy sobie bad jako kwadratową tabelkę, której wiersze i kolumny są indeksowane przez U. Tworzymy nową funkcję $U \to U$ biorąc elementy z przekątnej i modyfikując je za pomocą change.

Skoro bad jest surjekcją, to ta nowa funkcja musi być postaci bad u dla jakiegoś u: U. Aplikując obie strony jeszcze raz do u dostajemy równanie bad u u = change (bad u u), które jest sprzeczne na mocy lematu $change_neg$.

```
Definition U\_illegal: False. Proof. apply bad\_not\_sur. apply bad\_sur. Qed.
```

Ponieważ bad jednocześnie jest i nie jest surjekcją, nastepuje nagły atak sprzeczności.

Definicja uniwersum U przez indukcję-rekursję jest nielegalna. Tak właśnie prezentują się paradoksy Russella i Girarda w Coqowym wydaniu.

End PoorUniverse.

Ćwiczenie Tak naprawdę, to w tym podrozdziale byliśmy co najwyżej bieda-Thanosem, gdyż uniwersum, z którym się ścieraliśmy, samo było biedne. W niniejszym ćwiczeniu zmierzysz się z uniwersum, które zawiera też nazwy typu pustego, typu *unit* i liczb naturalnych, nazwy produktów, sum i funkcji, a także sum zależnych.

Mówiąc wprost: zakoduj aksjomatycznie poniższą definicję uniwersum U, a następnie udowodnij, że jest ona nielegalna. Nie powinno to być trudne - metoda jest podobna jak w przypadku biednego uniwersum.

Module NonPoorUniverse.

```
(*
  Fail Inductive U : Type :=
    | Empty : U
    | Unit : U
    | Nat : U
    | Prod : U -> U -> U
    | Sum : U -> U -> U
    | Arr : U -> U -> U
    | Pi : forall (A : U) (B : El A -> U), U
    | Sigma: forall (A : U) (B : El A -> U), U
    UU: U
  with El (u : U) : Type :=
  match u with
    | Empty => Empty_set
    | Unit => unit
    | Nat => nat
    | Prod A B => El A * El B
    | Sum A B => El A + El B
    | Arr A B => El A -> El B
    \mid Pi A B => forall x : El A, B x
    | Sigma A B \Rightarrow {x : El A & El (B x)}
    | UU => U
  end.
  *)
```

Theorem $U_{-}illegal : False$.

End NonPoorUniverse.

4.5.8 Pozytywne typy induktywne

Na koniec rozprawimy się z pozytywnymi typami "induktywnymi" (ale tylko do pewnego stopnia; tak po prawdzie, to raczej one rozprawią się z nami).

```
Fail Inductive Pos: Type := |Pos0:((Pos \rightarrow bool) \rightarrow bool) \rightarrow Pos. (* ===> The command has indeed failed with message: Non strictly positive occurrence of "Pos" in "((Pos -> bool) -> bool) -> Pos". *)
```

Coq odrzuca powyższą definicję typu Pos, gdyż pierwsze wystąpienie Pos w typie konstruktora $Pos\theta$ nie jest ściśle pozytywne. I faktycznie - gdy policzymy niedobrość tego wystąpienia zgodnie z naszym wzorem, to wyjdzie, że wynosi ona 2, gdyż Pos występuje na lewo od dwóch strzałek (pamiętaj, że najbardziej zewnętrzna strzałka, czyli ta, na prawo od której też jest Pos, nie liczy się - wzór dotyczy tylko argumentów konstruktora, a nie całego konstruktora).

```
(Pos: \mathsf{Type})
(Pos0: ((Pos 	o bool) 	o bool) 	o Pos)
(dcase: \  \  \, \forall \  \  \, (P: Pos 	o \mathsf{Type}) \  \  \, (Pos0: \forall g: (Pos 	o bool) 	o bool, P (Pos0 g)),
```

 $\{f: \forall x: Pos, P x \mid \forall g: (Pos \rightarrow bool) \rightarrow bool, f (Pos 0 g) = PPos 0 g\}).$

Axioms

Spróbujmy zawalczyć z typem Pos naszą metodą opartą o twierdzenie Cantora. Najpierw kodujemy typ Pos aksjomatycznie, a następnie spróbujemy zdefiniować bad, czyli surjekcję z Pos w $Pos \rightarrow bool$.

```
Definition bad: Pos \rightarrow (Pos \rightarrow bool). Proof.

apply (dcase \ (fun \ \_ \Rightarrow Pos \rightarrow bool)).

intros f \ x.

apply f. intro y.

apply f. intro z.

apply f. intro w.

(* ad infinitum *)

Abort.
```

Mogłoby się wydawać, że wyciągnięcie z Pos funkcji $Pos \rightarrow bool$ nie może być trudniejsze, niż zabranie dziecku cukierka. Niestety jednak nie jest tak, gdyż w Pos tak naprawdę nie ma żadnej takiej funkcji - jest funkcja $(Pos \rightarrow bool) \rightarrow bool$, a to już zupełnie coś innego.

Żeby lepiej zrozumieć tę materię, musimy metaforycznie zinterpretować znany nam już współ-

czynnik niedobrości i wynikający z niego podział na wystąpienia ściśle pozytywne, pozytywne i negatywne. Dzięki tej interpretacji dowiemy się też, dlaczego nieparzysta niedobrość jest negatywna, a niezerowa parzysta jest pozytywna.

Najprościej jest zinterpretować wystąpienia ściśle pozytywne, gdyż mieliśmy już z nimi sporo do czynienia. Weźmy konstruktor $cons:A\to list\ A\to list\ A$. Jest tutaj jedno ściśle pozytywne wystąpienie typu $list\ A$, które możemy interpretować tak: gdy używamy dopasowania do wzorca i dopasuje się $cons\ h\ t$, to "mamy" element t typu $list\ A$. Ot i cała filozofia.

Załóżmy teraz na chwilę, że Coq akceptuje negatywne i pozytywne typy induktywne. Co by było, gdybyśmy dopasowali konstruktor postaci $c:(T\to bool)\to T$? Tym razem nie mamy elementu typu T, lecz funkcję $f:T\to bool$. Parafrazując: musimy "dać" funkcji f element typu T, żeby dostać bool.

A co by było, gdybyśmy dopasowali konstruktor postaci $c:((T \to bool) \to bool) \to T$? Tym razem również nie mamy żadnego elementu typu T, lecz funkcję $f:((T \to bool) \to bool)$. Parafrazując: musimy dać funkcji f jakąś funkcję typu $T \to bool$, żeby dostać bool. Ale gdy konstruujemy funkcję $T \to bool$, to na wejściu dostajemy T. Tak więc początkowo nie mamy żadnego T, ale gdy o nie poprosimy, to możemy je dostać. Ba! Jak pokazuje przykład, możemy dostać bardzo dużo T.

Taka właśnie jest różnica między ścisłą pozytywnością (mamy coś), negatywnością (musimy coś dać) i pozytywnością (możemy coś dostać, i to nawet w dużej liczbie sztuk). Zauważmy, że jedynie w przypadku negatywnym możemy wyjąć z T funkcję $T \to co$ ś (chyba, że zawadza nam unit lub False), bo to jedyny przypadek, gdy żądają od nas T (a skoro żądają T, to muszą mieć funkcję, która coś z tym T zrobi). W przypadku pozytywnym nie ma żadnej takiej funkcji - to my dostajemy T i musimy coś z niego wyprodukować, więc to my jesteśmy tą funkcją!

Ufff... mam nadzieję, że powyższa bajeczka jest sformułowana zrozumiale, bo lepszego wytłumaczenia nie udało mi się wymyślić.

Moglibyśmy w tym miejscu zastanowić się, czy nie uda nam się pokazać sprzeczności choć na metapoziomie, poprzez napisanie nieterminującej funkcji loop. Szczerze pisząc, to niezbyt w to wierzę. Przypomnij sobie, że okazało się, że funkcja loop jest bardzo ściśle powiązana z funkcją bad, zaś esencja nieterminacji polegała na przekazaniu do loop jako argument czegoś, co zawierało loop jako podterm (jeżeli nie zauważyłeś, to wszystkie nasze nieterminujące funkcje udało nam się zdefiniować jedynie za pomocą reguły zależnej analizy przypadków - bez indukcji, bez rekursji!). To daje nam jako taką podstawę by wierzyć, że nawet nieterminacja nie jest w tym przypadku osiągalna.

W tym momencie należy sobie zadać zasadnicze pytanie: dlaczego w ogóle pozytywne typy induktywne są nielegalne? Przecież odróżnienie wystąpienia pozytywnego od negatywnego nie jest czymś trudnym, więc Coq nie może ich od tak po prostu nie rozróżniać - musi mieć jakiś powód!

I faktycznie, powód jest. Nie ma on jednak wiele wspólnego z mechanizmem (pozytywnych) typów induktywnych samym w sobie, a z impredykatywnością sortu Prop. Trudne słowo, co? Nie pamiętam, czy już to wyjaśniałem, więc wyjaśnię jeszcze raz.

Impredykatywność (lub też impredykatywizm) to pewna forma autoreferencji, która czasem jest nieszkodliwa, a czasem bardzo mordercza. Przyjrzyjmy się następującej definicji: "wujek Janusz to najbardziej wąsata osoba w tym pokoju". Definicja ta jest impredykatywna, gdyż definiuje ona wujka Janusza poprzez wyróżnienie go z pewnej kolekcji osób, ale definicja tej kolekcji osób musi odwoływać się do wujka Janusza ("w pokoju są wujek Janusz, ciocia Grażynka, Sebastianek i Karynka"). W Coqu impredykatywny jest sort Prop, co ilustruje przykład:

```
Definition X : \mathsf{Prop} := \forall P : \mathsf{Prop}, P.
```

Definicja zdania X jest impredykatywna, gdyż kwantyfikujemy w niej po wszystkich zdaniach ($\forall P : \texttt{Prop}$), a zatem kwantyfikujemy także po zdaniu X, które właśnie definiujemy.

Impredykatywność sortu Prop jest niegroźna (no chyba, że pragniemy pozytywnych typów induktywnych, to wtedy jest), ale impredykatywność dla Type byłaby zabójcza, co zresztą powinien nam był uświadomić paradoks Russella.

Dobra, koniec gadania. Poniższy przykład pośrednio pochodzi z sekcji 3.1 pracy "Inductively defined types", której autorami są Thierry Coquand oraz Christine Pauling-Mohring, zaś bezpośrednio jest przeróbką kodu wziętego z vilhelms.github.io/posts/why-must-inductive-types-be-strictly-positive

```
 \begin{array}{l} Fail \  \, {\tt Inductive} \  \, Pos': \  \, {\tt Type} := \\ \quad | \  \, Pos'0 : ((Pos' \to {\tt Prop}) \to {\tt Prop}) \to Pos'. \\ \\ {\tt Axioms} \\ \quad (Pos' : {\tt Type}) \\ \quad (Pos'0 : ((Pos' \to {\tt Prop}) \to {\tt Prop}) \to Pos') \\ \quad (dcase' : \\ \forall \\ \quad (P : Pos' \to {\tt Type}) \\ \quad (PPos'0 : \forall \ g : (Pos' \to {\tt Prop}) \to {\tt Prop}, \ P \ (Pos'0 \ g)), \\ \quad \{f : \forall \ x : \  \, Pos', \  \, P \ x \mid \\ \quad \forall \  \, g : (Pos' \to {\tt Prop}) \to {\tt Prop}, \\ \quad f \  \, (Pos'0 \ g) = PPos'0 \ g\}). \end{array}
```

Jak widać, podejrzanym typem jest Pos', bliźniaczo podobne do Pos, ale zamiast bool występuje tutaj Prop.

```
Definition unwrap: Pos' \rightarrow ((Pos' \rightarrow \texttt{Prop}) \rightarrow \texttt{Prop}). Proof. 
 apply\ (dcase'\ (\texttt{fun}\ \_ \Rightarrow (Pos' \rightarrow \texttt{Prop}) \rightarrow \texttt{Prop})). intros f. exact f. Defined.
```

Zaczynamy od zdefiniowania funkcji odwijającej konstruktor.

```
Lemma Pos'0\_inj:

\forall x \ y : (Pos' \rightarrow \texttt{Prop}) \rightarrow \texttt{Prop},

Pos'0 \ x = Pos'0 \ y \rightarrow x = y.
```

```
Proof. intros. apply (f_equal unwrap) in H. unfold unwrap in H. destruct (dcase'_) as [unwrap\ eq]. rewrite 2!eq in H. assumption. Qed.
```

Dzięki unwrap możemy łatwo pokazać, że konstruktor Pos'0 jest injekcją (to coś, co w przypadku zwykłych typów induktywnych dostajemy za darmo od taktyki inversion, ale cóż, nie tym razem!).

```
Definition i \{A: \mathsf{Type}\}: A \to (A \to \mathsf{Prop}):= \mathsf{fun}\ x\ y \Rightarrow x = y.
Lemma i\_inj:
\forall\ (A: \mathsf{Type})\ (x\ y:A),\ i\ x = i\ y \to x = y.
Proof.
unfold i. intros.
apply (\mathsf{f\_equal}\ (\mathsf{fun}\ f \Rightarrow f\ y)) in H.
rewrite H. reflexivity.
Qed.
```

Kolejnym krokiem jest zdefiniowanie funkcji i, która jest injekcją z dowolnego typu A w typ $A \to \mathsf{Prop}$. Zauważmy, że krok ten w kluczowy sposób korzysta z równości, żyjącej w sorcie Prop - gdyby zamiast Prop było bool, nie moglibyśmy zdefiniować tej injekcji.

```
Definition f (P:Pos' \to \text{Prop}):Pos' := Pos'0 \ (i\ P). Lemma f\_inj: \forall\ P\ Q:Pos' \to \text{Prop}, f\ P=f\ Q \to P=Q. Proof. unfold f. intros. apply (f\_equal\ unwrap) in H. unfold unwrap in H. destruct (dcase'\_) as [unwrap\ eq]. rewrite 2!eq in H. apply i\_inj in H. assumption. Qed.
```

Składając ze soba i oraz konstruktor Pos'0 dostajemy injekcję z $Pos' \to Prop \ w \ Pos'$.

```
Definition wut\ (x:Pos'): \texttt{Prop} := \exists\ P:Pos' \to \texttt{Prop}, f\ P = x \land \neg\ P\ x. Definition x:Pos' := f\ wut.
```

Tutaj następują największe czary, które używają impredykatywności. Nie mam żadnego dobrej bajeczki, która by je wyjaśniała.

```
Lemma paradox : wut x \leftrightarrow \neg wut x.
```

```
Proof. split. destruct 1 as (P \& H1 \& H2). intro H. unfold x in H1. apply f_-inj in H1. subst. contradiction. intro H. unfold wut. \exists \ wut. split. unfold x. reflexivity. assumption. Qed.
```

paradox to twierdzenie, które chwyta esencję całej sprawy. Z lewa na prawo rozbijamy dowód $wut\ x$ i dostajemy predykat P. Wiemy, że $f\ P=x$, ale $x=f\ wut$, a ponieważ f jest injekcją, to P=wut. To jednak kończy się sprzecznością, bo $wut\ x$, ale $\neg\ P\ x$.

Z prawa na lewo jest łatwiej. Mamy $\neg wut x$ i musimy udowodnić wut x. Wystarczy, że istnieje pewien predykat, na który wybieramy oczywiście wut, który spełnia f wut = x, co jest prawdą na mocy definicji x, oraz $\neg wut x$, co zachodzi na mocy założenia.

```
Theorem Pos'_illegal : False.

Proof.

pose paradox. firstorder.

Qed.
```

No i bum. Jak widać, pozytywne typy induktywne prowadzą do sprzeczności, ale nie ma to z nimi wiele wspólnego, za to ma wiele wspólnego z sortem Prop i jego impredykatywnościa.

4.6 Podsumowanie

To już koniec naszej przydługiej podróży przez mechanizmy definiowania typów przez indukcję. W jej trakcie nauczyliśmy się bardzo wielu rzeczy.

Zaczęliśmy od definiowania prostych enumeracji, operujących na nich funkcji definiowanych za pomocą dopasowania do wzorca oraz omówienia mechanizmu obliczania wyniku funkcji.

Następnie poznaliśmy różne rozszerzenia tego podstawowego pomysłu definiowania typu za pomocą konstruktorów reprezentujących możliwe wartości:

- rekurencję, dzięki której możemy definiować typy, których termy mają najprzeróżniejsze drzewiaste kształty
- parametryzowane typy induktywne, których głównym zastosowaniem jest definiowanie kontenerów o takich samych kształtach, ale różnych przechowywanych typach
- indukcję wzajemną, w praktyce niezbyt użyteczną, dzięki której możemy na raz zdefiniować wiele typów odnoszących się do siebie nawzajem
- indeksowane rodziny typów induktywnych, dzięki którym możemy przez indukcję definiować predykaty oraz relacje

- indukcję-indukcję, dzięki której możemy jednocześnie zdefiniować typ oraz indeksowaną nim rodzinę typów
- indukcję-rekursję, dzięki której możemy jednoczesnie zdefiniować typ oraz funkcję operującą na tym typie

Nauczyliśmy się definiować funkcje przez rekursję oraz dowodzić ich właściwości przez indukcję. Poznaliśmy definicje poznanych w pierwszym rozdziale spójników logicznych oraz odpowiadających im konstrukcji na typach, a także definicję bardzo ważnej rodziny typów, czyli równości.

Poznaliśmy podstawowe obiekty, którymi musi potrafić posługiwać się każdy programista, informatyk czy matematyk, a mianowicie wartości boolowskie, liczby naturalne oraz listy.

Nauczyliśmy się formułować i implementować reguły indukcyjne (TODO: opisać to w głównym tekście, a nie dopiero w przypomnieniu), a także, co powiązane, programować listy przy pomocy foldów i unfoldów.

Na końcu poznaliśmy kryterium ścisłej pozytywności, które obowiązuje wszystkie definicje typów induktywnych. Dowiedzieliśmy się, że negatywne typy induktywne prowadzą do nieterminacji, która jest złem wcielonym. Poznaliśmy pojęcie surjekcji oraz twierdzenie Cantora, które również zabrania negatywnym typom induktywnym istnienia.

Poznaliśmy też paradoks Russela/Girarda i jego związek z twierdzeniem Cantora, autoreferencją oraz ideą uniwersum zdefiniowanego za pomocą indukcji-rekursji.

Ostatecznie dowiedzieliśmy się, że pozytywne typy induktywne także są nielegalne, choć jesteśmy wobec nich raczej bezsilni, no chyba że chodzi o impredykatywny (tego słowa też się nauczyliśmy) sort Prop.

Całkiem sporo, prawda? Nie? No to w kolejnych rozdziałach będzie jeszcze więcej.

Rozdział 5

D2: Rekursja i indukcja

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).

5.1 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.
- W szczególności rekursja prymitywna to taka, w której funkcja wywołuje siebie jedynie na bezpośrednich podtermach argumentu głównego 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.

5.2 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\ {\tt 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 zatrzymuja.

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.

5.3 Rekursja po paliwie

Rekursja dobrze ufundowana to sirius byznys, więc zanim się nią zajmiemy wypadałoby nauczyć się robić robotę na odwal, byle działało. Jakkolwiek nie brzmi to zbyt profesjonalnie, dobrze jest mieć tego typu narzędzie w zanadrzu, choćby w celu szybkiego prototypowania. Czasem zdarza się też, że tego typu luźne podejście do problemu jest jedynym możliwym, bo nikt nie wie, jak to zrobić porządnie.

Narzędziem, o którym mowa, jest coś, co ja nazywam "rekursją po paliwie". Pozwala ona zasymulować definicję dowolnej funkcji o typie $A1 \to ... \to An \to B$ (w tym nawet częściowej czy nieterminującej, co już samo w sobie jest ciekawe) za pomocą funkcji o typie $nat \to A1 \to ... \to An \to option B$.

Trik jest dość banalny: argument typu nat jest argumentem głównym, po którym robimy rekursję. Jest on naszym "paliwem", które spalamy przy każdym wywołaniu rekurencyjnym. Jeżeli paliwo się nam skończy, zwracamy None. Jeżeli jeszcze starcza paliwa, możemy zdefiniować funkcję tak jak zamierzaliśmy, ale mamy też obowiązki biurokratyczne związane ze sprawdzaniem, czy wyniki wywołań rekurencyjnych to None czy Some.

Coby za dużo nie godoć, przykład.

```
Require Import List. Import ListNotations. Require Import Nat.
```

Będą nam potrzebne notacje dla list oraz funkcja even, która sprawdza, czy liczba naturalna jest parzysta. Będziemy chcieli zdefiniować funkcję Collatza. Gdyby Coq wspierał rekursję ogólną, jej definicja wyglądałaby tak:

```
Fail Fixpoint collatz (n:nat): list nat:= match n with \mid 0 \mid 1 \Rightarrow [n] \mid \mid - \Rightarrow n:: if even n then collatz (div2 n) else collatz (1 + 3 \times n) end.
```

Jest to bardzo wesoła funkcja. Przypadki bazowe to 0 i 1 - zwracamy wtedy po prostu listę z jednym elementem, odpowiednio [0] lub [1]. Ciekawiej jest dla n większego od 1. n zostaje głową listy, zaś w kwestii ogona mamy dwa przypadki. Jeżeli n jest parzyste, to argumentem wywołania rekurencyjnego jest n podzielone przez 2, zaś w przeciwnym przypadku jest to $1+3\times n$.

Funkcja ta nie ma żadnego ukrytego celu. Została wymyślona dla zabawy, a przyświecające jej pytanie to: czy funkcja ta kończy pracę dla każdego argumentu, czy może jest jakiś, dla którego się ona zapętla?

O ile funkcja jest prosta, o tyle odpowiedź jest bardzo skomplikowana i dotychczas nikt nie potrafił jej udzielić. Sprawdzono ręcznie (czyli za pomocą komputerów) bardzo dużo liczb i funkcja ta zawsze kończyła pracę, ale nikt nie umie udowodnić, że dzieje się tak dla wszystkich liczb.

```
Fixpoint collatz (fuel n : nat) : option (list nat) :=
match fuel with
      0 \Rightarrow None
      \mid S \; fuel' \Rightarrow
           match n with
                 \mid 0 \mid 1 \Rightarrow Some [n]
                 |  \rightarrow
                       if even n
                       then
                          match collatz fuel' (div2 n) with
                                 Some \ l \Rightarrow Some \ (n :: l)
                                 None \Rightarrow None
                          end
                       else
                          match collatz fuel' (1+3\times n) with
                                 Some \ l \Rightarrow Some \ (n :: l)
                                | None \Rightarrow None |
                          end
           end
end.
```

Definicja funkcji collatz za pomocą rekursji po paliwie wygląda dość groźnie, ale tak naprawdę jest całkiem banalna.

Ponieważ oryginalna funkcja była typu $nat \to list\ nat$, to ta nowa musi być typu $nat \to nat \to option\ (list\ nat)$. Tym razem zamiast dopasowywać n musimy dopasować paliwo, czyli fuel. Dla 0 zwracamy None, a gdy zostało jeszcze trochę paliwa, przechodzimy do właściwej części definicji. W przypadkach bazowych zwracamy [n], ale musimy zawinąć je w Some. W pozostałych przypadkach sprawdzamy, czy n jest parzyste, a następnie doklejamy odpowiedni ogon, ale musimy dopasować wywołania rekurencyjne żeby sprawdzić, czy zwracają one $None\ czy\ Some$.

```
: option (list nat) *)
```

Zaimplementowana za pomocą rekursji po paliwie funkcja oblicza się bez problemu, oczywiście o ile wystarczy jej paliwa. W powyższych przykładach 10 jednostek paliwa wystarcza, by obliczyć wynik dla 5, ale 2 jednostki paliwa to za mało. Jak więc widać, ilość potrzebnego paliwa zależy od konkretnej wartości na wejściu.

Interpretacja tego, czym tak naprawdę jest paliwo, nie jest zbyt trudna. Jest to maksymalna głębokość rekursji, na jaką może pozwolić sobie funkcja. Czym jest głębokość rekursji? Możemy wyobrazić sobie drzewo, którego korzeniem jest obecne wywołanie, a poddrzewami są drzewa dla wywołań rekurencyjnych. Głębokość rekursji jest po prostu głębokością (czyli wysokością) takiego drzewa.

W przypadku funkcji *collatz* głębokość rekursji jest równa długości zwróconej listy (gdy funkcja zwraca *Some*) lub większa niż ilość paliwa (gdy funkcja zwraca *None*).

Powyższe rozważania prowadzą nas do techniki, która pozwala z funkcji zrobionej rekursją po paliwie zrobić normalną, pełnoprawną funkcję. Wystarczy znaleźć "funkcję tankującą" $fill_tank: A1 \rightarrow ... \rightarrow An \rightarrow nat$, która oblicza, ile paliwa potrzeba dla danych argumentów wejściowych. Funkcja ta powinna mieć tę własność, że gdy nalejemy tyle paliwa, ile ona każe (lub więcej), zawsze w wyniku dostaniemy Some.

Trudnością, z którą nikt dotychczas w przypadku funkcji *collatz* nie potrafił się uporać, jest właśnie znalezienie funkcji tankującej. Jak więc widać, rekursja po paliwie nie zawsze jest fuszerką czy środkiem prototypowania, lecz czasem bywa faktycznie przydatna do reprezentowania funkcji, których inaczej zaimplementować się nie da.

Cwiczenie Zdefiniuj za pomocą rekursji po paliwie funkcję divFuel, która jest implementacją dzielenia (takiego zwykłego, a nie sprytnego jak ostatnio, tzn. divFuel fuel n 0 jest niezdefiniowane).

Čwiczenie Sporą zaletą rekursji po paliwie jest to, że definicje zrobionych za jej pomocą funkcji są jasne i czytelne (przynajmniej w porównaniu do rekursji dobrze ufundowanej, o czym już niedługo się przekonamy). To z kolei pozwala nam w dość łatwy sposób dowodzić interesujących nas właściwości tych funkcji.

Udowodnij kilka oczywistych właściwości dzielenia:

- divFuel? n = Some n, tzn. n/1 = n. Ile potrzeba paliwa?
- divFuel? n = Some 1, tzn. n/n = 1. Ile potrzeba paliwa?
- przy dzieleniu przez 0 nigdy nie starcza paliwa.

Cwiczenie (lemat o tankowaniu) Pokaż, że jeżeli wystarcza nam paliwa do obliczenia wyniku, ale zatankujemy jeszcze trochę, to dalej będzie nam wystarczać. Wniosek: tankującemu nie dzieje się krzywda.

Ćwiczenie Udowodnij, że funkcji collatz dla wejść o postaci $pow\ 2\ n$ (czyli potęg dwójki) wystarczy $S\ n$ jednostek paliwa.

Uwaga (trochę złośliwa): jeśli napotkasz trudności w trakcie dowodzenia (a moje uwagi przecież nie biorą się znikąd), to pamiętaj, że mają one charakter arytmetyczny, tzn. są związane z użyciem w definicji funkcji takich jak pow czy div2, nie są zaś spowodowane jakimiś problemami z samą techniką, jaką jest rekursja po paliwie.

5.4 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 domina bardziej przywodzą 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.

Co to znaczy, że niczego nie spartaczyliśmy, pytasz? Tutaj przydaje się spojrzenie na nasze domino "od tyłu". Żeby kostka domina się przewróciła, muszą przewrócić się na nią wszystkie bezpośrednio poprzedzające ją kostki, a żeby one się przewróciły, to przewrócić muszą się wszystkie poprzedzające je kostki i tak dalej. W związku z tym możemy powiedzieć, że kostka jest dostępna, jeżeli dostępne są wszystkie kostki ją poprzedzające.

Jeszcze jeden drobny detal: kiedy dostępne są kostki, które nie mają żadnych poprzedzających kostek? Odpowiedź: zawsze, a dowodem na to jest nasz palec, który je przewraca.

W ten oto wesoły sposób udało nam się uzyskać definicję elementu dostępnego oraz relacji dobrze ufundowanej.

```
Inductive Acc\ \{A: \mathsf{Type}\}\ (R:A\to A\to \mathsf{Prop})\ (x:A): \mathsf{Prop}:= |Acc\_intro:(\forall\ y:A,\ R\ y\ x\to Acc\ R\ y)\to 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 jest dostępna w układzie

domina R, jezeli każda kostka y, która poprzedza ją w układzie R, również jest dostępna.

Mniej poetycko: element x:A jest R-dostępny, jeżeli każdy R-mniejszy od niego element y:A również jest R-dostępny.

```
Definition well\_founded\ \{A: \mathtt{Type}\}\ (R:A\to A\to \mathtt{Prop}): \mathtt{Prop}:= \forall\ x:A,\ Acc\ R\ x.
```

Układ kostek reprezentowany przez R jest niespartaczony, jeżeli każda kostka domina jest dostępna.

Mniej poetycko: relacja R jest dobrze ufundowana, jeżeli każde x:A jest R-dostępne.

Uwaga: typem naszego układu kostek nie jest $A \to A \to \mathsf{Prop}$, lecz $A \to A \to \mathsf{Type}$, a zatem R jest tak naprawdę indeksowaną rodziną typów, a nie relacją. Różnica między relacją i rodziną typów jest taka, że relacją, 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 Sprawdź, czy relacje \leq , < są dobrze ufundowane.

Ćwiczenie Pokaż, że relacja dobrze ufundowana jest antyzwrotna oraz zinterpretuj ten fakt (tzn. powiedz, o co tak naprawdę chodzi w tym stwierdzeniu).

```
 \begin{array}{c} \texttt{Lemma} \ wf\_antirefl: \\ \forall \ (A: \texttt{Type}) \ (R: A \rightarrow A \rightarrow \texttt{Prop}), \\ well\_founded \ R \rightarrow \forall \ x: \ A, \ \neg \ R \ x \ x. \end{array}
```

Ćwiczenie 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 <.

Ćwiczenie Sprawdź, czy dobrze ufundowana jest następująca relacja porządku (mam nadzieję, że obrazek jest zrozumiały): 0 < 1 < ... < < +1 < ... < 2*

Oczywiście najpierw musisz wymyślić, w jaki sposób zdefiniować taką relację. Uwaga: istnieje bardzo sprytne rozwiązanie.

Nasza bajka powoli zbliża się do końca. Czas udowodnić ostateczne twierdzenie, do którego dążyliśmy: jeżeli układ kostek R jest niespartaczony (czyli gdy każda kostka jest dostępna), to każda kostka się przewraca.

Theorem $well_founded_rect$:

```
 \begin{array}{l} \forall \\ (A: \texttt{Type}) \; (R: A \rightarrow A \rightarrow \texttt{Prop}) \\ (\texttt{wf}: well\_founded \; R) \; (P: A \rightarrow \texttt{Type}), \\ (\forall \; x: \; A, \; (\forall \; y: A, \; R \; y \; x \rightarrow P \; y) \rightarrow P \; x) \rightarrow \end{array}
```

```
\forall \ x:\ A,\ P\ x. Proof. intros A\ R wf P\ H\ x. unfold well\_founded in wf. specialize (wf x). induction wf as [x\ \_\ IH]. apply H. exact IH. Defined.
```

Podobnie jak poprzednio, A to typ kostek domina, R to układ kostek, zaś wf: $well_founded$ R to dowód na to, że układ jest niespartaczony. $P:A\to \mathsf{Type}$ to dowolna rodzina typów indeksowana przez A, ale możemy myśleć, że P x znaczy "kostka x się przewraca". Mamy jeszcze hipotezę, która głosi, że kostka x przewraca się, gdy przewraca się każda kostka, która poprzedza ją w układzie R.

Dowód jest banalny. Zaczynamy od wprowadzenia zmiennych i hipotez do kontekstu. Następnie odwijamy definicję $well_founded$. Teraz hipoteza wf głosi, że każde x:A jest dostępne. Skoro tak, to specjalizujemy ją dla naszego konkretnego x, które mamy w kontekście.

Wiemy już zatem, że x jest dostępne. Jest to kluczowy fakt, gdyż oznacza to, że wszystkie kostki domina poprzedzające x również są dostępne. Co więcej, Acc jest zdefiniowane induktywnie, więc możemy pokazać, że x się przewraca, właśnie przez indukcję po dowodzie dostępności x.

Przypadek jest jeden (co nie znaczy, że nie ma przypadków bazowych - są nimi kostki domina, których nic nie poprzedza): musimy pokazać, że x się przewraca przy założeniu, że wszystkie poprzedzające je kostki również się przewracają. To, że x się przewraca, wynika z hipotezy H. Pozostaje nam jedynie pokazać, że przewraca się wszystko, co jest przed nim, ale to jest faktem na mocy hipotezy indukcyjnej IH.

```
Theorem well\_founded\_ind:
\forall \qquad \qquad (A: \texttt{Type}) \; (R: A \to A \to \texttt{Prop}) \\ \; (\texttt{wf}: well\_founded \; R) \; (P: A \to \texttt{Type}), \\ \; (\forall \; x: \; A, \; (\forall \; y: \; A, \; R \; y \; x \to P \; y) \to P \; x) \to \\ \; \forall \; x: \; A, \; P \; x.
\texttt{Proof}. \\ \; \texttt{intros} \; A \; R \; \texttt{wf} \; P \; H \; x. \\ \; \texttt{apply} \; (well\_founded\_rect \; \_ \; \texttt{wf} \; \_ H).
\texttt{Qed}.
```

Poprzednie twierdzenie, czyli $well_founded_rect$, to twierdzenie o rekursji dobrze ufundowanej. Powyższe, czyli $well_founded_ind$, które jest jego specjalizacją dla relacji binarnych (czyli bytów o typie $A \to A \to \mathsf{Prop}$), możemy nazwać twierdzeniem o indukcji dobrze ufundowanej.

Upewnij się, że dobrze rozumiesz oba twierdzenia, a także pojęcia dostępności i dobrego ufundowania, gdyż są one bardzo ważne przy rozwiązywaniu poważniejszych problemów.

Co to są "poważniejsze problemy"? Mam oczywiście na myśli dowodzenie twierdzeń i definiowanie funkcji, którego nie da się zrobić za pomocą prostej indukcji albo banalnego

dopasowania do wzorca. W tego typu sytuacjach nieodzowne będzie skorzystanie z indukcji i rekursji dobrze ufundowanej, o czym przekonamy się już natychmiast zaraz.

```
Require Import Lia.

Definition div: nat \rightarrow nat \rightarrow nat.

Proof.

apply (@well_founded_rect nat lt wf_lt (fun \_ \Rightarrow nat \rightarrow nat)).

intros n IH m.

destruct (le_-lt_-dec (S m) n).

2: exact 0.

refine (1 + IH (n - S m) \_ m). abstract lia. Show Proof.

Defined.
```

Poważniejszym problemem jest bowiem definicja dzielenia, z którą borykamy się od samiuśkiego początku niniejszego rozdziału. Powyższy kawałek kodu jest (nieudaną, jak się okaże) próbą uporania się z tym problemem.

Definiować będziemy w trybie dowodzenia, gdyż przy posługiwaniu się rekursją dobrze ufundowaną zazwyczaj tak jest dużo łatwiej. Zaczynamy od zaaplikowania reguły rekursji dobrze ufundowanej dla typu nat i porządku < (no i rzecz jasna wf_-lt , czyli dowodu na to, że lt jest dobrze ufundowany - bez tego ani rusz). Po typach widać, że rekursja będzie się odbywać po pierwszym argumencie. Wprowadzamy też zmienne do kontekstu.

```
Check le_{-}lt_{-}dec.
```

```
(* ===> le_lt_dec : forall n m : nat, {n <= m} + {m < n} *)
```

Następnie musimy sprawdzić, czy dzielna (czyli n) jest mniejsza od dzielnika (czyli S m - zauważ, że definiujemy tutaj "sprytną" wersję dzielenia, tzn. div n m = n/(m+1), żeby uniknąć problemów z dzieleniem przez 0). Jeżeli tak, wynikiem jest 0. Jeżeli nie, wynikiem jest wynik wywołania rekurencyjnego na argumencie n - S m powiększony o 1.

Na koniec musimy jeszcze tylko pokazać, że argument wywołania rekurencyjnego, czyli n - S m, jest mniejszy od argumentu obecnego wywołania, czyli n. Żeby za bardzo nie pobrudzić sobie rąk arytmetyką, zostawiamy ten cel taktyce lia, ale zawijamy jej użycie w kombinator abstract, który zapobiega "wylaniu się" rozumowania taktyki lia do definicji.

```
Print div.
```

```
: nat -> nat -> nat *)

Check div\_subproof.

(* ===> div\_subproof

: forall n m : nat, S m <= n -> n - S m < n *)

Print div\_subproof.

(* ===> du\dot{z}o różnych głupot, szkoda pisać *)
```

Mówiąc wprost, taktyka abstract *lia* zamiast wstawiać do definicji całe rozumowanie, tak jak zrobiłaby to taktyka *lia*, dowodzi sobie na boku odpowiedni lemat arytmetyczny, nazywa go *div_subproof* i dowodzi celu za jego pomocą.

```
Compute div \ 5 \ 2.
(* ===> = 1 : nat *)
```

Jak widać, definicja przechodzi bez problemu, a nasza funkcja elegancko się oblicza (pamiętaj, że div 5 2 to tak naprawdę 5/3, więc wynikiem faktycznie powinno być 1).

Jednak nie samymi definicjami żyje człowiek - czas trochę podowodzić. Spodziewamy się wszakże, że nasze dzielenie spełnia wszystkie właściwości, których się po nim spodziewamy, prawda?

```
Lemma div\_0\_r: \forall n: nat, div \ n \ 0 = n. Proof. apply (well\_founded\_ind\_\_\_wf\_lt). intros. unfold div. \ cbn. (* 0 Jezu, a cóż to za wojacy? *) Abort.
```

Niestety jednak, jak to w życiu, nie ma kolorowo.

Powyższy lemat głosi, że n/1 = n. Ponieważ div jest zdefiniowane za pomocą rekursji dobrze ufundowanej, to dowodzić będziemy oczywiście za pomocą indukcji dobrze ufundowanej. Tak, bedziemy dowodzić, hmmm... cóż... tylko jak?

Sytuacja wygląda beznadziejnie. Nie żeby lemat był nieprawdziwy - co to, to nie. Po prostu próba odwinięcia definicji i policzenia czegokolwiek daje inny wynik, niż byśmy chcieli - część definicji ukryta dotychczas w $div_subproof$ wylewa się i zaśmieca nam ekran.

Problem nie pochodzi jednak od taktyki *lia* (ani od abstract *lia*). Jest on dużo ogólniejszy i polega na tym, że wewnątrz definicji funkcji pojawiają się dowody, które są wymagane przez well_founded_rect, ale które zaorywują jej obliczeniową harmonię.

Nie jesteśmy jednak (jeszcze) skazani na porażkę. Spróbujemy uporać się z tą przeszkodą dzięki równaniu rekurencyjnemu. Równanie rekurencyjne to lemat, którego treść wygląda dokładnie tak, jak pożądana przez nas definicja funkcji, ale która nie może służyć jako definicja z różnych powodów, np. dlatego że nie jest strukturalnie rekurencyjna. Dzięki równaniu rekurencyjnemu możemy użyć taktyki rewrite do przepisania wystąpień funkcji div do pożądanej postaci zamiast rozwijać je za pomocą taktyki unfold lub obliczać za pomocą cbn.

Lemma $div_{-}eq$:

```
\forall \ n \ m: nat, \ div \ n \ m = \mbox{if} \ n <? \ S \ m \ \mbox{then} \ 0 \ \mbox{else} \ S \ (div \ (n - S \ m) \ m). Proof. apply (well\_founded\_ind \ \_ \ \_ \ wf\_lt \ (\mbox{fun} \ \_ \Rightarrow \forall \ m: nat, \ \_)). intros. unfold div. \ cbn. (* 0 Jezu, a cóż to za hołota? *) Admitted.
```

Powyższe równanie dokładnie opisuje, jak powinna zachowywać się funkcja div, ale za definicję służyć nie może, gdyż Coq nie byłby w stanie rozpoznać, że n - S m jest podtermem n. Zauważ, że używamy tu <? (czyli ltb) zamiast le_lt_dec . Możemy sobie na to pozwolić, gdyż użycie le_lt_dec w faktycznej definicji wynikało jedynie z tego, że potrzebowaliśmy dowodu odpowiedniego faktu arytmetycznego, żeby użyć go jako argumentu wywołania rekurencyjnego.

Niestety próba udowodnienia tego równania rekurencyjnego musi skończyć się taką samą porażką, jak próba udowodnienia $div_-\theta_-r$. Przyczyna jest taka sama jak ostatnio. Zresztą, naiwnym byłoby spodziewać się, że nam się uda - zarówno $div_-\theta_-r$, jak i div_-eq to nietrywialne właściwości funkcji div, więc gdybyśmy potrafili udowodnić równanie rekurencyjne, to z dowodem $div_-\theta_-r$ również poradzilibyśmy sobie bez problemu.

Żeby jednak przekonać się o użyteczności równania rekurencyjnego, jego "dowód" kończymy za pomocą komendy *Admitted*, która przerywa dowód i zamienia twierdzenie w aksjomat. Dzięki temu za chwilę zobaczymy, ile moglibyśmy zdziałać, mając równanie rekurencyjne.

```
Lemma div_- \theta_- r: \forall n: nat, \ div \ n \ 0 = n. Proof. apply (well\_founded\_ind \ \_ \ wf_- lt). intros n IH. rewrite div_- eq. destruct (Nat.ltb\_spec \ n \ 1). lia. rewrite IH; lia. Qed.
```

Jak widać, dzięki równaniu rekurencyjnemu dowody przebiegają dość gładko. W powyższym zaczynamy od indukcji dobrze ufundowanej po n (przy użyciu relacji < i dowodu wf_-lt), wprowadzamy zmienne do kontekstu, po czym przepisujemy równanie rekurencyjne. Po przeprowadzeniu analizy przypadków kończymy za pomocą rozumowań arytmetycznych, używając być może hipotezy indukcyjnej.

Ćwiczenie Zgadnij, jakie jest polecenie tego ćwiczenia, a następnie wykonaj je.

```
Lemma div_{-}n_{-}n:
 \forall n : nat, div (S n) n = 1.
```

Ćwiczenie 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 \rightarrow nat): \texttt{Prop} := \forall n: nat, f \ n \leq g \ n.
Definition lt'(f g: nat \rightarrow nat): \texttt{Prop} := \forall n: nat, f \ n < g \ n.
```

Ćwiczenie Niech B: Type i niech $R: B \to B \to \mathsf{Prop}$ będzie relacją dobrze ufundowaną. Zdefiniuj po współrzędnych relację porządku na funkcjach o typie $A \to B$ i rozstrzygnij, czy relacja ta jest dobrze ufundowana.

Uwaga: w zależności od okoliczności to zadanie może być trudne lub łatwe.

Ćwiczenie Pokaż, że jeżeli kodziedzina funkcji $f:A\to B$ jest dobrze ufundowana za pomocą relacji $R:B\to B\to \operatorname{Prop}$, to jej dziedzina również jest dobrze ufundowana.

```
Lemma wf\_inverse\_image:
\forall (A \ B : \texttt{Type}) \ (f : A \to B) \ (R : B \to B \to \texttt{Prop}),
well\_founded \ R \to well\_founded \ (\texttt{fun} \ x \ y : A \Rightarrow R \ (f \ x) \ (f \ y)).
```

5.5 Indukcja wykresowa

Skoro nie dla psa kiełbasa, to musimy znaleźć jakiś sposób na udowodnienie równania rekurencyjnego dla div. Zamiast jednak głowić się nad równaniami rekurencyjnymi albo nad funkcją div, zastanówmy się w pełnej ogólności: jak dowodzić właściwości funkcji rekurencyjnych?

No przez indukcję, czy to nie oczywiste? Jasne, ale jak dokładnie owa indukcja ma wyglądać? Odpowiedź jest prostsza niż można się spodziewać. Otóż gdy kupujesz but, ma on pasować do twojej stopy, zaś gdy kupujesz gacie, mają one pasować do twojej dupy. Podobnie jest z indukcją: jej kształt ma pasować do kształtu rekursji, za pomocą której zdefiniowana została funkcja.

Czym jest "kształt" rekursji (i indukcji)? Jest to raczej poetyckie pojęcie, które odnosi się do tego, jak zdefiniowano funkcję - ile jest przypadków, podprzypadków, podprzypadków etc., w jaki sposób są w sobie zagnieżdżone, gdzie są wywołania rekurencyjne, ile ich jest i na jakich argumentach etc.

Dowiedziawszy się, czym jest kształt rekursji i indukcji, powinniśmy zacząć szukać sposobu na dopasowanie kształtu indukcji w naszych dowodach do kształtu rekursji funkcji. Dotychczas indukcję zawsze robiliśmy po argumencie głównym, zaś z potencjalnymi niedopasowaniami kształtów radziliśmy sobie robiąc ad hoc analizy przypadków, które uznaliśmy za stosowne.

I tutaj przyda nam się nieco konceptualnej spostrzegawczości. Zauważyć nam bowiem trzeba, że robiąc indukcję po argumencie głównym, kształt indukcji odpowiada kształtowi typu argumentu głównego. Skoro zaś mamy dopasować go do kształtu rekursji funkcji, to nasuwa nam się oczywiste pytanie: czy da się zdefiniować typ, który ma taki sam kształt, jak definicja danej funkcji?

Odpowiedź brzmi: nie, ale da się zdefiniować rodzinę typów (a konkretniej pisząc, rodzinę zdań, czyli relację) o takiej właściwości. Owa relacja zwie się wykresem funkcji. Jaki ma to związek z bazgrołami znanymi ci ze szkoły (zakładam, że wiesz, że wykresem funkcji liniowej jest prosta, wykresem funkcji kwadratowej jest parabola, a wykresy sinusa i cosinusa to takie wesołe szlaczki)?

To, co w szkole nazywa się wykresem funkcji, jest jedynie graficznym przedstawieniem prawdziwego wykresu, czyli relacji. Samo słowo "wykres", wywodzące się w oczywisty sposób od kreślenia, sugeruje, że myślenie o wykresie jak o obrazku było pierwsze, a koncepcja wykresu jako relacji jest późniejsza.

W ramach ciekawostki być może warto napisać, że w dawnych czasach matematycy silnie utożsamiali funkcję z jej wykresem (w sensie obrazka) i przez to byty, których wykresu nie dało się narysować, nie były uznawane za funkcje.

W nieco późniejszym czasie zaszły jednak niemałe zmiany i obecnie panującym zabobonem jest utożsamianie funkcji z wykresem (w sensie relacji), przez co za funkcje uznawane są także byty, których nie da się obliczyć lub nikt nie potrafi pokazać, że terminują (takich jak np. "funkcja" Collatza).

Gdybyś zgłupiał od powyższych czterech akapitów, to przypominam, że dla nas zawarte w nich pojęcia oznaczają to:

- Funkcja to byt, którego typem jest $A \to B$ lub $\forall x: A, B x$. Można dać jej coś na wejściu i uzyskać wynik na wyjściu, tzn. można ją obliczyć. W Coqu wszystkie funkcje prędzej czy później kończą się obliczać.
- Wykres funkcji to relacja opisująca związek argumentu funkcji z jej wynikiem. Każda funkcja ma wykres, ale nie każda relacja jest wykresem jakiejś funkcji.
- \bullet Jeżeli typy A i B da się jakoś sensownie narysować, to możemy narysować obrazek przedstawiający wykres funkcji.

Definition is_graph

```
\{A\ B: \mathtt{Type}\}\ (f:A \to B)\ (R:A \to B \to \mathtt{Prop}): \mathtt{Prop}:= \ \forall\ (a:A)\ (b:B),\ R\ a\ b \leftrightarrow f\ a=b.
```

Žeby było nam raźniej, tak wygląda formalna definicja stwierdzenia, że relacja R jest wykresem funkcji f. Uwaga: jeżeli funkcja bierze więcej niż jeden argument (tzn. ma typ $A1 \rightarrow \dots \rightarrow An \rightarrow B$), to wtedy do powyższej definicji musimy wrzucić jej zmodyfikowaną wersję o typie $A1 \times \dots \times An \rightarrow B$.

Ćwiczenie Zdefiniuj funkcję $graph_of$, która każdej funkcji przyporządkowuje jej wykres. Następnie udowodnij, że faktycznie jest to wykres tej funkcji.

```
Lemma is\_graph\_graph\_of:

\forall (A \ B : \texttt{Type}) \ (f : A \rightarrow B),

is\_graph \ f \ (graph\_of \ f).
```

Ćwiczenie Wymyśl typy A i B oraz relację o typie $A \to B \to \mathsf{Prop}$, która nie jest wykresem żadnej funkcji. Następnie udowodnij formalnie, że nie mylisz się.

Ćwiczenie Pokaż, że wszystkie wykresy danej funkcji są równoważne w poniższym sensie.

Lemma $graph_unique$:

```
 \forall \{A \ B : \mathtt{Type}\} \ (f : A \to B) \ (R \ S : A \to B \to \mathtt{Prop}), \\ is\_graph \ f \ R \to is\_graph \ f \ S \to \\ \forall \ (a : A) \ (b : B), \ R \ a \ b \leftrightarrow S \ a \ b.
```

Skoro już wiemy czym są wykresy funkcji, czas nauczyć się definiować induktywne wykresy o kształtach odpowiednich dla naszych niecnych celów.

Zwróćmy tylko uwagę na fakt, że mówiąc o kształcie rekursji (lub po prostu o kształcie definicji) div nie mamy na myśli faktycznej definicji, która używa rekursji dobrze ufundowanej i jak już wiemy, jest dość problematyczna, lecz "docelowej" definicji, którą wyraża między innymi równanie rekurencyjne.

```
\begin{array}{l} \textbf{Inductive} \ divG : nat \rightarrow nat \rightarrow nat \rightarrow \texttt{Prop} := \\ \mid \ divG\_lt : \forall \ \{n \ m : nat\}, \ n < S \ m \rightarrow divG \ n \ m \ 0 \\ \mid \ divG\_ge : \\ \forall \ n \ m \ r : nat, \\ n \geq S \ m \rightarrow divG \ (n - S \ m) \ m \ r \rightarrow divG \ n \ m \ (S \ r). \end{array}
```

div jest funkcją typu $nat \to nat \to nat$, więc jej wykres to relacja typu $nat \to nat \to nat \to Prop$. Dwa pierwsze argumenty relacji reprezentują wejście, zaś ostatni argument reprezentuje wyjście, tzn. chcemy, żeby divG n m r było równoważne div n m = r.

Z równania rekurencyjnego widać, że mamy dwa przypadki, czyli konstruktory też będą dwa. Jeden odpowiada przypadkowi, gdy n < S m, tzn. dzielna jest mniejsza niż dzielnik (pamiętaj, że div n m oznacza n/(m+1), żeby uniknąć problemów z dzieleniem przez zero). Konkluzją jest wtedy divG n m 0, tzn. argumentami są n i m, zaś wynikiem jest 0.

Drugi przypadek to przyadek rekurencyjny. Jeżeli $n \geq S$ m, tzn. dzielna jest większa lub równa od dzielnika, to konkluzją jest divG n m (S r), tzn. argumentami są n i m, zaś wynikiem dzielenia jest S r. Czym jest r? Jest ono skwantyfikowane w tym konstruktorze

i pojawia się w przesłance divG (n - S m) m r, która mówi, że wynikiem dzielenia n - S m przez m jest r. Przesłanka ta jest wykresowym odpowiednikiem wywołania rekurencyjnego.

Ćwiczenie Mimo, że wszystkie wykresy danej funkcji są równoważne, to zdefiniować można je na wiele różnych sposobów. W zależności od sposobu definicja może być użyteczna lub nie, np. przy definicjach induktywnych dostajemy za darmo regułę indukcji.

Podaj inną definicję wykresu funkcji div, która nie używa typów induktywnych (ani nie odwołuje się do samej funkcji div - to byłoby za łatwe). Użyj kwantyfikatora egzystencjalnego, mnożenia, dodawania oraz relacji równości (i niczego więcej). Nazwij ją divG'.

Na razie nie musisz dowodzić, że wykres faktycznie jest wykresem div (póki co jest to za trudne), co oczywiście nie znaczy, że wolno ci się mylić - uzasadnij nieformalnie, że wykres faktycznie opisuje funkcję div. Do dowodu formalnego wrócimy później.

Mamy wykres. Fajnie, ale co możemy z nim zrobić? Jeszcze ważniejsze pytanie brzmi zaś: co powinniśmy z nim zrobić?

```
Lemma divG\_det: \forall n \ m \ r1 \ r2 : nat, divG \ n \ m \ r1 \rightarrow divG \ n \ m \ r2 \rightarrow r1 = r2. Proof. intros until 1. revert \ r2. induction H; inversion 1; subst. reflexivity. 1-2: lia. f_equal. apply IHdivG. assumption. Qed.
```

Pierwsza czynność po zdefiniowaniu wykresu, którą powinniśmy wykonać, to sprawdzenie, czy ów wykres jest relacją deterministyczną. Relacja deterministyczna to taka, której ostatni argument jest zdeterminowany przez poprzednie.

Jeżeli wykres jest deterministyczny to dobrze, a jeżeli nie, to definicja na pewno jest błędna, bo wykres ma opisywać funkcję, a żadna funkcja nie może dla tych samych argumentów dawać dwóch różnych wyników. Relacjom deterministycznym (i nie tylko) przyjrzymy się dokładniej w rozdziale o relacjach.

Dowód nie jest zbyt trudny. Robimy indukcję po dowodzie hipotezy divG n m r1, ale musimy pamiętać, żeby wcześniej zgeneralizować r2, bo w przeciwnym przypadku nasza hipoteza indukcyjna będzie za mało ogólna.

```
Lemma divG\_correct: \forall \ n \ m : nat, \\ divG \ n \ m \ (div \ n \ m).

Proof.

apply (well\_founded\_ind \_\_ wf\_lt \ (fun \_ \Rightarrow \forall \ m : nat, \_)).

intros n \ IH \ m.

rewrite div\_eq. destruct (Nat.ltb\_spec0 \ n \ (S \ m)).
```

```
constructor. assumption. constructor. lia. apply IH.\ lia. Qed.
```

Kolejna rzecz do udowodnienia to twierdzenie o poprawności, które mówi, że divG faktycznie jest wykresem div. Zauważ, że moglibyśmy równie dobrze sformułować je za pomocą is_graph , ale tak jak wyżej będzie praktyczniej.

Dowód zaczynamy od indukcji dobrze ufundowanej, po czym wprowadzamy zmienne do kontekstu i... aj waj, cóż to takiego? Używamy równania rekurencyjnego do rozpisania div, po czym kończymy przez rozważenie przypadków.

Ten dowód pokazuje, że nie udało nam się osiągnąć celu, który sobie postawiliśmy, czyli udowodnienia div_-eq za pomocą specjalnej reguły indukcji. Niestety, bez równania rekurencyjnego nie da się udowodnić twierdzenia o poprawności. Nie powinniśmy jednak za bardzo się tym przejmować - uszy do góry. Póki co dokończmy poszukiwań ostatecznej reguły indukcji, a tym nieszczęsnym równaniem rekurencyjnym zajmiemy się później.

```
Lemma divG\_complete:
\forall \ n \ m \ r: nat,
divG \ n \ m \ r \rightarrow r = div \ n \ m.

Proof.
intros. apply divG\_det with n \ m.
assumption.
apply divG\_correct.

Qed.
```

Kolejną, ostatnią już rzeczą, którą powinniśmy zrobić z wykresem, jest udowodnienie twierdzenia o pełności, które głosi, że jeżeli argumentom n i m odpowiada na wykresie wynik r, to r jest równe $div\ n\ m$. Dowód jest banalny i wynika wprost z twierdzeń o determinizmie i poprawności.

I po co nam to było? Ano wszystkie fikołki, które zrobiliśmy, posłużą nam jako lematy do udowodnienia reguły indukcji wykresowej dla div. Co to za reguła, jak wygląda i skąd ją wziąć?

```
Check divG_ind.
(* ===>
  divG_ind :
    forall
    P : nat -> nat -> nat -> Prop,
        (forall n m : nat, n < S m -> P n m 0) ->
        (forall n m r : nat,
            n >= S m -> divG (n - S m) m r ->
            P (n - S m) m r -> P n m (S r)) ->
        forall n m r : nat, divG n m r -> P n m r *)
```

Pierwowzorem reguły indukcji wykresowej dla danej funkcji jest reguła indukcji jej wykresu. Reguła indukcji dla div to w sumie to samo co powyższa reguła, ale z r wyspecjalizowanym do div n m. Chcemy też pozbyć się niepotrzebnej przesłanki divG n m r (po podstawieniu za r ma ona postać divG n m (div n m), gdyż nie jest potrzebna - jest zawsze prawdziwa na mocy twierdzenia divG_correct.

```
Lemma div_{-}ind:
  \forall
      (P: nat \rightarrow nat \rightarrow nat \rightarrow Prop)
      (Hlt: \forall n \ m: nat, n < S \ m \rightarrow P \ n \ m \ 0)
     (Hge:
        \forall n m : nat,
            n \geq S \ m \rightarrow P \ (n - S \ m) \ m \ (div \ (n - S \ m) \ m) \rightarrow
               P \ n \ m \ (S \ (div \ (n - S \ m) \ m))),
        \forall n \ m : nat, P \ n \ m \ (div \ n \ m).
Proof.
   intros P Hlt Hge n m.
   apply divG_{-}ind.
      assumption.
      intros. apply divG_{-}complete in H0. subst. apply Hge; assumption.
      apply divG\_correct.
Qed.
```

Przydałaby się jednak także i filozoficzna interpretacja reguły. Pozwoli nam ona dowodzić zdań, które zależą od n m : nat i wyniku dzielenia, czyli div n m.

Są dwa przypadki, jak w docelowej definicji div. Gdy n < S m, czyli dzielna jest mniejsza od dzielnika, wystarczy udowodnić P n m 0, bo wtedy div n m wynosi 0. W drugim przypadku, czyli gdy $n \ge S$ m, wystarczy udowodnić P n m (S (div (n - S m) m) (bo taki jest wynik div n m dla $n \ge S$ m) przy założeniu, że P zachodzi dla n - S m, m oraz div (n - S m) m, bo takie są argumenty oraz wynik wywołania rekurencyjnego.

Dowód jest prosty. Wprowadzamy zmienne do kontekstu, a następnie za pomocą zwykłego apply używamy reguły indukcji $divG_{-}ind$ - jako rzekło się powyżej, reguła $div_{-}ind$ nie jest niczym innym, niż lekką przeróbką $divG_{-}ind$.

Mamy trzy podcele. Pierwszy odpowiada przesłance Hlt. Drugi to przesłanka Hge, ale musimy wszędzie podstawić div (n' - S m') m' za r - posłuży nam do tego twierdzenie o pełności. Trzeci to zbędna przesłanka divG n m (div n m), którą załatwiamy za pomocą twierdzenia o poprawności.

Włala (lub bardziej wykwintnie: voilà)! Mamy regułę indukcji wykresowej dla div. Zobaczmy, co i jak można za jej pomocą udowodnić.

```
\label{eq:lemma_div_le} \begin{split} &\forall \ n \ m: \ nat, \\ & \ div \ n \ m \leq n. \end{split} Proof. apply (div\_ind \ (\text{fun } n \ m \ r: \ nat \Rightarrow r \leq n)); \ \text{intros}. \end{split}
```

```
\label{eq:liable_lia} \underset{lia.}{\text{apply}}\ le_-\theta_-n. Qed.
```

Ćwiczenie Udowodnij twierdzenie $div_{-}le$ za pomocą indukcji dobrze ufundowanej i równania rekurencyjnego, czyli bez użycia indukcji wykresowej. Jak trudny jest ten dowód w porównaniu do powyższego?

```
Lemma div_{-}le':

\forall n \ m : nat,

div \ n \ m \leq n.
```

Ćwiczenie Udowodnij za pomocą indukcji wykresowej, że twój alternatywny wykres funkcji div z jednego z poprzednich ćwiczeń faktycznie jest wykresem div.

Następnie udowodnij to samo za pomocą indukcji dobrze ufundowanej i równania rekurencyjnego. Która metoda dowodzenia jest lepsza (nie, to pytanie nie jest subiektywne - masz udzielić jedynej słusznej odpowiedzi).

```
 \begin{array}{l} \text{Lemma } divG'\_div: \\ \forall \ n \ m: nat, \\ divG' \ n \ m \ (div \ n \ m). \\ \\ \text{Lemma } divG'\_div': \\ \forall \ n \ m: nat, \\ divG' \ n \ m \ (div \ n \ m). \end{array}
```

Čwiczenie Napisz funkcję split o sygnaturze split (n:nat) $\{A: Type\}$ (l:list A): option $(list A \times list A)$, która rozdziela listę l na blok o długości n i resztę listy, lub zwraca None gdy lista jest za krótka.

Następnie udowodnij dla tej funkcji regułę indukcji wykresowej i użyj jej do udowodnienia kilku lematów.

Wszystkie te rzeczy przydadzą się nam w jednym z kolejnych zadań.

```
Definition lengthOrder \{A: {\tt Type}\}\ (l1\ l2: list\ A): {\tt Prop}:= length\ l1 < length\ l2.}
Lemma wf\_lengthOrder:
\forall\ A: {\tt Type},\ well\_founded\ (@lengthOrder\ A).
Proof.
intros. apply (wf\_inverse\_image\_\_\ (@length\ A)). apply wf\_lt.
Defined.
Lemma lengthOrder\_split\_aux:
\forall\ \{A: {\tt Type}\}\ (n:nat)\ (l:list\ A)\ (l1\ l2:list\ A),
split n\ l=Some\ (l1,l2)\to n=0\ \lor\ lengthOrder\ l2\ l.
```

```
Lemma lengthOrder\_split:

\forall (n:nat) (A: Type) (l: list A) (l1 l2: list A),

split (S n) l = Some (l1, l2) \rightarrow lengthOrder l2 l.
```

5.6 Metoda induktywnej dziedziny

Póki co nie jest źle - udało nam się wszakże wymyślić jedyną słuszną metodę dowodzenia właściwości funkcji rekurencyjnych. Jednak nasza implementacja kuleje przez to nieszczęsne równanie rekurencyjne. Jak możemy udowodnić je bez używania indukcji wykresowej?

Zeby znaleźć odpowiedź na to pytanie, znowu przyda się nam trochę konceptualnej jasności. Na czym tak naprawdę polega problem? Jak pamiętamy, problem wynika z tego, że definiując div przez rekursję dobrze ufundowaną musieliśmy jednocześnie dowodzić, że wywołania rekurencyjne odbywają się na argumencie mniejszym od argumentu obecnego wywołania.

Tak więc problemem jest połączenie w jednej definicji dwóch dość luźno powiązanych rzeczy, którymi są:

- Docelowa definicja, która określa obliczeniowe zachowanie funkcji. Jej manifestacją jest nasze nieszczęsne równanie rekurencyjne. Bywa ona czasem nazywana aspektem obliczeniowym (albo algorytmicznym) funkcji.
- Dowód terminacji, który zapewnia, że definicja docelowa jest legalna i nie prowadzi do sprzeczności. Jego manifestacją są występujące w definicji div dowody na to, że wywołanie rekurencyjne ma argument mniejszy od obecnego wywołania. Bywa on czasem nazywany aspektem logicznym funkcji.

Pani doktur, mamy diagnozę! Tylko co z nią zrobić? Czy jest jakaś metoda, żeby rozdzielić obliczeniowy i logiczny aspekt danej funkcji, a potem poskładać je do kupy?

Pomyślmy najpierw nad aspektem obliczeniowym. Czy da się zdefiniować funkcję bezpośrednio za pomocą jej definicji docelowej, czyli równania rekurencyjnego? Żeby to zrobić, musielibyśmy mieć możliwość robienia rekursji o dokładnie takim kształcie, jaki ma mieć ta funkcja...

Eureka! Przecież mamy coś, co pozwala nam na rekursję o dokładnie takim kształcie, a mianowicie induktywny wykres! Ale przecież wykres wiąże ze sobą argumenty i wynik, a my chcemy dopiero zdefiniować coś, co ów wynik obliczy... czyli nie eureka?

Nie do końca. Możemy zmodyfikować definicję wykresu, wyrzucając z niej wszystkie wzmianki o wyniku, uzyskując w ten sposób predykat będący induktywną charakteryzacją dziedziny naszej funkcji. Dzięki niemu możemy zdefiniować zmodyfikowaną wersję funkcji, w której dodatkowym argumentem jest dowód na to, że argumenty należą do dziedziny.

Logiczny aspekt funkcji, czyli dowód terminacji, sprowadza się w takiej sytuacji do pokazania, że wszystkie argumenty należą do dziedziny (czyli spełniają predykat dziedziny). Żeby zdefiniować oryginalną funkcję, wystarczy jedynie poskładać oba aspekty do kupy, czyli wstawić dowód terminacji do zmodyfikowanej funkcji. Żeby nie utonąć w ogólnościach, zobaczmy, jak nasz wspaniały wynalazek radzi sobie z dzieleniem.

```
\begin{array}{l} \textbf{Inductive} \ divD: nat \rightarrow nat \rightarrow \texttt{Type} := \\ \mid divD\_lt : \forall \ n \ m: nat, \ n < S \ m \rightarrow divD \ n \ m \\ \mid divD\_ge : \\ \forall \ n \ m: nat, \\ n > S \ m \rightarrow divD \ (n \text{-} S \ m) \ m \rightarrow divD \ n \ m. \end{array}
```

Tak wygląda predykat dziedziny dla dzielenia. Zauważmy, że tak naprawdę to nie jest to predykat, bo bierze dwa argumenty i co więcej nie zwraca Prop, lecz Type. Nie będziemy się tym jednak przejmować - dla nas divD będzie "predykatem dziedziny". Zauważmy też, że nie jest to predykat dziedziny dla div, lecz dla div, czyli zupełnie nowej funkcji, którą zamierzamy zdefiniować.

Ok, przejdźmy do konkretów. div' ma mieć typ $nat \to nat \to nat$, a zatem divD ma dwa indeksy odpowiadające dwóm argumentom div'. Pierwszy konstruktor głosi, że jeżeli n < S m, to oba te argumenty należą do dziedziny (bo będziemy chcieli w tym przypadku zwrócić 0). Drugi konstruktor głosi, że jeżeli $n \geq S$ m, to para argumentów n i m należy do dziedziny pod warunkiem, że para argumentów n - S m i m należy do dziedziny. Jest tak, gdyż w tym przypadku będziemy chcieli zrobić wywołanie rekurencyjne właśnie na n - S m oraz m.

```
Fixpoint div'\_aux \{n\ m: nat\} (H: divD\ n\ m): nat:= match H with  |\ divD\_lt\ \_\_\_ \Rightarrow 0 \\ |\ divD\_ge\ \_\_\_ H' \Rightarrow S\ (div'\_aux\ H')  end.
```

Dzięki divD możemy zdefiniować funkcję div'_aux , której typem jest $\forall \ n \ m: nat, \ divD \ n \ m \to nat$. Jest to funkcja pomocnicza, która posłuży nam do zdefiniowania właściwej funkcji div'.

Ponieważ divD jest zdefiniowane induktywnie, docelowa definicja div' jest strukturalnie rekurencyjna po argumencie H:divD n m, mimo że nie jest strukturalnie rekurencyjna po n ani m. To właśnie jest magia stojąca za metodą induktywnej dziedziny - możemy sprawić, żeby każda (no, prawie), nawet najdziwniejsza rekursja była strukturalnie rekurencyjna po dowodzie należenia do dziedziny.

Definicja jest banalna. Gdy natrafimy na konstruktor $divD_-lt$, zwracamy 0 (bo wiemy, że jednym z argumentów $divD_-lt$ jest dowód na to, że n < S m). Jeżeli trafimy na $divD_-ge$, to wiemy, że $n \ge S$ m, więc robimy wywołanie rekurencyjne na H': divD (n - S m) m i dorzucamy do wyniku S.

W ten sposób zdefiniowaliśmy obliczeniową część div', zupełnie nie przejmując się kwestią terminacji.

```
Lemma divD_-all:
 \forall n \ m : nat, \ divD \ n \ m.
Proof.
```

```
\begin{array}{l} \operatorname{apply}\ (well\_founded\_rect\ nat\ lt\ wf\_lt\ (\operatorname{fun}\ \_\Rightarrow\ \forall\ m:\ nat,\ \_)).\\ \operatorname{intros}\ n\ IH\ m.\\ \operatorname{destruct}\ (le\_lt\_dec\ (S\ m)\ n).\\ \operatorname{apply}\ divD\_ge.\\ \operatorname{unfold}\ ge.\ \operatorname{assumption}.\\ \operatorname{apply}\ IH.\ \operatorname{abstract}\ lia.\\ \operatorname{apply}\ divD\_lt.\ \operatorname{assumption}.\\ \\ \operatorname{Defined}. \end{array}
```

Dowód terminacji jest bliźniaczo podobny do naszej pierwszej definicji div. Zaczynamy przez rekursję dobrze ufundowaną z porządkiem lt (i dowodem wf_-lt na to, że lt jest dobrze ufundowany), wprowadzamy zmienne do kontekstu, po czym sprawdzamy, który z przypadków zachodzi.

Jeżeli $n \geq S$ m, używamy konstruktora $divD_ge$. $n \geq S$ m zachodzi na mocy założenia, zaś n - S m i m należą do dziedziny na mocy hipotezy indukcyjnej. Gdy n < S m, n i m należą do dziedziny na mocy założenia.

```
Definition div' (n m : nat) : nat := div'_aux (divD_all n m).
```

A oto i ostateczna definicja - wstawiamy dowód $divD_-all$ do funkcji pomocniczej div'_-aux i uzyskujemy pełnoprawną funkcję dzielącą $div': nat \rightarrow nat \rightarrow nat$.

```
Compute div' 666 7.
(* ===> = 83 : nat *)
```

Jak widać, wynik oblicza się bez problemu. Po raz kolejny przypominam, że div' n m oblicza n/(m+1), nie zaś n/m. Przypominam też, że dowód $divD_-all$ koniecznie musimy zakończyć za pomocą komendy Defined, a nie jak zazwyczaj Qed, gdyż w przeciwnym przypadku funkcja div' nie mogłaby niczego obliczyć.

```
Lemma divG\_div'\_aux:
\forall \ (n\ m:\ nat)\ (d:\ divD\ n\ m),
divG\ n\ m\ (div'\_aux\ d).
Proof.
induction d;\ cbn; constructor; assumption.
Qed.

Lemma divG\_correct':
\forall \ n\ m:\ nat,
divG\ n\ m\ (div'\ n\ m).
Proof.
intros. apply divG\_div'\_aux.
Qed.
```

Żeby udowodnić regułę indukcji wykresowej, będziemy potrzebowali tego samego co poprzednio, czyli twierdzeń o poprawności i pełności funkcji div' względem wykresu divG. Dowody są jednak dużo prostsze niż ostatnim razem.

Najpierw dowodzimy, że funkcja pomocnicza div'_-aux oblicza taki wynik, jakiego spodziewa się wykres divG. Dowód jest banalny, bo indukcja po d:divD n m ma dokładnie taki kształt, jakiego nam potrzeba. Właściwy dowód dla div' uzyskujemy przez wyspecjalizowanie $divG_-div'_-aux$ do div'.

```
Lemma divG\_complete':
  \forall n m r : nat,
     divG \ n \ m \ r \rightarrow r = div' \ n \ m.
Proof.
  intros. apply divG_{-}det with n m.
     assumption.
     apply divG\_correct'.
Qed.
Lemma div'_{-}ind:
     (P: nat \rightarrow nat \rightarrow nat \rightarrow Prop)
     (Hlt: \forall n \ m: nat, n < S \ m \rightarrow P \ n \ m \ 0)
     (Hge:
        \forall n m : nat, n \geq S m \rightarrow
           P(n-Sm)m(div'(n-Sm)m) \rightarrow
             P \ n \ m \ (S \ (div' \ (n - S \ m) \ m))),
        \forall n \ m : nat, P \ n \ m \ (div' \ n \ m).
Proof.
  intros P Hlt Hge n m.
  apply divG_{-}ind.
     assumption.
     intros. apply divG_{-}complete' in H0. subst. apply Hge; assumption.
     apply divG\_correct'.
Qed.
```

Dowód pełności i dowód reguły indukcji wykresowej są dokładnie takie same jak poprzednio. Zauważ, że tym razem zupełnie zbędne okazało się równanie rekurencyjne, bez którego nie mogliśmy obyć się ostatnim razem. Jednak jeżeli chcemy, możemy bez problemu je udowodnić, i to nawet na dwa sposoby.

```
Lemma div'\_eq: \forall \ n \ m: nat, div' \ n \ m = \text{if} \ n <? \ S \ m \ \text{then} \ 0 \ \text{else} \ S \ (div' \ (n - S \ m) \ m). Proof. intros. unfold div'. generalize (divD\_all \ n \ m) as d. induction d; cbn. rewrite leb\_correct. reflexivity. apply le\_S\_n. assumption.
```

```
rewrite leb\_correct\_conv.

f\_equal. apply divG\_det with (n-S \ m) m; apply divG\_div'\_aux. assumption.

Restart.

intros. apply div'\_ind; clear n m; intros; cbn.

rewrite leb\_correct.

reflexivity.

abstract lia.

rewrite leb\_correct\_conv.

reflexivity.

abstract lia.

Qed.
```

Pierwszy, trudniejszy sposób, to zgeneralizowanie $divD_-all\ n\ m$ do dowolnego d oraz indukcja po d (to tak, jakbyśmy najpierw udowodnili tę regułę dla div'_-aux , a potem wyspecjalizowali do div').

Drugi, łatwiejszy sposób, realizuje nasz początkowy pomysł, od którego wszystko się zaczęło: dowodzimy równania rekurencyjnego za pomocą reguły indukcji wykresowej.

Ćwiczenie Zdefiniuj funkcję rot, która bierze liczbę n oraz listę i zwraca listę, w której bloki o długości dokładnie n+1 zostały odwrócone, np.

```
rot \ 0 \ [1; \ 2; \ 3; \ 4; \ 5; \ 6; \ 7] = [1; \ 2; \ 3; \ 4; \ 5; \ 6; \ 7]

rot \ 1 \ [1; \ 2; \ 3; \ 4; \ 5; \ 6; \ 7] = [2; \ 1; \ 4; \ 3; \ 6; \ 5; \ 7]

rot \ 2 \ [1; \ 2; \ 3; \ 4; \ 5; \ 6; \ 7] = [3; \ 2; \ 1; \ 6; \ 5; \ 4; \ 7]
```

Wskazówka: rzecz jasna użyj metody induktywnej dziedziny. Nie bez przyczyny także w jednym z poprzednich zadań kazałem ci zdefiniować funkcję split, która odkraja od listy blok o odpowiedniej długości.

Następnie zdefiniuj wykres funkcji rot i udowodnij jej regułę indukcji wykresowej oraz równanie rekurencyjne. Użyj jej, żeby pokazać, że rot jest inwolucją dla dowolnego n, tzn. rot n (rot n l) = l. Uwaga: potrzebne będzie trochę lematów.

5.7 Komenda Function

Odkryliśmy uniwersalną metodę definiowania funkcji i dowodzenia ich właściwości. Czego chcieć więcej?

Po pierwsze, metoda definiowania nie jest uniwersalna (jeszcze), o czym przekonamy się w kolejnych podrozdziałach. Po drugie, mimo że metoda dowodzenia faktycznie jest uniwersalna, to komu normalnemu chciałoby się przy każdej funkcji tyle pisać? Jakieś wykresy, dziedziny, lematy, reguły indukcji, co to ma być?

Czy w celu sprawnego definiowania i dowodzenia właściwości funkcji trzeba zoutsourcować cały proces i zatrudnić milion Hindusów? Na szczęście nie, gdyż bóg dał nam komendę Function.

Require Import Recdef.

Komenda ta żyje w module *Recdef*, którego nazwa jest skrótem od słów "recydywista defraudator"... dobra, koniec żartów.

Definicja zaczyna się od słowa kluczowego Function, następnie mamy nazwę funkcji i argumenty, tak jak w zwykłych definicjach za pomocą Definition czy Fixpoint, a później tajemniczą klauzulę $\{\text{measure } id \ n\}$, do której zaraz wrócimy, i zwracany typ. Ciało definicji wygląda dokładnie jak docelowa definicja.

Jednak po kropce definicja nie kończy się - zamiast tego Coq każe nam udowodnić, że wywołanie rekurencyjne div" odbywa się na argumencie mniejszym niż n. Po zakończeniu dowodu funkcja zostaje zaakceptowana przez Coqa.

To jednak nie koniec. Komenda Function nie tylko pozwala bezboleśnie zdefiniować div", ale też generuje dla nas całą masę różnych rzeczy:

- $div''_{-}tcc$ to lemat, który mówi, że wszystkie wywołania rekurencyjne są na argumencie mniejszym od obecnego
- div''_terminate to dowód tego, że funkcja terminuje (czyli że się nie zapętla). Jeżeli przyjrzysz się jego typowi, to zobaczysz, że jest podobny zupełnie do niczego. Wynika to z faktu, że komenda Function tak naprawdę nie używa metody induktywnej dziedziny, ale pewnej innej metody definiowania funkcji ogólnie rekurencyjnych. Nie powinno nas to jednak martwić ważne, że działa.
- div''_{-ind} to reguła indukcji wykresowej dla div''. Jest też jej wariant div''_{-rect} , czyli "rekursja wykresowa", służąca raczej do definiowania niż dowodzenia.
- R_-div '' to induktywnie zdefiniowany wykres funkcji div''. Zauważ jednak, że nie jest on relacją, a rodziną typów nie wiem po co i nie ma co wnikać w takie detale.
- R_div''_correct to twierdzenie o poprawności wykresu.
- R_div''_complete to twierdzenie o pełności wykresu.

• div''_equation to równanie rekurencyjne

Jak więc widać, nastąpił cud automatyzacji i wszystko robi się samo. To jednak nie koniec udogodnień. Zobaczmy, jak możemy udowodnić jakiś fakt o div".

```
Lemma div''_le:

\forall \ n \ m : nat, \ div'' n \ m \leq n.

Proof.

intros. functional induction (div'' n \ m).

apply le_-\theta_-n.

apply leb_-complete_-conv in e.\ lia.

Defined.
```

Dowodzenie właściwości funkcji zdefiniowanych za pomocą Function jest bajecznie proste. Jeżeli wszystkie argumenty funkcji znajdują się w kontekście, to możemy użyć taktyki functional induction (nazwa_funkcji argument_1 ... argument_n), która odpala indukcję wykresową dla tej funkcji. Z powodu nazwy tej taktyki indukcja wykresowa bywa też nazywana indukcją funkcyjną.

Wujek Dobra Rada: nigdy nie odwijaj definicji funkcji zdefiniowanych za pomocą Function ani nie próbuj ręcznie aplikować reguły indukcji wykresowej, bo skończy się to jedynie bólem i zgrzytaniem zębów.

Na koniec wypadałoby jedynie dodać, że wcale nie złapaliśmy pana boga za nogi i komenda Function nie rozwiązuje wszystkich problemów pierwszego świata. W szczególności niektóre funkcje mogą być tak upierdliwe, że komenda Function odmówi współpracy z nimi. Radzeniu sobie z takimi ciężkimi przypadkami poświęcimy kolejne podrozdziały.

Ćwiczenie Zdefiniuj funkcję *rot* (i wszystkie funkcje pomocnicze) jeszcze raz, tym razem za pomocą komendy Function. Porównaj swoje definicje wykresu oraz reguły indukcji z tymi automatycznie wygenerowanymi. Użyj taktyki *functional* induction, żeby jeszcze raz udowodnić, że *rot* jest inwolucją. Policz, ile pisania udało ci się dzięki temu zaoszczędzić.

Czy w twoim rozwiązaniu są lematy, w których użycie indukcji funkcyjnej znacznie utrudnia przeprowadzenie dowodu? W moim poprzednim rozwiązaniu był jeden taki, ale wynikał z głupoty i już go nie ma.

5.8 Rekursja zagnieżdżona

Jakież to diabelstwo może być tak diabelskie, by przeciwstawić się metodzie induktywnej dziedziny oraz komendzie Function? Ano ano, rekursja zagnieżdżona - wywołanie rekurencyjne jest zagnieżdżone, jeżeli jego argumentem jest wynik innego wywołania rekurencyjnego.

```
Module McCarthy. 
 Fail Fixpoint f (n:nat):nat:= if 100 < ? n then n - 10 else f (f (n+11)).
```

```
Fail Function f(n:nat) {measure id n}: nat := if 100 <? n then <math>n-10 else f(f(n+11)).
```

Ta funkcja jest podobna zupełnie do niczego, co dotychczas widzieliśmy. Działa ona następująco:

- \bullet jeżeli n jest większe od 100, to zwróć n 10
- \bullet w przeciwnym wypadku wywołaj rekurencyjnie f na n+11, a następnie wywołaj f na wyniku tamtego wywołania.

Taka rekursja jest oczywiście nielegalna: n+11 nie jest strukturalnym podtermem n, gdyż jest od niego większe, zaś f (n+11) w ogóle nie wiadomo a priori, jak się ma do n. Nie dziwota więc, że Coq odrzuca powyższą definicję.

Być może wobec tego taka "funkcja" w ogóle nie jest funkcją, a definicja jest wadliwa? Otóż nie tym razem. Okazuje się bowiem, że istnieje funkcja zachowująca się zgodnie z zawartym w definicji równaniem. Żebyśmy mogli w to uwierzyć, zastanówmy się, ile wynosi f 100.

```
f 100 = f (f 111) = f 101 = 101 - 10 = 91 - poszło gładko. A co z 99? Mamy f 99 = f (f 110) = f 100 = 91 - znowu 91, czyżby spiseg? Dalej: f 98 = f (f 109) = f 99 = 91 - tak, to na pewno spiseg. Teraz możemy zwerbalizować nasze domysły: jeżeli n \le 100, to f n = 91. Jak widać, nieprzypadkowo funkcja ta bywa też nazywana "funkcją 91 McCarthy'ego".
```

Czy da się tę funkcję zaimplementować w Coqu? Pewnie!

```
Definition f_{-}troll\ (n:nat):nat:= if n<=? 100 then 91 else n - 10.
```

Ehhh... nie tego się spodziewałeś, prawda? $f_{-}troll$ jest wprawdzie implementacją opisanej powyżej nieformalnie funkcji f, ale definicja opiera się na tym, że z góry wiemy, jaki jest wynik f dla dowolnego argumentu. Nie trzeba chyba tłumaczyć, że dla żadnej ciekawej funkcji nie będziemy posiadać takiej wiedzy (a sama funkcja McCarthy'ego nie jest ciekawa, bo jest sztuczna, ot co!).

Czy więc da się zaimplementować f bezpośrednio, tzn. w sposób dokładnie oddający definicję nieformalną? Otóż tak, da się i to w sumie niewielkim kosztem: wystarczy jedynie nieco zmodyfikować naszą metodę induktywnej dziedziny. Zanim jednak to zrobimy, zobaczmy, dlaczego nie obejdzie się bez modyfikacji.

A oto i źródło całego problemu. Jeżeli $n \leq 100$, to chcemy zrobić dwa wywołania rekurencyjne: jedno na n+11, a drugie na f(n+11). Wobec tego należenie tych dwóch

argumentów do dziedziny jest warunkiem należenia n do dziedziny i stąd postać całego konstruktora.

Niestety, definicja jest zła - f(n + 11) nie jest poprawnym termem, gdyż f nie jest jeszcze zdefiniowane. Mamy więc błędne koło: żeby zdefiniować f, musimy zdefiniować predykat dziedziny fD, ale żeby zdefiniować fD, musimy zdefiniować f.

Jak wyrwać się z tego błędnego koła? Ratunek przychodzi ze strony być może nieoczekiwanej, ale za to już bardzo dobrze przez nas poznanej, a jest nim induktywna definicja wykresu. Tak tak - w definicji fD możemy (a nawet musimy) zastąpić wystąpienia f przez wystąpienia wykresu f.

Hej ho, po przykład by się szło.

```
\begin{array}{l} \text{Inductive } fG: nat \to nat \to \texttt{Prop} := \\ \mid fG\_gt100: \\ & \forall \; n: \; nat, \; 100 < n \to fG \; n \; (n \text{--} 10) \\ \mid fG\_le100: \\ & \forall \; n \; r1 \; r2: \; nat, \\ & n \leq 100 \to fG \; (n + 11) \; r1 \to fG \; r1 \; r2 \to fG \; n \; r2. \end{array}
```

Tak wygląda wykres funkcji f. Wywołanie rekurencyjne f (f (n+11) możemy zareprezentować jako dwa argumenty, mianowicie fG (n+11) r1 i fG r1 r2. Dosłownie odpowiada to wywołaniu rekurencyjnemu w stylu let r1 := f (n+11) in let r2 := f r1 in r2.

```
Lemma fG\_det:
\forall n \ r1 \ r2: nat,
fG \ n \ r1 \to fG \ n \ r2 \to r1 = r2.

Proof.

intros until 1. revert \ r2.
induction H; intros r \ Hr.
inversion Hr; subst.
reflexivity.
abstract \ lia.
inversion Hr; subst.
abstract \ lia.
assert \ (r1 = r0) by apply (IHfG1 \ L3); subst.
apply \ (IHfG2 \ L4).

Defined.
```

Po zdefiniowaniu wykresu dowodzimy, podobnie łatwo jak poprzednio, że jest on relacja deterministyczna.

```
\begin{array}{l} \textbf{Inductive } fD: nat \rightarrow \texttt{Type} := \\ \mid fD\_gt100: \\ \forall \; n: \; nat, \; 100 < n \rightarrow fD \; n \\ \mid fD\_le100: \\ \forall \; n \; r: \; nat, \; n \leq 100 \rightarrow \\ fG \; (n+11) \; r \rightarrow fD \; (n+11) \rightarrow fD \; r \rightarrow fD \; n. \end{array}
```

A tak wygląda definicja predykatu dziedziny. Zamiast fD (f (n + 11)) mamy fD r, gdyż r na mocy argumentu fG (n + 11) r reprezentuje wynik wywołania rekurencyjnego f (n + 11).

```
Fixpoint f' {n : nat} (d : fD n) : nat := match d with  | fD_-gt100 \_\_\_ \Rightarrow n-10 \\ | fD_-le100 \_\_\_\_ d2 \Rightarrow f' d2  end.
```

Definicja funkcji pomocniczej f' może być nieco zaskakująca: gdzie podziało się zagnieżdżone wywołanie rekurencyjne? Nie możemy jednak dać się zmylić przeciwnikowi. Ostatnią klauzulę dopasowania do wzorca możemy zapisać jako | fD_-le100 n r H g d1 $d2 \Rightarrow f$ ' d2. Widzimy, że d2 jest typu fD r, ale g: fG (n+11) r, więc możemy myśleć, że r to tak naprawdę f (n+11), a zatem d2 tak naprawdę jest typu fD (f (n+11)). Jeżeli dodatkowo napiszemy wprost domyślny argument f', to wywołanie rekurencyjne miałoby postać @f' (@f' (n+11) d1) d2, a więc wszystko się zgadza. Żeby jednak nie rzucać słów na wiatr, udowodnijmy to.

```
Lemma f'_correct: \forall (n:nat) \ (d:fD\ n), fG\ n\ (f'\ d). Proof. induction d; cbn. constructor assumption. econstructor 2. assumption. exact IHd1. assert (r=f'\ d1). apply fG\_det with (n+11); assumption. subst. assumption.
```

Defined.

Dowód twierdzenia o poprawności jest tylko odrobinkę trudniejsze niż ostatnio, gdyż w przypadku wystąpienia w kontekście dwóch hipotez o typie fG (n+11) _ musimy użyć twierdzenia o determinizmie wykresu.

```
Lemma f'\_complete: \forall (n\ r: nat)\ (d: fD\ n), fG\ n\ r \to f'\ d = r. Proof. intros. apply fG\_det with n. apply f'\_correct. assumption. Defined. Dowód twierdzenia o pełności pozostaje bez zmian. Lemma fG\_le100\_spec:
```

```
\forall n \ r : nat,
    fG \ n \ r \to n < 100 \to r = 91.
Proof.
  induction 1; intro.
     abstract lia.
     inversion H\theta; subst.
       inversion H1; subst.
          assert (n = 100) by abstract lia. subst. reflexivity.
          abstract lia.
       abstract lia.
Defined.
Lemma f'_{-}le100:
  \forall (n : nat) (d : fD n),
     n \le 100 \to f' \ d = 91.
Proof.
  intros. apply fG\_le100\_spec with n.
     apply f' - correct.
     assumption.
Defined.
Lemma f'_- qe100:
  \forall (n : nat) (d : fD \ n),
    100 < n \rightarrow f' \ d = n - 10.
Proof.
  destruct d; cbn; abstract lia.
Defined.
```

Teraz następuje mały twist. Udowodnienie, że każdy argument spełnia fD będzie piekielnie trudne i będziemy w związku z tym potrzebować charakteryzacji funkcji f'. Zaczynamy więc od udowodnienia, że dla $n \leq 100$ wynikiem jest 91. Najpierw robimy to na wykresie, bo tak jest łatwiej, a potem transferujemy wynik na funkcję. Charakteryzację dla 100 < n dostajemy wprost z definicji.

```
Lemma fD\_all: \forall \ n: nat, fD \ n. Proof. apply (well\_founded\_ind \ \_ (\text{fun } n \ m \Rightarrow 101 \ - \ n < 101 \ - \ m)). apply wf\_inverse\_image. apply wf\_lt. intros n IH. destruct (le\_lt\_dec \ n \ 100). assert (d: fD \ (n+11)) by (\text{apply } IH; \ lia). apply fD\_le100 with (f'\ d). assumption. apply f'\_correct. assumption.
```

```
apply IH. inversion d; subst. rewrite f '_ge100. abstract lia. assumption. rewrite f '_le100; abstract lia. constructor. assumption.
```

Defined.

Dowód jest przez indukcję dobrze ufundowaną po n, a relacja dobrze ufundowana, której używamy, to fun n m: $nat \Rightarrow 101$ - n < 101 - m. Dlaczego akurat taka? Przypomnijmy sobie, jak dokładnie oblicza się funkcja f, np. dla 95:

```
f 95 = f (f 106) = f 96 = f (f 107) = f 97 = f (f 108) = f 98 = f (f 109) = f 99 = f (f 110) = f 100 = f (f 111) = f 101 = 91.
```

Jak więc widać, im dalej w las, tym bardziej zbliżamy się do magicznej liczby 101. Wyrażenie 101 - n mówi nam, jak blisko przekroczenia 101 jesteśmy, a więc 101 - n < 101 - m oznacza, że każde wywołanie rekurencyjne musi być bliżej 101 niż poprzednie wywołanie. Oczywiście zamiast 101 może być dowolna większa liczba - jeżeli zbliżamy się do 101, to zbliżamy się także do 1234567890.

Dowód dobrego ufundowania jest banalny, ale tylko pod warunkiem, że zrobiłeś wcześniej odpowiednie ćwiczenie. Jeszcze jedna uwaga: jak wymyślić relację dobrze ufundowaną, jeżeli jest nam potrzebna przy dowodzie takim jak ten? Mógłbym ci tutaj naopowiadać frazesów o... w sumie nie wiem o czym, ale prawda jest taka, że nie wiem, jak się je wymyśla. Tej powyższej wcale nie wymyśliłem sam - znalazłem ją w świerszczyku dla bystrzaków.

Dobra, teraz właściwa część dowodu. Zaczynamy od analizy przypadków. Drugi przypadek, gdy 100 < n, jest bardzo łatwy. W pierwszym zaś przypadku z hipotezy indukcyjnej dostajemy fD (n+11), tzn. n+11 należy do dziedziny. Skoro tak, to używamy konstruktora fD_-le100 , a jako r (czyli wynik wywołania rekurencyjnego) dajemy mu f' d.

Dwa podcele zachodzą na mocy założenia, a jedna wynika z twierdzenia o poprawności. Pozostaje nam zatem pokazać, że f ' d także należy do dziedziny. W tym celu po raz kolejny używamy hipotezy indukcyjnej. Na zakończenie robimy analizę przypadków po d, używamy charakteryzacji f ' do uproszczenia celu i kończymy rozumowaniami arytmetycznymi.

```
Definition f (n:nat):nat:=f' (fD_-all\ n). (* Compute f 101. *)
```

Teraz możemy zdefiniować oryginalne f. Niestety, funkcja f się nie oblicza i nie wiem nawet dlaczego.

```
Lemma f\_correct: \forall n: nat, fG \ n \ (f \ n). Proof. intros. apply f'\_correct. Qed. Lemma f\_complete: \forall n \ r: nat,
```

```
fG \ n \ r \to f \ n = r.
Proof.
  intros. apply f'\_complete. assumption.
Qed.
Lemma f_-91:
  \forall (n : nat),
     n < 100 \to f \ n = 91.
Proof.
  intros. apply f'_{-}le100. assumption.
Qed.
    Twierdzenia o poprawności i pełności oraz charakteryzacja dla f wynikają za darmo z
odpowiednich twierdzeń dla f'.
Lemma f_{-}ind:
  \forall
     (P: nat \rightarrow nat \rightarrow Prop)
     (H_{-}gt100 : \forall n : nat, 100 < n \rightarrow P \ n \ (n - 10))
     (H_{-}le100:
       \forall~n:~nat,~n\leq 100\rightarrow
          P\ (n+11)\ (f\ (n+11)) 
ightarrow P\ (f\ (n+11))\ (f\ (f\ (n+11))) 
ightarrow
             P \ n \ (f \ (n + 11))),
     \forall n : nat, P n (f n).
Proof.
  intros. apply fG_{-}ind.
     assumption.
     intros. apply f_{-}complete in H0. apply f_{-}complete in H2.
       subst. apply H_{-}le100; assumption.
     apply f_{-}correct.
Defined.
    Reguły indukcji wykresowej dowodzimy tak samo jak poprzednio, czyli za pomocą twier-
dzeń o pełności i poprawności.
Lemma f_{-}eq:
  \forall n : nat.
     f \ n =  if \ 100 < ? \ n \ then \ n - 10 \ else \ f \ (f \ (n + 11)).
Proof.
  intros. apply fG_-det with n.
     apply f_{-}correct.
     unfold ltb. destruct (Nat.leb\_spec0 \ 101 \ n).
       constructor. assumption.
       econstructor.
          lia.
          apply f_{-}correct.
```

```
apply f_{-}correct.
```

Qed.

Na koniec również mały twist, gdyż równanie rekurencyjne najprościej jest udowodnić za pomocą właściwości wykresu funkcji f - jeśli nie wierzysz, to sprawdź (ale będzie to bardzo bolesne sprawdzenie).

Podsumowując: zarówno oryginalna metoda induktywnej dziedziny jak i komenda Function nie radzą sobie z zagnieżdżonymi wywołaniami rekurencyjmi, czyli takimi, w których argumentem jest wynik innego wywołania rekurencyjnego. Możemy jednak poradzić sobie z tym problemem za pomocą ulepszonej metody induktywnej dziedziny, w której funkcję w definicji predykatu dziedziny reprezentujemy za pomocą jej induktywnie zdefiniowanego wykresu.

Ćwiczenie Przyjrzyjmy się poniższej fikuśnej definicji funkcji:

```
Fail Fixpoint g (n:nat):nat:= match n with \mid 0 \Rightarrow 0 \mid S \mid n \Rightarrow g \mid (g \mid n) end.
```

Wytłumacz, dlaczego Coq nie akceptuje tej definicji. Następnie wymyśl twierdzenie charakteryzujące tę funkcję, a na koniec zdefiniuj ją za pomocą metody zaprezentowanej w tym podrozdziale.

End McCarthy.

5.9 Metoda induktywno-rekurencyjnej dziedziny

Zapoznawszy się z metodą induktywnej dziedziny i jej ulepszoną wersją, która potrafi poskromić nawet rekursję zagnieżdżoną, dobrze byłoby na koniec podumać sobie trochę, co by było gdyby... Coq raczył wspierać indukcję-rekursję?

Ano, trochę ułatwiłoby to nasze nędzne życie, gdyż metoda induktywnej dziedziny przepoczwarzyłaby się w metodę induktywno-rekurencyjnej dziedziny: dzięki indukcji-rekursji moglibyśmy jednocześnie zdefiniować funkcję (nawet taką, w której jest rekursja zagnieżdżona) jednocześnie z jej predykatem dziedziny, co oszczędziłoby nam nieco pisania.

Zobaczmy, jak to wygląda na przykładzie funkcji McCarthy'ego. Ponieważ Coq jednak nie wspiera indukcji-rekursji, będziemy musieli użyć kodowania aksjomatycznego, co zapewne nieco umniejszy atrakcyjności tej metody.

```
Module McCarthy'.
```

```
forall d : fD (n + 11), fD (f (n + 11) d) -> fD n
with Fixpoint f' (n : nat) (d : fD n) : nat :=
match d with
   | fD_gt100 n H => n - 10
   | fD_le100 n H d1 d2 => f' (f' (n + 11) d1) d2
end.
*)
```

Tak wyglądałoby induktywno-rekurencyjna definicja zmodyfikowanej funkcji f' wraz z jej dziedziną. Ponieważ w definicji fD możemy napisać po prostu fD (f (n+11) d), wykres nie jest nam do niczego potrzebny. Definicja funkcji wygląda dokładnie tak samo jak ostatnio.

Variables

```
(fD: nat \rightarrow \mathsf{Type})
(f': \forall n: nat, fD \ n \rightarrow nat)
(fD_{-}gt100 : \forall n : nat, 100 < n \rightarrow fD \ n)
(fD_{-}le100 :
  \forall n : nat, n \leq 100 \rightarrow
     \forall d: fD (n+11), fD (f'(n+11) d) \rightarrow fD n
(f'-eq1:
  \forall (n : nat) (H : 100 < n), f' n (fD_gt100 n H) = n - 10)
(f'_-eq2:
  \forall
     (n: nat) (H: n \leq 100)
     (d1: fD (n + 11)) (d2: fD (f' (n + 11) d1)),
        f' \ n \ (fD\_le100 \ n \ H \ d1 \ d2) = f' \ (f' \ (n+11) \ d1) \ d2)
(fD_{-}ind:
     (P: \forall n: nat, fD n \rightarrow Type)
     (P_{-}gt100:
        \forall (n : nat) (H : 100 < n),
           P n (fD_-gt100 n H)
     (P_{-}le100:
        \forall
           (n: nat) (H: n \le 100)
           (d1: fD (n + 11)) (d2: fD (f' (n + 11) d1)),
             P(n+11) d1 \rightarrow P(f'(n+11) d1) d2 \rightarrow
                P \ n \ (fD_{-}le100 \ n \ H \ d1 \ d2)),
     \{g: \forall (n: nat) (d: fD n), P n d \mid
        (\forall (n : nat) (H : 100 < n),
           g \ n \ (fD_gt100 \ n \ H) = P_gt100 \ n \ H) \wedge
           (n: nat) (H: n \leq 100)
```

```
(d1:fD\;(n+11))\;(d2:fD\;(f'\;(n+11)\;d1)), \ g\;n\;(fD\_le100\;n\;H\;d1\;d2) = \ P\_le100\;n\;H\;d1\;d2 \ (g\;(n+11)\;d1) \ (g\;(f'\;(n+11)\;d1)\;d2)) \ \}).
```

Aksjomatyczne kodowanie tej definicji działa tak, jak nauczyliśmy się w poprzednim rozdziale: najpierw deklarujemy fD, potem f, potem konstruktory fD, potem równania definiujące f, a na samym końcu regułę indukcji.

Reguła indukcji powstaje analogicznie jak dla slist z poprzedniego rozdziału. Definiujemy tylko jedną rodzinę typów fD, więc reguła da nam tylko jedną funkcję, g, o typie $\forall (n : nat) (d : fD n), P n d$, gdzie $P : \forall n : nat, fD n \rightarrow$ Type reprezentuje przeciwdziedzinę g.

Mamy dwa przypadki: nieindukcyjny P_-gt100 odpowiadający konstruktorowi fD_-gt100 oraz P_-le100 odpowiadający za fD_-le100 , w którym mamy do dyspozycji dwie hipotezy indukcyjne. Otrzymana z reguły funkcja spełnia oczekiwane równania.

```
Lemma fD_{-}inv:
```

```
 \forall \; (n:nat) \; (d:fD \; n), \\ \{H:100 < n \; | \; d=fD\_gt100 \; n \; H\} \; + \\ \{H:n \leq 100 \; \& \\ \; \{d1:fD \; (n+11) \; \& \\ \; \{d2:fD \; (f' \; (n+11) \; d1) \; | \; d=fD\_le100 \; n \; H \; d1 \; d2\}\}\}.  Proof. apply fD\_ind. intros. left. \exists \; H. reflexivity. intros. right. \exists \; H, d1, \; d2. reflexivity. Defined.
```

Będziemy też chcieli używać **inversion** na hipotezach postaci fD n, ale fD nie jest induktywne (tylko aksjomatyczne), więc musimy pożądaną przez nas inwersję zamknąć w lemat. Dowodzimy go oczywiście za pomocą reguły indukcji.

```
Lemma f\_spec:
```

```
orall \ (n:nat) \ (d:fD\ n), \ n \leq 100 	o f'\ n\ d = 91. Froof.

apply (fD\_ind\ (\text{fun}\ n\ d \Rightarrow n \leq 100 \to f'\ n\ d = 91)).

intros n\ H\ H'.\ lia.

intros n\ H\ d1\ d2\ IH1\ IH2\ \_.

destruct (fD\_inv\ \_\ d1) as

[[H'\ eq]\ |\ (H'\ \&\ d1'\ \&\ d2'\ \&\ eq)].

destruct (fD\_inv\ \_\ d2) as

[[H''\ eq']\ |\ (H''\ \&\ d1''\ \&\ d2''\ \&\ eq')].

rewrite f'\_eq2,\ eq',\ f'\_eq1,\ eq,\ f'\_eq1\ in\ *.
```

```
clear IH1 eq eq 'H 'H ''. lia.
rewrite f ' eq 2. apply IH2. assumption.
rewrite f ' eq 2. apply IH2. rewrite IH1.
lia.
assumption.
```

Qed.

Możemy też udowodnić charakteryzację funkcji f. Dowód wygląda dużo groźniej niż ostatnio, ale to wszystko wina narzutu związanego z aksjomatycznym kodowaniem.

Dowód idzie tak: najpierw używamy indukcji, a potem naszego inwersjowego lematu na hipotezach postaci fD_- . W kluczowych momentach obliczamy funkcję f za pomocą definiujących ją równań oraz posługujemy się taktyką lia do przemielenia oczywistych, ale skomplikowanych formalnie faktów z zakresu arytmetyki liczb naturalnych.

```
\operatorname{Lemma} fD_{-}all:
  \forall n : nat, fD n.
Proof.
  apply (well\_founded\_ind \_ (fun \ n \ m \Rightarrow 101 - n < 101 - m)).
     apply wf_inverse_image. apply wf_it.
     intros n IH. destruct (le_{-}lt_{-}dec \ n \ 100).
        assert (d: fD (n + 11)) by (apply IH; lia).
          apply fD_{-}le100 with d.
             assumption.
             apply IH. destruct (fD_{-}inv_{-}d) as [[H\ eq]\ |\ [H\ _{-}]].
               rewrite eq, f'_-eq1. lia.
               rewrite f_spec.
                  lia.
                  assumption.
       apply fD_{-}qt100. assumption.
Qed.
```

Dowód tego, że wszystkie argumenty spełniają predykat dziedziny, jest taki sam jak ostatnio. Jedyna różnica jest taka, że zamiast inversion musimy ręcznie aplikować $fD_{-}inv$.

```
Definition f (n:nat): nat := f' \ n \ (fD_all \ n). Compute f 42. (* ===> = f' 42 \ (fD_all \ 42) : nat *)
```

Mając f' oraz dowód fD_-all możemy zdefiniować f, które niestety się nie oblicza, gdyż f' jest dane aksjomatycznie.

```
Lemma f'\_ext: \forall (n:nat) (d1 \ d2:fD \ n), \ f' \ n \ d1 = f' \ n \ d2. Proof. apply (fD\_ind \ (fun \ \_d1 \Rightarrow \forall \ d2, \ \_)); intros. rewrite <math>f'\_eq1. destruct (fD\_inv \ \_d2) as [[H' \ eq] \ | \ [H' \ \_]].
```

```
rewrite eq, f'_eq1. reflexivity. lia. rewrite f'_eq2. destruct (fD\_inv\_d0) as [[H'\ eq]\ |\ (H'\ \&\ d1'\ \&\ d2'\ \&\ eq)]. lia. subst. rewrite f'_eq2. specialize (H0\ d1'). destruct H0. apply H1. Qed.
```

Żeby udowodnić regułę indukcyjną, potrzebny nam będzie lemat mówiacy, że konkretny dowód tego, że n spełnia predykat dziedziny fD, nie wpływa na wynik obliczany przez f'. Dowód jest prosty: używamy indukcji po d1, a następnie inwersji po pozostałych hipotezach, przepisujemy równania definiujące f' i kończymy za pomocą rozumowań arytmetycznych albo hipotezy indukcyjnej.

```
Lemma f_{-}ind:
  \forall
     (P: nat \rightarrow nat \rightarrow Prop)
     (P_{-}qt100 : \forall n : nat, 100 < n \rightarrow P \ n \ (n - 10))
     (P_{-}le100:
        \forall n : nat, n \leq 100 \rightarrow
           P(n + 11) (f(n + 11)) \rightarrow
           P (f (n + 11)) (f (f (n + 11))) \rightarrow
              P \ n \ (f \ (f \ (n+11)))),
     \forall n : nat, P n (f n).
Proof.
  intros. apply (fD_{-}ind \text{ (fun } n \ d \Rightarrow P \ n \ (f' \ n \ d))); intros.
     rewrite f'_{-}eq1. apply P_{-}qt100. assumption.
     rewrite f'_{-}eq2. specialize (P_{-}le100 - H).
        unfold f in P_{-}le100.
        rewrite !(f'_-ext_- d1), !(f'_-ext_- d2) in P_-le100.
        apply P_{-}le100; assumption.
Qed.
```

Dowód samej reguły też jest dość prosty. Zaczynamy od indukcji po dowodzie faktu, że n: nat spełnia predykat dziedziny fD (którym to dowodem jest $fD_-all\ n$, a który schowany jest w definicji f). W przypadku nierekurencyjnym przepisujemy równanie definiujące f' i kończymy założeniem.

W przypadku rekurencyjnym również zaczynamy od przepisania definicji f'. Następnie korzystamy z założenia P_le100 , choć technicznie jest to dość skomplikowane - najpierw specjalizujemy je częściowo za pomocą hipotezy H, a potem odwijamy definicję f i dwukrotnie korzystamy z lematu f'_ext, żeby typy się zgadzały. Po tej obróbce możemy śmiało skorzystać z P_le100 - jej przesłanki zachodzą na mocy założenia.

 $\acute{\mathbf{C}}$ wiczenie Rozwiąż jeszcze raz ćwiczenie o funkcji g z poprzedniego podrozdziału, ale tym razem wykorzystaj metodę induktywno-rekurencyjnej dziedziny zaprezentowaną w ni-

niejszym podrozdziale.

```
Fail Fixpoint g (n:nat):nat:= match n with \mid 0 \Rightarrow 0 \mid S \mid n \Rightarrow g \mid (g \mid n) end.
```

End McCarthy'.

5.10 Metoda induktywnej dziedziny 2

Na koniec została nam do omówienia jeszcze jedna drobna kwestia. Poznając metodę induktywnej dziedziny, dowiedzieliśmy się, że "predykat" dziedziny tak naprawdę wcale nie jest predykatem, ale rodziną typów. Czas naprawić ten szkopuł.

W niniejszym podrozdziale najpierw zapoznamy się (na przykładzie dzielenia - znowu) z wariantem metody induktywnej dziedziny, w którym dziedzina faktycznie jest predykatem, a na koniec podumamy, dlaczego powinno nas to w ogóle obchodzić.

Module again.

```
\begin{array}{l} \textbf{Inductive} \ divD: nat \rightarrow nat \rightarrow \texttt{Prop} := \\ \mid divD\_lt: \forall \ n \ m: nat, \ n < S \ m \rightarrow divD \ n \ m \\ \mid divD\_ge: \\ \forall \ n \ m: nat, \\ n \geq S \ m \rightarrow divD \ (n \text{-} S \ m) \ m \rightarrow divD \ n \ m. \end{array}
```

Definicja dziedziny jest taka sama jak ostatnio, ale z tą drobną różnicą, że teraz faktycznie jest to predykat.

Skoro mamy dziedzinę, spróbujmy zdefiniować funkcję pomocniczą tak samo jak ostatnio.

```
Fail Fixpoint div_aux \ \{n \ m : nat\} \ (d : divD \ n \ m) : nat :=  match d with  | \ divD_blt_{---} \Rightarrow 0 \ | \ divD_ge_{---} d' \Rightarrow S \ (div_aux \ d')  end.  (* ===> \text{ The command has indeed failed with message:}  Incorrect elimination of "d" in the inductive type "divD": the return type has sort Set" 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. *)
```

Cóż, nie da się i nie dziwota - gdyby się dało, to zrobiliśmy tak już na samym początku. Powód porażki jest całkiem prozaiczny - nie możemy definiować programów przez dopaso-

wanie do wzorca dowodów, czyli parafrazując, nie możemy konstruować elementów typów sortów Set ani Type przez eliminację elementów typów sortu Prop.

Wynika to z faktu, że sort Prop z założenia dopuszcza możliwość przyjęcia aksjomatu irrelewancji dowodów (ang. proof irrelevance), który głosi, że wszystkie dowody danego zdania są równe. Gdybyśmy mogli dopasowywać do wzorca dowody zdań definiując programy, irrelewancja wyciekłaby do świata programów i wtedy wszystko byłoby równe wszystkiemu, co oczywiście daje sprzeczność.

Jeżeli powyższy opis nie jest przekonujący, zobaczmy to na szybkim przykładzie.

Module $proof_irrelevance_example$.

```
Inductive bool': Prop := |true':bool'
|false':bool'.
```

Najpierw definiujemy typ bool', który wygląda jak bool, ale żyje w sorcie Prop.

Axiom

```
proof\_irrelevance: \forall (P: Prop) (p1 \ p2: P), p1 = p2.
```

Następnie przyjmujemy aksjomat irrelewancji dowodów, przez co bool' staje się tym samym co zdanie True.

Axioms

```
(f:bool' \rightarrow bool)

(eq1:f\ true' = true)

(eq2:f\ false' = false).
```

Załóżmy, że Coq pozwolił nam zdefiniować funkcję $f:bool' \to bool$, która potrafi odróżnić true' od false'.

```
Theorem wut: true = false. Proof. rewrite \leftarrow eq1. rewrite \ (proof\_irrelevance \_ \_ false'). rewrite \ eq2. reflexivity. Qed.
```

Jak widać, true to to samo co f true', ale true' to false' na mocy irrelewancji, a f false' to przecież false. Konkluzja: prawda to falsz, a tego zdecydowanie nie chcemy. Żeby uniknąć sprzeczności, nie wolno definiować programów przez eliminację zdań.

Od powyższej zasady są jednak wyjątki, mianowicie przy konstrukcji programów wolno eliminować dowody zdań, które:

- nie mają konstruktorów, np. False
- maja jeden konstruktor, którego wszystkie argumenty również są dowodami zdań

Powyższy wyjątek od reguły nazywa się "eliminacją singletonów" i jest zupełnie niegroźny, gdyż dla takich zdań możemy bez żadnych aksjomatów udowodnić, że wszystkie ich dowody są równe.

End proof_irrelevance_example.

Dobra, koniec tej przydługiej dygresji. Wracamy do metody induktywnej dziedziny, gdzie dziedzina naprawdę jest predykatem. Skoro nie możemy zdefiniować funkcji bezpośrednio przez eliminację $d: divD \ n \ m$, to jak inaczej?

Tutaj ujawnia się pewna chytra sztuczka: nie możemy dopasować d za pomocą matcha, ale wciąż możemy robić wywołania rekurencyjne na podtermach d. Wystarczy więc napisać funkcję, która wyjmuje z d jego podterm (oczywiście jedynie pod warunkiem, że $n \geq S$ m, bo tylko wtedy d będzie miało jakiś podterm). Ponieważ kodziedziną takiej funkcji będzie divD (n - S m) m, dopasowanie d do wzorca będzie już legalne.

Brzmi... chytrze? Zobaczmy, jak wygląda to w praktyce.

```
{\tt Lemma}\ div D\_ge\_inv:
```

```
 \forall \ n \ m: \ nat, \ n \geq S \ m \rightarrow divD \ n \ m \rightarrow divD \ (n - S \ m) \ m.  Proof. destruct 2. apply le\_not\_lt in H. contradiction. exact H1. Defined.
```

Jeżeli mamy d: divD n m i wiemy, że $n \geq S$ m, to d musiało zostać zrobione konstruktorem $divD_-ge$. Możemy to udowodnić po prostu rozbijając d. W pierwszym przypadkiem dostajemy sprzeczność arytmetyczną (bo $n \geq S$ m i jednocześnie n < S m), zaś w drugim wynikiem jest pożądany podterm.

```
Fixpoint div'\_aux \{n \ m : nat\} (d : divD \ n \ m) : nat := match le\_lt\_dec (S \ m) n with | right \_\Rightarrow 0 | left H\Rightarrow S (div'\_aux (divD\_ge\_inv \ n \ m \ H \ d)) end.
```

Żeby zdefiniować div'_-aux (czyli, przypomnijmy, zmodyfikowaną wersję dzielenia, którego argumentem głównym jest d:divD n m, nie zaś samo n), sprawdzamy najpierw, czy mamy do czynienia z przypadkiem n < S m, czy z $n \ge S$ m. W pierwszym przypadku po prostu zwracamy 0, zaś w drugim robimy wywołanie rekurencyjne, którego argumentem jest $divD_-ge_-inv$ n m H d.

Term ten, jak się okazuje, jest uznawany przez Coqa za podterm d, a więc wywołanie rekurencyjne na nim jest legalne. Dlaczego jest to podterm d? Jeżeli odwiniemy definicję $divD_{-}ge_{-}inv$ i wykonamy występujące tam dopasowanie d do wzorca, to wiemy, że nie może być ono postaci $divD_{-}lt_{-}$, a zatem musi być postaci $divD_{-}ge_{-}$ d' i wynikiem wykonania funkcji jest d', które faktycznie jest podtermem d.

```
Lemma divD_-all:
```

```
\forall n \ m : nat, divD \ n \ m.
```

```
Proof.
  apply (well\_founded\_rect\ nat\ lt\ wf\_lt\ (fun\ \_ \Rightarrow \forall\ m:\ nat,\ \_)).
  intros n IH m.
  destruct (le_{-}lt_{-}dec\ (S\ m)\ n).
    apply div D_{-} qe.
      unfold ge. assumption.
      apply IH. abstract lia.
    apply div D_{-}lt. assumption.
Defined.
   Dowód tego, że każde n m: nat należą do dziedziny, jest dokładnie taki sam jak po-
przednio.
Definition div' (n m : nat) : nat :=
  div'_-aux (divD_-all\ n\ m).
Compute div' 666 7.
(* ===> = 83 : nat *)
   Ostateczna definicja funkcji div' również wygląda identycznie jak poprzednio i podobnie
elegancko się oblicza, a skoro tak, to czas udowodnić, że wykresem div' jest divG. Nie musimy
redefiniować wykresu - jest on zdefiniowany dokładnie tak samo jak ostatnio.
Lemma divG_{-}div'_{-}aux :
  \forall (n \ m : nat) (d : divD \ n \ m),
    divG \ n \ m \ (div'\_aux \ d).
Proof.
  Fail induction d.
  (* ===> The command has indeed failed with message:
            Abstracting over the terms \(\hat{n}''\) and "m" leads to a term
            fun n0 m0 : nat => divG n0 m0 (div'_aux d)
            which is ill-typed. *)
Abort.
   Pierwsza próba dowodu kończy się zupełnie niespodziewaną porażką już przy pierwszym
kroku, czyli próbie odpalenia indukcji po d.
Check divD_{-}ind.
(* ===>
  divD_ind :
    forall P : nat -> nat -> Prop,
    (forall n m : nat, n < S m -> P n m) ->
     (forall n m : nat,
       n \ge S m -> divD (n - S m) m -> P (n - S m) m -> P n m) ->
    forall n n0 : nat, divD n n0 -> P n n0
```

Powód jest prosty: konkluzja, czyli divG n m $(div'_-aux$ d) zależy od d, ale domyślna reguła indukcji wygenerowana przez Coqa, czyli $divD_-ind$, nie jest ani trochę zależna i nie

*)

dopuszcza możliwości, by konkluzja zależała od d. Potrzebna jest więc nam zależna reguła indukcji.

Na szczęście nie musimy implementować jej ręcznie - Coq potrafi zrobić to dla nas automatycznie (ale skoro tak, to dlaczego nie zrobił tego od razu? - nie pytaj, niezbadane są wyroki...).

```
Scheme divD_{-}ind' := Induction for divD Sort Prop.
```

Do generowania reguł indukcji służy komenda Scheme. $divD_{-}ind'$ to nazwa reguły, Induction for divD mówi nam, dla jakiego typu lub rodziny typów chcemy regułę, zaś Sort Prop mówi, że chcemy regułę, w której przeciwdziedziną motywu jest Prop (tak na marginesie - motyw eliminacji to typ lub rodzina typów, której element chcemy za pomocą eliminacji skonstruować - powinienem był wprowadzić tę nazwę wcześniej).

```
Check divD_ind'.
(* ===>
  divD_ind' :
    forall P : forall n n0 : nat, divD n n0 -> Prop,
    (forall (n m : nat) (1 : n < S m), P n m (divD_lt n m l)) ->
    (forall (n m : nat) (g : n >= S m) (d : divD (n - S m) m),
        P (n - S m) m d -> P n m (divD_ge n m g d)) ->
    forall (n n0 : nat) (d : divD n n0), P n n0 d
    *)
```

Jak widać, reguła wygenerowana przez komendę Scheme jest zależna, gdyż jednym z argumentów P jest divD n $n\theta$. Czas więc wrócić do dowodu faktu, że divG jest wykresem div'.

```
 \begin{array}{l} \text{Lemma } divG\_div'\_aux : \\ \forall \; (n \; m: \; nat) \; (d: \; divD \; n \; m), \\ divG \; n \; m \; (@div'\_aux \; n \; m \; d). \\ \text{Proof.} \\ \text{induction } d \; \text{using } divD\_ind'. \\ \text{unfold } div'\_aux. \; \text{destruct } (le\_lt\_dec \; (S \; m) \; n). \\ lia. \\ \text{constructor. } \text{assumption.} \\ \text{unfold } div'\_aux. \; \text{destruct } (le\_lt\_dec \; (S \; m) \; n). \\ \text{constructor.} \\ \text{assumption.} \\ \text{exact } IHd. \\ lia. \\ \text{Qed.} \end{array}
```

Indukcję z użyciem innej niż domyślna reguły możemy zrobić za pomocą taktyki induction d using $divD_{-}ind$. Tym razem reguła jest wystarczająco ogólna, więc indukcja się udaje.

Następnym krokiem w obu przypadkach jest odwinięcie definicji div'_-aux i sprawdzenie, czy n < S m, czy może $n \ge S$ m. Taki sposób postępowania jest kluczowy, gdyż próba

użycia tu taktyki cbn skończyłaby się katastrofą - zamiast uprościć cel, wyprulibyśmy z niego flaki, które zalałyby nam ekran, a wtedy nawet przeczytanie celu byłoby trudne. Jeżeli nie wierzysz, spróbuj.

Mamy więc dowód poprawności div'_aux względem wykresu. Wszystkie pozostałe dowody przechodzą bez zmian, więc nie będziemy ich tutaj powtarzać.

End again.

Do rozstrzygnięcia pozostaje nam ostatnia już kwestia - po cholerę w ogóle bawić się w coś takiego? Powyższe trudności z eliminacją d, dowodzeniem lematów wyciągających z d podtermy, dowodzeniem przez indukcję po d, generowaniem lepszych reguł indukcyjnych i unikaniem użycia taktyki cbn powinny jako żywo uzmysłowić nam, że uczynienie dziedziny divD prawdziwym predykatem było raczej upośledzonym pomysłem.

Odpowiedź jest krótka i mało przekonująca, a jest nią mechanizm ekstrakcji. Cóż to takiego? Otóż Coq dobrze sprawdza się przy definiowaniu programów i dowodzeniu ich właściwości, ale raczej słabo w ich wykonywaniu (komendy Compute czy Eval są dość kiepskie).

Mechanizm ekstrakcji to coś, co nas z tej nędzy trochę ratuje: daje on nam możliwość przetłumaczenia naszego programu w Coqu na program w jakimś nieco bardziej mainstreamowym języku funkcyjnym (jak OCaml, Haskell czy Scheme), w których programy da się normalnie odpalać i działają nawet szybko.

Mechanizm ten nie będzie nas interesował, ponieważ moim zdaniem jest on ślepą uliczką ewolucji - zamiast niego trzeba będzie po prostu wymyślić sposób efektywnej kompilacji Coqowych programow, ale to już temat na inną bajkę.

Nie będziemy zatem zbytnio zagłębiać się w szczegóły ekstrakcji - zanim zupełnie o niej zapomnimy, zobaczmy tylko jeden przykład.

Extraction Language Haskell.

Komenda Extraction Language ustawia język, do którego chcemy ekstrahować. My użyjemy Haskella, gdyż pozostałych dostępnych języków nie lubię.

Extraction again.div'.

Komenda Extraction p tłumaczy Coqowy program p na program Haskellowy. Mimo że nie znamy Haskella, spróbujmy przeczytać wynikowy program.

Wynikiem ekstrakcji jest Haskellowa funkcja div' o typie $Nat \rightarrow Nat$, gdzie Nat to Haskellowa wersja Coqowego nat (podwójny dwukropek :: oznacza w Haskellu to samo, co pojedynczy dwukropek : w Coqu). Funkcja div' jest zdefiniowana jako... i tu zaskoczenie... div'_-aux . Ale jak to? Rzućmy jeszcze raz okiem na oryginalną, Coqową definicję.

Print again.div'.

Gdzież w wyekstrahowanym programie podział się dowód divD_all n m? Otóż nie ma

go, bo Haskell to zwykły język programowania, w którym nie można dowodzić. O to właśnie chodzi w mechanizmie ekstrakcji - w procesie ekstrakcji wyrzucić z Coqowego programu wszystkie dowody i przetłumaczyć tylko tę część programu, która jest niezbędna, by wyekstrahowany program się obliczał.

Mogłoby się wydawać dziwne, że najpierw w pocie czoła dowodzimy czegoś w Coqu, a potem mechanizm ekstrakcji się tego pozbywa. Jest to jednak całkiem naturalne - skoro udało nam się udowodnić jakąś właściwość naszego programu, to wiemy, że ma on tę właściwość i dowód nie jest nam już do niczego potrzebny, a zatem ekstrakcja może się go pozbyć.

```
Print again.div'_aux.
(* ===>
    again.div'_aux =
    fix div'_aux (n m : nat) (d : again.divD n m) {struct d} : nat :=
    match le_lt_dec (S m) n with
        | left H =>
            S (div'_aux (n - S m) m (again.divD_ge_inv n m H d))
        | right _ => 0
    end
      : forall n m : nat, again.divD n m -> nat *)
Extraction again.div'_aux.
(* ===>
    div'_aux :: Nat -> Nat -> Nat
    div'_aux n m =
      case le_lt_dec (S m) n of {
       Left -> S (div'_aux (sub n (S m)) m);
       Right -> 0} *)
```

A tak wygląda wyekstrahowana funkcja div'_aux . Jeżeli pominiemy różnice składniowe między Coqiem i Haskellem (w Coqu typ jest na dole, po dwukropku, a w Haskellu na górze, przed definicją; w Coqu mamy match, a w Haskellu case etc.) to wygląda całkiem podobnie. Kluczową różnicą jest widniejący w Coqowej wersji dowód należenia do dziedziny $again.divD_ge_inv\ n\ m\ H\ d$, który w Haskellowym ekstrakcie został usunięty.

Cały ten cyrk z przerabianiem divD na prawdziwy predykat był po to, żeby dowód należenia do dziedziny mógł zostać usunięty podczas ekstrakcji. Dzięki temu wyekstrahowany program w Haskellu wygląda jak gdyby został napisany ręcznie. Jest też szybszy i krótszy, bo nie ma tam wzmianki o $divD_-all$, która musiałaby się pojawić, gdyby divD było rodziną typów, a nie predykatem.

```
Extraction div'_{-}aux.
```

```
(* ===>
    div'_aux :: Nat -> Nat -> DivD -> Nat
    div'_aux _ _ h =
        case h of {
        DivD_lt _ _ -> 0;
```

$$DivD_ge n m h' \rightarrow S (div'_aux (sub n (S m)) m h')} *)$$

Tak wygląda ekstrakt oryginalnego div'_-aux , tzn. tego, w którym divD nie jest predykatem, lecz rodziną typów. W wyekstrahowanym programie, w typie funkcji div'_-aux pojawia się złowieszczy typ DivD, który jest zupełnie zbędny, bo Haskell (i żaden inny funkcyjny język programowania, który nie jest przeznaczony do dowodzenia) nie narzuca żadnych ograniczeń na wywołania rekurencyjne.

No, to by było na tyle. Życzę ci, żebyś nigdy nie musiał stosować wariantu metody induktywnej dziedziny opisanej w tym podrozdziale ani nie musiał niczego ekstrahować.

5.11 Plugin Equations

Čwiczenie Zainstaluj plugin Equations: https://github.com/mattam82/Coq-Equations
Przeczytaj tutorial: https://raw.githubusercontent.com/mattam82/Coq-Equations/master/doc/equati
Następnie znajdź jakiś swój dowód przez indukcję, który był skomplikowany i napisz
prostszy dowód za pomocą komendy Equations i taktyki funelim.

5.12 Podsumowanie

Póki co nie jest źle, wszakże udało nam się odkryć indukcję wykresową, czyli metodę dowodzenia właściwości funkcji za pomocą specjalnie do niej dostosowanej reguły indukcji, wywodzącej się od reguły indukcji jej wykresu.

Rozdział 6

D2ipół

W tym rozdziale będą różne formy indukcji/rekursji, których chwilowo nie chcę wstawiać do głównego tekstu rozdziałów D1 i D2, bo tam nie pasują. Prędzej czy później zostaną one z tymi rozdział zintegrowane (albo i nie - nie mamy pańskiego płaszcza i co nam pan zrobi?).

6.1 Rekursja prymitywna (TODO)

Wiemy już, że rekursja ogólna prowadzi do sprzeczności, a jedyną legalną formą rekursji jest rekursja prymitywna (i niektóre formy rekursji strukturalnej, o czym dowiemy się później). Funkcje rekurencyjne, które dotychczas pisaliśmy, były prymitywnie rekurencyjne, więc potrafisz już całkiem sprawnie posługiwać się tym rodzajem rekursji. Pozostaje nam zatem jedynie zbadać techniczne detale dotyczące sposobu realizacji rekursji prymitywnej w Coqu. W tym celu przyjrzyjmy się ponownie definicji dodawania:

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 $\{structn\}$. 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 skończonej wysokości. 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)

Čwiczenie 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\_antireft:
\forall n: nat, \neg subterm\_nat \ n \ n.
Lemma subterm\_nat\_trans:
\forall a \ b \ c: nat.
```

 $subterm_nat \ a \ b \rightarrow subterm_nat \ b \ c \rightarrow subterm_nat \ a \ c.$

```
Lemma subterm\_list\_antirefl:
\forall (A: \mathtt{Type}) \ (l: list\ A), \ \neg\ subterm\_list\ l\ l.
Lemma subterm\_list\_trans:
\forall\ (A: \mathtt{Type}) \ (l1\ l2\ l3: list\ A),
subterm\_list\ l1\ l2 \rightarrow subterm\_list\ l2\ l3 \rightarrow 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 \mid 0 \Rightarrow tt \mid S \mid n' \Rightarrow weird \mid 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 \mid 0 \Rightarrow 0 \mid 1 \Rightarrow 1 \mid S\ (S\ n') \Rightarrow 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
Ś n'" instead of one of the following variables:
n0" n'". *)
```

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 $n\theta$, której w naszym kodzie nigdzie nie ma. Sposobem na poradzenie sobie z problemem jest pokazanie Coqowi palcem, o co nam chodzi:

```
\begin{split} \text{Fixpoint } f\!ib \; (n: nat): \; nat:= \\ \text{match } n \; \text{with} \\ \mid 0 \Rightarrow 0 \\ \mid 1 \Rightarrow 1 \\ \mid S \; (S \; n' \; \text{as} \; n'') \Rightarrow f\!ib \; n' + f\!ib \; n'' \\ \text{end.} \end{split}
```

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 kolejnych podrozdziałach, wcale tak nie jest. Dzięki pewnej sztuczce 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.

Čwiczenie (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.

6.2 Jak działa indukcja (nie, nie kuchenka)

6.3 Rekursja strukturalna (TODO)

6.4 Rekursja jako najlepszość

Znamy już podstawowe typy induktywne, jak liczby naturalne oraz listy elementów typu A. Wiemy też, że ich induktywność objawia się głównie w tym, że możemy definiować funkcje przez rekursję strukturalną po argumentach tych typów oraz dowodzić przez indukcję.

W takim podejściu indukcja i sama induktywność typów induktywnych wydają się być czymś w rodzaju domina - wystarczy popchnąć pierwsze kilka kostek (przypadki bazowe) i zapewnić, że pozostałe kostki są dobrze ułożone (przypadki rekurencyjne), aby zainicjować reakcję łańcuchową, która będzie przewracać kostki w nieskończoność.

Nie jest to jednak jedyny sposób patrzenia na typy induktywne. W tym podrozdziale spróbuję przedstawić inny sposób patrzenia, w którym typ induktywny to najlepszy typ do robienia termów o pewnym kształcie, a rekursja to zmiana kształtu z lepszego na gorszy, ale bardziej użyteczny.

Żeby móc patrzeć z tej perspektywy musimy najpierw ustalić, czym jest kształt. Uwaga: "kształt" nie jest pojęciem technicznym i nie ma ścisłej definicji - używam tego słowa, żeby ułatwić pracę twojej wyobraźni.

Czym jest kształt termu? Najprościej rzecz ujmując każdy term jest drzewkiem, którego korzeniem jest jakiś konstrukt językowy (stała, konstruktor, uprzednio zdefiniowana funkcja, dopasowanie do wzorca, let, lub cokolwiek innego), a jego poddrzewa to argumenty tego konstruktu.

Dla przykładu, termy typu nat moga mieć takie kształty:

- 0 stała
- S(S(S(S))) konstruktor
- plus 0 5, mult 0 5 uprzednio zdefiniowana funkcja
- ullet if andb false false then 42 else S 42 if
- match 0 with $\mid 0 \Rightarrow 666 \mid S \rightarrow 123$ dopasowanie do wzorca
- length [true; false] uprzednio zdefiniowana funkcja
- let x := Prop in 16 let
- ... i wiele, wiele innych!

Tak wiele różnych sposobów robienia termów to niesamowite bogactwo, więc żeby zgodnie z przysłowiem od tego przybytku nie rozbolała nas głowa, musimy pomyśleć o nich w nieco bardziej jednorodny sposób. Rozwiązanie jest na szczęście bajecznie proste: zauważ, że wszystkie powyższe konstrukty językowe można po prostu zawinąć w funkcję, która bierze pewną liczbę argumentów (być może zero) i zwraca coś typu nat.

To jednak nie w pełni mityguje przyszły-niedoszły ból głowy. O ile mamy teraz jednorodny sposób myślenia o kształtach termów, to i tak kształtów tych mogą być olbrzymie ilości. Z tego powodu dokonamy samoograniczenia i zamiast o wszystkich możliwych kształtach termów będziemy wybiórczo skupiać naszą uwagę tylko na tych kształtach, które akurat będą nas interesować.

Dla przykładu, możemy interesować się termami typu *nat* zrobionymi wyłącznie za pomocą:

- \bullet konstruktorów 0 i S
- konstruktora 0, stałej 1 oraz funkcji plus 2
- funkcji plus i stałych 5 oraz 42
- funkcji *mult* i stałej 1
- funkcji $length: list\ nat \rightarrow nat$

Čwiczenie Narysuj jakieś nietrywialne termy typu *nat* o takich kształtach.

Ćwiczenie Liczbę n da się wyrazić za pomocą termu t, jeżeli t oblicza się do n, tzn. komenda Compute t daje w wyniku n.

Pytanie: termy o których z powyższych kształtów mogą wyrazić wszystkie liczby naturalne?

Ćwiczenie Liczba n ma unikalną reprezentację za pomocą termów o danym kształcie, gdy jest tylko jeden term t, który reprezentuje n.

Pytanie: które z powyższych sposobów unikalnie reprezentują wszystkie liczby naturalne? Sporo już osiągnęliśmy w wyklarowywaniu pojęcia kształtu, ale zatrzymajmy się na chwilę i zastanówmy się, czy jest ono zgodne z naszymi intuicjami.

Okazuje się, że otóż nie do końca, bo w naszej obecnej formulacji kształty (0, plus) oraz (1, mult) są różne, podczas gdy obrazki (narysuj je!) jasno pokazują nam, że np. plus 0 (plus 0 0) oraz mult 1 (mult 1 1) wyglądają bardzo podobnie, tylko nazwy są różne.

Dlatego też modyfikujemy nasze pojęcie kształtu - teraz kształtem zamiast stałych i funkcji, jak 0 i plus, nazywać będziemy typy tych stałych i funkcji. Tak więc kształtem termów zrobionych z 0 i plus będzie nat (bo 0:nat) i $nat \rightarrow nat \rightarrow nat$ (bo $plus:nat \rightarrow nat \rightarrow nat$). Teraz jest już jasne, że 1 i mult dają dokładnie ten sam kształt, bo typem 1 jest nat, zaś typem mult jest $nat \rightarrow nat$.

Zauważmy, że można nasze pojęcie kształtu jeszcze troszkę uprościć:

- po pierwsze, każdą stałą można traktować jako funkcję biorącą argument typu unit, np. możemy 0:nat reprezentować za pomocą funkcji $Z:unit \to nat$ zdefiniowanej jako Z:= fun $_{-}:unit \Rightarrow 0$
- po drugie, funkcje biorące wiele argumentów możemy reprezentować za pomocą funkcji biorących jeden argument, np. $plus: nat \rightarrow nat$ możemy reprezentować za pomocą $plus': nat \times nat \rightarrow nat$, który jest zdefiniowany jako $plus':= fun'(n, m) \Rightarrow plus n m$
- po trzecie, ponieważ kodziedzina wszystkich funkcji jest taka sama (w naszym przypadku nat), możemy połączyć wiele funkcji w jedną, np. 0 i plus możemy razem reprezentować jako $Zplus: unit + nat \times nat \rightarrow nat$, zdefiniowaną jako Zplus:= fun $x \Rightarrow$ match x with $|inl| \Rightarrow 0$ $|inr|(n, m) \Rightarrow plus| n m$ end

Dzięki tym uproszczeniom (albo utrudnieniom, zależy kogo spytacie) możemy teraz jako kształt traktować nie funkcje albo same ich typy, lecz tylko jeden typ, który jest dziedziną takiej połączonej funkcji. Tak więc zarówno (0, plus) jak i (1, mult) są kształtu $unit + nat \times nat$. Ma to sporo sensu: drzewa reprezentujące te termy są albo liściem (reprezentowanym przez unit), albo węzłem, który rozgałęzia się na dwa poddrzewa (reprezentowanym przez $nat \times nat$).

Ale to jeszcze nie wszystko. Przecież nat to nie jedyny typ, w którym można robić termy o kształcie $unit + nat \times nat$. Jeżeli przyjrzymy się, jak wyglądają termy zrobione za pomocą (true, andb) albo (false, orb), to okaże się, że... mają one dokładnie ten sam kształt, mimo że według naszej definicji ich kształt to $unit + bool \times bool$, czyli niby coś innego.

Ostatnim stadium ewolucji naszego pojęcia kształtu jest taki oto zestaw definicji:

- kształt to funkcja $F: \mathsf{Type} \to \mathsf{Type}$
- \bullet realizacją kształtu F jest typ X oraz funkcja $f: F X \to X$

Widzimy teraz, że (0, plus), (1, mult), (true, andb) oraz (false, orb) nie są kształtami, lecz realizacjami kształtu $F := fun \ X : Type \Rightarrow 1 + X \times X$.

Pora powoli zmierzać ku konkluzji. Na początku powiedzieliśmy, że typ induktywny to najlepszy typ do robienia termów o pewnym kształcie. Jakim kształcie, zapytasz pewnie, i jak objawia się owa najlepszość? Czas się tego dowiedzieć.

Definiując typ induktywny podajemy jego konstruktory, a całą resztę, czyli możliwość definiowania przez dopasowanie do wzorca i rekursję, reguły eliminacji etc. dostajemy za darmo. Nie dziwota więc, że to właśnie konstruktory są realizacją kształtu, którego dany typ jest najlepszym przykładem.

Napiszmy to jeszcze raz, dla jasności: typ induktywny to najlepszy sposób robienia termów o kształcie realizowanym przez jego konstruktory.

W naszym natowym przykładzie oznacza to, że nat jest najlepszym sposobem robienia termów o kształcie $F := \text{fun } X \Rightarrow unit + X$, czyli termów w kształcie "sznurków" (konstruktor S to taki supełek na sznurku, a 0 reprezentuje koniec sznurka). Są też inne realizacje tego

sznurkowego kształtu, jak np. stała 42 : nat i funkcja plus 8 : $nat \rightarrow nat$ albo stała true : bool i funkcja negb : $bool \rightarrow bool$, albo nawet zdanie False : Prop oraz negacja not : Prop \rightarrow Prop, ale żadna z nich nie jest najlepsza.

Jak objawia się najlepszość typu induktywnego? Ano, dwojako:

- po pierwsze, objawia się w postaci rekursora, który bierze jako argument docelową realizację danego kształtu i przerabia term typu induktywnego, podmieniając najlepszą realizację na docelową
- po drugie, rekursor jest unikalny, czyli powyższa podmiana realizacji odbywa się w jedyny słuszny sposób

Żeby nie być gołosłownym, zobaczmy przykłady:

```
Fixpoint nat\_rec' \{X: \mathtt{Type}\}\ (z:X)\ (s:X\to X)\ (n:nat):X:= match n with  \mid 0\Rightarrow z \\ \mid S\ n'\Rightarrow s\ (nat\_rec\ 'z\ s\ n') end.
```

Tak wygląda rekursor dla liczb naturalnych. Widzimy, że "zmiana realizacji" termu o danym kształcie intuicyjnie polega na tym, że bierzemy term i zamieniamy 0 na z, a S na s, czyli dla przykładu liczba 4 (czyli S (S (S (S 0)))) zostanie zamieniona na s (s (s (s 2))). Jeszcze konkretniejszy przykład: nat_rec ' true negb zamieni liczbę S (S (S 0))) w negb (negb (negb true))). Oczywiście term ten następnie oblicza się do true.

Ćwiczenie Mamy nat_rec ' $true\ negb$: $nat \to bool$, a zatem zmiana realizacji sznurka z (0, S) na $(true,\ negb)$ odpowiada sprawdzeniu jakiejś właściwości liczb naturalnych. Jakiej?

Pisząc wprost: zdefiniuj bezpośrednio przez rekursję taką funkcję $f: nat \to bool$, że $\forall n$: nat, nat_rec ' true $negb\ n = f\ n$ (oczywiście musisz udowodnić, że wszystko się zgadza).

Uwaga: Coq domyślnie generuje dla typu "rekursor", ale ma on na myśli coś innego, niż my:

Check nat_rec .

Coqowe nat_rec to w zasadzie to samo, co nat_ind , czyli reguła indukcji, tyle że kodziedziną motywu nie jest Prop, lecz Set (możesz myśleć, że Set to to samo co Type).

Podobieństwo naszego nat_rec ' oraz reguły indukcji nie jest przypadkowe - myślenie o typach induktywnych w przedstawiony wyżej sposób jest najlepszym sposobem na spamiętanie wszystkich możliwych reguł rekursji, indukcji i tympodobnych. A robi się to tak (naszym przykładem tym razem będzie typ $list\ A$).

Krok pierwszy: każda lista to albo nil: $list\ A$ albo cons: $A \to list\ A \to list\ A$ zaaplikowany do głowy h: A i ogona t: $list\ A$.

Krok drugi: skoro tak, to $list\ A$ jest najlepszym sposobem na robienie termów w kształcie $(nil,\ cons).$

Krok trzeci: wobec tego mamy (a raczej musimy sobie zrobić) rekursor $list_rec$ ', który, gdy damy mu inną realizację kształtu $F := fun \ X \Rightarrow unit + A \times X$, to podmieni on nil i consy w dowolnej liście l na tą inną realizację. Jego typ wygląda tak:

```
Definition list\_rec'_type: Type :=
     (A: Type) (* parametr list *)
     (P: Type) (* inna realizacja kształtu - typ *)
     (n:P) (* inna realizacja kształtu – nil *)
     (c:A \rightarrow P \rightarrow P) (* inna realizacja kształtu - cons *)
    (l: list A), (* lista, w której chcemy zrobić podmianę *)
       P. (* wynik podmiany *)
   Krócej można ten typ zapisać tak:
Definition list\_rec'\_type': Type :=
  \forall A P : \mathsf{Type}, P \to (A \to P \to P) \to \mathit{list} A \to P.
   Implementacja jest banalna:
Fixpoint list_rec'
  \{A\ P: \mathtt{Type}\}\ (n:P)\ (c:A 	o P 	o P)\ (l:\mathit{list}\ A):P:=
match l with
      nil \Rightarrow n \text{ (* podmieniamy } nil \text{ na } n... \text{*)}
      cons \ h \ t \Rightarrow c \ h \ (list\_rec' \ n \ c \ t) \ (* \ldots a \ cons \ na \ c \ *)
end.
```

Krok czwarty: żeby uzyskać regułę indukcji, bierzemy rekursor i zamieniamy P: Type na P:list $A\to \mathsf{Prop}$. Żeby uzyskać najbardziej ogólną regułę eliminacji, używamy P:list $A\to \mathsf{Type}$.

```
 \begin{array}{l} \texttt{Definition } \mathit{list\_ind'\_type} : \texttt{Type} := \\ \forall \\ \{A : \texttt{Type}\} \\ \{P : \mathit{list } A \to \texttt{Prop}\} \\ (n : P \ \mathit{nil}) \\ (c : \forall \ (h : A) \ (t : \mathit{list } A), \ P \ t \to P \ (\mathit{cons } h \ t)) \\ (l : \mathit{list } A), \ P \ l. \end{array}
```

Oczywiście musimy też dostosować typy argumentów. Może to prowadzić do pojawienia się nowych argumentów. c w rekursorze miało typ $A \to P \to P$. Pierwszy argument typu A musimy nazwać h, żeby móc go potem użyć. Ostatnie P to konkluzja, która musi być postaci P (cons h t), ale t: list A nigdzie nie ma, więc je dodajemy. Pierwsze P zmienia się w hipotezę indukcyjną P t.

Krok piąty: definicja reguły indukcji jest prawie taka sama jak poprzednio (musimy uwzględnić pojawienie się t: list A jako argumentu w c. Poza tym drobnym detalem zmieniają się tylko typy:

```
Fixpoint list\_ind'
\{A: \mathtt{Type}\}
\{P: list\ A \to \mathtt{Prop}\}
(n:P\ nil)
(c:\forall\ (h:A)\ (t:list\ A),\ P\ t \to P\ (cons\ h\ t))
(l: list\ A)
:\ P\ l:=
match l with
|\ nil \Rightarrow n
|\ cons\ h\ t \Rightarrow c\ h\ t\ (list\_ind'\ n\ c\ t)
end.
```

Włala, mamy regułę indukcji.

Na sam koniec wypadałoby jeszcze opisać drobne detale dotyczące najlepszości. Czy skoro nat jest najlepszym typem do robienia termów w kształcie sznurków, to znaczy, że inne realizacje tego kształtu są gorsze? I w jaki sposób objawia się ich gorszość?

Odpowiedź na pierwsze pytanie jest skomplikowańsza niż bym chciał: *nat* jest najlepsze, ale inne typy też mogą być najlepsze. Rozważmy poniższy typ:

```
Inductive nat': Type := \mid Z' : nat' \mid S' : nat' \rightarrow nat'.
```

Jako, że nat' jest typem induktywnym, to jest najlepszym sposobem robienia termów w kształcie $F := \text{fun } X \Rightarrow unit + X$. Ale jak to? Przecież najlepsze dla tego kształtu jest nat! Tak, to prawda. Czy zatem nat' jest gorsze? Nie: oba te typy, nat i nat', są najlepsze.

Wynika stąd ciekawa konkluzja: nat' to w zasadzie to samo co nat, tylko inaczej nazwane. Fakt ten łatwo jest udowodnić: mając natowy sznurek możemy za pomocą nat_rec ' przerobić go na nat'owy sznurek. Podobnie nat'owy sznurek można za pomocą nat'_rec' przerobić na natowy sznurek. Widać na oko, że obie te funkcje są swoimi odwrotnościami, a zatem typy nat i nat' są izomorficzne, czyli mają takie same elementy i takie same właściwości.

Ćwiczenie Zdefiniuj funkcje $f: nat \to nat'$ i $g: nat \to nat'$, które spełniają $\forall n: nat, g$ $(f \ n) = n$ oraz $\forall n: nat', f \ (g \ n) = n$. Nie musisz w tym celu używać nat_rec' ani nat'_rec' (no chyba, że chcesz).

Drugie pytanie brzmiało: w czym objawia się brak najlepszości innych realizacji danego kształtu? Odpowiedź jest prosta: skoro najlepszość to unikalny rekursor, to brak najlepszości oznacza brak unikalnego rekursora. Przeżyjmy to na przykładzie:

Używając rekursora dla nat możemy podmienić S na negację, a 0 na false, i otrzymać dzięki temu funkcję sprawdzającą, czy długość sznurka (czyli liczby naturalnej) jest nieparzysta. Czy dla innych realizacji tego samego kształtu też możemy tak zrobić?

Nie zawsze. Rozważmy typ unit wraz ze stałą tt i funkcją $f := \mathbf{fun} \ _ \Rightarrow tt$, które realizują ten sam kształt co nat, 0 i S. Zauważmy, że tt = f tt, a zatem różne sznurki obliczają się do tej samej wartości. Jest to sytuacja zgoła odmienna od natowej - różne ilości Sów dają różne liczby naturalne.

Gdybyśmy mieli dla tej realizacji rekursor podmieniający f na jakąś funkcję g, zaś tt na stałą x, to niechybnie doszłoby do katastrofy. Dla przykładu, gdybyśmy próbowali tak jak wyżej sprawdzić, czy długość sznurka jest nieparzysta, zamieniając tt na false, zaś f na negb, to wynikiem zamiany dla tt byłoby false, zaś dla f tt byłoby to negb false = true. To jednak prowadzi do sprzeczności, bo tt = f tt. Wyniki podmiany dla sznurków obliczających się do równych wartości musza być takie same.

Oczywiście unit, tt i f to bardzo patologiczna realizacja sznurkowego kształtu. Czy są jakieś mniej patologiczne realizacje, które umożliwiają podmiankę, która pozwala sprawdzić nieparzystość długości sznurka?

Tak. Przykładem takiej realizacji jest... bool, false i negb (a rzeczona podmianka, to w tym przypadku po prostu funkcja identycznościowa).

Czy znaczy to, że bool, false i negb to najlepsza realizacja sznurkowego kształtu? Nie - da się znaleźć całą masę podmianek, które nat umożliwia, a bool, false i negb - nie (joł, sprawdź to - wcale nie kłamię).

Cóż, to by było na tyle. W ramach pożegnania z tym spojrzeniem na typy induktywne napiszę jeszcze tylko, że nie jest ono skuteczne zawsze i wszędzie. Działa jedynie dla prostych typów zrobionych z enumeracji, rekurencji i parametrów. Żeby myśleć w ten sposób np. o indeksowanych rodzinach typów trzeba mieć nieco mocniejszą wyobraźnię.

Ćwiczenie Rozważmy poniższe typy:

- unit
- bool
- option A
- \bullet $A \times B$
- \bullet A + B
- $\bullet \exists x : A, P x$
- nat
- list A

Dla każdego z nich:

- znajdź kształt, którego jest on najlepszą realizacją
- napisz typ rekursora

- zaimplementuj rekursor
- zaimplementuj bezpośrednio za pomocą rekursora jakąś ciekawą funkcję
- z typu rekursora wyprowadź typ reguły indukcji (oczywiście bez podglądania za pomocą komendy Print... nie myśl też o białym niedźwiedziu)
- zaimplementuj regułę indukcji
- spróbuj bezpośrednio użyć reguły indukcji, by udowodnić jakiś fakt na temat zaimplementowanej uprzednio za pomocą rekursora funkcji

6.5 Reguły eliminacji (TODO)

6.6 Rekursja monotoniczna

Require Import X3.

Czas na omówienie pewnej ciekawej, ale średnio użytecznej formy rekursji (z pamięci nie jestem w stanie przytoczyć więcej niż dwóch sztampowych przykładów jej użycia), a jest nią rekursja monotoniczna (zwana też czasem rekursją zagnieżdżoną, ale nie będziemy używać tej nazwy, gdyż dotychczas używaliśmy jej na określenie rekursji, w której arguemntem wywołania rekurencyjnego jest wynik innego wywołania rekurencyjnego).

Cóż to za zwierzątko, rekursja monotoniczna? Żeby się tego dowiedzieć, przypomnijmy sobie najpierw, jak technicznie w Coqu zrealizowana jest rekursja strukturalna.

```
Fixpoint plus\ (n:nat):nat \to nat:= match n with  \mid 0 \Rightarrow \text{fun } m:nat \Rightarrow m \\ \mid S\ n' \Rightarrow \text{fun } m:nat \Rightarrow S\ (plus\ n'\ m)  end.
```

Tak oto definicja funkcji plus, lecz zapisana nieco inaczej, niż gdy widzieliśmy ją ostatnim razem. Tym razem prezentujemy ją jako funkcję biorącą jeden argument typu nat i zwracającą funkcję z typu nat w typ nat.

```
\begin{array}{l} \text{Definition } plus': nat \rightarrow nat \rightarrow nat := \\ \text{fix } f \ (n: nat): nat \rightarrow nat := \\ \text{match } n \text{ with} \\ \mid 0 \Rightarrow \text{fun } m: nat \Rightarrow m \\ \mid S \ n' \Rightarrow \text{fun } m: nat \Rightarrow S \ (f \ n' \ m) \\ \text{end.} \end{array}
```

Ale komenda Fixpoint jest jedynie cukrem syntaktycznym - funkcję plus możemy równie dobrze zdefiniować bez niej, posługując się jedynie komendą Definition, a wyrażeniem,

które nam to umożliwia, jest fix. fix działa podobnie jak fun, ale pozwala dodatkowo nadać definiowanej przez siebie funkcji nazwe, dzięki czemu możemy robić wywołania rekurencyjne.

Czym więc jest rekursja monotoniczna? Z rekursją monotoniczną mamy do czynienia, gdy za pomocą fixa (czyli przez rekursję) definiujemy funkcję, która zwraca inną funkcję, i ta zwracana funkcja także jest zdefiniowana za pomocą fixa (czyli przez rekursję). Oczywiście to tylko pierwszy krok - wynikowa funkcja również może zwracać funkcję, która jest zdefiniowana za pomocą fixa i tak dalej.

Widać zatem jak na dłoni, że plus ani plus' nie są przykładami rekursji monotonicznej. Wprawdzie definiują one za pomocą fixa (lub komendy Fixpoint) funkcję, która zwraca inną funkcję, ale ta zwracana funkcja nie jest zdefiniowana za pomocą fixa, lecz za pomocą fun, a więc nie jest rekurencyjna.

Podsumowując: rekursja jest monotoniczna, jeżeli w definicji funkcji pojawiają się co najmniej dwa wystąpienia fix, jedno wewnątrz drugiego (przy czym rzecz jasna Fixpoint też liczy się jako fix).

No to skoro już wiemy, czas zobaczyć przykład jakiejś funkcji, która jest zdefiniowana przez rekursję monotoniczną.

```
Fail Fixpoint ack\ (n\ m:nat):nat:= match n,\ m with  \begin{array}{c} \mid 0,\ m\Rightarrow S\ m\\ \mid S\ n',\ 0\Rightarrow ack\ n'\ 1\\ \mid S\ n',\ S\ m'\Rightarrow ack\ n'\ (ack\ (S\ n')\ m') \end{array}  end.  (* ===> \ \text{The command has indeed failed with message:}  Cannot guess decreasing argument of fix. *)
```

Powyższa funkcja zwana jest funkcją Ackermanna, gdyż wymyślił ją... zgadnij kto. Jest ona całkiem sławna, choć z zupełnie innych powodów niż te, dla których my się jej przyglądamy. Nie oblicza ona niczego specjalnie użytecznego - jej wynikami są po prostu bardzo duże liczby. Jeżeli nie wierzysz, spróbuj policzyć ręcznie ack 4 2 - zdziwisz się.

Jak widać, Coq nie akceptuje powyższej definicji. Winny temu jest rzecz jasna kształt rekursji. Dla n równego 0 zwracamy S m, co jest ok. Dla n postaci S n' i m równego 0 robimy wywołanie rekurencyjne na n' i 1, co również jest ok. Jednak jeżeli n i m odpowednio są postaci S n' i S m', to robimy wywołanie rekurencyjne postaci ack n' (ack (S n') m'). W wewnętrznym wywołaniu rekurencyjnym pierwszy argument jest taki sam jak obecny. Gdyby argumentem głównym był drugi argument, to jest tym bardziej źle, gdyż w zewnętrznym wywołaniu rekurencyjnym nie jest nim m', lecz ack (S n') m'. Nie ma się więc co dziwić, że Coq nie może zgadnać, który argument ma być argumentem głównym.

Mimo, że Coq nie akceptuje tej definicji, to wydaje się ona być całkiem spoko. Żaden z argumentów nie może wprawdzie posłużyć nam za argument główny, ale jeżeli rozważymy ich zachowanie jako całość, to okazuje się, że w każdym wywołaniu rekurencyjnym mamy dwie możliwości:

• albo pierwszy argument się zmniejsza

• albo pierwszy argument się nie zmienia, ale drugi argument się zmniejsza

Możemy z tego wywnioskować, że jeżeli wywołamy ack na argumentach n i m, to w ogólności najpierw m będzie się zmniejszał, ale ponieważ musi kiedyś spaść do zera, to wtedy n będzie musiał się zmniejszyć. Oczywiście wtedy w kolejnym wywołaniu zaczynamy znowu z jakimś m, które potem się zmniejsza, aż w końcu znowu zmniejszy się n i tak dalej, aż do chwili, gdy n spadnie do zera. Wtedy rekursja musi się skończyć.

Jednym z typowych zastosowań rekursji zagnieżdżonej jest radzenie sobie z takimi właśnie przypadkami, w których mamy ciąg argumentów i pierwszy maleje, lub pierwszy stoi w miejscu a drugi maleje i tak dalej. Zobaczmy więc, jak techniki tej można użyć do zdefiniowania funkcji Ackermanna.

```
Fixpoint ack\ (n:nat):nat \to nat:= match n with  \begin{array}{c} \mid 0 \Rightarrow S \\ \mid S\ n' \Rightarrow \\ & \text{fix } ack'\ (m:nat):nat:=\\ & \text{match } m \text{ with } \\ & \mid 0 \Rightarrow ack\ n'\ 1 \\ & \mid S\ m' \Rightarrow ack\ n'\ (ack'\ m') \\ & \text{end} \end{array}  end.
```

Zauważmy przede wszystkim, że nieco zmienia się wygląd typu naszej funkcji. Jest on wprawdzie dokładnie taki sam $(nat \rightarrow nat \rightarrow nat)$, ale zapisujemy go inaczej. Robimy to by podkreslić, że wynikiem ack jest funkcja. W przypadku gdy n jest postaci S n' zdefiniowana jest ona za pomocą f ixa tak, jak wyglądają dwie ostatnie klauzule dopasowania z oryginalnej definicji, ale z wywołaniem ack (S n') m' zastąpionym przez ack' m'. Tak więc funkcja ack' reprezentuje częściową aplikację ack n.

```
 \begin{array}{l} \text{Lemma } ack\_eq: \\ \forall \ n \ m: nat, \\ ack \ n \ m = \\ \text{match } n, \ m \ \text{with} \\ \mid 0, \ \_ \Rightarrow S \ m \\ \mid S \ n', \ 0 \Rightarrow ack \ n' \ 1 \\ \mid S \ n', \ S \ m' \Rightarrow ack \ n' \ (ack \ (S \ n') \ m') \\ \text{end.} \\ \\ \text{Proof.} \\ \text{destruct } n, \ m; \ \text{reflexivity.} \\ \\ \text{Qed.} \\ \\ \text{Lemma } ack\_big: \\ \forall \ n \ m: nat, \\ m < ack \ n \ m. \\ \end{array}
```

```
Proof.
  induction n as [\mid n' \mid].
     cbn. intro. apply le_-n.
     induction m as [\mid m'].
        cbn. apply lt\_trans with 1.
          apply le_n.
          apply IHn'.
       specialize (IHn' (ack (S n') m')).
          rewrite ack_{-}eq. lia.
Qed.
Lemma ack_{-}big':
  \forall n1 \ n2 : nat, n1 \leq n2 \rightarrow
     \forall m1 \ m2 : nat, m1 \leq m2 \rightarrow
       ack \ n1 \ m1 \le ack \ n2 \ m2.
Proof.
  induction 1.
     induction 1.
       reflexivity.
       rewrite IHle. destruct n1.
          cbn. apply le_-S, le_-n.
          rewrite (ack_{-}eq\ (S\ n1)\ (S\ m)).
            pose (ack\_big\ n1\ (ack\ (S\ n1)\ m)). lia.
     induction 1.
       destruct m1.
          cbn. apply IHle. do 2 constructor.
          rewrite (ack_{-}eq\ (S\ m)\ (S\ m1)).
            rewrite (IHle\ (S\ m1)\ (ack\ (S\ m)\ m1)).
               reflexivity.
               apply ack_big.
       rewrite (ack_{-}eq\ (S\ m)). apply IHle. apply le_{-}trans with (S\ m\theta).
          apply le_-S. exact H0.
          apply ack_big.
Qed.
```

Ćwiczenie Require Import Arith.

Zdefiniuj funkcję merge o typie \forall (A: Type) $(cmp: A \rightarrow A \rightarrow bool)$, list $A \rightarrow list$ $A \rightarrow list$ A, która scala dwie listy posortowane według porządku wyznaczanego przez cmp w jedną posortowaną listę. Jeżeli któraś z list posortowana nie jest, wynik może być dowolny.

Wskazówka: dlaczego niby to ćwiczenie pojawia się w podrozdziale o rekursji zagnieżdżonej?

Compute merge leb [1; 4; 6; 9] [2; 3; 5; 8].

```
(* ===> = [1; 2; 3; 4; 5; 6; 8; 9]
: list nat *)
```

Obie listy są posortowane według leb, więc wynik też jest.

```
Compute merge leb [5; 3; 1] [4; 9].

(* ===> = [4; 5; 3; 1; 9]

: list nat *)
```

Pierwsza lista nie jest posortowana według leb, więc wynik jest z dupy.

Ćwiczenie Skoro już udało ci się zdefiniować *merge*, to udowodnij jeszcze parę lematów, cobyś nie miał za dużo wolnego czasu.

```
Lemma merge\_eq:
  \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{l1 \ l2 : list \ A\},\
     merge \ cmp \ l1 \ l2 =
     match l1, l2 with
           | [], \_ \Rightarrow l2
           |-,[] \Rightarrow l1
           | h1 :: t1, h2 :: t2 \Rightarrow
                 if cmp \ h1 \ h2
                 then h1 :: merge cmp t1 l2
                 else h2 :: merge\ cmp\ l1\ t2
     end.
Lemma merge\_nil\_l:
  \forall \{A : \mathtt{Type}\} \{cmp : A \rightarrow A \rightarrow bool\} \{l : list A\},\
     merge cmp \mid \mid l = l.
Proof.
  reflexivity.
Qed.
Lemma merge\_nil\_r:
  \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{l : list A\},\
     merge cmp \ l \ [] = l.
Proof.
  destruct l; reflexivity.
Qed.
{\tt Lemma}\ Permutation\_merge:
  \forall \{A : \mathsf{Type}\} \{f : A \to A \to bool\} \{l1 \ l2 : list \ A\},\
     Permutation (merge f l1 l2) (l1 ++ l2).
Lemma merge\_length:
  \forall \{A : \mathsf{Type}\} \{f : A \to A \to bool\} \{l1 \ l2 : list \ A\},\
     length (merge f l1 l2) = length l1 + length l2.
Lemma merge\_map:
```

```
\forall \{A \ B : \mathtt{Type}\} \{cmp : B \rightarrow B \rightarrow bool\} \{f : A \rightarrow B\} \{l1 \ l2 : list \ A\},\
         merge \ cmp \ (map \ f \ l1) \ (map \ f \ l2) =
         map\ f\ (merge\ (fun\ x\ y: A \Rightarrow cmp\ (f\ x)\ (f\ y))\ l1\ l2).
Lemma merge\_replicate:
   \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{x \ y : A\} \{n \ m : nat\},\
      merge\ cmp\ (replicate\ n\ x)\ (replicate\ m\ y) =
      if cmp \ x \ y
      then replicate \ n \ x \ ++ \ replicate \ m \ y
      else replicate m y ++ replicate n x.
Fixpoint ins
   \{A: \mathsf{Type}\}\ (\mathit{cmp}: A \to A \to \mathit{bool})\ (x:A)\ (\mathit{l}:\mathit{list}\ A): \mathit{list}\ A:=
match l with
      | | | \Rightarrow |x|
      |h::t\Rightarrow if \ cmp \ x \ h \ then \ x::h::t \ else \ h::ins \ cmp \ x \ t
end.
Lemma merge\_ins\_l:
   \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{l : list A\} \{x : A\},\
      merge cmp |x| l = ins cmp x l.
Lemma merge\_ins\_r:
   \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{l : list A\} \{x : A\},\
      merge cmp \ l \ [x] = ins \ cmp \ x \ l.
Lemma merge\_ins':
   \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{l1 \ l2 : list \ A\} \{x : A\},\
      merge cmp (ins cmp x l1) (ins cmp x l2) =
      ins cmp \ x \ (ins \ cmp \ x \ (merge \ cmp \ l1 \ l2)).
Lemma merge\_all\_true:
   \forall \{A : \mathsf{Type}\} \{cmp : A \to A \to bool\} \{l : list A\} \{x : A\},\
      all (fun y: A \Rightarrow cmp \ x \ y) l = true \rightarrow
         merge cmp[x] l = x :: l.
Lemma merge\_ind:
   \forall \{A : \mathsf{Type}\}\ (P : \mathit{list}\ A \to \mathit{list}\ A \to \mathit{list}\ A \to \mathsf{Prop})
      \{f: A \to A \to bool\},\
         (\forall l: list A, P [] l l) \rightarrow
         (\forall l: list A, P l [] l) \rightarrow
         (\forall (h1 \ h2 : A) (t1 \ t2 \ r : list \ A),
            f h1 h2 = true \rightarrow
                P \ t1 \ (h2 :: t2) \ r \rightarrow P \ (h1 :: t1) \ (h2 :: t2) \ (h1 :: r)) \rightarrow
         (\forall (h1 \ h2 : A) (t1 \ t2 \ r : list \ A),
            f h1 h2 = false \rightarrow
                P \ (h1 :: t1) \ t2 \ r \rightarrow P \ (h1 :: t1) \ (h2 :: t2) \ (h2 :: r)) \rightarrow
```

```
\forall \ l1 \ l2 : list \ A, \ P \ l1 \ l2 \ (merge \ f \ l1 \ l2). Lemma merge\_filter: \forall \ \{A : \texttt{Type}\} \ \{cmp : A \rightarrow A \rightarrow bool\} \ \{p : A \rightarrow bool\} \ \{l1 \ l2 : list \ A\}, merge \ cmp \ (filter \ p \ l1) \ (filter \ p \ l2) = filter \ p \ (merge \ cmp \ l1 \ l2).
```

6.7 Rząd rżnie głupa, czyli o pierwszym i wyższym rzędzie

6.8 Rekursja wyższego rzędu (TODO)

ACHTUNG: bardzo upośledzona wersja alfa.

Pozostaje kwestia rekursji wyższego rzędu. Co to takiego? Ano dotychczas wszystkie nasze wywołania rekurencyjne były konkretne, czyli zaaplikowane do argumentów.

Mogłoby się wydawać, że jest to jedyny możliwy sposób robienia wywołań rekurencyjnych, jednak nie jest tak. Wywołania rekurencyjne mogą mieć również inną, wyższorzędową postać, a mianowicie - możemy przekazać funkcję, którą właśnie definiujemy, jako argument do innej funkcji.

Dlaczego jest to wywołanie rekurencyjne, skoro nie wywołujemy naszej funkcji? Ano dlatego, że tamta funkcja, która dostaje naszą jako argument, dostaje niejako możliwość robienia wywołań rekurencyjnych. W zależności od tego, co robi ta funkcja, wszystko może być ok (np. gdy ignoruje ona naszą funkcję i w ogóle jej nie używa) lub śmiertelnie niebezpieczne (gdy próbuje zrobić wywołanie rekurencyjne na strukturalnie większym argumencie).

Sztoby za dużo nie godoć, bajszpil:

```
\begin{array}{l} \texttt{Inductive} \ \mathit{Tree} \ (A : \texttt{Type}) : \ \texttt{Type} := \\ \mid \mathit{Node} : A \rightarrow \mathit{list} \ (\mathit{Tree} \ A) \rightarrow \mathit{Tree} \ A. \\ Arguments \ \mathit{Node} \ \{A\} \_ \_. \end{array}
```

Tree to typ drzew niepustych, które mogą mieć dowolną (ale skończoną) ilość poddrzew. Spróbujmy zdefiniować funkcję, która zwraca lustrzane odbicie drzewa.

```
*
Fixpoint mirror {A : Type} (t : Tree A) : Tree A :=
match t with
   | Node x ts => Node x (rev (map mirror ts))
end.
*)
```

Nie jest to zbyt trudne. Rekurencyjnie odbijamy wszystkie poddrzewa za pomocą map mirror, a następnie odwracamy kolejność poddrzew z użyciem rev. Chociaż poszło gładko, to mamy tu do czynienia z czymś, czego wcześniej nie widzieliśmy. Nie ma tu żadnego wywołania rekurencyjnego, a mimo to funkcja działa ok. Dlaczego? Właśnie dlatego, że

wywołania rekurencyjne są robione przez funkcję map. Mamy więc do czynienia z rekursją wyższego rzędu.

```
(*
   Require Import List.
   Import ListNotations.
   Print Forall2.
   Inductive mirrorG {A : Type} : Tree A -> Tree A -> Prop :=
  | mirrorG_0 :
       forall (x : A) (ts rs : list (Tree A)),
         Forall2 mirrorG ts rs -> mirrorG (Node x ts) (Node x (rev rs)).
   Definition mab {A B : Type} (f : A -> B) :=
  fix mab (1 : list A) : list B :=
  match 1 with
       =>
       | h :: t => f h :: mab t
  end.
   Inductive mirrorFG
  \{A : Type\}\ (f : Tree A -> Tree A) : Tree A -> Tree A -> Prop :=
    | mirrorFG_0 :
         forall (x : A) (ts : list (Tree A)),
            mirrorG (Node x ts) (Node x (rev (map f ts))).
   *)
   Inny przykład:
Inductive Tree'(A: Type): Type :=
    | Node' : A \rightarrow \forall \{B : \mathtt{Type}\}, (B \rightarrow Tree' A) \rightarrow Tree' A.
Arguments Node' \{A\} _ _ _ ...
   Tym razem mamy drzewo, które może mieć naprawdę dowolną ilość poddrzew, ale jego
poddrzewa są nieuporządkowane.
Fixpoint mirror' \{A: \mathtt{Type}\}\ (t: \mathit{Tree'}\ A): \mathit{Tree'}\ A:=
match t with
    | Node' \times B \ ts \Rightarrow Node' \times B \ (fun \ b : B \Rightarrow mirror' \ (ts \ b))
end.
```

Rozdział 7

D3: Logika boolowska, rozstrzygalność i reflekcja [TODO]

UWAGA: ten rozdział właśnie ulega konceptualnemu przeobrażeinu i może być nie na miejscu, tzn. zawierać rzeczy, o których jeszcze nie było mowy.

7.1 Logika boolowska

Zadania z funkcji boolowskich, sprawdzające radzenie sobie w pracy z enumeracjami, definiowaniem funkcji przez dopasowanie do wzorca 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.

7.2 Definicje

Zdefiniuj następujące funkcje boolowskie:

- negb (negacja)
- andb (koniunkcja)
- orb (alternatywa)
- \bullet implb (implikacja)
- eqb (równoważność)
- xorb (alternatywa wykluczająca)
- nor

nand

```
Notation "b1 && b2":= (andb \ b1 \ b2).
Notation "b1 || b2":= (orb \ b1 \ b2).
```

7.3 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 \rightarrow b1 = true.
Theorem andb\_elim\_r: b1 && b2 = true \rightarrow b2 = true.
Theorem andb\_elim: b1 && b2 = true \rightarrow b1 = true \land b2 = true.
Theorem orb\_elim: b1 \mid\mid b2 = true \rightarrow 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 \mid\mid b = b.
Theorem orb\_false\_neutral\_r: b \mid\mid 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 \mid\mid b = true. Theorem orb\_true\_annihilation\_r: b \mid\mid true = true.
```

Łączność

```
Theorem andb\_assoc: b1 \&\& (b2 \&\& b3) = (b1 \&\& b2) \&\& b3. Theorem orb\_assoc: b1 \mid\mid (b2 \mid\mid b3) = (b1 \mid\mid b2) \mid\mid b3.
```

Przemienność

Theorem $andb_comm: b1 \&\& b2 = b2 \&\& b1.$

Theorem $orb_comm: b1 \mid\mid b2 = b2 \mid\mid b1.$

Rozdzielność

Theorem $andb_dist_orb$:

$$b1 \&\& (b2 || b3) = (b1 \&\& b2) || (b1 \&\& b3).$$

Theorem orb_dist_andb :

$$b1 \mid | (b2 \&\& b3) = (b1 \mid | b2) \&\& (b1 \mid | b3).$$

Wyłączony środek i niesprzeczność

Theorem $andb_negb:b$ && negb b=false.

Theorem $orb_negb: b \mid\mid negb \mid b = true.$

Prawa de Morgana

Theorem $negb_andb$: negb (b1 && b2) = negb b1 || negb b2.

Theorem $negb_orb$: negb (b1 || 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 $norb_spec$:

$$norb \ b1 \ b2 = negb \ (b1 \ || \ b2).$$

Theorem $nandb_spec$:

$$nandb \ b1 \ b2 = negb \ (b1 \ \&\& \ b2).$$

Różne

Theorem $andb_eq_orb$:

$$b1 \&\& b2 = b1 \mid b2 \to b1 = b2.$$

Theorem $all \beta_spec$:

```
(b1 && b2) || (negb b1 || negb b2) = true.
Theorem noncontradiction_bool:
  negb (eqb b (negb b)) = true.
Theorem excluded_middle_bool:
  b || negb b = true.
End boolean_functions.
```

7.4 Rozstrzygalność

```
Theorem excluded\_middle: \forall P : \texttt{Prop}, P \lor \neg 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 $\forall P: \texttt{Prop}, P \lor \neg P$ jest przyjęcie go jako aksjomat.

```
Theorem True\_dec: True \lor \neg 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 rozstrzy-

galne, 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

```
Čwiczenie Theorem eq\_nat\_dec: \forall n \ m : nat, \ n = m \lor \neg n = m.
```

7.4.1 Techniczne apekty rozstrzygalności

Podsumowując powyższe rozważania, moglibyśmy stwierdzić: zdanie P jest rozstrzygalne, jeżeli istnieje term typu $P \vee \neg P$. 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  \begin{array}{c|c} |nil\Rightarrow false\\ |cons\ h\ t\Rightarrow\\ &\text{match}\ eq\_nat\_dec\ n\ h\ \text{with}\\ &|or\_introl\ \_\Rightarrow true\\ &|\_\Rightarrow inb\_nat\ n\ t\\ &\text{end} \end{array}
```

Coq nie akceptuje powyższego kodu, racząc nas informacją o błędzie:

(* Error:

```
Incorrect elimination of eq_nat_dec n h0" in the inductive type or": the return type has sort "Type" while it should be "Prop". Elimination of an inductive object of sort Prop is not allowed on a predicate in sort Type
```

```
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 Type) 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 **Type** 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 $P \vee \neg P$ są mocno bezużyteczne.

Uszy do góry: nie wszystko stracone! Jest to tylko drobna przeszkoda, którą bardzo łatwo ominąć:

```
Inductive sumbool\ (A\ B: \texttt{Prop}): \texttt{Type} := | \texttt{left}: A \rightarrow sumbool\ A\ B | \texttt{right}: B \rightarrow sumbool\ A\ B.
```

Typ sumbool jest niemal dokładną kopią or, jednak nie żyje on w Prop, lecz w Type. 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.

7.5 Typy hybrydowe

Ostatnim z typów istotnych z punktu widzenia silnych specyfikacji jest typ o wdzięcznej nazwie sumor.

Module sumor.

```
\begin{array}{l} \texttt{Inductive} \ sumor \ (A : \texttt{Type}) \ (B : \texttt{Prop}) : \texttt{Type} := \\ \mid inleft : A \rightarrow sumor \ A \ B \\ \mid inright : B \rightarrow sumor \ A \ B. \end{array}
```

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.

7.6 Reflekcja w małej skali, czyli jak odbijać żeby się nie zmęczyć

```
Inductive even: nat \rightarrow \texttt{Prop} :=
    \mid even\theta : even 0
    | evenSS : \forall n : nat, even n \rightarrow even (S(S n)).
(*
   Function evenb (n : nat) : bool :=
   match n with
    | 0 => true
    | 1 =  false
    | S (S n') => evenb n'
   end.
   Lemma evenb_spec :
  forall n : nat, evenb n = true -> even n.
   Proof.
  intros. functional induction evenb n.
    constructor.
    congruence.
    constructor. auto.
   Qed.
   Goal even 666.
   Proof.
  apply evenb_spec. cbn. trivial.
   Qed.
   Print Unnamed_thm.
   Print evenb_spec.
   *)
```

Wrzucić tu przykład z porządkiem leksykograficznym z bloga Mondet. Dać też przykład z permutacjami?

7.7 Poradnik hodowcy, czyli jak nie rozmnażać definicji

7.8 Metoda encode-decode, czyli o rozwiązaywaniu problemów,

które sami sobie tworzymy

Jak powiedział śp. Stefan Kisielewski: "teoria typów bohatersko zwalcza problemy nieznane w żadnej innej teorii".

Rozdział 8

D4: 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.

8.1 Podstawy

8.1.1 Definicja i notacje

Zdefiniuj liczby naturalne.

```
Notation "0":= O.
Notation "1":= (S \ 0).
```

8.1.2 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 \rightarrow S \ n \neq S \ m.
```

```
Lemma S_{-}injective: \forall n \ m : nat, S \ n = S \ m \rightarrow n = m.
```

8.1.3 Poprzednik

Zdefiniuj funkcję zwracającą poprzednik danej liczby naturalnej. Poprzednikiem 0 jest 0.

```
Lemma pred_{-}\theta: pred_{-}\theta = 0.

Lemma pred_{-}Sn:

\forall n : nat, pred_{-}(S_{-}n) = n.
```

8.2 Proste działania

8.2.1 Dodawanie

Zdefiniuj dodawanie (rekurencyjnie po pierwszym argumencie) i udowodnij jego właściwości.

```
Lemma plus_0_l :
  \forall n : nat, plus 0 \ n = n.
Lemma plus_-\theta_-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 \rightarrow plus \ i \ n \neq 0.
```

```
Lemma plus\_no\_inverse\_r\_strong:

\forall n \ i : nat, \ n \neq 0 \rightarrow plus \ n \ i \neq 0.
```

8.2.2 Alternatywne definicje dodawania

Udowodnij, że poniższe alternatywne metody zdefiniowania dodawania rzeczywiście definiują dodawanie.

```
Fixpoint plus' (n m : nat) : nat :=
{\tt match}\ m\ {\tt with}
      \mid 0 \Rightarrow n
      \mid S \mid m' \Rightarrow S \mid (plus' \mid n \mid m')
end.
Lemma plus'_is_plus:
   \forall n \ m : nat, plus' n \ m = plus \ n \ m.
Fixpoint plus " (n m : nat) : nat :=
match n with
      \mid 0 \Rightarrow m
      \mid S \mid n' \Rightarrow plus'' \mid n' \mid (S \mid m)
end.
Lemma plus "_iis_plus :
   \forall n \ m : nat, plus'' \ n \ m = plus \ n \ m.
Fixpoint plus''' (n m : nat) : nat :=
{\tt match}\ m with
      \mid 0 \Rightarrow n
      S m' \Rightarrow plus''' (S n) m'
end.
Lemma plus'''_{-}is_{-}plus:
   \forall n \ m : nat, plus''' \ n \ m = plus \ n \ m.
```

8.2.3 Odejmowanie

Zdefiniuj odejmowanie i udowodnij jego właściwości.

```
Lemma minus\_pred:
\forall \ n: nat, \ minus \ n \ 1 = pred \ n.
Lemma minus\_0\_l:
\forall \ n: nat, \ minus \ 0 \ n = 0.
Lemma minus\_0\_r:
\forall \ 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.
{\tt 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.
8.2.4
          Mnożenie
Zdefiniuj mnożenie i udowodnij jego właściwości.
Lemma mult_{-}\theta_{-}l :
  \forall n : nat, mult 0 n = 0.
Lemma mult_{-}\theta_{-}r:
```

 $\forall n : nat, mult \ n \ 0 = 0.$

 $\forall n : nat, mult \ 1 \ n = n.$

 $\forall n : nat, mult \ n \ 1 = n.$

 $mult \ n \ m = mult \ m \ n.$

 $mult\ (plus\ a\ b)\ c = plus\ (mult\ a\ c)\ (mult\ b\ c).$

Lemma $mult_plus_distr_r$:

Lemma $mult_minus_distr_l$:

Lemma $mult_1l$:

Lemma $mult_1r:$

Lemma $mult_comm$: $\forall n \ m : nat$,

 $\forall a b c : nat$,

```
\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_2plus:
  \forall n : nat, mult (S(S(0))) = plus n n.
```

8.2.5 Potęgowanie

Lemma pow_mult :

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).
```

```
\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).
```

8.3 Porządek

8.3.1 Porządek \leq

Zdefiniuj relację "mniejszy lub równy" i udowodnij jej właściwości.

Notation $\hat{\mathbf{n}} \leq \mathbf{m}" := (le \ n \ m).$

Lemma $le_-\theta_-n$:

 $\forall n : nat, 0 \leq n.$

Lemma $le_{-}n_{-}Sm$:

 $\forall n \ m : nat, n \leq m \rightarrow n \leq S \ m.$

Lemma le_Sn_m :

 $\forall n \ m : nat, S \ n \leq m \rightarrow n \leq m.$

Lemma $le_{-}n_{-}S$:

 $\forall n \ m : nat, n \leq m \rightarrow S \ n \leq S \ m.$

Lemma le_-S_-n :

 $\forall n \ m : nat, S \ n \leq S \ m \rightarrow n \leq m.$

Lemma le_-Sn_-n :

 $\forall n : nat, \neg S \ n \leq n.$

Lemma le_refl :

 $\forall n : nat, n < n.$

Lemma le_trans :

 $\forall a \ b \ c : nat,$

 $a \le b \to b \le c \to a \le c$.

Lemma $le_antisym$:

 $\forall n m : nat,$

 $n \leq m \rightarrow m \leq n \rightarrow n = m$.

Lemma le_pred :

 $\forall n : nat, pred n \leq n.$

Lemma le_npred :

 $\forall n m : nat,$

 $n \leq m \rightarrow pred \ n \leq pred \ m.$

Lemma $no_le_pred_n$:

```
\neg \ \forall \ n \ m : nat,
pred \ n \leq pred \ m \rightarrow n \leq m.
```

Lemma le_plus_l :

$$\forall \ a \ b \ c: \ nat, \\ b \leq c \rightarrow plus \ a \ b \leq plus \ a \ c.$$

Lemma le_plus_r :

$$\forall a b c : nat,$$

$$a \leq b \rightarrow plus \ a \ c \leq plus \ b \ c.$$

Lemma le_plus :

$$\forall a b c d : nat,$$

$$a \le b \to c \le d \to plus \ a \ c \le plus \ b \ d.$$

Lemma le_minus_S :

$$\forall n m : nat,$$

$$minus \ n \ (S \ m) \leq minus \ n \ m.$$

Lemma le_minus_l :

$$\forall a b c : nat$$
,

$$b \le c \to minus \ a \ c \le minus \ a \ b.$$

Lemma le_minus_r :

$$\forall a b c : nat,$$

$$a \leq b \rightarrow minus \ a \ c \leq minus \ b \ c.$$

Lemma le_-mult_-l :

$$\forall a b c : nat$$
,

$$b \le c \to mult \ a \ b \le mult \ a \ c.$$

Lemma $le_{-}mult_{-}r$:

$$\forall a b c : nat$$
,

$$a \leq b \rightarrow mult \ a \ c \leq mult \ b \ c.$$

Lemma $le_{-}mult$:

$$\forall a b c d : nat,$$

$$a \leq b \rightarrow c \leq d \rightarrow mult \ a \ c \leq mult \ b \ d.$$

Lemma le_plus_exists :

$$\forall n \ m : nat,$$

$$n \leq m \rightarrow \exists k : nat, plus \ n \ k = m.$$

Lemma le_pow_l :

$$\forall a b c : nat,$$

$$a \neq 0 \rightarrow b \leq c \rightarrow pow \ a \ b \leq pow \ a \ c.$$

Lemma $le_{-}pow_{-}r$:

$$\forall a b c : nat$$
,

$$a \leq b \rightarrow pow \ a \ c \leq pow \ b \ c.$$

8.3.2 Porządek <

```
Definition lt\ (n\ m:nat): \operatorname{Prop}:=S\ n \leq m. Notation n < m":=(lt\ n\ m). Lemma lt\_irrefl: \forall\ n:nat, \neg\ n < n. Lemma lt\_trans: \forall\ a\ b\ c:nat,\ a < b \to b < c \to a < c. Lemma lt\_asym: \forall\ n\ m:nat,\ n < m \to \neg\ m < n.
```

8.3.3 Minimum i maksimum

Zdefiniuj operacje brania minimum i maksimum z dwóch liczb naturalnych oraz udowodnij ich właściwości.

```
Lemma min_-\theta_-l:
  \forall n : nat, min \ 0 \ n = 0.
Lemma min_-\theta_-r:
  \forall n : nat, min \ n \ 0 = 0.
Lemma max_{-}\theta_{-}l :
  \forall n : nat, max \ 0 \ n = n.
Lemma max_{-}\theta_{-}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)).
```

8.4 Rozstrzygalność

8.4.1 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.
```

8.4.2 Rozstrzygalność równości

Zdefiniuj funkcję eqb, która sprawdza, czy n = m.

```
Lemma eqb\_spec:

\forall n \ m : nat,

n = m \leftrightarrow eqb \ n \ m = true.
```

8.5 Dzielenie i podzielność

 $(P: nat \rightarrow Prop) (H0: P 0) (H1: P 1)$

8.5.1 Dzielenie przez 2

Fixpoint nat_ind_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.

```
(HSS: \forall n: nat, P n \rightarrow P (S (S n))) (n: nat): P n.
    Zdefiniuj dzielenie całkowitoliczbowe przez 2 oraz funkcję obliczająca resztę z dzielenia
przez 2.
Notation "2":= (S (S 0)).
Lemma div2_even:
  \forall n : nat, div2 (mult 2 n) = n.
Lemma div2\_odd:
  \forall n : nat, div2 (S (mult 2 n)) = n.
Lemma mod2\_even:
  \forall n : nat, mod2 (mult 2 n) = 0.
Lemma mod2\_odd:
  \forall n : nat, mod2 (S (mult 2 n)) = 1.
Lemma div2\_mod2\_spec:
  \forall n : nat, plus (mult 2 (div2 n)) (mod2 n) = n.
Lemma div2\_le :
  \forall n : nat, div2 \ n \leq n.
Lemma div2\_pres\_le:
  \forall n \ m : nat, n \leq m \rightarrow div2 \ n \leq div2 \ m.
Lemma mod2\_le:
  \forall n : nat, mod 2 \ n < n.
Lemma mod2\_not\_pres\_e:
  \exists n \ m : nat, n \leq m \land mod2 \ m \leq mod2 \ n.
Lemma div2\_lt :
  \forall n : nat,
     0 \neq n \rightarrow div2 \ n < n.
```

8.5.2 Podzielność

```
Definition divides (k \ n : nat) : Prop :=
   \exists m : nat, mult \ k \ m = n.
Notation "k \mid n":= (divides \ k \ n) (at level 40).
     k dzieli n jeżeli n jest wielokrotnością k. Udowodnij podstawowe właściwości tej relacji.
Lemma divides_{-}\theta :
   \forall n : nat, n \mid 0.
Lemma not\_divides\_\theta:
   \forall n : nat, n \neq 0 \rightarrow \neg 0 \mid n.
Lemma divides\_1:
   \forall n : nat, 1 \mid n.
Lemma divides\_reft:
   \forall n : nat, n \mid n.
{\tt 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 9

D5: 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)\ ..).
```

9.1 Proste funkcje

9.1.1 is Empty

Zdefiniuj funkcję isEmpty, która sprawdza, czy lista jest pusta.

9.1.2 length

```
Zdefiniuj funkcję length, która oblicza długość listy. 
 Przykład: length [1; 2; 3] = 3
 Lemma length\_nil:
```

```
orall A: {\tt Type}, \ length \ (@nil \ A) = 0. Lemma length\_cons: \forall \ (A: {\tt Type}) \ (h:A) \ (t: list \ A), \exists \ n: nat, \ length \ (h::t) = S \ n. Lemma length\_0: \forall \ (A: {\tt Type}) \ (l: list \ A), length \ l = 0 \ \rightarrow l = \|.
```

9.1.3 snoc

Zdefiniuj funkcję snoc, która dokleja element x na koniec listy l. Przykład: snoc 42 [1; 2; 3] = [1; 2; 3; 42]

Lemma $snoc_isEmpty$:

$$\forall (A: \mathtt{Type}) (x:A) (l: \mathit{list}\ A), \\ \mathit{isEmpty}\ l = \mathit{true} \rightarrow \mathit{snoc}\ x\ l = [x].$$

Lemma $isEmpty_snoc$:

$$\forall (A : \mathtt{Type}) (x : A) (l : \mathit{list} \ A), isEmpty (snoc x \ l) = \mathit{false}.$$

Lemma $length_snoc$:

$$\forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A), \\ \mathit{length} (\mathit{snoc} \ x \ l) = S (\mathit{length} \ l).$$

Lemma $snoc_inv$:

$$\forall$$
 $(A: \mathsf{Type})$ $(l1\ l2: list\ A)\ (x\ y: A),$ $snoc\ x\ l1 = snoc\ y\ l2 \to x = y \land l1 = l2.$

9.1.4 app

Zdefiniuj funkcję app, która skleja dwie listy.

Przykład:
$$app [1; 2; 3] [4; 5; 6] = [1; 2; 3; 4; 5; 6]$$

Notation $l1 ++ l2" := (app \ l1 \ l2).$

Lemma app_nil_l :

$$\forall \ (A: \mathtt{Type}) \ (l: \mathit{list} \ A), \\ [] \ ++ \ l = \mathit{l}.$$

Lemma app_nil_r :

$$\forall (A : \mathtt{Type}) (l : \mathit{list} \ A), \ l ++ [] = l.$$

Lemma app_assoc :

$$orall \; (A: { t Type}) \; (l1 \; l2 \; l3: \; list \; A), \ l1 \; ++ \; (l2 \; ++ \; l3) = (l1 \; ++ \; l2) \; ++ \; l3.$$

```
Lemma isEmpty\_app :
```

$$\forall (A: \mathtt{Type}) \ (l1 \ l2: list \ A), \ is Empty \ (l1 \ ++ \ l2) = andb \ (is Empty \ l1) \ (is Empty \ l2).$$

Lemma $length_app$:

$$orall \; (A: { t Type}) \; (l1 \; l2: list \; A), \ length \; (l1 \; ++ \; l2) = length \; l1 \; + \; length \; l2.$$

Lemma $snoc_app$:

$$\forall (A: {\tt Type}) \; (x:A) \; (l1\; l2: list\; A), \\ snoc \; x \; (l1\; ++\; l2) = l1\; ++\; snoc \; x \; l2.$$

Lemma app_snoc_l :

$$\forall (A: \mathtt{Type}) (x:A) (l1 \ l2: list \ A), snoc x \ l1 \ ++ \ l2 = l1 \ ++ \ x :: l2.$$

Lemma app_snoc_r :

$$\forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A), \\ l1 \ ++ \ snoc \ x \ l2 = snoc \ x \ (l1 \ ++ \ l2).$$

Lemma $snoc_app_singl$:

$$\forall (A : \texttt{Type}) (x : A) (l : list A), snoc x l = l ++ [x].$$

Lemma app_cons_l :

$$\forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A), (x :: l1) ++ l2 = x :: (l1 ++ l2).$$

Lemma app_cons_r :

$$\forall (A : \mathtt{Type}) (x : A) (l1 \ l2 : list \ A), \ l1 \ ++ \ x :: l2 = (l1 \ ++ \ [x]) \ ++ \ l2.$$

Lemma $no_infinite_cons$:

$$\forall (A: \mathtt{Type}) (x:A) (l:\mathit{list}\ A), \ l=x::l o \mathit{False}.$$

Lemma $no_infinite_app$:

$$\forall (A: \mathtt{Type}) \ (l \ l': \mathit{list} \ A), \ l' \neq [] \rightarrow l = l' ++ l \rightarrow \mathit{False}.$$

Lemma $app_{-}inv_{-}l$:

$$\forall (A: \mathtt{Type}) (l \ l1 \ l2: list \ A),$$

 $l++ \ l1 = l++ \ l2 \rightarrow l1 = l2.$

Lemma app_inv_r :

$$\forall (A: {\tt Type}) \ (l \ l1 \ l2: list \ A), \ l1 \ ++ \ l = l2 \ ++ \ l \ o l1 \ = l2.$$

Lemma $app_{-}eq_{-}nil$:

$$\forall (A : \texttt{Type}) (l1 \ l2 : list \ A), \ l1 \ ++ \ l2 = [] \rightarrow l1 = [] \land l2 = [].$$

$9.1.5 \quad rev$

```
Zdefiniuj funkcję rev, która odwraca listę.
    Przykład: rev [1; 2; 3] = [3; 2; 1]
Lemma isEmpty\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      isEmpty (rev \ l) = isEmpty \ l.
Lemma length\_rev:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      length (rev l) = length l.
Lemma snoc\_rev:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      snoc \ x \ (rev \ l) = rev \ (x :: l).
Lemma rev\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      rev (snoc x l) = x :: rev l.
Lemma rev\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      rev (l1 ++ l2) = rev l2 ++ rev l1.
Lemma rev_inv:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      rev(rev l) = l.
9.1.6
            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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      map id l = l.
Lemma map\_map:
   \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B) \ (g : B \to C) \ (l : \mathit{list} \ A),
      map \ g \ (map \ f \ l) = map \ (fun \ x : A \Rightarrow g \ (f \ x)) \ l.
Lemma isEmpty\_map :
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A),
      isEmpty (map f l) = isEmpty l.
Lemma length_{-}map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{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.
```

9.1.7 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 $join_snoc$:

$$\forall (A: \mathtt{Type}) (x: \mathit{list}\ A) (l: \mathit{list}\ (\mathit{list}\ A)), \ \mathit{join}\ (\mathit{snoc}\ x\ l) = \mathit{join}\ l++x.$$

Lemma $join_app$:

$$\forall (A: \mathtt{Type}) \ (l1 \ l2: list \ (list \ A)), \\ join \ (l1 \ ++ \ l2) = join \ l1 \ ++ \ join \ l2.$$

Lemma $rev_{-}join$:

$$\forall (A : \mathtt{Type}) (l : list (list A)),$$
 $rev (join \ l) = join (rev (map \ rev \ l)).$

Lemma map_join :

$$\forall (A \ B : \mathtt{Type}) \ (f : A \rightarrow B) \ (l : list \ (list \ A)),$$

 $map \ f \ (join \ l) = join \ (map \ (map \ f) \ l).$

9.1.8 bind

Napisz funkcję bind, która spełnia specyfikację $bind_spec$. Użyj rekursji, ale nie używaj funkcji join ani map.

Lemma $bind_spec$:

```
\forall (A \ B : \mathtt{Type}) \ (f : A \rightarrow list \ B) \ (l : list \ A), bind f \ l = join \ (map \ f \ l).
```

Lemma $bind_snoc$:

```
\forall (A \ B : \mathtt{Type}) \ (f : A \rightarrow list \ B) \ (x : A) \ (l : list \ A), \ bind \ f \ (snoc \ x \ l) = bind \ f \ l ++ f \ x.
```

9.1.9 replicate

```
Napisz funkcję replicate, która powiela dany element n razy, tworząc listę. Przykład: replicate 5 0 = [0; 0; 0; 0]
```

```
Definition isZero\ (n:nat):bool:= match n with  \mid 0 \Rightarrow true \\ \mid \_ \Rightarrow false end.
```

Lemma $isEmpty_replicate$:

$$\forall (A : \mathsf{Type}) \ (n : nat) \ (x : A),$$
 $isEmpty \ (replicate \ n \ x) = \mathsf{if} \ isZero \ n \ \mathsf{then} \ true \ \mathsf{else} \ false.$

Lemma $length_replicate$:

$$\forall (A: \mathtt{Type}) (n: nat) (x: A),$$

 $length (replicate \ n \ x) = n.$

Lemma $snoc_replicate$:

$$\forall (A : \mathsf{Type}) (x : A) (n : nat),$$

 $snoc \ x \ (replicate \ n \ x) = replicate \ (S \ n) \ x.$

Lemma $replicate_plus$:

$$\forall (A: \mathtt{Type}) \ (n \ m: nat) \ (x:A),$$
 $replicate \ (n+m) \ x = replicate \ n \ x ++ replicate \ m \ x.$

Lemma $replicate_plus_comm$:

```
\forall (A : \mathtt{Type}) (n \ m : nat) (x : A),
replicate (n + m) x = replicate (m + n) x.
```

Lemma $rev_replicate$:

$$\forall (A : \mathtt{Type}) (n : nat) (x : A),$$

 $rev (replicate \ n \ x) = replicate \ n \ x.$

Lemma $map_replicate$:

$$\forall (A \ B : \mathtt{Type}) \ (f : A \rightarrow B) \ (n : nat) \ (x : A),$$
 map $f \ (replicate \ n \ x) = replicate \ n \ (f \ x).$

9.1.10 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 : \mathsf{Type}) (f : A \to A) (n : nat) (x : A),
     isEmpty\ (iterate\ f\ n\ x) =
     match n with
           \mid 0 \Rightarrow true
           | \_ \Rightarrow false
     end.
Lemma length\_iterate:
  \forall (A : \mathsf{Type}) (f : A \to A) (n : nat) (x : A),
     length (iterate f n x) = n.
Lemma snoc\_iterate:
  \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (n : nat) (x : A),
     iterate id n x = replicate n x.
```

9.1.11 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:
   \forall (A : \mathsf{Type}), head [] = (@None A).
Lemma head\_cons:
  \forall (A : \mathsf{Type}) (h : A) (t : \mathit{list} A),
      head (h :: t) = Some h.
Lemma head\_isEmpty\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      isEmpty\ l=true \rightarrow head\ l=None.
\forall (A : \mathsf{Type}) (l : \mathit{list} A),
      head \ l \neq None \rightarrow isEmpty \ l = false.
Lemma head\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      head (snoc x l) =
      if isEmpty \ l then Some \ x else head \ l.
Lemma head\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      head (l1 ++ l2) =
     match l1 with
           | | | \Rightarrow head l2
            | h :: \_ \Rightarrow Some h
      end.
Lemma head\_map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A),
      head (map f l) =
     {\tt match}\ l with
           | | | \Rightarrow None
            | h :: \_ \Rightarrow Some (f h)
      end.
Lemma head\_replicate\_S:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      head\ (replicate\ (S\ n)\ x) = Some\ x.
Lemma head\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      head (replicate \ n \ x) =
     {\tt match}\ n\ {\tt with}
            | 0 \Rightarrow None
            | \  \  \Rightarrow Some \ x
      end.
```

```
Lemma head\_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A) (n : nat) (x : A),
      head\ (iterate\ f\ n\ x) =
      {\tt match}\ n\ {\tt with}
            \mid 0 \Rightarrow None
            \mid S \mid n' \Rightarrow Some \mid 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:
   \forall A : \mathsf{Type}, \ tail \ (@nil \ A) = None.
Lemma tail\_cons:
   \forall (A : \mathsf{Type}) (h : A) (t : \mathit{list} A),
      tail\ (h::t) = Some\ t.
Lemma tail\_isEmpty\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      isEmpty\ l=true \rightarrow tail\ l=None.
Lemma isEmpty\_tail\_not\_None:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      tail \ l \neq None \rightarrow isEmpty \ l = false.
Lemma tail\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      tail (snoc x l) =
      match \ tail \ l \ with
            | None \Rightarrow Some []
            | Some \ t \Rightarrow Some \ (snoc \ x \ t) |
      end.
Lemma tail\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      tail (l1 ++ l2) =
      match l1 with
            | | | \Rightarrow tail \ l2
            | h :: t \Rightarrow Some (t ++ l2)
      end.
Lemma tail\_map:
```

 $\forall (A B : \mathsf{Type}) (f : A \to B) (l : list A),$

tail (map f l) =

```
{\tt match}\ l\ {\tt with}
            | | | \Rightarrow None
            end.
Lemma tail\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      tail\ (replicate\ n\ x) =
      {\tt match}\ n\ {\tt with}
             \mid 0 \Rightarrow None
             \mid S \mid n' \Rightarrow Some \ (replicate \mid n' \mid x)
      end.
Lemma tail\_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A) (n : nat) (x : A),
      tail\ (iterate\ f\ n\ x) =
      {\tt match}\ n\ {\tt with}
             | 0 \Rightarrow None
             \mid S \mid n' \Rightarrow Some \ (iterate \ f \mid n' \ (f \mid 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:
\forall (A: \texttt{Type}) \ (l: list \ A),
uncons \ l =
\texttt{match} \ head \ l, \ tail \ l \ \texttt{with}
\mid Some \ h, \ Some \ t \Rightarrow Some \ (h, \ t)
\mid \_, \_ \Rightarrow None
end.
```

9.1.12 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 $last_nil$:

```
\forall (A : \mathsf{Type}), last [] = (@None A).
```

Lemma $last_isEmpty_true$:

```
\forall (A : \mathsf{Type}) (l : \mathit{list} A),
      isEmpty\ l=true \rightarrow last\ l=None.
Lemma isEmpty\_last\_not\_None:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      last \ l \neq None \rightarrow isEmpty \ l = false.
Lemma last\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      last (snoc x l) = Some x.
Lemma last\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      last (l ++ [x]) = Some x.
Lemma last\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      last (l1 ++ l2) =
      match l2 with
             | | | \Rightarrow last l1
             | \_ \Rightarrow last \ l2
      end.
Lemma last\_replicate\_S:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      last (replicate (S n) x) = Some x.
Lemma last\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      last (replicate \ n \ x) =
      match n with
             | 0 \Rightarrow None
             | \_ \Rightarrow Some \ x
      end.
Lemma last\_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A) (n : nat) (x : A),
      last (iterate f n x) =
      {\tt match}\ n\ {\tt with}
             \mid 0 \Rightarrow None
             \mid S \mid n' \Rightarrow Some (iter f \mid n' \mid 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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      init \ l = None \rightarrow l = [].
Lemma init\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      init (snoc x l) = Some l.
Lemma init\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      init (l1 ++ l2) =
      match init\ l2 with
             | None \Rightarrow init l1
             | Some i \Rightarrow Some (l1 ++ i)
      end.
Lemma init\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      init (l ++ [x]) = Some l.
Lemma init\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A),
      init (map f l) =
      {\tt match}\ l\ {\tt with}
             | | | \Rightarrow None
             | h :: t \Rightarrow
                   match init t with
                            None \Rightarrow Some
                            Some \ i \Rightarrow Some \ (map \ f \ (h :: i))
                   end
      end.
Lemma init\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      init (replicate \ n \ x) =
      {\tt match}\ n with
             \mid 0 \Rightarrow None
             \mid S \mid n' \Rightarrow Some \ (replicate \mid n' \mid x)
      end.
Lemma init\_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A) (n : nat) (x : A),
      init\ (iterate\ f\ n\ x) =
      {\tt match}\ n\ {\tt with}
             | 0 \Rightarrow None
             \mid S \mid n' \Rightarrow Some \ (iterate \ f \mid n' \mid x)
      end.
```

```
Lemma init\_last: \forall (A: \mathsf{Type}) \ (l\ l': list\ A) \ (x:A), init\ l = Some\ l' \to last\ l = Some\ x \to 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:
\forall (A : Type) (l : list A),
unsnoc \ l = None \rightarrow l = [].

Lemma unsnoc\_spec:
\forall (A : Type) (l : list A),
unsnoc \ l =
match \ last \ l, \ init \ l \ with
| \ Some \ x, \ Some \ l' \Rightarrow Some \ (x, \ l')
| \ \_, \ \_ \Rightarrow None
end.
```

Dualności head i last, tail i init oraz ciekawostki

```
Lemma last\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      last (rev l) = head l.
Lemma head\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      head (rev l) = last l.
Lemma tail\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      tail (rev l) =
      match \ init \ l \ with
             | None \Rightarrow None |
             | Some t \Rightarrow Some (rev t) |
      end.
Lemma init\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      init (rev l) =
      match tail l with
             | None \Rightarrow None
```

```
| Some \ t \Rightarrow Some \ (rev \ t) |
      end.
Lemma init\_decomposition:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      l = [] \vee
      \exists (h : A) (t : list A),
         init\ l = Some\ t \wedge last\ l = Some\ h \wedge l = t ++ [h].
      end hide *)
Lemma bilateral\_decomposition:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      l = [] \vee
      (\exists x : A, l = [x]) \lor
      \exists (x \ y : A) \ (l' : list \ A), \ l = x :: l' ++ [y].
9.1.13
               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:
   \forall (A : \mathsf{Type}) (n : nat),
      nth \ n \ (@nil \ A) = None.
Lemma nth\_isEmpty\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      isEmpty\ l=true \rightarrow nth\ n\ l=None.
Lemma isEmpty\_nth\_not\_None:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      nth \ n \ l \neq None \rightarrow isEmpty \ l = false.
Lemma nth\_length\_lt:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length \ l \rightarrow \exists \ x : A, \ nth \ n \ l = Some \ x.
Lemma nth\_length\_ge:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length l \leq n \rightarrow nth \ n \ l = None.
Lemma nth\_snoc\_length\_lt:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length \ l \rightarrow nth \ n \ (snoc \ x \ l) = nth \ n \ l.
```

Lemma $nth_snoc_length_eq$:

```
\forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      nth (length l) (snoc x l) = Some x.
Lemma nth\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n : \mathit{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:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      nth \ n \ (l1 \ ++ \ l2) =
      match nth n l1 with
             | None \Rightarrow nth (n - length l1) l2
             | Some x \Rightarrow Some x
      end.
Lemma nth\_app\_l:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      n < length \ l1 \rightarrow nth \ n \ (l1 ++ l2) = nth \ n \ l1.
Lemma nth\_app\_r:
   \forall (A : \mathsf{Type}) (n : nat) (l1 \ l2 : list \ A),
      length \ l1 \leq n \rightarrow nth \ n \ (l1 ++ l2) = nth \ (n - length \ l1) \ l2.
Lemma nth\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length \ l \rightarrow nth \ n \ (rev \ l) = nth \ (length \ l - S \ n) \ l.
Lemma nth-None:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      nth \ n \ l = None \rightarrow length \ l \leq n.
Lemma nth\_Some:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      nth \ n \ l = Some \ x \rightarrow n < length \ l.
Lemma nth\_spec':
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      match nth \ n \ l with
             | None \Rightarrow length | l < n
            \mid Some \ x \Rightarrow \exists \ l1 \ l2 : list \ A,
                                   l = l1 ++ x :: l2 \wedge length \ l1 = n
      end.
Lemma nth\_map\_Some:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (n : nat) (x : A),
      nth \ n \ l = Some \ x \rightarrow nth \ n \ (map \ f \ l) = Some \ (f \ x).
```

```
Lemma nth_{-}map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (n : nat),
      nth \ n \ (map \ f \ l) =
      match nth \ n \ l with
            | None \Rightarrow None
            \mid Some \ x \Rightarrow Some \ (f \ x)
      end.
Lemma nth\_replicate:
   \forall (A : \mathsf{Type}) (n \ i : nat) (x : A),
      i < n \rightarrow nth \ i \ (replicate \ n \ x) = Some \ x.
Lemma nth_{-}iterate:
   \forall (A : \mathsf{Type}) (f : A \to 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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      nth \ 0 \ l = head \ l.
Lemma last_nth:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      last l = nth (length l - 1) l.
9.1.14
               take
Zdefiniuj funkcję take, która bierze z listy n początkowych elementów.
    Przykład:
    take \ 2 \ [1; \ 2; \ 3] = [1; \ 2]
Lemma take_{-}\theta:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      take \ 0 \ l = [].
Lemma take\_nil:
   \forall (A : \mathsf{Type}) (n : nat),
      take \ n \ [] = @nil \ A.
Lemma take\_S\_cons:
   \forall (A : \mathsf{Type}) (n : nat) (h : A) (t : list A),
      take (S n) (h :: t) = h :: take n t.
Lemma isEmpty\_take:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      isEmpty\ (take\ n\ l)=orb\ (beq\_nat\ 0\ n)\ (isEmpty\ l).
Lemma take\_length:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
```

```
take\ (length\ l)\ l=l.
Lemma take\_length':
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length \ l \leq n \rightarrow take \ n \ l = l.
Lemma length\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length (take \ n \ l) = min (length \ l) \ n.
(* TODO: zabij *) Lemma take\_snoc\_lt:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A) (n : \mathit{nat}),
      n < length \ l \rightarrow take \ n \ (snoc \ x \ l) = take \ n \ l.
Lemma take\_snoc\_le:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A) (n : \mathit{nat}),
      n \leq length \ l \rightarrow take \ n \ (snoc \ x \ l) = take \ n \ l.
Lemma take\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      take \ n \ (l1 \ ++ \ l2) = take \ n \ l1 \ ++ \ take \ (n \ - \ length \ l1) \ l2.
Lemma take\_app\_l:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      n \leq length \ l1 \rightarrow take \ n \ (l1 ++ l2) = take \ n \ l1.
Lemma take\_app\_r:
   \forall (A : \mathsf{Type}) (n : nat) (l1 \ l2 : list \ A),
      length l1 < n \rightarrow
         take \ n \ (l1 ++ l2) = l1 ++ take \ (n - length \ l1) \ l2.
Lemma take\_map:
   \forall (A B : \mathsf{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 : \mathsf{Type}) (n \ m : nat) (x : A),
      take \ m \ (replicate \ n \ x) = replicate \ (min \ n \ m) \ x.
Lemma take\_iterate:
   \forall (A : \mathsf{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 : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      head (take \ n \ l) =
      if beq\_nat \ 0 n then None else head \ l.
Lemma last\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      last\ (take\ (S\ n)\ l) = nth\ (min\ (length\ l-1)\ n)\ l.
```

```
Lemma tail\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       tail (take \ n \ l) =
      match n, l with
              \mid 0, \bot \Rightarrow None
              | _{-}, [] \Rightarrow None
             \mid S \mid n', h :: t \Rightarrow Some (take \mid n' \mid t)
       end.
Lemma init\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       init (take \ n \ l) =
      match n, l with
              \mid 0, \bot \Rightarrow None
             | -, [] \Rightarrow None
              \mid S \mid n', h :: t \Rightarrow Some (take (min \mid n' (length \mid l - 1)) \mid l)
       end.
Lemma nth\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat),
       nth \ m \ (take \ n \ l) =
       if leb (S m) n then nth m l else None.
Lemma take\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat),
       take \ m \ (take \ n \ l) = take \ (min \ n \ m) \ l.
Lemma take\_interesting:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       (\forall n : nat, take \ n \ l1 = take \ n \ l2) \rightarrow l1 = l2.
                drop
```

9.1.15

Zdefiniuj funkcję drop, która wyrzuca z listy n początkowych elementów i zwraca to, co zostało.

```
Przykład:
     drop \ 2 \ [1; \ 2; \ 3] = [3]
Lemma drop_{-}\theta:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      drop \ 0 \ l = l.
Lemma drop_nil:
   \forall (A : \mathsf{Type}) (n : nat),
      drop \ n \ [] = @nil \ A.
Lemma drop\_S\_cons:
```

```
\forall (A : \mathsf{Type}) (n : nat) (h : A) (t : list A),
      drop (S n) (h :: t) = drop n t.
Lemma isEmpty\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      isEmpty\ (drop\ n\ l) = leb\ (length\ l)\ n.
Lemma drop\_length:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      drop\ (length\ l)\ l = [].
Lemma drop\_length':
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length l \leq n \rightarrow drop \ n \ l = [].
Lemma length\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length (drop \ n \ l) = length \ l - n.
Lemma drop\_snoc\_le:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n \leq length \ l \rightarrow drop \ n \ (snoc \ x \ l) = snoc \ x \ (drop \ n \ l).
Lemma drop\_app :
   \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (f : A \to B) (l : list A) (n : nat),
       drop \ n \ (map \ f \ l) = map \ f \ (drop \ n \ l).
Lemma drop\_replicate:
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      drop \ m \ (replicate \ n \ x) = replicate \ (n - m) \ x.
Lemma drop_{-}iterate:
   \forall (A : \mathsf{Type}) (f : A \to 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 : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
```

 $head (drop \ n \ l) = nth \ n \ l.$

```
Lemma last\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      last\ (drop\ n\ l) = if\ leb\ (S\ n)\ (length\ l) then last\ l else None.
Lemma tail\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      tail (drop \ n \ l) =
      if leb (S n) (length l) then Some (drop (S n) l) else None.
Lemma init\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      init (drop \ n \ l) =
      if n < ? length l
      then
         {\tt match}\ init\ l\ {\tt with}
                 None \Rightarrow None
                | Some l' \Rightarrow Some (drop n l')
          end
      else None.
Lemma nth\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat),
      nth \ m \ (drop \ n \ l) = nth \ (n + m) \ l.
Lemma nth\_spec\_Some:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      nth \ n \ l = Some \ x \rightarrow l = take \ n \ l ++ x :: drop \ (S \ n) \ l.
Lemma nth\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      match nth \ n \ l with
             | None \Rightarrow length | l < n
             |Some x \Rightarrow l = take \ n \ l ++ x :: drop \ (S \ n) \ l
      end.
Lemma drop\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat),
      drop \ m \ (drop \ n \ l) = drop \ (n + m) \ l.
Lemma drop\_not\_so\_interesting:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      (\forall n: nat, drop \ n \ l1 = drop \ n \ l2) \rightarrow l1 = l2.
Dualność take i drop
```

```
Lemma take\_rev:
 \forall (A : Type) (l : list A) (n : nat),
```

```
take \ n \ (rev \ l) = rev \ (drop \ (length \ l - n) \ l).
Lemma rev\_take:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      rev (take \ n \ l) = drop (length \ l - n) (rev \ l).
Lemma drop\_rev:
  \forall (A : \mathsf{Type}) (n : nat) (l : list A),
      drop \ n \ (rev \ l) = rev \ (take \ (length \ l - n) \ l).
Lemma take\_drop:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat),
      take \ m \ (drop \ n \ l) = drop \ n \ (take \ (n + m) \ l).
Lemma drop_{-}take:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat),
      drop \ m \ (take \ n \ l) = take \ (n - m) \ (drop \ m \ l).
Lemma app\_take\_drop:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      take \ n \ l ++ \ drop \ n \ l = l.
9.1.16
              cycle
Napisz funkcję cycle: \forall A: Type, nat \rightarrow list A \rightarrow list A, która obraca listę cyklicznie.
Udowodnij jej właściwości.
Compute cycle\ 3\ [1;\ 2;\ 3;\ 4;\ 5].
(* ===> 4; 5; 1; 2; 3 : list nat *)
Compute cycle\ 6\ [1;\ 2;\ 3;\ 4;\ 5].
(* ===> 2; 3; 4; 5; 1 : list nat *)
Lemma cycle_{-}\theta:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      cycle\ 0\ l=l.
Lemma cycle_nil:
  \forall (A : \mathsf{Type}) (n : nat),
      @ cycle A n [] = [].
Lemma isEmpty\_cycle:
  \forall (A : \mathsf{Type}) (n : nat) (l : list A),
      isEmpty (cycle \ n \ l) = isEmpty \ l.
Lemma length\_cycle:
  \forall (A : \mathsf{Type}) (n : nat) (l : list A),
      length (cycle \ n \ l) = length \ l.
```

Lemma $cycle_length_app$:

```
\forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      cycle\ (length\ l1\ +\ n)\ (l1\ ++\ l2) = cycle\ n\ (l2\ ++\ l1).
Lemma cycle\_length:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      cycle (length l) l = l.
Lemma cycle\_plus\_length:
  \forall (A : \mathsf{Type}) (n : nat) (l : list A),
      cycle\ (length\ l+n)\ l=cycle\ n\ l.
    Łamigłówka: jaki jest związek cycle ze snoc, i rev?
Compute cycle 2 [1; 2; 3; 4; 5; 6].
Compute rev (cycle 4 (rev [1; 2; 3; 4; 5; 6])).
Lemma cycle\_map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (n : nat) (l : list A),
      cycle \ n \ (map \ f \ l) = map \ f \ (cycle \ n \ l).
    A z join i bind?
Lemma cycle\_replicate:
  \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      cycle\ m\ (replicate\ n\ x) = replicate\ n\ x.
Lemma cycle\_cycle:
  \forall (A : \mathsf{Type}) (n \ m : nat) (l : list \ A),
      cycle \ n \ (cycle \ m \ l) = cycle \ (m + n) \ l.
```

9.1.17splitAt

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:
```

```
\forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
   match splitAt n l with
          | None \Rightarrow length | l \leq n
          |Some(l1, x, l2) \Rightarrow l = l1 ++ x :: l2
   end.
```

Lemma $splitAt_spec$ ':

```
\forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (x : A) (n : \mathit{nat}),
    splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
       l1 = take \ n \ l \wedge l2 = drop \ (S \ n) \ l.
```

Lemma $splitAt_megaspec$:

```
\forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      match splitAt n l with
             | None \Rightarrow length | l \leq n
             | Some (l1, x, l2) \Rightarrow
                    nth \ n \ l = Some \ x \land 
                    l1 = take \ n \ l \wedge
                    l2 = drop (S n) l \wedge
                    l = l1 ++ x :: l2
      end.
Lemma splitAt\_isEmpty\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      isEmpty\ l=true \rightarrow \forall\ n:\ nat,\ splitAt\ n\ l=None.
Lemma isEmpty\_splitAt\_false:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      splitAt \ n \ l \neq None \rightarrow isEmpty \ l = false.
Lemma splitAt\_length\_inv:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      splitAt \ n \ l \neq None \leftrightarrow n < length \ l.
Lemma splitAt\_Some\_length:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (x : A) (n : \mathit{nat}),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow n < length \ l.
Lemma splitAt\_length\_lt:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length \ l \rightarrow \exists \ x : A,
          splitAt \ n \ l = Some \ (take \ n \ l, \ x, \ drop \ (S \ n) \ l).
Lemma splitAt\_length\_ge:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length l \leq n \rightarrow splitAt \ n \ l = None.
Lemma splitAt\_snoc:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      splitAt \ n \ (snoc \ x \ l) =
      if n < ? length l
      then
         match splitAt \ n \ l with
                 | None \Rightarrow None
                |Some(b, y, e)| \Rightarrow Some(b, y, snoc x e)
          end
      else
          if beq_nat \ n \ (length \ l)
          then Some (l, x, ||)
```

```
else None.
Lemma splitAt_-app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      splitAt \ n \ (l1 \ ++ \ l2) =
     match splitAt n l1 with
            |Some(l11, x, l12) \Rightarrow Some(l11, x, l12 ++ l2)
            | None \Rightarrow
                 match splitAt (n - length l1) l2 with
                         Some\ (l21,\ x,\ l22) \Rightarrow Some\ (l1\ ++\ l21,\ x,\ l22)
                         None \Rightarrow None
                  end
      end.
Lemma splitAt\_app\_lt:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      n < length l1 \rightarrow
         splitAt \ n \ (l1 \ ++ \ l2) =
         match splitAt n l1 with
                None \Rightarrow None
               | Some (x, l11, l12) \Rightarrow Some (x, l11, l12 ++ l2)
         end.
Lemma splitAt\_app\_ge:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      length l1 \leq n \rightarrow
         splitAt \ n \ (l1 \ ++ \ l2) =
         match splitAt (n - length l1) l2 with
               | None \Rightarrow None |
               |Some(l21, x, l22)| \Rightarrow Some(l1 ++ l21, x, l22)|
         end.
Lemma splitAt\_rev\_aux:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length l \rightarrow
         splitAt \ n \ l =
         match splitAt (length \ l - S \ n) (rev \ l) with
               | None \Rightarrow None
               | Some (l1, x, l2) \Rightarrow Some (rev l2, x, rev l1)
         end.
Lemma splitAt\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length l \rightarrow
         splitAt \ n \ (rev \ l) =
         match splitAt (length l - S n) l with
```

```
| None \Rightarrow None |
               | Some (l1, x, l2) \Rightarrow Some (rev l2, x, rev l1)
         end.
Lemma splitAt_{-}map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A) (n : \mathit{nat}),
      splitAt \ n \ (map \ f \ l) =
      match splitAt n l with
            | None \Rightarrow None
            |Some(l1, x, l2)| \Rightarrow Some(map f l1, f x, map f l2)
      end.
Lemma splitAt\_replicate:
  \forall (A : \mathsf{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:
   \forall (A : \mathsf{Type}) (f : A \to 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:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (x : A) (n : \mathit{nat}),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         head l1 =
         match n with
               \mid 0 \Rightarrow None
               | \_ \Rightarrow head l
         end.
Lemma splitAt\_head\_r:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (x : A) (n : \mathit{nat}),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         head l2 = nth (S n) l.
Lemma splitAt\_last\_l:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (x : A) (n : \mathit{nat}),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         last l1 =
         match n with
               \mid 0 \Rightarrow None
               \mid S \mid n' \Rightarrow nth \mid n' \mid l
```

```
end.
```

```
Lemma splitAt\_last\_r:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (x : A) (n : \mathit{nat}),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         last l2 =
         if length \ l <= ? \ S \ n
         then None
         else last l2.
(* TODO: init, unsnoc *)
Lemma take\_splitAt:
  \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (n \ m : nat) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         take \ m \ l1 = take \ (min \ n \ m) \ l.
Lemma take\_splitAt':
  \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         take \ m \ l2 = take \ m \ (drop \ (S \ n) \ l).
Lemma drop\_splitAt\_l:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (n \ m : nat) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         drop \ m \ l1 = take \ (n - m) \ (drop \ m \ l).
Lemma drop\_splitAt\_r :
  \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (n \ m : nat) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         drop \ m \ l2 = drop \ (S \ n + m) \ l.
9.1.18
               insert
Napisz funkcję insert, która wstawia do listy l na n-tą pozycję element x.
    Przykład:
    insert [1; 2; 3; 4; 5] [2] [42] [42; 42; 3; 4; 5]
Lemma insert_{-}\theta:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      insert l \ 0 \ x = x :: l.
Lemma isEmpty\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      isEmpty\ (insert\ l\ n\ x) = false.
Lemma length\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      length (insert \ l \ n \ x) = S (length \ l).
```

```
Lemma insert\_length:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      insert l (length l) x = snoc x l.
Lemma insert\_snoc:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
      insert (snoc x l) n y =
      if n \le n length l then snoc \ x (insert l n y) else snoc \ y (snoc \ x l).
Lemma insert\_app:
   \forall (A : \mathsf{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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      insert \ (rev \ l) \ n \ x = rev \ (insert \ l \ (length \ l - n) \ x).
Lemma rev\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      rev (insert \ l \ n \ x) = insert (rev \ l) (length \ l - n) \ x.
Lemma map\_insert:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (n : nat) (x : A),
      map \ f \ (insert \ l \ n \ x) = insert \ (map \ f \ l) \ n \ (f \ x).
Lemma\ insert\_join:
   \forall (A : \mathsf{Type}) (ll : \mathit{list} (\mathit{list} A)) (n : \mathit{nat}) (x : A) (l : \mathit{list} A),
      join (insert ll n [x]) = l \rightarrow
         \exists m : nat, l = insert (join ll) m x.
Lemma insert\_replicate:
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      insert (replicate n x) m x = replicate (S n) x.
Lemma head\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      head (insert \ l \ n \ x) =
      match l, n with
             | [], \bot \Rightarrow Some \ x
             \mid -, 0 \Rightarrow Some \ x
             | \_, \_ \Rightarrow head \ l
      end.
Lemma tail\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      tail\ (insert\ l\ n\ x) =
```

```
match l, n with
             | [], \_ \Rightarrow Some []
             \mid -, 0 \Rightarrow Some \ l
             | h :: t, S n' \Rightarrow Some (insert t n' x)
      end.
Lemma last\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      last (insert \ l \ n \ x) =
      if is Empty l
      then Some x
      else if leb (S n) (length l) then last l else Some x.
Lemma nth_{-}insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      n \leq length \ l \rightarrow nth \ n \ (insert \ l \ n \ x) = Some \ x.
Lemma nth_insert':
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      nth \ n \ (insert \ l \ n \ x) =
      if leb \ n \ (length \ l) then Some \ x else None.
Lemma insert\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      insert\ l\ n\ x=take\ n\ l\ ++\ x::\ drop\ n\ l.
Lemma insert\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : \mathit{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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      take (S n) (insert l n x) = snoc x (take n l).
Lemma take\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
      take \ m \ (insert \ l \ n \ x) =
      if m \le ? n then take m l else snoc x l.
Lemma drop\_S\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      drop (S n) (insert l n x) = drop n l.
Lemma insert\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
      insert (drop \ n \ l) \ m \ x =
```

```
drop\ (n-1)\ (insert\ l\ (n+m)\ x).
```

9.1.19 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:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
         isEmpty \ l' = isEmpty \ l.
Lemma length\_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow length \ l' = length \ l.
Lemma replace\_length\_lt:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      n < length l \rightarrow
         \exists l': list A, replace l n x = Some l'.
Lemma replace\_length\_ge:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      length \ l \leq n \rightarrow \texttt{replace} \ l \ n \ x = None.
Lemma replace\_snoc\_eq:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
      n = length \ l \rightarrow replace \ (snoc \ x \ l) \ n \ y = Some \ (snoc \ y \ l).
Lemma replace\_snoc\_neq:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
      n \neq length l \rightarrow
         replace (snoc \ x \ l) \ n \ y =
         match replace l n y with
                | None \Rightarrow None |
                | Some l' \Rightarrow Some (snoc x l') |
         end.
Lemma replace\_snoc:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
      replace (snoc \ x \ l) \ n \ y =
      if beq_nat \ n \ (length \ l)
      then Some (snoc y l)
      else
         \mathtt{match\ replace}\ l\ n\ y\ \mathtt{with}
                | None \Rightarrow None
```

```
| Some l' \Rightarrow Some (snoc x l') |
         end.
Lemma replace\_app:
  \forall (A : \mathsf{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 \Rightarrow None |
             Some l', \bot \Rightarrow Some (l' ++ l2)
            | \cdot |, Some l' \Rightarrow Some (l1 ++ l')
      end.
Lemma replace\_spec:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{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':
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      n < length l \rightarrow
         replace l \ n \ x = Some \ (take \ n \ l ++ x :: drop \ (S \ n) \ l).
Lemma replace_spec'':
  \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow l' = take \ n \ l ++ x :: drop (S \ n) \ l.
Lemma replace\_rev\_aux:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      n < length l \rightarrow
         replace l n x =
         match replace (rev \ l) \ (length \ l - S \ n) \ x with
                | None \Rightarrow None |
               | Some l' \Rightarrow Some (rev l') |
         end.
Definition omap \{A \ B : \text{Type}\}\ (f : A \to B)\ (oa : option \ A) : option \ B :=
match oa with
       None \Rightarrow None
      | Some \ a \Rightarrow Some \ (f \ a)
end.
Lemma replace\_rev:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      n < length l \rightarrow
         replace (rev \ l) \ n \ x = omap \ rev \ (replace \ l \ (length \ l - S \ n) \ x).
Lemma map\_replace:
```

```
\forall (A B : \mathsf{Type}) (f : A \to B) (l \ l' : list \ A) (n : nat) (x : A),
     replace l \ n \ x = Some \ l' \rightarrow
         Some\ (map\ f\ l') = \texttt{replace}\ (map\ f\ l)\ n\ (f\ x).
Lemma replace\_join:
  \forall (A : \mathsf{Type}) (ll : list (list A)) (n : nat) (x : A) (l : list A),
     replace (join ll) n x = Some l \rightarrow
        \exists n m : nat,
           match nth n ll with
                  | None \Rightarrow False
                  \mid Some \ l' \Rightarrow
                       match replace l' m x with
                              | None \Rightarrow False
                              \mid Some \ l^{"} \Rightarrow
                                    match replace ll \ n \ l'' with
                                           None \Rightarrow False
                                          | Some ll' \Rightarrow join ll' = l
                                    end
                        end
            end.
Lemma replace\_replicate:
  \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n \ m : \mathit{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:
  \forall (A : \mathsf{Type}) (f : A \to A) (l : \mathit{list} A) (n \ m : \mathit{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:
  \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
     replace l \ n \ x = Some \ l' \rightarrow
         head l' =
        match n with
               \mid 0 \Rightarrow Some \ x
                _{-} \Rightarrow head l
         end.
Lemma tail\_replace:
```

```
\forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
           tail l' =
          match n with
                 \mid 0 \Rightarrow tail \mid l
                 \mid S \mid n' \Rightarrow
                        match tail l with
                               | None \Rightarrow None
                                | Some \ t \Rightarrow replace \ t \ n' \ x
                        end
           end.
Lemma replace\_length\_aux:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
       replace l \ n \ x = Some \ l' \rightarrow length \ l = length \ l'.
Lemma nth\_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
       replace l \ n \ x = Some \ l' \rightarrow
          nth \ m \ l' = if \ n = ? \ m \ then \ Some \ x \ else \ nth \ m \ l.
Lemma replace_nth_eq:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
          l = l' \leftrightarrow nth \ n \ l = Some \ x.
Lemma last\_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
           last l' =
           if n = ? length l - 1
          then Some x
          else last l.
Lemma init\_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
           init l' =
          match \ init \ l \ with
                   None \Rightarrow None
                  \mid Some \; i \Rightarrow 	ext{if} \; length \; i <=? \; n \; 	ext{then} \; Some \; i \; 	ext{else} \; 	ext{replace} \; i \; n \; x
          end.
Lemma take\_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
       replace l \ n \ x = Some \ l' \rightarrow
           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:
  \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
     replace l \ n \ x = Some \ l' \rightarrow
         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:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
     n \leq length l \rightarrow
        replace (insert l n x) n y = Some (insert l n y).
Lemma replace\_plus:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n \ m : nat) (x : A),
     replace l(n + m) x =
     match replace (drop \ n \ l) \ m \ x with
           | None \Rightarrow None |
           | Some \ l' \Rightarrow Some \ (take \ n \ l ++ l')
      end.
```

9.1.20 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:
\forall\ (A:\ \mathsf{Type})\ (n:\ nat)\ (h:\ A)\ (t:\ list\ A),
remove'\ (S\ n)\ (h::\ t) = h::\ remove'\ n\ t.
Lemma remove\_isEmpty\_true:
\forall\ (A:\ \mathsf{Type})\ (l:\ list\ A)\ (n:\ nat),
isEmpty\ l = true \to remove\ n\ l = None.
Lemma isEmpty\_remove\_not\_None:
\forall\ (A:\ \mathsf{Type})\ (l:\ list\ A)\ (n:\ nat),
remove\ n\ l \neq None \to isEmpty\ l = false.
```

```
Lemma isEmpty\_remove:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      remove n \mid l = Some(x, l') \rightarrow
         isEmpty\ l'=isEmpty\ l\ ||\ ((length\ l<=?\ 1)\ \&\&\ isZero\ n).
Lemma length\_remove:
   \forall (A : \mathsf{Type}) (h : A) (l \ t : list \ A) (n : nat),
      remove n \mid Some(h, t) \rightarrow length(l) = S(length(t)).
Lemma remove\_length\_lt:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length l \rightarrow
         nth \ n \ l =
         match remove \ n \ l with
                 None \Rightarrow None
                | Some (h, \bot) \Rightarrow Some h
         end.
Lemma remove_length_lt':
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length \ l \rightarrow remove \ n \ l \neq None.
Lemma remove\_length\_ge:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      length l < n \rightarrow remove \ n \ l = None.
Lemma remove\_length\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      remove (length l) (snoc x l) = Some (x, l).
Lemma remove\_snoc\_lt:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A) (n : \mathit{nat}),
      n < length l \rightarrow
         remove \ n \ (snoc \ x \ l) =
         match remove \ n \ l with
                | None \Rightarrow None |
                | Some (h, t) \Rightarrow Some (h, snoc x t)
         end.
Lemma remove\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      remove \ n \ (l1 \ ++ \ l2) =
      match remove n l1 with
            | Some (h, t) \Rightarrow Some (h, t ++ l2)
            | None \Rightarrow
                  match remove (n - length l1) l2 with
                          Some (h, t) \Rightarrow Some (h, l1 ++ t)
                          None \Rightarrow None
```

```
end
```

end.

```
Lemma remove\_app\_lt:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      n < length \ l1 \rightarrow
         remove \ n \ (l1 \ ++ \ l2) =
         match remove n l1 with
               | None \Rightarrow None |
               | Some (h, t) \Rightarrow Some (h, t ++ l2)
         end.
Lemma remove\_app\_ge:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      length l1 \leq n \rightarrow
         remove \ n \ (l1 \ ++ \ l2) =
         match remove (n - length l1) l2 with
               | None \Rightarrow None |
               | Some (h, t) \Rightarrow Some (h, l1 ++ t)
         end.
Lemma remove'_-app:
  \forall (A : \mathsf{Type}) (n : nat) (l1 \ l2 : list \ A),
      n < length l1 \rightarrow
         remove' \ n \ (l1 \ ++ \ l2) = remove' \ n \ l1 \ ++ \ l2.
Lemma remove\_app':
  \forall (A : \mathsf{Type}) (n : nat) (l1 \ l2 : list \ A),
      length l1 \leq n \rightarrow
         remove' \ n \ (l1 \ ++ \ l2) = l1 \ ++ \ remove' \ (n - length \ l1) \ l2.
Lemma remove\_rev\_aux:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length l \rightarrow
         remove \ n \ l =
         match remove (length l - S n) (rev l) with
               | None \Rightarrow None
               | Some (h, t) \Rightarrow Some (h, rev t) |
         end.
Lemma remove\_rev:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length l \rightarrow
         remove \ n \ (rev \ l) =
         match remove (length l - S n) l with
               | None \Rightarrow None |
               | Some (h, t) \Rightarrow Some (h, rev t) |
```

```
end.
```

```
Lemma remove\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (n : nat),
      remove \ n \ (map \ f \ l) =
      match remove \ n \ l with
            | None \Rightarrow None |
            | Some (x, l') \Rightarrow Some (f x, map f l')
      end.
Lemma remove\_replicate:
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      m < n \rightarrow remove \ m \ (replicate \ n \ x) = Some \ (x, replicate \ (n - 1) \ x).
Lemma remove\_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x : A),
      m < n \rightarrow
         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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      nth \ n \ l = Some \ x \leftrightarrow
      remove n \mid l = Some (x, take \mid n \mid l + l + drop (S \mid n) \mid l).
Lemma remove\_insert:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      n < length l \rightarrow
         remove n (insert l n x) = Some (x, l).
Lemma remove'_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
         remove' n l' = remove' n l.
```

9.1.21 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 :

$$\forall (A B : \mathsf{Type}) \ (l : \mathit{list} \ B), \ \mathit{zip} \ (@\mathit{nil} \ A) \ l = [].$$

Lemma zip_nil_r :

```
\forall (A B : \mathsf{Type}) (l : list A), zip \ l (@nil B) = [].
Lemma isEmpty\_zip :
  \forall (A B : \mathsf{Type}) (la : \mathit{list} A) (lb : \mathit{list} B),
      isEmpty\ (zip\ la\ lb) = orb\ (isEmpty\ la)\ (isEmpty\ lb).
Lemma length\_zip:
  \forall (A B : \mathsf{Type}) (la : list A) (lb : list B),
      length (zip la lb) = min (length la) (length lb).
Lemma zip\_not\_rev:
   \exists (A B : \mathsf{Type}) (la : list A) (lb : list B),
      zip (rev la) (rev lb) \neq rev (zip la lb).
Lemma head\_zip:
  \forall (A B : \mathsf{Type}) (la : list A) (lb : list B) (a : A) (b : B),
      head\ la = Some\ a \rightarrow head\ lb = Some\ b \rightarrow
         head\ (zip\ la\ lb) = Some\ (a,\ b).
Lemma tail\_zip:
  \forall (A B : \mathsf{Type}) (la \ ta : \mathit{list} \ A) (\mathit{lb} \ \mathit{tb} : \mathit{list} \ B),
      tail \ la = Some \ ta \rightarrow tail \ lb = Some \ tb \rightarrow
         tail\ (zip\ la\ lb) = Some\ (zip\ ta\ tb).
Lemma zip\_not\_app:
   \exists (A B : \mathsf{Type}) (la \ la' : \mathit{list} \ A) (\mathit{lb} \ \mathit{lb'} : \mathit{list} \ B),
      zip (la ++ la') (lb ++ lb') \neq zip la lb ++ zip la' lb'.
Lemma zip_{-}map:
  \forall (A \ B \ A' \ B' : \mathsf{Type}) \ (f : A \rightarrow A') \ (g : B \rightarrow B')
   (la: list A) (lb: list B),
      zip (map f la) (map g lb) =
      map \text{ (fun } x \Rightarrow (f \text{ (fst } x), g \text{ (snd } x))) \text{ (zip la lb)}.
Lemma zip_-replicate:
  \forall (A B : \mathsf{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 : \mathsf{Type}) (fa : A \to A) (fb : B \to B) (na \ nb : nat) (a : A) (b : B),
         zip (iterate fa na a) (iterate fb nb b) =
         iterate (fun'(a, b) \Rightarrow (fa \ a, fb \ b)) (min \ na \ nb) (a, b).
Lemma nth\_zip:
  \forall (A B : \mathsf{Type}) (la : list A) (lb : list B) (n : nat),
      nth \ n \ (zip \ la \ lb) =
      if n \le n (length la) (length lb)
      then
```

```
match nth n la, nth n lb with
               | Some a, Some b \Rightarrow Some (a, b)
               | \_, \_ \Rightarrow None
         end
      else None.
Lemma nth_zip':
   \forall (A B : \mathsf{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 \Rightarrow Some \ (a, b)
            | \_, \_ \Rightarrow None
      end.
Lemma zip_{-}take:
   \forall (A B : \mathsf{Type}) (la : list A) (lb : list B) (n : nat),
      zip (take \ n \ la) (take \ n \ lb) = take \ n (zip \ la \ lb).
Lemma zip_{-}drop:
   \forall (A B : \mathsf{Type}) (la: \mathit{list} A) (lb: \mathit{list} B) (n: nat),
      zip (drop \ n \ la) (drop \ n \ lb) = drop \ n \ (zip \ la \ lb).
Lemma splitAt\_zip:
   \forall (A B : \mathsf{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) \Rightarrow
                 Some (zip \ la1 \ lb1, (a, b), zip \ la2 \ lb2)
            | \_, \_ \Rightarrow None
      end.
Lemma insert\_zip:
  \forall (A B : \mathsf{Type}) (la : list A) (lb : list B) (a : A) (b : B) (n : nat),
      insert (zip la lb) n (a, b) =
      if n \le 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 : \mathsf{Type}) (la \ la' : \mathit{list} \ A) (lb \ lb' : \mathit{list} \ B)
      (n: nat) (a: A) (b: B),
        replace la \ n \ a = Some \ la' \rightarrow
        replace lb \ n \ b = Some \ lb' \rightarrow
            replace (zip \ la \ lb) \ n \ (a, b) = Some \ (zip \ la' \ lb').
Lemma replace_zip':
   \forall
```

```
(A B : \mathsf{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' \Rightarrow Some (zip la' lb')
              | \_, \_ \Rightarrow None
        end.
Lemma remove\_zip:
  \forall (A B : \mathsf{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') \Rightarrow Some ((a, b), zip la' lb')
           | \_, \_ \Rightarrow None
     end.
9.1.22
             unzip
```

Zdefiniuj funkcję unzip, która jest w pewnym sensie "odwrotna" do zip.

$$unzip [(1, 2); (3, 4); (5, 6)] = ([1; 3; 5], [2; 4; 6])$$

Lemma zip_-unzip :

```
\forall (A B : \mathsf{Type}) (l : list (A \times B)),
   zip (fst (unzip l)) (snd (unzip l)) = l.
```

Lemma $unzip_-zip$:

```
\exists (A B : \mathsf{Type}) (la : list A) (lb : list B),
   unzip (zip la lb) \neq (la, lb).
```

Lemma $isEmpty_unzip$:

```
\forall (A B : \mathsf{Type}) (l : \mathit{list} (A \times B)) (la : \mathit{list} A) (lb : \mathit{list} B),
    unzip \ l = (la, lb) \rightarrow isEmpty \ l = orb \ (isEmpty \ la) \ (isEmpty \ lb).
```

Lemma $unzip_snoc$:

```
\forall (A B : \mathsf{Type}) (x : A \times B) (l : \mathit{list} (A \times B)),
   unzip (snoc x l) =
      let (la, lb) := unzip \ l \ in \ (snoc \ (fst \ x) \ la, snoc \ (snd \ x) \ lb).
```

9.1.23zip With

Zdefiniuj funkcję zip With, która zachowuje się jak połączenie zip i map. Nie używaj zip ani map - użyj rekursji.

```
Przykład:
```

```
zip With plus [1; 2; 3] [4; 5; 6] = [5; 7; 9]
```

Lemma $zip With_spec$:

```
\forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C)
   (la: list A) (lb: list B),
      zip With f la lb =
      map (fun '(a, b) \Rightarrow f \ a \ b) (zip \ la \ lb).
Lemma zip With\_pair:
   \forall (A B : \mathsf{Type}) (la : list A) (lb : list B),
      zip With pair la lb = zip la lb.
Lemma isEmpty\_zipWith:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (la : list \ A) \ (lb : list \ B),
      isEmpty\ (zipWith\ f\ la\ lb)=orb\ (isEmpty\ la)\ (isEmpty\ lb).
Lemma zip With\_snoc:
  \forall
      (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C)
     (a : A) (b : B) (la : list A) (lb : list B),
         length la = length lb \rightarrow
            zip With f (snoc a la) (snoc b lb) =
            snoc (f \ a \ b) (zipWith f \ la \ lb).
Lemma zip With\_iterate:
  A
      (A \ B \ C : \mathsf{Type}) \ (fa : A \to A) \ (fb : B \to B) \ (g : A \to B \to C)
     (na \ nb : nat) (a : A) (b : B),
         zipWith \ g \ (iterate \ fa \ na \ a) \ (iterate \ fb \ nb \ b) =
         map (fun '(a, b) \Rightarrow g \ a \ b)
            (iterate\ (fun\ '(a, b) \Rightarrow (fa\ a, fb\ b))\ (min\ na\ nb)\ (a, b)).
Lemma take\_zipWith:
     (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (la : \mathit{list} \ A) \ (lb : \mathit{list} \ B) \ (n : \mathit{nat}),
         take \ n \ (zip With \ f \ la \ lb) = zip With \ f \ (take \ n \ la) \ (take \ n \ lb).
Lemma drop\_zipWith:
     (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (la : \mathit{list} \ A) \ (lb : \mathit{list} \ B) \ (n : \mathit{nat}),
         drop \ n \ (zip With \ f \ la \ lb) = zip With \ f \ (drop \ n \ la) \ (drop \ n \ lb).
Lemma splitAt\_zipWith:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to 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) \Rightarrow
                     Some (zipWith\ f\ la1\ lb1, f\ a\ b, zipWith\ f\ la2\ lb2)
               | \_, \_ \Rightarrow None
         end.
```

```
Lemma replace\_zipWith:
  \forall
     (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (la \ la' : list \ A) \ (lb \ lb' : list \ B)
     (n: nat) (a: A) (b: B),
        replace la \ n \ a = Some \ la' \rightarrow
        replace lb \ n \ b = Some \ lb' \rightarrow
           replace (zip With f la lb) n (f a b) = Some (zip With f la' lb').
Lemma remove\_zipWith:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C)
     (la: list A) (lb: list B) (n: nat),
        remove \ n \ (zip With \ f \ la \ lb) =
        match remove n la, remove n lb with
              | Some (a, la'), Some (b, lb') \Rightarrow
                   Some (f a b, zipWith f la' lb')
              | \_, \_ \Rightarrow None
        end.
```

$9.1.24 \quad unzip With$

Zdefiniuj funkcję unzip With, która ma się tak do zip With, jak unzip do zip. Oczywiście użyj rekursji i nie używaj żadnych funkcji pomocniczych.

```
 \begin{array}{l} \text{Lemma } is Empty\_unzip With: \\ \forall \ (A \ B \ C : \texttt{Type}) \ (f: A \to B \times C) \ (l: list \ A) \\ (lb: list \ B) \ (lc: list \ C), \\ unzip With \ f \ l = (lb, lc) \to \\ is Empty \ l = orb \ (is Empty \ lb) \ (is Empty \ lc). \\ \\ \text{Lemma } unzip With\_spec: \\ \forall \ (A \ B \ C : \texttt{Type}) \ (f: A \to B \times C) \ (l: list \ A), \\ unzip With \ f \ l = unzip \ (map \ f \ l). \\ \\ \text{Lemma } unzip With\_id: \\ \forall \ (A \ B : \texttt{Type}) \ (l: list \ (A \times B)), \\ unzip With \ id \ l = unzip \ l. \end{array}
```

9.2 Funkcje z predykatem boolowskim

9.2.1 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:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      isEmpty\ l=true \rightarrow any\ p\ l=false.
Lemma isEmpty\_any\_true:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      any p \mid l = true \rightarrow isEmpty \mid l = false.
Lemma any\_length:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      any p \mid l = true \rightarrow 1 \leq length \mid l.
Lemma any\_snoc:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      any \ p \ (snoc \ x \ l) = orb \ (any \ p \ l) \ (p \ x).
Lemma any\_app:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      any \ p \ (l1 ++ l2) = orb \ (any \ p \ l1) \ (any \ p \ l2).
Lemma any\_rev:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      any \ p \ (rev \ l) = any \ p \ l.
Lemma any_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (p : B \to bool) (l : list A),
      any \ p \ (map \ f \ l) = any \ (fun \ x : A \Rightarrow p \ (f \ x)) \ l.
Lemma any\_join:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list (list A)),
      any \ p \ (join \ l) = any \ (any \ p) \ l.
Lemma any\_replicate:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
      any p (replicate n(x) = andb (leb 1 n) (p(x)).
Lemma any\_iterate:
   \forall (A : \mathsf{Type}) (p : A \to bool) (f : A \to A) (n : nat) (x : A),
      (\forall x: A, p (f x) = p x) \rightarrow
         any p (iterate f n x) =
         match n with
               \mid 0 \Rightarrow false
               | \  \  \Rightarrow p \ x
         end.
Lemma any_nth:
  \forall (A : \mathsf{Type}) (p : A \to 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 : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      any \ p \ (take \ n \ l) = true \rightarrow any \ p \ l = true.
Lemma any\_take\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      any \ p \ l = false \rightarrow any \ p \ (take \ n \ l) = false.
Lemma any\_drop:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      any \ p \ (drop \ n \ l) = true \rightarrow any \ p \ l = true.
Lemma any\_drop\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      any \ p \ l = false \rightarrow any \ p \ (drop \ n \ l) = false.
Lemma any\_cycle:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (l : list A),
      any \ p \ (cycle \ n \ l) = any \ p \ l.
Lemma any\_insert:
   \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
         any p \mid l' = any \mid p \mid (take \mid n \mid l) \mid \mid p \mid x \mid \mid any \mid p \mid (drop \mid (S \mid n) \mid l).
Lemma any\_replace':
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
         any p \mid l = true \rightarrow p \mid x = true \rightarrow any \mid p \mid l' = true.
Lemma any\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      any (fun \implies true) l = negb (isEmpty l).
Lemma any\_false:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      any (fun \_ \Rightarrow false) l = false.
Lemma any\_orb:
   \forall (A : \mathsf{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 : \mathsf{Type}) (p \ q : A \to bool) (l : list A),
```

any (fun $x: A \Rightarrow andb (p x) (q x)$) $l = true \rightarrow$

any p $l = true \land any q l = true$.

9.2.2 all

Napisz funkcję all, która sprawdza, czy wszystkie wartości na liście l spełniają predykat boolowski p.

```
Przykład: all even [2; 4; 6] = true
```

Lemma $all_isEmpty_true$:

$$\forall (A: \mathsf{Type}) \ (p: A \to bool) \ (l: list \ A),$$
 $isEmpty \ l = true \to all \ p \ l = true.$

Lemma $isEmpty_all_false$:

$$\forall \ (A: {\tt Type}) \ (p: A \to bool) \ (l: list \ A), \ all \ p \ l = false \to is Empty \ l = false.$$

Lemma all_length :

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (l: list \ A),$$
 all $p \ l = false \rightarrow 1 \leq length \ l.$

Lemma all_snoc :

$$\forall (A: \mathtt{Type}) \ (p:A \rightarrow bool) \ (x:A) \ (l: list\ A),$$
 all $p \ (snoc\ x\ l) = andb \ (all\ p\ l) \ (p\ x).$

Lemma all_app :

$$\forall (A: \mathtt{Type}) (p: A \rightarrow bool) (l1 \ l2: list \ A),$$
 all $p (l1 ++ l2) = andb (all \ p \ l1) (all \ p \ l2).$

Lemma all_rev :

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (l: list \ A),$$
 all $p \ (rev \ l) = all \ p \ l.$

Lemma $all_{-}map$:

$$\forall \ (A \ B : \mathtt{Type}) \ (f : A \rightarrow B) \ (p : B \rightarrow bool) \ (l : \mathit{list} \ A), \\ \mathit{all} \ p \ (\mathit{map} \ f \ l) = \mathit{all} \ (\mathtt{fun} \ x : A \Rightarrow p \ (f \ x)) \ \mathit{l}.$$

Lemma $all_{-}join$:

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (l: list \ (list \ A)),$$
 all $p \ (join \ l) = all \ (all \ p) \ l.$

Lemma $all_replicate$:

$$\forall (A: \mathtt{Type}) (p: A \rightarrow bool) (n: nat) (x: A),$$
 all p (replicate $n(x) = orb (n <=? 0) (p(x)).$

Lemma $all_iterate$:

$$\forall (A: \mathsf{Type}) \ (p: A \to bool) \ (f: A \to A) \ (n: nat) \ (x: A),$$
 $(\forall x: A, p \ (f \ x) = p \ x) \to all \ p \ (iterate \ f \ n \ x) = orb \ (isZero \ n) \ (p \ x).$

Lemma all_-nth :

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (l: list \ A),$$
 all $p \ l = true \leftrightarrow$

```
\forall n : nat, n < length l \rightarrow \exists x : A,
          nth \ n \ l = Some \ x \land p \ x = true.
Lemma all\_take:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
       all\ p\ (take\ n\ l) = false \rightarrow all\ p\ l = false.
Lemma all\_take\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
       all p \mid l = true \rightarrow all \mid p \mid (take \mid n \mid l) = true.
Lemma all_{-}drop:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
       all\ p\ (drop\ n\ l) = false \rightarrow all\ p\ l = false.
Lemma all\_drop\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
       all \ p \ l = true \rightarrow all \ p \ (drop \ n \ l) = true.
Lemma all\_insert:
   \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
          all p \mid l' = all \mid p \mid (take \mid n \mid l) \&\& \mid p \mid x \&\& \mid all \mid p \mid (drop \mid (S \mid n) \mid l).
Lemma all\_replace':
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
          all p \mid l = true \rightarrow p \mid x = true \rightarrow all \mid p \mid l' = true.
Lemma all\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
       all (fun \_ \Rightarrow true) l = true.
Lemma all\_false:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
       all (fun \_ \Rightarrow false) l = isEmpty l.
Lemma all\_orb:
   \forall (A : \mathsf{Type}) (p \ q : A \to bool) (l : list A),
```

 $orb (all \ p \ l) (all \ q \ l) = true \rightarrow$

andb (all p l) (all q l).

Lemma all_andb :

all (fun $x : A \Rightarrow orb (p x) (q x)$) l = true.

 $\forall (A : \mathsf{Type}) (p \ q : A \rightarrow bool) (l : list \ A),$ all (fun $x : A \Rightarrow andb (p \ x) (q \ x)) l =$

```
 \begin{array}{l} \text{Lemma } any\_all: \\ \forall \; (A: \texttt{Type}) \; (p:A \rightarrow bool) \; (l:list\; A), \\ any \; p \; l = negb \; (all \; (\texttt{fun} \; x:A \Rightarrow negb \; (p\; x)) \; l). \\ \text{Lemma } all\_any: \\ \forall \; (A: \texttt{Type}) \; (p:A \rightarrow bool) \; (l:list\; A), \\ all \; p \; l = negb \; (any \; (\texttt{fun} \; x:A \Rightarrow negb \; (p\; x)) \; l). \\ \text{Lemma } all\_cycle: \\ \forall \; (A: \texttt{Type}) \; (p:A \rightarrow bool) \; (n:nat) \; (l:list\; A), \\ all \; p \; (cycle\; n\; l) = all\; p\; l. \\ \text{Lemma } isEmpty\_join: \\ \forall \; (A: \texttt{Type}) \; (l:list\; (list\; A)), \\ isEmpty \; (join\; l) = all\; isEmpty\; l. \\ \end{array}
```

9.2.3 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:
```

```
\forall (A : \mathsf{Type}) (l : \mathit{list} \ A), \\ \mathit{find} \ (\mathsf{fun} \ \_ \Rightarrow \mathit{false}) \ l = \mathit{None}.
```

Lemma $find_{-}true$:

$$\forall (A: \mathtt{Type}) \ (l: \mathit{list}\ A), \\ \mathit{find}\ (\mathtt{fun}\ _ \Rightarrow \mathit{true}) \ l = \mathit{head}\ l.$$

Lemma $find_isEmpty_true$:

$$\forall \ (A: {\tt Type}) \ (p: A \rightarrow bool) \ (l: list \ A), \ is Empty \ l = true \rightarrow find \ p \ l = None.$$

Lemma $isEmpty_find_not_None$:

$$\forall (A: \mathsf{Type}) \ (p: A \to bool) \ (l: list \ A),$$
 find $p \ l \neq None \to isEmpty \ l = false.$

Lemma $find_length$:

```
\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l: list \ A), find p \ l = Some \ x \rightarrow 1 \leq length \ l.
```

Lemma $find_snoc$:

```
\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l: list \ A), find p \ (snoc \ x \ l) = match find p \ l with
```

```
| None \Rightarrow if p x then Some x else None |
            \mid Some \ y \Rightarrow Some \ y
      end.
Lemma findLast\_snoc:
   \forall (A : \mathsf{Type}) (p : A \to 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 : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l : list A),
     find \ p \ (rev \ l) = findLast \ p \ l.
Lemma find\_findLast:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     find p l = findLast p (rev l).
Lemma find_{-}map:
   \forall (A B : \mathsf{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
            \mid Some \ a \Rightarrow Some \ (f \ a)
      end.
Lemma find_{-}join:
   \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
```

```
find p (replicate n x) =
     match n, p x with
            \mid 0, \bot \Rightarrow None
             \_, false \Rightarrow None
            \mid \_, true \Rightarrow Some x
      end.
Lemma find\_iterate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (f : A \to A) (n : nat) (x : A),
      (\forall x : A, p (f x) = p x) \rightarrow
         find p (iterate f n x) =
         if isZero n then None else if p x then Some x else None.
Lemma findLast\_iterate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (f : A \to A) (n : nat) (x : A),
      (\forall x: A, p (f x) = p x) \rightarrow
         findLast \ p \ (iterate \ f \ n \ x) =
         match n with
               \mid 0 \Rightarrow None
               \mid S \mid n' \Rightarrow \text{if } p \mid x \text{ then } Some \ (iter f \mid n' \mid x) \text{ else } None
         end.
Lemma find_nth:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      (\exists (n: nat) (x: A), nth \ n \ l = Some \ x \land p \ x = true) \leftrightarrow
     find p \mid l \neq None.
Lemma find_{-}tail:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l \ t : list \ A),
      tail l = Some \ t \rightarrow find \ p \ t \neq None \rightarrow find \ p \ l \neq None.
Lemma find_init:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l \ t : list \ A),
      init l = Some \ t \rightarrow find \ p \ t \neq None \rightarrow find \ p \ l \neq None.
Lemma find\_take\_Some:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat) (x : A),
     find p (take n l) = Some x \rightarrow find p l = Some x.
Lemma find\_take\_None:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     find p \mid l = None \rightarrow find \mid p \mid (take \mid n \mid l) = None.
Lemma find\_drop\_not\_None:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      find p (drop \ n \ l) \neq None \rightarrow find \ p \ l \neq None.
Lemma find\_drop\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
```

```
find p \mid l = None \rightarrow find \mid p \mid (drop \mid n \mid l) = None.
Lemma findLast\_take:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     findLast\ p\ (take\ n\ l) \neq None \rightarrow findLast\ p\ l \neq None.
Lemma findLast\_drop:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat) (x : A),
     findLast\ p\ (drop\ n\ l) = Some\ x \rightarrow findLast\ p\ l = Some\ x.
Lemma find\_replace:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
         find p l' =
         match find p (take n l), p x with
                | Some \ y, \bot \Rightarrow Some \ y
                1 - true \Rightarrow Some x
               | \_, \_ \Rightarrow find \ p \ (drop \ (S \ n) \ l)
         end.
Lemma replace\_findLast:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
     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.
```

9.2.4 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]) \mathsf{Lemma}\ removeFirst\_isEmpty\_true: \forall\ (A:\ \mathsf{Type})\ (p:A\to bool)\ (l:\ list\ A), isEmpty\ l=true\to removeFirst\ p\ l=None.
```

Lemma $isEmpty_removeFirst_not_None:$ $\forall (A : Type) (p : A \rightarrow bool) (l : list A),$

 $removeFirst \ p \ l \neq None \rightarrow isEmpty \ l = false.$

Lemma $removeFirst_satisfies$:

```
\forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (x : A),
     removeFirst\ p\ l = Some\ (x, l') \rightarrow p\ x = true.
Lemma removeFirst\_length:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     length l = 0 \rightarrow removeFirst \ p \ l = None.
Lemma removeFirst\_snoc:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
     removeFirst \ p \ (snoc \ x \ l) =
     match removeFirst p l with
            \mid None \Rightarrow \texttt{if} \ p \ x \ \texttt{then} \ Some \ (x, \ l) \ \texttt{else} \ None
            | Some (h, t) \Rightarrow Some (h, snoc x t) |
     end.
Lemma removeLast\_snoc:
  \forall (A : \mathsf{Type}) (p : A \to 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 \Rightarrow None |
               | Some (h, t) \Rightarrow Some (h, snoc x t)
        end.
Lemma removeFirst\_app:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     removeFirst \ p \ (l1 \ ++ \ l2) =
     match removeFirst p l1, removeFirst p l2 with
            | Some (h, t), \_ \Rightarrow Some (h, t ++ l2)
            | \cdot | Some (h, t) \Rightarrow Some (h, l1 ++ t)
           | \_, \_ \Rightarrow None
     end.
Lemma removeLast\_app:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     removeLast p (l1 ++ l2) =
     match removeLast p l2, removeLast p l1 with
            Some (y, l'), \bot \Rightarrow Some (y, l1 ++ l')
            | \cdot | Some (y, l') \Rightarrow Some (y, l' ++ l2)
            | \_, \_ \Rightarrow None
     end.
Lemma removeFirst\_rev:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     removeFirst \ p \ (rev \ l) =
```

```
match removeLast p l with
           | Some (x, l) \Rightarrow Some (x, rev l) |
           | None \Rightarrow None |
      end.
Lemma removeLast\_rev:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     removeLast \ p \ (rev \ l) =
     match removeFirst p l with
           | None \Rightarrow None |
           | Some (x, l) \Rightarrow Some (x, rev l)
      end.
Lemma removeFirst\_map:
  \forall (A B : \mathsf{Type}) (p : B \to bool) (f : A \to B) (l : list A),
     removeFirst \ p \ (map \ f \ l) =
     match removeFirst (fun x \Rightarrow p(f x)) l with
           | Some (x, l) \Rightarrow Some (f x, map f l) |
           | None \Rightarrow None
      end.
Lemma removeFirst\_join:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list (list A)),
      removeFirst \ p \ (join \ l) =
     (fix f (l : list (list A)) : option (A \times list A) :=
     {\tt match}\ l\ {\tt with}
           | | | \Rightarrow None
           | hl :: tl \Rightarrow
                 match removeFirst p hl with
                        Some (x, l') \Rightarrow Some (x, join (l' :: tl))
                        None \Rightarrow
                            match f tl with
                                  | Some (x, l) \Rightarrow Some (x, hl ++ l)
                                  | None \Rightarrow None
                             end
                 end
      end) l.
Lemma removeFirst\_replicate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
      removeFirst \ p \ (replicate \ n \ x) =
     if p x
     then
           match n with
                 \mid 0 \Rightarrow None
```

```
\mid S \mid n' \Rightarrow Some (x, replicate \mid n' \mid x)
            end
      else None.
Lemma removeFirst\_nth\_None:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst \ p \ l = None \leftrightarrow
         \forall (n : nat) (x : A), nth \ n \ l = Some \ x \rightarrow p \ x = false.
Lemma removeFirst\_nth\_Some:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ l' : list \ A),
      removeFirst \ p \ l = Some \ (x, l') \rightarrow
      \exists n : nat, nth \ n \ l = Some \ x \land p \ x = true.
Lemma removeFirst\_nth\_Some':
   \exists (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x y : A) (l l' : list A),
      removeFirst \ p \ l = Some \ (x, l') \land
      nth \ n \ l = Some \ y \wedge p \ y = true.
Lemma head\_removeFirst:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ l' : list \ A),
      removeFirst \ p \ l = Some \ (x, l') \rightarrow
      head l' =
      {\tt match}\ l\ {\tt with}
            | [] \Rightarrow None
            \mid h :: t \Rightarrow \text{if } p \text{ } h \text{ then } head \text{ } t \text{ else } Some \text{ } h
    end.
Lemma removeFirst\_take\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (l : list A),
      removeFirst\ p\ l = None \rightarrow removeFirst\ p\ (take\ n\ l) = None.
Lemma removeFirst\_take:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A) (l \ l' : list \ A),
      removeFirst \ p \ (take \ n \ l) = Some \ (x, l') \rightarrow
         removeFirst \ p \ l = Some \ (x, \ l' ++ \ drop \ n \ l).
Lemma removeLast\_drop:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A) (l \ l' : list \ A),
      removeLast \ p \ (drop \ n \ l) = Some \ (x, l') \rightarrow
         removeLast \ p \ l = Some \ (x, take \ n \ l ++ l').
Lemma removeFirst\_replace:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
         removeFirst p l' =
         match removeFirst\ p\ (take\ n\ l),\ p\ x,\ removeFirst\ p\ (drop\ (S\ n)\ l) with
               |Some(y, l''), \bot, \bot \Rightarrow Some(y, l'' ++ x :: drop(S n) l)|
```

```
| \_, true, \_ \Rightarrow Some (x, take \ n \ l ++ drop (S \ n) \ l)
               \mid _, _, Some (y, l'') \Rightarrow Some (y, take \ n \ l ++ x :: l'')
               | \_, \_, \_ \Rightarrow None
         end.
Lemma removeLast\_replace:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
     replace l \ n \ x = Some \ l' \rightarrow
      removeLast p l' =
     match removeLast \ p \ (drop \ (S \ n) \ l), \ p \ x, \ removeLast \ p \ (take \ n \ l) with
             Some (y, l''), \_, \_ \Rightarrow Some (y, take \ n \ l ++ x :: l'')
            \mid _, true, _ \Rightarrow Some (x, take n l ++ drop <math>(S n) l)
            | \_, \_, Some (y, l'') \Rightarrow Some (y, l'' ++ x :: drop (S n) l)
            | -, -, - \Rightarrow None
      end.
Lemma removeFirst\_any\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst \ p \ l = None \leftrightarrow any \ p \ l = false.
Lemma removeFirst\_not\_None\_any:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst \ p \ l \neq None \leftrightarrow any \ p \ l = true.
Lemma removeFirst\_None\_iff\_all:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst \ p \ l = None \leftrightarrow
      all (fun x: A \Rightarrow negb(p x)) l = true.
            findIndex
```

9.2.5

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:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      findIndex (fun \_ \Rightarrow false) l = None.
Lemma findIndex\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      findIndex (fun \_ \Rightarrow true) l =
      match l with
            | | | \Rightarrow None
            | \bot \Rightarrow Some 0
      end.
```

```
Lemma findIndex\_orb:
  \forall (A : \mathsf{Type}) (p \ q : A \to bool) (l : list A),
     findIndex (fun x: A \Rightarrow orb (p x) (q x)) l =
     match findIndex \ p \ l, findIndex \ q \ l with
            | Some n, Some m \Rightarrow Some (min \ n \ m)
            | Some \ n, None \Rightarrow Some \ n
            | None, Some m \Rightarrow Some m
            | \_, \_ \Rightarrow None
     end.
Lemma findIndex\_isEmpty\_true:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     isEmpty\ l=true \rightarrow findIndex\ p\ l=None.
Lemma isEmpty\_findIndex\_not\_None:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     findIndex \ p \ l \neq None \rightarrow isEmpty \ l = false.
Lemma findIndex\_length:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     findIndex \ p \ l = Some \ n \rightarrow n < length \ l.
Lemma findIndex\_snoc:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
     findIndex \ p \ (snoc \ x \ l) =
     match findIndex p l with
           | None \Rightarrow if p x then Some (length l) else None
           \mid Some \ n \Rightarrow Some \ n
     end.
Lemma findIndex\_app\_l:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A) (n : nat),
     findIndex \ p \ l1 = Some \ n \rightarrow findIndex \ p \ (l1 \ ++ \ l2) = Some \ n.
Lemma findIndex\_app\_r:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A) (n : nat),
     findIndex \ p \ l1 = None \rightarrow findIndex \ p \ l2 = Some \ n \rightarrow
        findIndex \ p \ (l1 ++ l2) = Some \ (length \ l1 + n).
Lemma findIndex\_app\_None:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     findIndex \ p \ l1 = None \rightarrow findIndex \ p \ l2 = None \rightarrow
        findIndex \ p \ (l1 ++ l2) = None.
Lemma findIndex\_app:
  \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
     findIndex p (l1 ++ l2) =
     match findIndex p l1, findIndex p l2 with
```

```
| Some \ n, \_ \Rightarrow Some \ n
           | \_, Some \ n \Rightarrow Some \ (length \ l1 + n)
           | \_, \_ \Rightarrow None
      end.
Lemma findIndex\_map:
  \forall (A B : \mathsf{Type}) (p : B \to bool) (f : A \to B) (l : list A) (n : nat),
     findIndex (fun x: A \Rightarrow p(f x)) l = Some n \rightarrow
        findIndex \ p \ (map \ f \ l) = Some \ n.
Lemma findIndex\_map\_conv:
  \forall (A B : \mathsf{Type}) (p : B \to bool) (f : A \to B) (l : list A) (n : nat),
     (\forall x \ y : A, f \ x = f \ y \rightarrow x = y) \rightarrow
     findIndex \ p \ (map \ f \ l) = Some \ n \rightarrow
        findIndex (fun x: A \Rightarrow p (f x)) l = Some \ n.
Lemma findIndex\_join:
  \forall (A : \mathsf{Type}) (p : A \to bool) (ll : list (list A)),
     findIndex p (join ll) =
     {\tt match}\ \mathit{ll}\ {\tt 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 : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
     findIndex p (replicate n x) =
     match n with
           \mid 0 \Rightarrow None
           | \_ \Rightarrow  if p x then Some \ 0 else None
      end.
Lemma findIndex_nth:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     findIndex \ p \ l = Some \ n \rightarrow
        \exists x : A, nth \ n \ l = Some \ x \land p \ x = true.
Lemma findIndex\_nth\_conv:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat) (x : A),
      nth \ n \ l = Some \ x \rightarrow p \ x = true \rightarrow
        \exists m : nat, findIndex \ p \ l = Some \ m \land m \leq n.
Lemma findIndex\_nth':
```

```
\forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     findIndex \ p \ l = Some \ n \rightarrow find \ p \ l = nth \ n \ l.
Lemma findIndex\_head:
  \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l : list A),
     findIndex \ p \ l = Some \ (length \ l - 1) \leftrightarrow
     \exists x: A,
         last \ l = Some \ x \land
         p \ x = true \land
        \forall (n : nat) (y : A),
            n < length \ l - 1 \rightarrow nth \ n \ l = Some \ y \rightarrow p \ y = false.
Lemma findIndex\_spec:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     findIndex \ p \ l = Some \ n \rightarrow
        \forall m : nat, m < n \rightarrow
            \exists x : A, nth \ m \ l = Some \ x \land p \ x = false.
Lemma findIndex\_take:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n m : nat),
     findIndex \ p \ (take \ n \ l) = Some \ m \rightarrow
        findIndex \ p \ l = Some \ m \land m \le n.
Lemma findIndex\_drop:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n m : nat),
     findIndex \ p \ l = Some \ m \rightarrow n \leq m \rightarrow
        findIndex \ p \ (drop \ n \ l) = Some \ (m - n).
Lemma findIndex\_zip:
  \forall (A B : \mathsf{Type}) (pa : A \to bool) (pb : B \to bool)
   (la: list A) (lb: list B) (n: nat),
     findIndex \ pa \ la = Some \ n \rightarrow findIndex \ pb \ lb = Some \ n \rightarrow
        findIndex (fun '(a, b) \Rightarrow andb (pa a) (pb b)) (zip la lb) = Some n.
Lemma findIndex\_zip\_conv:
  \forall (A \ B : \mathsf{Type}) \ (pa : A \to bool) \ (pb : B \to bool)
   (la: list A) (lb: list B) (n: nat),
     findIndex (fun '(a, b) \Rightarrow andb (pa a) (pb b)) (zip la lb) = Some n \rightarrow
     \exists na \ nb : nat,
        findIndex pa la = Some na \land
        findIndex \ pb \ lb = Some \ nb \wedge
         na < n \wedge
         nb \leq n.
```

9.2.6 count

Napisz funkcję count, która liczy, ile jest na liście l elementów spełniających predykat bolowski p.

```
Przykład:
    count even [1; 2; 3; 4] = 2
Lemma count\_isEmpty:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      isEmpty\ l=true \rightarrow count\ p\ l=0.
Lemma isEmpty\_count\_not\_\theta:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count p \mid l \neq 0 \rightarrow isEmpty \mid l = false.
Lemma count\_length:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count p \mid l \leq length \mid l.
Lemma count\_snoc:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      count \ p \ (snoc \ x \ l) = count \ p \ l + if \ p \ x \ then \ 1 \ else \ 0.
Lemma count\_app:
   \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
      count \ p \ (l1 \ ++ \ l2) = count \ p \ l1 \ + \ count \ p \ l2.
Lemma count\_rev:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count \ p \ (rev \ l) = count \ p \ l.
Lemma count\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (p : B \to bool) (l : list A),
      count \ p \ (map \ f \ l) = count \ (fun \ x : A \Rightarrow p \ (f \ x)) \ l.
(* Lemma count_join *)
Lemma count\_replicate:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
      count \ p \ (replicate \ n \ x) =
      if p x then n else 0.
Lemma count\_insert:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat) (x : A),
      count \ p \ (insert \ l \ n \ x) =
      (if p x then 1 else 0) + count p l.
Lemma count\_replace:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
```

replace $l \ n \ x = Some \ l' \rightarrow$

```
count \ p \ l' = count \ p \ (take \ n \ l) + count \ p \ [x] + count \ p \ (drop \ (S \ n) \ l).
Lemma count\_remove:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
      remove n \ l = Some \ (x, \ l') \rightarrow
         S(count \ p \ l') = if \ p \ x \ then \ count \ p \ l \ else \ S(count \ p \ l).
Lemma count\_take:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      count \ p \ (take \ n \ l) \leq n.
Lemma count\_take':
   \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      count \ p \ (drop \ n \ l) < length \ l - n.
Lemma count\_cycle:
   \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (l : list A),
      count \ p \ (cycle \ n \ l) = count \ p \ l.
Lemma count\_splitAt:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l1 \ l2 : list \ A) (n : nat) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         count \ p \ l = (if \ p \ x \ then \ 1 \ else \ 0) + count \ p \ l1 + count \ p \ l2.
Lemma count\_false:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      count (fun \_ \Rightarrow false) l = 0.
Lemma count\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      count (fun \_ \Rightarrow true) l = length l.
Lemma count\_negb:
   \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{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.
9.2.7
            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:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
     filter (fun = \Rightarrow false) l = [].
Lemma filter\_true:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
     filter (fun = \Rightarrow true) l = l.
Lemma filter\_andb:
   \forall (A : \mathsf{Type}) (f \ g : A \to bool) (l : list \ A),
     filter (fun x: A \Rightarrow andb (f x) (g x)) l =
     filter f (filter g l).
Lemma isEmpty\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      isEmpty\ (filter\ p\ l)=all\ (fun\ x:A\Rightarrow negb\ (p\ x))\ l.
Lemma length\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      length (filter p l) \leq length l.
Lemma filter\_snoc:
   \forall (A : \mathsf{Type}) (p : A \to 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:
   \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
```

 $filter \ p \ (l1 \ ++ \ l2) = filter \ p \ l1 \ ++ \ filter \ p \ l2.$

```
Lemma filter\_rev:
        \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
                filter \ p \ (rev \ l) = rev \ (filter \ p \ l).
Lemma filter\_map:
        \forall (A \ B : \mathsf{Type}) \ (f : A \to B) \ (p : B \to bool) \ (l : list \ A),
                filter p \pmod{f(l)} = map(f(filter(fun(x)); A \Rightarrow p(f(x)))).
Lemma filter\_join:
        \forall (A : \mathsf{Type}) (p : A \to bool) (lla : list (list A)),
                filter \ p \ (join \ lla) = join \ (map \ (filter \ p) \ lla).
Lemma filter\_replicate:
        \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
                 filter \ p \ (replicate \ n \ x) =
                 if p x then replicate n x else [].
Lemma filter\_iterate:
        \forall (A : \mathsf{Type}) (p : A \to bool) (f : A \to A) (n : nat) (x : A),
                 (\forall x: A, p (f x) = p x) \rightarrow
                         filter \ p \ (iterate \ f \ n \ x) =
                          if p x then iterate f n x else [].
Lemma head\_filter:
        \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
                 head (filter p l) = find p l.
Lemma last\_filter:
        \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
                 last (filter p l) = findLast p l.
Lemma filter\_nth:
        \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat) (x : A),
                 nth \ n \ l = Some \ x \rightarrow p \ x = true \rightarrow p \ x = t
                          \exists m : nat, m \leq n \wedge nth \ m \ (filter \ p \ l) = Some \ x.
Lemma splitAt_-filter:
        \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l1 \ l2 : list \ A) (x : A) (n : nat),
                 splitAt \ n \ (filter \ p \ l) = Some \ (l1, x, l2) \rightarrow l
                          \exists m: nat,
                         match \ splitAt \ m \ l \ with
                                            | None \Rightarrow False
                                            | Some (l1', y, l2') \Rightarrow
                                                             x = y \wedge l1 = filter p l1' \wedge l2 = filter p l2'
                          end.
Lemma filter\_insert:
        \forall (A : \mathsf{Type}) (p : A \to 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:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
     replace l \ n \ x = Some \ l' \rightarrow
        filter p l' =
        filter \ p \ (take \ n \ l) ++ filter \ p \ [x] \ ++ filter \ p \ (drop \ (S \ n) \ l).
Lemma remove\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (x : A) (n : nat),
      remove n (filter p l) = Some (x, l') \rightarrow
        \exists m : nat,
        match remove m l with
               | None \Rightarrow False
               | Some (y, l'') \Rightarrow x = y \land l' = filter p l''
         end.
Lemma filter\_idempotent:
   \forall (A : \mathsf{Type}) (f : A \to bool) (l : list A),
     filter f (filter f l) = filter f l.
Lemma filter\_comm:
   \forall (A : \mathsf{Type}) (f \ g : A \to bool) (l : list \ A),
     filter f (filter g l) = filter g (filter f l).
Lemma zip\_not\_filter:
   \exists (A \ B : \mathsf{Type}) \ (pa : A \rightarrow bool) \ (pb : B \rightarrow bool)
   (la: list A) (lb: list B),
      zip (filter pa la) (filter pb lb) \neq
     filter (fun x \Rightarrow andb (pa (fst x)) (pb (snd x))) (zip la lb).
Lemma any_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      any \ p \ l = negb \ (isEmpty \ (filter \ p \ l)).
Lemma all\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      all p (filter p l) = true.
Lemma all\_filter':
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      all p \mid l = isEmpty (filter (fun x : A \Rightarrow negb (p x)) \mid l).
Lemma filter\_all:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      all p \mid l = true \rightarrow filter \mid p \mid l = l.
Lemma removeFirst\_filter:
```

```
\forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst \ p \ (filter \ p \ l) =
      match filter p l with
            | | | \Rightarrow None
            | h :: t \Rightarrow Some (h, t)
      end.
Lemma removeFirst\_negb\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst (fun \ x : A \Rightarrow negb \ (p \ x)) (filter \ p \ l) = None.
Lemma findIndex\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      findIndex \ p \ (filter \ p \ l) = None \lor
      findIndex \ p \ (filter \ p \ l) = Some \ 0.
Lemma count\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count \ p \ (filter \ p \ l) = length \ (filter \ p \ l).
Lemma count\_filter':
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count \ p \ (filter \ p \ l) = count \ p \ l.
```

9.2.8 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: \mathtt{Type}) \ (p: A \rightarrow bool) \ (l: list \ A), partition p \ l = (filter \ p \ l, filter \ (\mathtt{fun} \ x \Rightarrow negb \ (p \ x)) \ l).
```

Lemma $partition_true$:

```
\forall (A: \mathtt{Type}) (p: A \rightarrow bool) (l: list A),

partition (\mathtt{fun} \ \_ \Rightarrow true) \ l = (l, []).
```

Lemma $partition_false$:

```
\forall (A : \mathsf{Type}) (p : A \to bool) (l : list A), partition (fun <math>\_ \Rightarrow false) l = (\parallel, l).
```

Lemma $partition_cons_true$:

```
\forall (A: \mathsf{Type}) \ (p: A \to bool) \ (h: A) \ (t \ l1 \ l2: list \ A),
p \ h = true \to partition \ p \ t = (l1, l2) \to
partition \ p \ (h:: t) = (h:: l1, l2).
```

Lemma $partition_cons_false$:

```
\forall (A : \mathsf{Type}) (p : A \to 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).
```

9.2.9findIndices

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 : \mathsf{Type}) (l : \mathit{list} A),
     findIndices (fun \_ \Rightarrow false) l = [].
Lemma findIndices\_true:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
     findIndices (fun \_ \Rightarrow true) l =
     if isEmpty\ l then [] else iterate\ S\ (length\ l)\ 0.
Lemma findIndices\_isEmpty\_true:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     isEmpty\ l=true \rightarrow findIndices\ p\ l=[].
Lemma isEmpty\_findIndices\_not\_nil:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     findIndices\ p\ l \neq [] \rightarrow isEmpty\ l = false.
Lemma length\_findIndices:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     length (findIndices p l) = count p l.
Lemma findIndices\_snoc:
  \forall (A : \mathsf{Type}) (p : A \to 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:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     findIndices p (l1 ++ l2) =
     findIndices p l1 ++ map (plus (length l1)) (findIndices p l2).
Lemma findIndices\_rev\_aux:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     rev (findIndices p (rev l)) =
     map (fun \ n : nat \Rightarrow length \ l - S \ n) (findIndices \ p \ l).
```

```
Lemma findIndices\_rev:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     findIndices p (rev l) =
      rev \ (map \ (fun \ n : nat \Rightarrow length \ l - S \ n) \ (findIndices \ p \ l)).
Lemma rev\_findIndices:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      rev (findIndices p l) =
      map \ (fun \ n : nat \Rightarrow length \ l - S \ n) \ (findIndices \ p \ (rev \ l)).
Lemma findIndices\_map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (p : B \to bool) (l : list A),
     findIndices p (map f l) =
     findIndices (fun x: A \Rightarrow p(f x)) l.
Lemma findIndices\_replicate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
     findIndices p (replicate n x) =
     {\tt match}\ n\ {\tt with}
           \mid 0 \Rightarrow \mid \mid
           |S| n' \Rightarrow \text{if } p x \text{ then } iterate |S| n |0| \text{ else } ||
      end.
Lemma map\_nth\_findIndices:
  \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l : list A),
     head (findIndices p l) = findIndex p l.
Lemma tail\_findIndices:
  \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat) (x : A),
     findIndices p (insert l n x) =
        findIndices p (take n l) ++
        (if p x then [min (length l) n] else []) ++
        map\ (plus\ (S\ n))\ (findIndices\ p\ (drop\ n\ l)).
Lemma findIndices\_replace:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (n : nat) (x : A),
     replace l \ n \ x = Some \ l' \rightarrow
     findIndices p l' =
     findIndices p (take n l) ++
     map\ (plus\ n)\ (findIndices\ p\ [x])\ ++
     map\ (plus\ (S\ n))\ (findIndices\ p\ (drop\ (S\ n)\ l)).
```

9.2.10 take While i drop While

Zdefiniuj funkcje takeWhile oraz dropWhile, które, dopóki funkcja p zwraca true, odpowiednio biora lub usuwa ja elementy z listy.

```
Przykład: take While \ even \ [2;\ 4;\ 6;\ 1;\ 8;\ 10;\ 12] = [2;\ 4;\ 6] drop While \ even \ [2;\ 4;\ 6;\ 1;\ 8;\ 10;\ 12] = [1;\ 8;\ 10;\ 12] Lemma take While \_drop While \_spec: \forall\ (A:\ \mathsf{Type})\ (p:\ A\to bool)\ (l:\ list\ A), take While\ p\ l ++\ drop\ While\ p\ l = l. Lemma take\ While\ _false: \forall\ (A:\ \mathsf{Type})\ (l:\ list\ A), take\ While\ (\texttt{fun}\ _\to\ false)\ l = []. Lemma drop\ While\ _false: \forall\ (A:\ \mathsf{Type})\ (l:\ list\ A), drop\ While\ (\texttt{fun}\ _\to\ false)\ l = l.
```

```
Lemma takeWhile\_andb:
  \forall (A : \mathsf{Type}) (p \ q : A \to bool) (l : list A),
     takeWhile (fun x : A \Rightarrow andb (p x) (q x)) l =
      takeWhile \ p \ (takeWhile \ q \ l).
Lemma isEmpty\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     isEmpty\ (takeWhile\ p\ l) =
     match l with
           | | | \Rightarrow true
           | h :: t \Rightarrow negb (p h)
     end.
Lemma isEmpty\_dropWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     isEmpty\ (drop\ While\ p\ l)=all\ p\ l.
Lemma takeWhile\_snoc\_all:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
     all p \mid l = true \rightarrow
        takeWhile \ p \ (snoc \ x \ l) = if \ p \ x \ then \ snoc \ x \ l \ else \ l.
Lemma takeWhile\_idempotent:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      takeWhile \ p \ (takeWhile \ p \ l) = takeWhile \ p \ l.
Lemma drop While\_idempotent:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     drop While \ p \ (drop While \ p \ l) = drop While \ p \ l.
Lemma takeWhile\_replicate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
      takeWhile \ p \ (replicate \ n \ x) =
     if p x then replicate n x else [].
Lemma takeWhile\_iterate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (f : A \to 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 drop While\_replicate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (x : A),
      drop While p (replicate n x) =
     if p x then [] else replicate n x.
Lemma drop While\_iterate:
  \forall (A : \mathsf{Type}) (p : A \to bool) (f : A \to A) (n : nat) (x : A),
     (\forall x : A, p (f x) = p x) \rightarrow
```

```
drop While \ p \ (iterate \ f \ n \ x) =
        if p x then [] else iterate f n x.
Lemma any\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     any \ p \ (takeWhile \ p \ l) = negb \ (isEmpty \ (takeWhile \ p \ l)).
Lemma any\_drop While:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      any (fun x: A \Rightarrow negb(p x)) (drop While p l) =
     negb (isEmpty (dropWhile p l)).
Lemma any\_takeWhile\_dropWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      any \ p \ l = orb \ (any \ p \ (takeWhile \ p \ l)) \ (any \ p \ (dropWhile \ p \ l)).
Lemma all\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     all \ p \ (takeWhile \ p \ l) = true.
Lemma all\_takeWhile':
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     all p \mid l = true \rightarrow takeWhile \mid p \mid l = l.
Lemma all_{-}dropWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     all \ p \ (drop While \ p \ l) = all \ p \ l.
Lemma takeWhile\_app\_all:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     all \ p \ l1 = true \rightarrow takeWhile \ p \ (l1 \ ++ \ l2) = l1 \ ++ \ takeWhile \ p \ l2.
Lemma removeFirst\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     removeFirst \ p \ (takeWhile \ p \ l) =
     match take While p l with
           | [] \Rightarrow None
           | h :: t \Rightarrow Some (h, t)
     end.
Lemma removeLast\_dropWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     removeFirst\ p\ (drop\ While\ (fun\ x:A\Rightarrow negb\ (p\ x))\ l) =
     match drop While (fun x : A \Rightarrow negb (p x)) l with
           | | | \Rightarrow None
           | h :: t \Rightarrow Some (h, t)
     end.
Lemma findIndex\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n m : nat),
```

```
findIndex \ p \ (takeWhile \ p \ l) = Some \ n \rightarrow
        findIndex \ p \ l = Some \ n \rightarrow n \leq m.
Lemma findIndex\_spec':
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
     findIndex \ p \ l = Some \ n \rightarrow
        takeWhile (fun x : A \Rightarrow negb (p x)) l = take n l.
Lemma findIndex\_drop While:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n m : nat),
     findIndex \ p \ (drop While \ p \ l) = Some \ m \rightarrow
        findIndex \ p \ l = Some \ n \rightarrow n \leq m.
Lemma count\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count \ p \ (takeWhile \ p \ l) = length \ (takeWhile \ p \ l).
Lemma count\_drop While:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count \ p \ (drop While \ p \ l) < count \ p \ l.
Lemma count\_takeWhile\_dropWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      count\ p\ (takeWhile\ p\ l) + count\ p\ (dropWhile\ p\ l) = count\ p\ l.
```

$9.2.11 \quad span$

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: {\tt Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A), \\ span \ p \ l = Some \ (b, x, \ e) \rightarrow \\ is Empty \ l = false.
```

Lemma $length_span$:

```
\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A), span p \ l = Some \ (b, x, e) \rightarrow length \ b + length \ e < length \ l.
```

Lemma $length_span'$:

```
orall \; (A: {\tt Type}) \; (p:A 	o bool) \; (x:A) \; (l\;b\;e:list\;A), \ span\; p\; l = Some\; (b,x,e) 	o \ length\; b < length\; l \wedge \ length\; e < length\; l.
```

```
Lemma span\_snoc:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
     span p (snoc x l) =
     match span p l with
           | None \Rightarrow \text{if } p \text{ } x \text{ then } Some (l, x, ||) \text{ else } None
           | Some (b, y, e) \Rightarrow Some (b, y, snoc x e) |
      end.
Lemma span\_app:
  \forall (A : \mathsf{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), \bot \Rightarrow Some (b, x, e ++ l2)
            | \_, Some (b, x, e) \Rightarrow Some (l1 ++ b, x, e)
            | \_, \_ \Rightarrow None
      end.
Lemma span_{-}map:
  \forall (A \ B : \mathsf{Type}) \ (f : A \to B) \ (p : B \to bool) \ (l : list \ A),
      span \ p \ (map \ f \ l) =
     match span (fun x : A \Rightarrow p (f x)) l with
            | None \Rightarrow None
           | Some (b, x, e) \Rightarrow Some (map f b, f x, map f e) |
      end.
Lemma span_{-}join:
  \forall (A : \mathsf{Type}) (p : A \to bool) (lla : list (list A)),
      span p (join lla) =
     match span (any p) lla with
            | None \Rightarrow None
           | Some (bl, l, el) \Rightarrow
                 match span p l with
                        None \Rightarrow None
                        Some (b, x, e) \Rightarrow Some (join bl ++ b, x, e ++ join el)
                 end
      end.
Lemma span\_replicate:
  \forall (A : \mathsf{Type}) (p : A \to 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:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ b \ e : list \ A),
```

```
span \ p \ l = Some \ (b, x, e) \rightarrow any \ p \ l = true. Lemma span\_all:
```

$$\forall \ (A: \texttt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A), \\ span \ p \ l = Some \ (b, x, \ e) \rightarrow \\ all \ p \ l = andb \ (beq_nat \ (length \ b) \ 0) \ (all \ p \ e).$$

 $Lemma\ span_find$:

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A),$$
 span $p \ l = Some \ (b, x, e) \rightarrow find \ p \ l = Some \ x.$

Lemma $span_removeFirst$:

$$\forall$$
 $(A: \mathsf{Type}) \ (p: A \to bool) \ (x: A) \ (l \ b \ e: list \ A),$ $span \ p \ l = Some \ (b, x, e) \to removeFirst \ p \ l = Some \ (x, b ++ e).$

Lemma $count_span_l$:

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A),$$
 span $p \ l = Some \ (b, x, e) \rightarrow count \ p \ b = 0.$

Lemma $count_span_r$:

$$\begin{array}{l} \forall \ (A: {\tt Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A), \\ span \ p \ l = Some \ (b, x, \ e) \rightarrow \\ count \ p \ e < length \ l - length \ b. \end{array}$$

 $Lemma\ span_filter$:

$$\begin{array}{l} \forall\; (A: {\tt Type})\; (p: A \to bool)\; (l: list\; A),\\ span\; p\; (filter\; p\; l) =\\ {\tt match}\; filter\; p\; l\; {\tt with}\\ {\;\;|\;\;} |\;\; | \to None\\ {\;\;|\;\;} h\; ::\; t \Rightarrow Some\; ([],\; h,\; t)\\ {\tt end}. \end{array}$$

 $Lemma\ filter_span_l$:

$$\forall (A: \mathtt{Type}) (p: A \rightarrow bool) (x: A) (l \ b \ e: list \ A),$$
 span $p \ l = Some \ (b, x, e) \rightarrow filter \ p \ b = [].$

 ${\tt Lemma}\ take While_span\ :$

$$\forall (A: \mathtt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A),$$
 $span \ p \ l = Some \ (b, x, e) \rightarrow take While \ (\mathtt{fun} \ x: A \Rightarrow neqb \ (p \ x)) \ l = b.$

 ${\tt Lemma}\ drop\ While_span\ :$

$$\forall \ (A: \texttt{Type}) \ (p: A \rightarrow bool) \ (x: A) \ (l \ b \ e: list \ A), \\ span \ p \ l = Some \ (b, x, \ e) \rightarrow \\ drop \ While \ (\texttt{fun} \ x: A \Rightarrow negb \ (p \ x)) \ l = x :: e.$$

Związki span i rev

Zdefiniuj funkcję naps, która działa tak jak span, tyle że "od tylu". Udowodnij twierdzenie $span_rev$.

```
Lemma span\_rev\_aux:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
     span p l =
     match naps p (rev l) with
          | None \Rightarrow None
          | Some (b, x, e) \Rightarrow Some (rev e, x, rev b) |
     end.
Lemma span_rev:
  \forall (A : \mathsf{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.
         Sekcja mocno ad hoc
9.3
9.3.1
           pmap
```

 $pmap \ f \ (snoc \ a \ l) =$ match f a with

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 \text{ (fun } n: nat \Rightarrow \text{ if } even \text{ } n \text{ then } None \text{ else } Some \text{ } (n+42)) \text{ } [1; 2; 3] = [43; 45]
Lemma isEmpty\_pmap\_false:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l : list A),
      isEmpty\ (pmap\ f\ l) = false \rightarrow isEmpty\ l = false.
Lemma isEmpty\_pmap\_true:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l : list A),
      isEmpty\ l = true \rightarrow isEmpty\ (pmap\ f\ l) = true.
Lemma length\_pmap:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l : list A),
     length (pmap f l) \leq length l.
Lemma pmap\_snoc:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (a : A) (l : list A),
```

```
| None \Rightarrow pmap f | l
            | Some b \Rightarrow snoc b (pmap f l)
      end.
Lemma pmap\_app:
  \forall (A B : \mathsf{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 : \mathsf{Type}) (f : A \to option B) (l : list A),
     pmap \ f \ (rev \ l) = rev \ (pmap \ f \ l).
Lemma pmap_{-}map:
   \forall (A \ B \ C : \mathsf{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:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (l : list (list A)),
      pmap \ f \ (join \ l) = join \ (map \ (pmap \ f) \ l).
Lemma pmap\_bind:
   \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to list \ B) \ (g : B \to option \ C) \ (l : list \ A),
      pmap \ g \ (bind \ f \ l) = bind \ (fun \ x : A \Rightarrow pmap \ g \ (f \ x)) \ l.
Lemma pmap\_replicate:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (n : nat) (x : A),
      pmap \ f \ (replicate \ n \ x) =
     {\tt match}\; f\; x\; {\tt with}
            | None \Rightarrow []
            \mid Some \ y \Rightarrow replicate \ n \ y
      end.
Definition isSome \{A : Type\} (x : option A) : bool :=
{\tt match}\ x\ {\tt with}
      | None \Rightarrow false
      | \ \_ \Rightarrow true
end.
Lemma head\_pmap:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l : list A),
      head (pmap f l) =
     match find isSome (map f l) with
            | None \Rightarrow None |
            \mid Some \ x \Rightarrow x
      end.
Lemma pmap\_zip:
      (A B C : Type)
```

```
(fa: A \rightarrow option \ C) \ (fb: B \rightarrow option \ C)
      (la: list A) (lb: list B),
         pmap
             (fun'(a, b) \Rightarrow
            match fa a, fb b with
                    | Some a', Some b' \Rightarrow Some (a', b')
                    | \_, \_ \Rightarrow None
             end)
             (zip \ la \ lb) \neq
         zip (pmap fa la) (pmap fb lb).
Lemma any_pmap:
   \forall (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (p : B \to bool) \ (l : list \ A),
      any \ p \ (pmap \ f \ l) =
      any
         (\operatorname{fun} x : A \Rightarrow
         \mathrm{match}\,f\,x\,\mathrm{with}
                | Some b \Rightarrow p b
                | None \Rightarrow false
         end)
         l.
Lemma all_pmap:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (p : B \to bool) (l : list A),
      all \ p \ (pmap \ f \ l) =
      all
         (\operatorname{fun} x : A \Rightarrow
         \mathtt{match}\ f\ x\ \mathtt{with}
                | Some b \Rightarrow p b
                | None \Rightarrow true
         end)
         l.
Lemma find_pmap:
   \forall (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (p : B \to bool) \ (l : list \ A),
      find p (pmap f l) =
      let oa :=
         find (fun \ x : A \Rightarrow match \ f \ x \ with \ Some \ b \Rightarrow p \ b \mid \_ \Rightarrow false \ end) \ l
      in
      match oa with
             | Some a \Rightarrow f a
             | None \Rightarrow None
      end.
Lemma findLast\_pmap:
```

```
\forall (A B : \mathsf{Type}) (f : A \to option B) (p : B \to bool) (l : list A),
     findLast \ p \ (pmap \ f \ l) =
      let oa :=
         findLast
            (fun x: A \Rightarrow \text{match } f \ x \text{ with } Some \ b \Rightarrow p \ b \mid \_ \Rightarrow false \ \text{end}) l
      in
     match oa with
            \mid Some \ a \Rightarrow f \ a
            | None \Rightarrow None
      end.
Lemma count_pmap:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (p : B \to bool) (l : list A),
      count \ p \ (pmap \ f \ l) =
      count
         (\operatorname{fun} x : A \Rightarrow
         match f x with
               | Some b \Rightarrow p b
               | None \Rightarrow false
         end)
         l.
(* TODO *) Definition aux \{A \ B : Type\} \ (p : B \rightarrow bool) \ (f : A \rightarrow option \ B)
   (dflt:bool)(x:A):bool:=
\mathtt{match}\,f\,x\,\mathtt{with}
       Some b \Rightarrow p \ b
       None \Rightarrow dflt
end.
Lemma pmap\_filter:
  \forall (A B : \mathsf{Type}) (p : B \to bool) (f : A \to option B) (l : list A),
     filter \ p \ (pmap \ f \ l) =
      pmap \ f \ (filter \ (aux \ p \ f \ false) \ l).
Lemma pmap\_takeWhile:
  \forall (A B : \mathsf{Type}) (p : B \to bool) (f : A \to option B) (l : list A),
      takeWhile \ p \ (pmap \ f \ l) =
     pmap f
         (take While
            (fun x: A \Rightarrow \text{match } f \ x \text{ with } | Some \ b \Rightarrow p \ b | \_ \Rightarrow true \ \text{end})
            l).
Lemma pmap\_drop While:
  \forall (A B : \mathsf{Type}) (p : B \to bool) (f : A \to option B) (l : list A),
      drop While p (pmap f l) =
     pmap f
```

```
(drop While)
            (fun x: A \Rightarrow \text{match } f \ x \text{ with } | Some \ b \Rightarrow p \ b | \_ \Rightarrow true \ \text{end})
            l).
Lemma pmap\_span:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (p : B \to bool) (l : list A),
      match
         span
            (fun x: A \Rightarrow \text{match } f \text{ } x \text{ with } None \Rightarrow false \mid Some \text{ } b \Rightarrow p \text{ } b \text{ end})
      with
             | None \Rightarrow True
             | Some (b, x, e) \Rightarrow
                   \exists y: B, f \ x = Some \ y \land
                      span \ p \ (pmap \ f \ l) = Some \ (pmap \ f \ b, \ y, \ pmap \ f \ e)
      end.
Lemma pmap\_nth\_findIndices:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      pmap (fun \ n : nat \Rightarrow nth \ n \ l) (findIndices \ p \ l) =
      filter p l.
           Bardziej skomplikowane funkcje
```

9.4

9.4.1intersperse

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 : \mathsf{Type}) (x : A) (l : \mathit{list} A),
   isEmpty (intersperse \ x \ l) = isEmpty \ l.
```

Lemma $length_intersperse$:

```
\forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
   length (intersperse \ x \ l) = 2 \times length \ l - 1.
```

Lemma $intersperse_snoc$:

```
\forall (A : \mathsf{Type}) (x \ y : A) (l : \mathit{list} \ A),
   intersperse \ x \ (snoc \ y \ l) =
   if isEmpty\ l then [y] else snoc\ y\ (snoc\ x\ (intersperse\ x\ l)).
```

Lemma $intersperse_app$:

```
\forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
```

```
intersperse \ x \ (l1 ++ l2) =
      match l1, l2 with
            | [], \_ \Rightarrow intersperse \ x \ l2
            | -, [] \Rightarrow intersperse \ x \ l1
            \mid h1 :: t1, h2 :: t2 \Rightarrow
                   intersperse \ x \ l1 \ ++ \ x :: intersperse \ x \ l2
      end.
Lemma intersperse\_app\_cons:
   \forall (A : \mathsf{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 : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      intersperse \ x \ (rev \ l) = rev \ (intersperse \ x \ l).
Lemma intersperse\_map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (a : A) (b : B),
      f \ a = b \rightarrow intersperse \ b \ (map \ f \ l) = map \ f \ (intersperse \ a \ l).
Lemma head\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      head (intersperse x l) = head l.
Lemma last\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      last (intersperse x l) = last l.
Lemma tail\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{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 : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n : \mathit{nat}),
      n < length l \rightarrow
         nth (2 \times n) (intersperse \ x \ l) = nth \ n \ l.
Lemma nth\_intersperse\_odd:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n : \mathit{nat}),
      0\,<\,n\,\rightarrow\,n\,<\,\mathit{length}\ l\,\rightarrow\,
         nth (2 \times n - 1) (intersperse \ x \ l) = Some \ x.
Lemma intersperse\_take:
```

```
\forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n : \mathit{nat}),
      intersperse \ x \ (take \ n \ l) =
      take (2 \times n - 1) (intersperse \ x \ l).
Lemma any\_intersperse:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      any p (intersperse x l) =
       orb (any \ p \ l) \ (andb \ (2 \le ? length \ l) \ (p \ x)).
Lemma all\_intersperse:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      all p (intersperse x l) =
      all p \mid k \& ((length \mid l <=? \mid 1) \mid \mid p \mid x).
Lemma findIndex\_intersperse:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      findIndex \ p \ (intersperse \ x \ l) =
      if p x
      then
         {\tt match}\ l\ {\tt with}
                | | | \Rightarrow None
                | [h] \Rightarrow \text{if } p \text{ } h \text{ then } Some \text{ } 0 \text{ else } None
                | h :: t \Rightarrow \text{if } p \text{ } h \text{ then } Some \text{ } 0 \text{ else } Some \text{ } 1
          end
      else
         match findIndex p l with
                 None \Rightarrow None
                | Some \ n \Rightarrow Some \ (2 \times n) |
          end.
Lemma count\_intersperse:
   \forall (A : \mathsf{Type}) (p : A \to 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:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      p \ x = false \rightarrow filter \ p \ (intersperse \ x \ l) = filter \ p \ l.
Lemma pmap\_intersperse:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (x : A) (l : list A),
      f \ x = None \rightarrow pmap \ f \ (intersperse \ x \ l) = pmap \ f \ l.
```

9.5 Proste predykaty

9.5.1 elem

Zdefiniuj induktywny predykat elem. elem x l jest spełniony, gdy x jest elementem listy l.

```
Lemma elem\_not\_nil:
   \forall (A : \mathsf{Type}) (x : A), \neg elem x [].
Lemma elem\_not\_cons:
   \forall (A : \mathsf{Type}) (x \ h : A) (t : \mathit{list} \ A),
       \neg elem \ x \ (h :: t) \rightarrow x \neq h \land \neg elem \ x \ t.
Lemma elem\_cons':
   \forall (A : \mathsf{Type}) (x \ h : A) (t : \mathit{list} \ A),
       elem \ x \ (h :: t) \leftrightarrow x = h \lor elem \ x \ t.
Lemma elem\_snoc:
   \forall (A : \mathsf{Type}) (x \ y : A) (l : \mathit{list} \ A),
       elem \ x \ (snoc \ y \ l) \leftrightarrow elem \ x \ l \lor x = y.
Lemma elem\_app\_l:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       elem \ x \ l1 \rightarrow elem \ x \ (l1 ++ l2).
Lemma elem\_app\_r:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       elem \ x \ l2 \rightarrow elem \ x \ (l1 \ ++ \ l2).
Lemma elem\_or\_app:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       elem \ x \ l1 \ \lor \ elem \ x \ l2 \rightarrow \ elem \ x \ (l1 \ ++ \ l2).
Lemma elem\_app\_or:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       elem \ x \ (l1 \ ++ \ l2) \rightarrow elem \ x \ l1 \ \lor \ elem \ x \ l2.
Lemma elem\_app:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       elem \ x \ (l1 \ ++ \ l2) \leftrightarrow elem \ x \ l1 \ \lor \ elem \ x \ l2.
Lemma elem\_spec:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       elem \ x \ l \leftrightarrow \exists \ l1 \ l2 : list \ A, \ l = l1 \ ++ \ x :: l2.
Lemma elem_{-}map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (x : A),
       elem \ x \ l \rightarrow elem \ (f \ x) \ (map \ f \ l).
Lemma elem_{-}map_{-}conv:
   \forall (A \ B : \mathsf{Type}) \ (f : A \to B) \ (l : list \ A) \ (y : B),
```

```
elem\ y\ (map\ f\ l) \leftrightarrow \exists\ x:A,f\ x=y \land elem\ x\ l.
Lemma elem_{-}map_{-}conv':
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A) (x : A),
       (\forall x y : A, f x = f y \rightarrow x = y) \rightarrow
           elem (f x) (map f l) \rightarrow elem x l.
Lemma map\_ext\_elem:
   \forall (A B : \mathsf{Type}) (f g : A \to B) (l : \mathit{list} A),
       (\forall x: A, elem \ x \ l \rightarrow f \ x = g \ x) \rightarrow map \ f \ l = map \ g \ l.
Lemma elem_{-}join:
   \forall (A : \mathsf{Type}) (x : A) (ll : list (list A)),
       elem \ x \ (join \ ll) \leftrightarrow \exists \ l : list \ A, \ elem \ x \ l \land \ elem \ l \ ll.
Lemma elem\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x y : A),
       elem y (replicate n x) \leftrightarrow n \neq 0 \land x = y.
Lemma nth\_elem:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       n < length \ l \rightarrow \exists \ x : A, \ nth \ n \ l = Some \ x \land elem \ x \ l.
(* TOOD: ulepszyć? *) Lemma iff_-elem_-nth:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       elem x \mid l \leftrightarrow \exists n : nat, nth \mid n \mid l = Some \mid x.
Lemma elem\_rev\_aux :
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       elem \ x \ l \rightarrow elem \ x \ (rev \ l).
Lemma elem\_rev:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       elem \ x \ (rev \ l) \leftrightarrow elem \ x \ l.
Lemma elem\_remove\_nth:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A) (n : \mathit{nat}),
       elem \ x \ l \rightarrow nth \ n \ l \neq Some \ x \rightarrow
      match remove \ n \ l with
              | None \Rightarrow True
              | Some (\_, l') \Rightarrow elem x l'
       end.
Lemma elem_{-}take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
       elem \ x \ (take \ n \ l) \rightarrow elem \ x \ l.
Lemma elem\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
       elem \ x \ (drop \ n \ l) \rightarrow elem \ x \ l.
```

```
Lemma elem\_splitAt':
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
      splitAt \ n \ l = Some \ (l1, y, l2) \rightarrow
          elem \ x \ l \leftrightarrow x = y \lor elem \ x \ l1 \lor elem \ x \ l2.
Lemma elem\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
       elem\ y\ (insert\ l\ n\ x) \leftrightarrow x = y \lor elem\ y\ l.
Lemma elem_{-}replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
      replace l \ n \ x = Some \ l' \rightarrow
          elem y \mid l' \leftrightarrow elem y \ (take \mid n \mid l) \lor x = y \lor elem y \ (drop \ (S \mid n) \mid l).
Lemma elem_-filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
       elem \ x \ (filter \ p \ l) \leftrightarrow p \ x = true \land elem \ x \ l.
Lemma elem\_filter\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
       elem \ x \ l \leftrightarrow
       elem \ x \ (filter \ p \ l) \land p \ x = true \lor
       elem x (filter (fun x : A \Rightarrow negb (p x)) l) \land p x = false.
Lemma elem\_partition:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ l1 \ l2 : list \ A),
      partition p \ l = (l1, l2) \rightarrow
          elem \ x \ l \leftrightarrow
          (elem \ x \ l1 \land p \ x = true) \lor (elem \ x \ l2 \land p \ x = false).
Lemma elem_{-}takeWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
       elem\ x\ (takeWhile\ p\ l) \rightarrow elem\ x\ l \wedge p\ x = true.
Lemma elem\_drop While:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
       elem \ x \ (drop While \ p \ l) \rightarrow elem \ x \ l.
Lemma elem\_takeWhile\_dropWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
       elem \ x \ l \leftrightarrow elem \ x \ (takeWhile \ p \ l) \lor elem \ x \ (dropWhile \ p \ l).
Lemma elem\_drop While\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
       elem \ x \ l \rightarrow \neg \ elem \ x \ (drop While \ p \ l) \rightarrow p \ x = true.
     TODO: span i intersperse, groupBy *)
Lemma span\_spec':
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
```

```
match span p l with
            | None \Rightarrow \forall x : A, elem x | l \rightarrow p | x = false
            | Some (b, x, e) \Rightarrow
                  b = takeWhile (fun \ x : A \Rightarrow negb \ (p \ x)) \ l \land
                  Some x = find p l \wedge
                  x :: e = drop While (fun x : A \Rightarrow negb (p x)) l \wedge
                  Some (x, b ++ e) = removeFirst p l
      end.
Lemma elem\_span\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      span \ p \ l = None \rightarrow \forall \ x : A, \ elem \ x \ l \rightarrow p \ x = false.
Lemma elem\_span\_Some:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ b \ e : list \ A),
      span \ p \ l = Some \ (b, x, e) \rightarrow
         \forall y: A, elem \ y \ l \leftrightarrow elem \ y \ b \lor y = x \lor elem \ y \ e.
Lemma elem\_span:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      match span p l with
            | None \Rightarrow \forall x : A, elem x l \rightarrow p x = false
            | Some (b, x, e) \Rightarrow
                  \forall y: A, elem \ y \ l \leftrightarrow elem \ y \ b \lor y = x \lor elem \ y \ e
      end.
Lemma elem\_removeFirst\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      removeFirst \ p \ l = None \rightarrow
         \forall x: A, elem x l \rightarrow p x = false.
Lemma elem_-zip:
  \forall (A B : \mathsf{Type}) (a : A) (b : B) (la : list A) (lb : list B),
      elem\ (a,\ b)\ (zip\ la\ lb) \rightarrow elem\ a\ la\ \land\ elem\ b\ lb.
Lemma zip\_not\_elem:
   \exists (A B : \mathsf{Type}) (a : A) (b : B) (la : list A) (lb : list B),
      elem\ a\ la\ \land\ elem\ b\ lb\ \land\ \neg\ elem\ (a,\ b)\ (zip\ la\ lb).
Lemma elem\_findIndices:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (n : nat),
      elem \ n \ (findIndices \ p \ l) \rightarrow
         \exists x: A, nth \ n \ l = Some \ x \land p \ x = true.
Lemma isEmpty\_bind:
   \forall (A B : \mathsf{Type}) (f : A \to list B) (l : list A),
      isEmpty\ (bind\ f\ l) = true \leftrightarrow
      l = [] \lor l \neq [] \land \forall x : A, elem \ x \ l \rightarrow f \ x = [].
```

```
Lemma elem\_pmap: \forall \ (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (l : list \ A) \ (a : A) \ (b : B), f \ a = Some \ b \to elem \ a \ l \to elem \ b \ (pmap \ f \ l). Lemma elem\_pmap': \forall \ (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (l : list \ A) \ (b : B), (\exists \ a : A, \ elem \ a \ l \land f \ a = Some \ b) \to elem \ b \ (pmap \ f \ l). Lemma elem\_pmap\_conv: \forall \ (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (l : list \ A) \ (b : B), elem \ b \ (pmap \ f \ l) \to \exists \ a : A, \ elem \ a \ l \land f \ a = Some \ b. Lemma elem\_intersperse: \forall \ (A : \mathsf{Type}) \ (x \ y : A) \ (l : list \ A), elem \ x \ (intersperse \ y \ l) \leftrightarrow elem \ x \ l \lor (x = y \land 2 \le length \ l).
```

9.5.2 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: \mathtt{Type}\}\ (x:A)\ (l:\mathit{list}\ A): \mathtt{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".

```
 \begin{array}{c} \operatorname{Lemma} \ In\_elem : \\ \forall \ (A: \mathtt{Type}) \ (x:A) \ (l: \mathit{list} \ A), \\ In \ x \ l \leftrightarrow \mathit{elem} \ x \ l. \\ \\ \operatorname{Lemma} \ In\_not\_nil : \\ \forall \ (A: \mathtt{Type}) \ (x:A), \ \neg \ In \ x \ []. \\ \\ \operatorname{Lemma} \ In\_not\_cons : \\ \forall \ (A: \mathtt{Type}) \ (x \ h:A) \ (t: \mathit{list} \ A), \end{array}
```

```
\neg \ In \ x \ (h :: t) \rightarrow x \neq h \land \neg \ In \ x \ t. Lemma In\_cons :
```

$$\forall \ (A: \mathtt{Type}) \ (x \ h: A) \ (t: \mathit{list} \ A), \\ \mathit{In} \ x \ (h:: t) \leftrightarrow x = h \ \lor \mathit{In} \ x \ t.$$

Lemma In_snoc :

$$\forall (A: \mathtt{Type}) \ (x \ y: A) \ (l: \mathit{list} \ A), \\ \mathit{In} \ x \ (\mathit{snoc} \ y \ l) \leftrightarrow \mathit{In} \ x \ l \ \lor x = y.$$

Lemma In_app_l :

$$\forall (A: \mathtt{Type}) (x:A) (l1 \ l2: list \ A),$$

 $In \ x \ l1 \rightarrow In \ x \ (l1 \ ++ \ l2).$

Lemma In_app_r :

$$\forall \ (A: \mathtt{Type}) \ (x:A) \ (l1 \ l2: \mathit{list} \ A), \\ \mathit{In} \ x \ l2 \ \rightarrow \mathit{In} \ x \ (l1 \ ++ \ l2).$$

Lemma In_or_app :

$$\forall (A: \mathtt{Type}) (x:A) (l1 \ l2: list \ A), \\ In \ x \ l1 \ \lor In \ x \ l2 \rightarrow In \ x \ (l1 \ ++ \ l2).$$

Lemma In_app_or :

$$\forall \ (A: \mathtt{Type}) \ (x:A) \ (l1 \ l2: list \ A),$$

$$In \ x \ (l1 \ ++ \ l2) \rightarrow In \ x \ l1 \ \lor In \ x \ l2.$$

Lemma In_app :

$$\forall (A: \mathtt{Type}) \ (x:A) \ (l1 \ l2: list \ A),$$

$$In \ x \ (l1 \ ++ \ l2) \leftrightarrow In \ x \ l1 \ \lor In \ x \ l2.$$

 ${\tt Lemma}\ In_spec:$

$$\begin{array}{c} \forall \ (A: {\tt Type}) \ (x:A) \ (l: \mathit{list} \ A), \\ \mathit{In} \ x \ l \leftrightarrow \exists \ \mathit{l1} \ \mathit{l2} : \mathit{list} \ A, \ \mathit{l} = \mathit{l1} \ ++ \ x :: \mathit{l2}. \end{array}$$

 $\texttt{Lemma}\ In_map :$

$$\forall (A \ B : \mathtt{Type}) \ (f : A \rightarrow B) \ (l : \mathit{list} \ A) \ (x : A),$$
 $\mathit{In} \ x \ l \rightarrow \mathit{In} \ (f \ x) \ (\mathit{map} \ f \ l).$

 ${\tt Lemma}\; In_map_conv :$

$$\forall \ (A \ B : \mathtt{Type}) \ (f : A \to B) \ (l : \mathit{list} \ A) \ (y : B),$$

$$\mathit{In} \ y \ (\mathit{map} \ f \ l) \leftrightarrow \exists \ x : A, f \ x = y \land \mathit{In} \ x \ l.$$

Lemma In_map_conv' :

$$\forall \ (A \ B : \mathtt{Type}) \ (f : A \rightarrow B) \ (l : \mathit{list} \ A) \ (x : A), \\ (\forall \ x \ y : A, f \ x = f \ y \rightarrow x = y) \rightarrow \\ \mathit{In} \ (f \ x) \ (\mathit{map} \ f \ l) \rightarrow \mathit{In} \ x \ l.$$

Lemma map_ext_In :

$$\forall (A \ B : \mathtt{Type}) \ (f \ g : A \to B) \ (l : \mathit{list} \ A), \\ (\forall x : A, \mathit{In} \ x \ l \to f \ x = g \ x) \to \mathit{map} \ f \ l = \mathit{map} \ g \ l.$$

 ${\tt Lemma}\ In_join\ :$

```
\forall (A : \mathsf{Type}) (x : A) (ll : list (list A)),
       In x (join ll) \leftrightarrow
       \exists l : list A, In x l \wedge In l ll.
Lemma In\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x y : A),
       In y (replicate n x) \leftrightarrow n \neq 0 \land x = y.
Lemma In_{-}iterate:
    \forall (A : \mathsf{Type}) (f : A \to 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 : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       n < length \ l \rightarrow \exists \ x : A, \ nth \ n \ l = Some \ x \land In \ x \ l.
Lemma iff_{-}In_{-}nth:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       In x \mid l \leftrightarrow \exists n : nat, nth \mid n \mid l = Some \mid x.
Lemma In\_rev\_aux:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       In x \mid l \rightarrow In \mid x \mid (rev \mid l).
Lemma In\_rev:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       In \ x \ (rev \ l) \leftrightarrow In \ x \ l.
Lemma In\_take:
    \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
       In x (take n l) \rightarrow In x l.
Lemma In\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
       In \ x \ (drop \ n \ l) \rightarrow In \ x \ l.
Lemma In\_splitAt:
   \forall (A : \mathsf{Type}) (l \ b \ e : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
       splitAt \ n \ l = Some \ (b, x, e) \rightarrow
           In y \mid l \leftrightarrow In \mid y \mid b \lor x = y \lor In \mid y \mid e.
Lemma In\_insert:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
       In y (insert l n x) \leftrightarrow x = y \lor In y l.
Lemma In\_replace:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x \ y : A),
       replace l \ n \ x = Some \ l' \rightarrow
           In y \mid l' \leftrightarrow In \ y \ (take \ n \mid l) \lor x = y \lor In \ y \ (drop \ (S \mid n) \mid l).
```

Lemma $In_{-}filter$:

```
\forall (A : \mathsf{Type}) (p : A \to 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 : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A),
      In x \ l \leftrightarrow
      In x (filter p l) \wedge p x = true \vee
      In x (filter (fun x : A \Rightarrow negb(p x)) l) \land p x = false.
Lemma In\_partition:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ l1 \ l2 : list \ A),
      partition p \ l = (l1, l2) \rightarrow
         In x \mid l \leftrightarrow
         (In \ x \ l1 \land p \ x = true) \lor (In \ x \ l2 \land p \ x = false).
Lemma In\_takeWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
      In x (takeWhile p l) \rightarrow In x l \land p x = true.
Lemma In\_drop While:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
      In x (drop While p l) \rightarrow In x l.
Lemma In\_takeWhile\_dropWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
      In x l \rightarrow
         In x (takeWhile p l) \vee
         In x (drop While p l).
Lemma In\_drop While\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A) (x : A),
      In x \mid l \rightarrow \neg In x \mid (drop While \mid p \mid l) \rightarrow p \mid x = true.
(* TODO: jak elem *)
Lemma In\_span:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x y : A) (l b e : list A),
      span \ p \ l = Some \ (b, x, e) \rightarrow
         In y \mid l \leftrightarrow In \mid y \mid b \lor y = x \lor In \mid y \mid e.
Lemma In_{-}zip :
   \forall (A B : \mathsf{Type}) (a : A) (b : B) (la : list A) (lb : list B),
      In (a, b) (zip \ la \ lb) \rightarrow In \ a \ la \wedge In \ b \ lb.
Lemma zip\_not\_In:
   \exists (A B : \mathsf{Type}) (a : A) (b : B) (la : list A) (lb : list B),
      In a la \wedge In \ b \ lb \wedge \neg In \ (a, b) \ (zip \ la \ lb).
Lemma In\_intersperse:
   \forall (A : \mathsf{Type}) (x \ y : A) (l : \mathit{list} \ A),
```

In x (intersperse y l) \leftrightarrow

```
In x \mid l \lor (x = y \land 2 \leq length \mid l).
```

9.5.3 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 pożadane właściwości.

```
Lemma NoDup\_singl:
   \forall (A : \mathsf{Type}) (x : A), NoDup [x].
Lemma NoDup\_cons\_inv:
   \forall (A : \mathsf{Type}) (h : A) (t : \mathit{list} A),
      NoDup\ (h :: t) \rightarrow NoDup\ t.
Lemma NoDup\_length:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      \neg NoDup \ l \rightarrow 2 \leq length \ l.
Lemma NoDup\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      NoDup\ (snoc\ x\ l) \leftrightarrow NoDup\ l \land \neg\ elem\ x\ l.
Lemma NoDup\_app :
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      NoDup\ (l1\ ++\ l2) \leftrightarrow
      NoDup \ l1 \ \land
      NoDup \ l2 \ \land
      (\forall x : A, elem \ x \ l1 \rightarrow \neg elem \ x \ l2) \land
      (\forall x : A, elem \ x \ l2 \rightarrow \neg elem \ x \ l1).
Lemma NoDup\_app\_comm:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      NoDup\ (l1\ ++\ l2) \leftrightarrow NoDup\ (l2\ ++\ l1).
Lemma NoDup\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      NoDup \ (rev \ l) \leftrightarrow NoDup \ l.
Lemma NoDup\_map :
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A),
      NoDup\ (map\ f\ l) \to NoDup\ l.
Lemma NoDup\_map\_inj:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{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 : \mathsf{Type}) (n : nat) (x : A),
      NoDup (replicate n x) \leftrightarrow n = 0 \lor n = 1.
Lemma NoDup\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      NoDup \ l \rightarrow NoDup \ (take \ n \ l).
Lemma NoDup\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      NoDup \ l \rightarrow NoDup \ (drop \ n \ l).
Lemma NoDup_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      NoDup \ l \rightarrow NoDup \ (filter \ p \ l).
Lemma NoDup\_partition:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l1 \ l2 : list \ A),
      partition p \mid l = (l1, l2) \rightarrow NoDup \mid l \leftrightarrow NoDup \mid l1 \land NoDup \mid l2.
Lemma NoDup\_takeWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      NoDup \ l \rightarrow NoDup \ (takeWhile \ p \ l).
Lemma NoDup\_drop\,While:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      NoDup \ l \rightarrow NoDup \ (drop While \ p \ l).
Lemma NoDup\_zip:
   \forall (A B : \mathsf{Type}) (la : list A) (lb : list B),
      NoDup\ la \wedge NoDup\ lb \rightarrow NoDup\ (zip\ la\ lb).
Lemma NoDup\_zip\_conv:
   \exists (A B : \mathsf{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 : \mathsf{Type}) (f : A \to option B) (l : list A),
      NoDup\ l \land \neg\ NoDup\ (pmap\ f\ l).
Lemma NoDup\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      NoDup (intersperse x \ l) \rightarrow length \ l \leq 2.
Lemma NoDup\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      \neg NoDup l \leftrightarrow
      \exists (x : A) (l1 \ l2 \ l3 : list \ A),
         l = l1 ++ x :: l2 ++ x :: l3.
```

9.5.4 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 : \mathsf{Type}, \neg Dup \ (@nil \ A).
Lemma Dup\_cons:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Dup(x::l) \leftrightarrow elem(x) l \vee Dup(l)
Lemma Dup\_singl:
   \forall (A : \mathsf{Type}) (x : A), \neg Dup [x].
Lemma Dup\_cons\_inv:
   \forall (A : \mathsf{Type}) (h : A) (t : \mathit{list} A),
       \neg Dup (h :: t) \rightarrow \neg elem \ h \ t \land \neg Dup \ t.
Lemma Dup\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      Dup \ l \leftrightarrow
      \exists (x : A) (l1 \ l2 \ l3 : list \ A),
          l = l1 ++ x :: l2 ++ x :: l3.
Lemma Dup_-NoDup:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      \neg Dup \ l \leftrightarrow NoDup \ l.
Lemma Dup\_length:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Dup l \to 2 < length l.
Lemma Dup\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Dup\ (snoc\ x\ l) \leftrightarrow Dup\ l \lor elem\ x\ l.
Lemma Dup_-app_-l:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Dup \ l1 \rightarrow Dup \ (l1 ++ l2).
Lemma Dup\_app\_r:
```

```
\forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       Dup \ l2 \rightarrow Dup \ (l1 ++ l2).
Lemma Dup\_app\_both:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       elem \ x \ l1 \rightarrow elem \ x \ l2 \rightarrow Dup \ (l1 \ ++ \ l2).
Lemma Dup_{-}app:
   \forall (A : \mathsf{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 : \mathsf{Type}) (l : \mathit{list} \ A),
       Dup (rev \ l) \leftrightarrow Dup \ l.
Lemma Dup_{-}map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A),
       Dup \ l \rightarrow Dup \ (map \ f \ l).
Lemma Dup_{-}map_{-}conv:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : list A),
      (\forall x y : A, f x = f y \rightarrow x = y) \rightarrow
          Dup\ (map\ f\ l) \to Dup\ l.
Lemma Dup\_join:
   \forall (A : \mathsf{Type}) (ll : list (list A)),
       Dup\ (join\ ll) \rightarrow
      (\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 : \mathsf{Type}) (n : nat) (x : A),
       Dup\ (replicate\ n\ x) \to 2 \le n.
Lemma Dup_nth:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
       Dup \ l \leftrightarrow
       \exists (x : A) (n1 \ n2 : nat),
          n1 < n2 \wedge nth \ n1 \ l = Some \ x \wedge nth \ n2 \ l = Some \ x.
Lemma Dup\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       Dup\ (take\ n\ l) \to Dup\ l.
Lemma Dup\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       Dup (drop \ n \ l) \rightarrow Dup \ l.
      TODO: Dup dla insert i replace *)
```

```
TODO: findIndex, takeWhile, dropWhile dla replace *)
Lemma Dup_{-}filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Dup (filter \ p \ l) \rightarrow Dup \ l.
Lemma Dup\_filter\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Dup \ l \rightarrow
         Dup (filter p l) \vee
         Dup (filter (fun x : A \Rightarrow negb (p x)) l).
Lemma Dup_partition:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l \ l1 \ l2 : list \ A),
      partition p \mid l = (l1, l2) \rightarrow Dup \mid l \leftrightarrow Dup \mid l1 \lor Dup \mid l2.
Lemma Dup_{-}takeWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Dup\ (takeWhile\ p\ l) \to Dup\ l.
Lemma Dup\_dropWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Dup\ (drop\ While\ p\ l) \to Dup\ l.
Lemma Dup\_takeWhile\_dropWhile\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Dup \ l \rightarrow
         Dup (takeWhile p l) \lor
         Dup (drop While p l) \lor
         \exists x: A,
            elem x (takeWhile p l) \land elem x (dropWhile p l).
Lemma Dup\_span:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ b \ e : list \ A),
      span \ p \ l = Some \ (b, x, e) \rightarrow
         Dup \ l \leftrightarrow Dup \ b \lor Dup \ e \lor elem \ x \ b \lor elem \ x \ e \lor
            \exists y: A, elem y b \land elem y e.
      TODO: NoDup, Rep *)
Lemma Dup_-zip:
  \forall (A B : \mathsf{Type}) (la : list A) (lb : list B),
      Dup\ (zip\ la\ lb) \rightarrow Dup\ la\ \land\ Dup\ lb.
Lemma Dup\_zip\_conv:
   \forall (A B : \mathsf{Type}) (la : list A) (lb : list B),
      \neg Dup \ la \land \neg Dup \ lb \rightarrow \neg Dup \ (zip \ la \ lb).
Lemma Dup_{-}pmap:
   \exists (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (l : list \ A),
```

```
\begin{array}{c} Dup \ l \wedge \neg \ Dup \ (pmap \ f \ l). \\ \\ \text{Lemma} \ Dup\_intersperse : \\ \\ \forall \ (A: \texttt{Type}) \ (x: A) \ (l: \textit{list } A), \\ \\ Dup \ (intersperse \ x \ l) \rightarrow 2 \leq \textit{length } l. \end{array}
```

9.5.5 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:
   \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       elem \ x \ l \rightarrow Rep \ x \ 1 \ l.
Lemma Rep_elem:
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l : list A),
       1 \leq n \rightarrow Rep \ x \ n \ l \rightarrow elem \ x \ l.
Lemma Dup_{-}Rep :
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
       Dup \ l \rightarrow \exists \ x : A, Rep \ x \ 2 \ l.
Lemma Rep_{-}Dup :
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l : list A),
       2 \leq n \rightarrow Rep \ x \ n \ l \rightarrow Dup \ l.
Lemma Rep_{-}le:
   \forall (A : \mathsf{Type}) (x : A) (n \ m : nat) (l : list \ A),
      n \leq m \rightarrow Rep \ x \ m \ l \rightarrow Rep \ x \ n \ l.
Lemma Rep_-S_-inv:
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l : list A),
      Rep \ x \ (S \ n) \ l \rightarrow Rep \ x \ n \ l.
```

```
Lemma Rep_length:
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l : list A),
      Rep \ x \ n \ l \rightarrow n \leq length \ l.
Lemma Rep\_S\_snoc:
  \forall (A : \mathsf{Type}) (x : A) (n : nat) (l : list A),
      Rep \ x \ n \ l \rightarrow Rep \ x \ (S \ n) \ (snoc \ x \ l).
Lemma Rep\_snoc:
   \forall (A : \mathsf{Type}) (x \ y : A) (n : nat) (l : list A),
      Rep \ x \ n \ l \rightarrow Rep \ x \ n \ (snoc \ y \ l).
Lemma Rep\_app\_l:
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l1 \ l2 : list \ A),
      Rep \ x \ n \ l1 \rightarrow Rep \ x \ n \ (l1 ++ l2).
Lemma Rep\_app\_r:
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l1 \ l2 : list \ A),
      Rep \ x \ n \ l2 \rightarrow Rep \ x \ n \ (l1 ++ l2).
Lemma Rep\_app:
   \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (x : A) (n : nat) (l : list A),
      Rep \ x \ n \ (rev \ l) \leftrightarrow Rep \ x \ n \ l.
Lemma Rep_{-}map:
   \forall (A B : \mathsf{Type}) (f : A \to 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 : \mathsf{Type}) (f : A \to 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 : \mathsf{Type}) (x : A) (n : nat),
      Rep x n (replicate n x).
Lemma Rep\_replicate\_general:
   \forall (A : \mathsf{Type}) (x : A) (n \ m : nat),
```

 $n \leq m \rightarrow Rep \ x \ n \ (replicate \ m \ x).$

```
Lemma Rep_{-}take:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A) (n \ m : \mathit{nat}),
      Rep \ x \ n \ (take \ m \ l) \rightarrow Rep \ x \ n \ l.
Lemma Rep_{-}drop:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} \ A) (n \ m : \mathit{nat}),
      Rep \ x \ n \ (drop \ m \ l) \rightarrow Rep \ x \ n \ l.
Lemma Rep_filter:
   \forall (A : \mathsf{Type}) (p : A \to 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 : \mathsf{Type}) (p : A \to 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 : \mathsf{Type}) (p : A \to 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 : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A) (n : nat),
      Rep \ x \ n \ (takeWhile \ p \ l) \rightarrow Rep \ x \ n \ l.
Lemma Rep_dropWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l : list A) (n : nat),
      Rep \ x \ n \ (drop While \ p \ l) \rightarrow Rep \ x \ n \ l.
Lemma Rep_zip:
   \forall (A B : \mathsf{Type}) (a : A) (b : B) (la : list A) (lb : list B) (n : nat),
      Rep\ (a,\ b)\ n\ (zip\ la\ lb) \to Rep\ a\ n\ la\ \land Rep\ b\ n\ lb.
Lemma Rep\_intersperse:
   \forall (A : \mathsf{Type}) (x \ y : A) (n : nat) (l : list A),
      Rep \ x \ n \ (intersperse \ y \ l) \leftrightarrow
      Rep \ x \ n \ l \lor x = y \land Rep \ x \ (S \ n - length \ l) \ l.
```

9.5.6 Exists

Zaimplementuj induktywny predykat Exists. Exists P l zachodzi, gdy lista l zawiera taki element, który spełnia predykat P.

```
 \begin{array}{c} \mathtt{Lemma} \ Exists\_spec : \\ \forall \ (A : \mathtt{Type}) \ (P : A \to \mathtt{Prop}) \ (l : \mathit{list} \ A), \\ Exists \ P \ l \leftrightarrow \exists \ x : A, \ \mathit{elem} \ x \ l \land P \ x. \\ \\ \mathtt{Lemma} \ Exists\_\mathit{nil} : \\ \forall \ (A : \mathtt{Type}) \ (P : A \to \mathtt{Prop}), \\ Exists \ P \ [] \leftrightarrow \mathit{False}. \end{array}
```

```
Lemma Exists\_cons:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (h : A) (t : \mathit{list} A),
      Exists P(h :: t) \leftrightarrow P(h \lor Exists P(t))
Lemma Exists\_length:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       Exists P \mid l \rightarrow 1 \leq length \mid l.
Lemma Exists\_snoc:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (l : \mathit{list} A),
      Exists P (snoc x l) \leftrightarrow Exists P l \lor P x.
Lemma Exists\_app:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Exists P (l1 ++ l2) \leftrightarrow Exists P l1 \lor Exists P l2.
Lemma Exists\_rev:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       Exists P (rev l) \leftrightarrow Exists P l.
Lemma Exists\_map:
   \forall (A B : \mathsf{Type}) (P : B \to \mathsf{Prop}) (f : A \to B) (l : \mathit{list} A),
       Exists P \ (map \ f \ l) \rightarrow Exists \ (fun \ x : A \Rightarrow P \ (f \ x)) \ l.
Lemma Exists\_join:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (ll : list (list A)),
       Exists P (join ll) \leftrightarrow
      Exists (fun l: list A \Rightarrow Exists P l) ll.
Lemma Exists\_replicate:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A),
      Exists P (replicate n x) \leftrightarrow 1 \leq n \land P x.
Lemma Exists\_nth:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
      Exists P \ l \leftrightarrow
      \exists (n : nat) (x : A), nth \ n \ l = Some \ x \land P \ x.
Lemma Exists\_remove:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
      Exists P \ l \rightarrow
      match remove \ n \ l with
             | None \Rightarrow True
             | Some (x, l') \Rightarrow \neg P x \rightarrow Exists P l'
      end.
Lemma Exists\_take:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
      Exists P (take n l) \rightarrow Exists P l.
```

Lemma $Exists_drop$:

```
\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
      Exists P (drop n l) \rightarrow Exists P l.
Lemma Exists\_take\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
      Exists P \mid l \rightarrow Exists \mid P \mid (take \mid n \mid l) \lor Exists \mid P \mid (drop \mid n \mid l).
Lemma Exists\_cycle:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l : list A),
       Exists P (cycle n l) \leftrightarrow Exists P l.
Lemma Exists\_splitAt:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l1 \ l2 : list \ A) (n : nat) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
          Exists P \ l \leftrightarrow P \ x \lor Exists \ P \ l1 \lor Exists \ P \ l2.
Lemma Exists\_insert:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}) (x : A),
      Exists P (insert l n x) \leftrightarrow P x \lor Exists P l.
Lemma Exists\_replace:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
          Exists P l' \leftrightarrow
          Exists P (take n l) \vee P x \vee Exists P (drop (S n) l).
Lemma Exists\_filter:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Exists P (filter p l) \rightarrow Exists P l.
Lemma Exists\_filter\_conv:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      Exists P l \rightarrow
          Exists P (filter p l) \vee
          Exists P (filter (fun x : A \Rightarrow negb (p x)) l).
Lemma Exists\_filter\_compat:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = false) \rightarrow \neg Exists P (filter p l).
Lemma Exists\_partition:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l \ l1 \ l2 : \mathit{list} \ A),
      partition p \ l = (l1, l2) \rightarrow
          Exists P \ l \leftrightarrow Exists \ P \ l1 \lor Exists \ P \ l2.
Lemma Exists\_takeWhile:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Exists P (takeWhile p l) \rightarrow Exists P l.
```

Lemma $Exists_takeWhile_compat$:

```
\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = false) \rightarrow \neg Exists P (takeWhile p l).
Lemma Exists\_drop While:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      Exists P (drop While p l) \rightarrow Exists P l.
Lemma Exists\_takeWhile\_dropWhile:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      Exists P \mid l \rightarrow Exists \mid P \mid (takeWhile \mid p \mid l) \lor Exists \mid P \mid (dropWhile \mid p \mid l).
Lemma Exists\_span:
      (A: \mathsf{Type}) \ (P: A \to \mathsf{Prop}) \ (p: A \to 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
             Exists P \ l \leftrightarrow Exists \ P \ b \lor P \ x \lor Exists \ P \ e.
Lemma Exists\_interesting:
   \forall (A B : \mathsf{Type}) (P : A \times B \to \mathsf{Prop}) (la : list A) (hb : B) (tb : list B),
       Exists (fun a:A\Rightarrow Exists (fun b:B\Rightarrow P(a,b)) tb) la\rightarrow
      Exists (fun a: A \Rightarrow Exists (fun b: B \Rightarrow P(a, b)) (hb :: tb)) la.
Lemma Exists\_zip:
   \forall (A B : \mathsf{Type}) (P : A \times B \to \mathsf{Prop}) (la : list A) (lb : list B),
      Exists P (zip la lb) \rightarrow
          Exists (fun a: A \Rightarrow Exists (fun b: B \Rightarrow P(a, b)) lb) la.
Lemma Exists\_pmap:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (P : B \to \mathsf{Prop}) (l : list A),
      Exists P (pmap \ f \ l) \leftrightarrow
          Exists (fun x: A \Rightarrow \text{match } f \ x \text{ with } | \ Some \ b \Rightarrow P \ b \ | \ \_ \Rightarrow False \ \text{end}) \ l.
Lemma Exists\_intersperse:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (l : \mathit{list} A),
      Exists P (intersperse x \ l) \leftrightarrow
      Exists P \mid l \lor (P \mid x \land 2 \leq length \mid l).
```

9.5.7 *Forall*

Zaimplementuj induktywny predykat Forall. Forall P l jest spełniony, gdy każdy element listy l spełnia predykat P.

```
Lemma Forall\_spec:
```

```
\forall (A: \mathtt{Type}) (P: A \rightarrow \mathtt{Prop}) (l: \mathit{list}\ A), Forall\ P\ l \leftrightarrow \forall\ x: A, \mathit{elem}\ x\ l \rightarrow P\ x.
```

Lemma $Forall_nil$:

$$\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}),$$

```
Forall P [] \leftrightarrow True.
Lemma Forall\_cons:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (h : A) (t : \mathit{list} A),
       Forall P(h :: t) \leftrightarrow P(h \land Forall P(t))
Lemma Forall\_snoc:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (l : \mathit{list} A),
       Forall P (snoc x l) \leftrightarrow Forall P l \land P x.
Lemma Forall\_app:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
       Forall P (l1 ++ l2) \leftrightarrow Forall P l1 \land Forall P l2.
Lemma Forall\_rev:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       Forall P (rev l) \leftrightarrow Forall P l.
Lemma Forall\_map:
   \forall (A B : \mathsf{Type}) (P : B \to \mathsf{Prop}) (f : A \to B) (l : \mathit{list} A),
       Forall P (map f l) \rightarrow Forall (fun x : A \Rightarrow P (f x)) l.
Lemma Forall\_join:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (ll : list (list A)),
       Forall P (join ll) \leftrightarrow Forall (fun l : list A \Rightarrow Forall P l) ll.
Lemma Forall\_replicate:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A),
       Forall P (replicate n x) \leftrightarrow n = 0 \lor P x.
Lemma Forall\_nth:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       Forall P \mid l \leftrightarrow \forall n : nat, n < length \mid l \rightarrow l
          \exists x : A, nth \ n \ l = Some \ x \land P \ x.
Lemma Forall\_remove:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
       Forall P \ l \rightarrow
      match remove n l with
              | None \Rightarrow True
              | Some (x, l') \Rightarrow Forall P l'
       end.
Lemma Forall\_take:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
       Forall P \mid l \rightarrow Forall \mid P \mid (take \mid n \mid l).
Lemma Forall\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
```

Forall $P \mid l \rightarrow Forall \mid P \mid (drop \mid n \mid l)$.

```
Lemma Forall\_take\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
       Forall P (take n l) \rightarrow Forall P (drop n l) \rightarrow Forall P l.
Lemma Forall\_splitAt:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l1 \ l2 : list \ A) (n : nat) (x : A),
       splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
          Forall P \ l \leftrightarrow P \ x \wedge Forall \ P \ l1 \wedge Forall \ P \ l2.
Lemma Forall\_insert:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}) (x : A),
          Forall P (insert l n x) \leftrightarrow P x \land Forall P l.
Lemma Forall\_replace:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow
          Forall P l' \leftrightarrow
          Forall P (take n l) \wedge P x \wedge Forall P (drop (S n) l).
Lemma Forall\_filter:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Forall P \mid l \rightarrow Forall \mid P \mid (filter \mid p \mid l).
Lemma Forall\_filter\_conv:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Forall P (filter p \mid l) \rightarrow
       Forall P (filter (fun x : A \Rightarrow negb (p x)) l) \rightarrow
          Forall P l.
Lemma Forall\_filter\_compat:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow Forall P (filter p l).
Lemma Forall\_partition:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l \ l1 \ l2 : list \ A),
      partition p \ l = (l1, l2) \rightarrow
          Forall P \ l \leftrightarrow Forall \ P \ l1 \land Forall \ P \ l2.
Lemma Forall\_takeWhile:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Forall P \mid l \rightarrow Forall \mid P \mid (takeWhile \mid p \mid l).
Lemma Forall\_takeWhile\_compat:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P \ x \leftrightarrow p \ x = true) \rightarrow Forall \ P \ (takeWhile \ p \ l).
Lemma Forall\_drop While:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Forall P \mid l \rightarrow Forall \mid P \mid (drop While \mid p \mid l).
```

Lemma $Forall_takeWhile_dropWhile$:

```
\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
       Forall P (takeWhile p l) \rightarrow Forall P (dropWhile p l) \rightarrow Forall P l.
Lemma Forall\_span:
      (A: \mathsf{Type}) \ (P: A \to \mathsf{Prop}) \ (p: A \to 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 : \mathsf{Type}) \ (PA : A \to \mathsf{Prop}) \ (PB : B \to \mathsf{Prop})
   (la: list A) (lb: list B),
       Forall PA la \rightarrow Forall PB \ lb \rightarrow
          Forall (fun '(a, b) \Rightarrow PA a \land PB b) (zip la lb).
Lemma Forall\_pmap:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (P : B \to \mathsf{Prop}) (l : list A),
       For all\ (	extsf{fun}\ x:A\Rightarrow 	extsf{match}\ f\ x\ 	ext{with}\ |\ Some\ b\Rightarrow P\ b\ |\ \_\Rightarrow False\ 	ext{end})\ l\rightarrow
          Forall P (pmap f l).
Lemma Forall\_intersperse:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (l : \mathit{list} A),
      Forall P (intersperse x \ l) \leftrightarrow
       Forall P \mid l \land (2 \leq length \mid l \rightarrow P \mid x).
Lemma Forall\_impl:
   \forall (A : \mathsf{Type}) (P \ Q : A \to \mathsf{Prop}) (l : \mathit{list} \ A),
      (\forall x: A, P x \rightarrow Q x) \rightarrow
          Forall P l \rightarrow Forall Q l.
Lemma Forall\_Exists:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       Forall P \mid l \rightarrow \neg Exists (fun x : A \Rightarrow \neg P \mid x) l.
Lemma Exists\_Forall:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       Exists P \mid l \rightarrow \neg Forall (fun x : A \Rightarrow \neg P \mid x) l.
```

9.5.8 At Least

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: \mathsf{Type}) (P: A \to \mathsf{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 : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (h : A) (t : list A),
      AtLeast\ P\ (S\ n)\ (h::t) \to AtLeast\ P\ n\ t.
Lemma AtLeast\_length:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l : list A),
       AtLeast\ P\ n\ l \rightarrow n < length\ l.
Lemma AtLeast\_snoc:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A) (l : list A),
       AtLeast\ P\ n\ l \rightarrow AtLeast\ P\ n\ (snoc\ x\ l).
Lemma AtLeast\_S\_snoc:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A) (l : list A),
       AtLeast\ P\ n\ l \rightarrow P\ x \rightarrow AtLeast\ P\ (S\ n)\ (snoc\ x\ l).
Lemma AtLeast\_Exists:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       AtLeast\ P\ 1\ l \leftrightarrow Exists\ P\ l.
Lemma AtLeast\_Forall:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
       AtLeast\ P\ (length\ l)\ l \leftrightarrow Forall\ P\ l.
Lemma AtLeast\_Rep:
   \forall (A : \mathsf{Type}) (x : A) (n : nat) (l : list A),
      AtLeast (fun y: A \Rightarrow x = y) n \ l \leftrightarrow Rep \ x \ n \ l.
Lemma AtLeast\_app\_l:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l1 \ l2 : list \ A),
       AtLeast\ P\ n\ l1 \rightarrow AtLeast\ P\ n\ (l1\ ++\ l2).
Lemma AtLeast\_app\_r:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l1 \ l2 : list \ A),
       AtLeast\ P\ n\ l2 \rightarrow AtLeast\ P\ n\ (l1\ ++\ l2).
Lemma AtLeast\_plus\_app:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n1 \ n2 : nat) (l1 \ l2 : list \ A),
       AtLeast\ P\ n1\ l1\ 
ightarrow\ AtLeast\ P\ n2\ l2\ 
ightarrow
          AtLeast P (n1 + n2) (l1 ++ l2).
Lemma AtLeast\_app\_conv:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l1 \ l2 : list \ A),
      AtLeast\ P\ n\ (l1\ ++\ l2) \rightarrow
          \exists \ \mathit{n1} \ \mathit{n2} : \mathit{nat}, \mathit{AtLeast} \ \mathit{P} \ \mathit{n1} \ \mathit{l1} \land \mathit{AtLeast} \ \mathit{P} \ \mathit{n2} \ \mathit{l2} \land \mathit{n} = \mathit{n1} + \mathit{n2}.
Lemma AtLeast\_app:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l1 \ l2 : list \ A),
```

 $AtLeast\ P\ n\ (l1\ ++\ l2) \leftrightarrow$

```
\exists n1 \ n2 : nat,
         AtLeast\ P\ n1\ l1\ \land\ AtLeast\ P\ n2\ l2\ \land\ n=n1+n2.
Lemma AtLeast\_app\_comm:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l1 \ l2 : list \ A),
      AtLeast\ P\ n\ (l1\ ++\ l2) \rightarrow AtLeast\ P\ n\ (l2\ ++\ l1).
Lemma AtLeast\_rev:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l : list A),
      AtLeast\ P\ n\ (rev\ l) \leftrightarrow AtLeast\ P\ n\ l.
Lemma AtLeast\_map:
   \forall (A B : \mathsf{Type}) (P : B \to \mathsf{Prop}) (f : A \to B) (n : nat) (l : list A),
      AtLeast (fun \ x : A \Rightarrow P \ (f \ x)) \ n \ l \rightarrow
         AtLeast P n (map f l).
Lemma AtLeast\_map\_conv:
   \forall (A B : \mathsf{Type}) (P : B \to \mathsf{Prop}) (f : A \to B) (n : nat) (l : list A),
      (\forall x \ y : A, f \ x = f \ y \rightarrow x = y) \rightarrow AtLeast \ P \ n \ (map \ f \ l) \rightarrow
         AtLeast (fun x: A \Rightarrow P(f x)) n l.
Lemma AtLeast\_replicate:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A),
      n \neq 0 \rightarrow P \ x \rightarrow AtLeast \ P \ n \ (replicate \ n \ x).
Lemma AtLeast\_replicate\_conv:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n \ m : nat) (x : A),
      AtLeast\ P\ m\ (replicate\ n\ x) \to m = 0 \lor m \le n \land P\ x.
Lemma AtLeast\_remove:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (m : \mathit{nat}),
      AtLeast\ P\ m\ l \rightarrow \forall\ n:\ nat,
         match remove n l with
                | None \Rightarrow True
                | Some (\_, l') \Rightarrow AtLeast P (m-1) l'
         end.
Lemma AtLeast\_take:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n \ m : \mathit{nat}),
      AtLeast\ P\ m\ (take\ n\ l) \rightarrow AtLeast\ P\ m\ l.
Lemma AtLeast\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n \ m : \mathit{nat}),
      AtLeast\ P\ m\ (drop\ n\ l) \rightarrow AtLeast\ P\ m\ l.
Lemma AtLeast\_take\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n \ m : nat) (l : list A),
      AtLeast\ P\ n\ l \rightarrow
      \exists n1 \ n2 : nat,
         AtLeast\ P\ n1\ (take\ m\ l) \land AtLeast\ P\ n2\ (drop\ m\ l) \land n=n1+n2.
```

```
Lemma AtLeast\_splitAt:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l1 \ l2 : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         \forall m : nat,
            AtLeast\ P\ m\ l \rightarrow
            \exists m1 \ mx \ m2 : nat,
                AtLeast\ P\ m1\ l1\ \land\ AtLeast\ P\ mx\ [x]\ \land\ AtLeast\ P\ m2\ l2\ \land
                m1 + mx + m2 = m.
Lemma AtLeast\_insert:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n : \mathit{nat}),
      AtLeast\ P\ n\ l \rightarrow \forall\ (m:nat)\ (x:A),
         AtLeast\ P\ n\ (insert\ l\ m\ x).
Lemma AtLeast\_replace:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l' : list \ A) (n \ m : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow AtLeast \ P \ m \ l \rightarrow
         AtLeast P (m-1) l'.
Lemma AtLeast\_replace':
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l' : \mathit{list} \ A) (n \ m : \mathit{nat}) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow AtLeast \ P \ m \ l \rightarrow P \ x \rightarrow
         AtLeast P m l'.
Lemma AtLeast\_replace\_conv:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l' : list \ A) (n \ m : nat) (x : A),
      replace l \ n \ x = Some \ l' \rightarrow AtLeast \ P \ m \ l' \rightarrow AtLeast \ P \ (m-1) \ l.
Lemma AtLeast\_replace\_conv':
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l' : \mathit{list} \ A) (n \ m : \mathit{nat}) (x \ y : A),
      replace l \ n \ x = Some \ l' \rightarrow nth \ n \ l = Some \ y \rightarrow P \ y \rightarrow
         AtLeast\ P\ m\ l' \rightarrow AtLeast\ P\ m\ l.
     TODO: Exactly, AtMost dla replace *)
Lemma AtLeast\_filter:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (n : nat) (l : list A),
      AtLeast\ P\ n\ (filter\ p\ l) \rightarrow AtLeast\ P\ n\ l.
Lemma AtLeast\_filter\_compat\_true:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
         AtLeast\ P\ (length\ (filter\ p\ l))\ (filter\ p\ l).
Lemma AtLeast\_filter\_compat\_false:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (n : nat) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = false) \rightarrow
         AtLeast\ P\ n\ (filter\ p\ l) \rightarrow n = 0.
Lemma AtLeast\_takeWhile:
```

```
\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (n : nat) (l : list A),
      AtLeast\ P\ n\ (takeWhile\ p\ l) \rightarrow AtLeast\ P\ n\ l.
Lemma AtLeast\_drop While:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (n : nat) (l : list A),
      AtLeast\ P\ n\ (drop\ While\ p\ l) \to AtLeast\ P\ n\ l.
Lemma AtLeast\_takeWhile\_true:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
         AtLeast\ P\ (length\ (takeWhile\ p\ l))\ (takeWhile\ p\ l).
Lemma AtLeast\_takeWhile\_false:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (n : nat) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = false) \rightarrow
         AtLeast\ P\ n\ (takeWhile\ p\ l) \rightarrow n = 0.
Lemma AtLeast\_dropWhile\_true:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A) (n : nat),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
         AtLeast\ P\ n\ l \rightarrow AtLeast\ P\ (n - length\ (takeWhile\ p\ l))\ (dropWhile\ p\ l).
Lemma AtLeast\_dropWhile\_false:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A) (n : nat),
      (\forall x: A, P x \leftrightarrow p x = false) \rightarrow
         AtLeast\ P\ n\ l \rightarrow AtLeast\ P\ n\ (drop\ While\ p\ l).
Lemma AtLeast\_zip :
   \forall (A \ B : \mathsf{Type}) \ (PA : A \to \mathsf{Prop}) \ (PB : B \to \mathsf{Prop})
   (la: list A) (lb: list B) (n: nat),
      AtLeast (fun '(a, b) \Rightarrow PA \ a \land PB \ b) \ n \ (zip \ la \ lb) \rightarrow
         AtLeast\ PA\ n\ la \land AtLeast\ PB\ n\ lb.
Lemma AtLeast\_findIndices:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A) (n : nat),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
         AtLeast\ P\ n\ l \rightarrow n < length\ (findIndices\ p\ l).
Lemma AtLeast\_1\_elem:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A),
      AtLeast\ P\ 1\ l \leftrightarrow \exists\ x: A,\ elem\ x\ l \wedge P\ x.
Lemma AtLeast\_intersperse:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (l : \mathit{list} A),
      P x \rightarrow AtLeast \ P \ (length \ l - 1) \ (intersperse \ x \ l).
Lemma AtLeast\_intersperse':
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (n : nat) (l : list A),
      AtLeast\ P\ n\ l \rightarrow P\ x \rightarrow
         AtLeast\ P\ (n+(length\ l-1))\ (intersperse\ x\ l).
```

```
\forall (A: \mathsf{Type}) \ (P: A \to \mathsf{Prop}) \ (x: A) \ (n: nat) \ (l: list \ A), AtLeast \ P \ n \ l \to \neg \ P \ x \to AtLeast \ P \ n \ (intersperse \ x \ l).
```

9.5.9 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:
```

$$\forall (A: \mathsf{Type}) (P: A \to \mathsf{Prop}) (x: A) (l: \mathit{list}\ A), Exactly\ P\ 0\ (x: l) \leftrightarrow \neg\ P\ x \land Exactly\ P\ 0\ l.$$

Lemma $Exactly_S_cons$:

$$\forall \ (A: \mathtt{Type}) \ (P: A \rightarrow \mathtt{Prop}) \ (n: nat) \ (x: A) \ (l: list \ A), \\ Exactly \ P \ (S \ n) \ (x:: l) \leftrightarrow \\ P \ x \wedge Exactly \ P \ n \ l \lor \neg P \ x \wedge Exactly \ P \ (S \ n) \ l.$$

Lemma $Exactly_AtLeast$:

$$\forall (A: \mathtt{Type}) \ (P: A \rightarrow \mathtt{Prop}) \ (n: nat) \ (l: list \ A),$$
 Exactly $P \ n \ l \rightarrow AtLeast \ P \ n \ l.$

Lemma $Exactly_eq$:

$$\forall \; (A: \texttt{Type}) \; (P: A \rightarrow \texttt{Prop}) \; (n \; m: nat) \; (l: list \; A), \\ Exactly \; P \; n \; l \rightarrow Exactly \; P \; m \; l \rightarrow n = m.$$

Lemma $Exactly_length$:

$$\forall \ (A: \mathtt{Type}) \ (P: A \to \mathtt{Prop}) \ (n: nat) \ (l: \mathit{list} \ A), \\ \mathit{Exactly} \ P \ n \ l \to n \leq \mathit{length} \ l.$$

Lemma $Exactly_snoc$:

$$\forall (A: \mathtt{Type}) \ (P: A \rightarrow \mathtt{Prop}) \ (n: nat) \ (x: A) \ (l: list \ A),$$

 $Exactly \ P \ n \ l \rightarrow \neg \ P \ x \rightarrow Exactly \ P \ n \ (snoc \ x \ l).$

Lemma $Exactly_S_snoc$:

$$\forall (A: \mathtt{Type}) \ (P: A \rightarrow \mathtt{Prop}) \ (n: nat) \ (x: A) \ (l: list \ A),$$

 $Exactly \ P \ n \ l \rightarrow P \ x \rightarrow Exactly \ P \ (S \ n) \ (snoc \ x \ l).$

Lemma $Exactly_app$:

$$\forall \ (A: {\tt Type}) \ (P: A \to {\tt Prop}) \ (n1 \ n2: nat) \ (l1 \ l2: list \ A), \ Exactly \ P \ n1 \ l1 \to Exactly \ P \ n2 \ l2 \to Exactly \ P \ (n1 \ + n2) \ (l1 \ ++ \ l2).$$

Lemma $Exactly_app_conv$:

```
orall \; (A: {\tt Type}) \; (P:A 	o {\tt Prop}) \; (n:nat) \; (l1\; l2: list\; A), \ Exactly \; P \; n \; (l1\; ++\; l2) 	o \ \exists \; n1\; n2: nat, \ Exactly \; P \; n1\; l1\; \wedge \; Exactly \; P \; n2\; l2\; \wedge \; n = n1\; +\; n2.
```

Lemma $Exactly_app_comm$:

```
\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l1 \ l2 : list \ A),
      Exactly P n (l1 ++ l2) \rightarrow Exactly P n (l2 ++ l1).
Lemma Exactly\_rev:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l : list A),
      Exactly P n (rev \ l) \leftrightarrow Exactly P n l.
Lemma Exactly\_map:
   \forall (A B : \mathsf{Type}) (P : B \to \mathsf{Prop}) (f : A \to B) (n : nat) (l : list A),
      (\forall x \ y : A, f \ x = f \ y \rightarrow x = y) \rightarrow
      Exactly (fun x: A \Rightarrow P(f x)) n l \leftrightarrow
         Exactly P n (map f l).
Lemma Exactly\_replicate:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A),
      P x \rightarrow Exactly P n (replicate n x).
Lemma Exactly\_replicate\_conv:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A),
      Exactly P n (replicate n x) \rightarrow n = 0 \lor P x.
Lemma Exactly_replicate':
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n \ m : nat) (x : A),
      Exactly P n (replicate m x) \leftrightarrow
      n = 0 \land m = 0 \lor
      n = 0 \land \neg P x \lor
      n = m \wedge P x.
Lemma Exactly\_take:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n \ \mathit{m1} \ \mathit{m2} : \mathit{nat}),
      Exactly P m1 (take n l) \rightarrow Exactly P m2 l \rightarrow m1 \leq m2.
Lemma Exactly\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n \ \mathit{m1} \ \mathit{m2} : \mathit{nat}),
      Exactly P m1 (drop n l) \rightarrow Exactly P m2 l \rightarrow m1 < m2.
Lemma Exactly\_take\_drop:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l : \mathit{list} A) (n \ m : nat),
      Exactly P \ n \ l \rightarrow \exists \ n1 \ n2 : nat,
          n = n1 + n2 \wedge Exactly P n1 \ (take m l) \wedge Exactly P n2 \ (drop m l).
Lemma Exactly\_splitAt:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l \ l1 \ l2 : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      splitAt \ n \ l = Some \ (l1, x, l2) \rightarrow
         \forall m : nat,
             Exactly P \ m \ l \leftrightarrow
             \exists m1 \ mx \ m2 : nat,
                Exactly P m1 l1 \wedge Exactly P mx [x] \wedge Exactly P m2 l2 \wedge
                m1 + mx + m2 = m.
```

```
Lemma Exactly\_filter:
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
         Exactly P (length (filter p l)) (filter p l).
Lemma Exactly\_takeWhile:
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
         Exactly P (length (takeWhile p l)) (takeWhile p l).
Lemma Exactly\_drop While:
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (p : A \to bool) (n : nat) (l : list A),
      (\forall x: A, P x \leftrightarrow p x = true) \rightarrow
      Exactly P \ n \ l \rightarrow
         Exactly P (n - length (take While p l)) (drop While p l).
Lemma Exactly\_span:
  \forall
      (A: \mathsf{Type}) \ (P: A \to \mathsf{Prop}) \ (p: A \to bool)
      (n:nat)(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
            Exactly P \ n \ l \leftrightarrow
            \exists n1 \ n2 : nat,
               Exactly P n1 b \wedge Exactly P n2 e \wedge
               if p x then S (n1 + n2) = n else n1 + n2 = n.
      TODO: span i AtLeast, AtMost *)
Lemma Exactly\_intersperse:
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (n : nat) (l : list A),
      Exactly P \ n \ l \rightarrow P \ x \rightarrow
         Exactly P(n + (length \ l - 1)) (intersperse x \ l).
Lemma Exactly\_intersperse':
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (n : nat) (l : list A),
      Exactly P \ n \ l \rightarrow \neg P \ x \rightarrow
         Exactly P n (intersperse x l).
              AtMost
9.5.10
Lemma AtMost_{-}\theta:
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (x : A) (l : \mathit{list} A),
      AtMost\ P\ 0\ (x::l) \leftrightarrow \neg\ P\ x \land AtMost\ P\ 0\ l.
Lemma AtMost\_nil:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat),
```

```
AtMost\ P\ n\ [] \leftrightarrow True.
Lemma AtMost\_le:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l : list A),
      AtMost\ P\ n\ l \to \forall\ m:\ nat,\ n \leq m \to AtMost\ P\ m\ l.
Lemma AtMost\_S\_cons:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A) (l : list A),
      AtMost\ P\ (S\ n)\ (x::l) \leftrightarrow
      (\tilde{P} \times AtMost P (S n) l) \vee AtMost P n l.
Lemma AtMost\_S\_snoc:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{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 : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (x : A) (l : list A),
      AtMost P n l \rightarrow \neg P x \rightarrow AtMost <math>P n (snoc x l).
Lemma AtMost\_S:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (n : nat) (l : list A),
      AtMost\ P\ n\ l \to AtMost\ P\ (S\ n)\ l.
```

9.6 Relacje między listami

9.6.1 Listy jako termy

```
Lemma Sublist\_length: \forall (A: \mathtt{Type}) \ (l1\ l2: list\ A), Sublist\ l1\ l2 \rightarrow length\ l1 < length\ l2. Lemma Sublist\_cons\_l:
```

```
\forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A), \neg \mathit{Sublist} (x :: l) l.
Lemma Sublist\_cons\_l':
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Sublist (x :: l1) l2 \rightarrow Sublist l1 l2.
Lemma Sublist\_nil\_cons:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A), \mathit{Sublist} [] (x :: l).
Lemma Sublist\_irrefl:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A), \neg \mathit{Sublist} l \ \mathit{l}.
Lemma Sublist\_antisym:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow \neg Sublist l2 l1.
Lemma Sublist\_trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Sublist l1\ l2 \rightarrow Sublist\ l2\ l3 \rightarrow Sublist\ l1\ l3.
Lemma Sublist\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow Sublist (snoc x l1) (snoc x l2).
Lemma Sublist\_snoc\_inv:
   \forall (A : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      l1 \neq \parallel \rightarrow Sublist \ l2 \ (l1 ++ l2).
Lemma Sublist\_app\_l':
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Sublist l1 l2 \rightarrow Sublist l1 (l3 ++ l2).
Lemma Sublist\_app\_r:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
       Sublist l1\ l2 \rightarrow Sublist\ (l1\ ++\ l3)\ (l2\ ++\ l3).
Lemma Sublist\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : \mathit{list} \ A),
       Sublist l1 l2 \rightarrow Sublist (map f l1) (map f l2).
Lemma Sublist\_join:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
       \neg elem [] l2 \rightarrow Sublist \ l1 \ l2 \rightarrow Sublist \ (join \ l1) \ (join \ l2).
Lemma Sublist\_replicate:
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      Sublist (replicate n x) (replicate m x) \leftrightarrow n < m.
```

Lemma $Sublist_replicate'$:

```
\forall (A : \mathsf{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 : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x : A),
      Sublist (iterate f n x) (iterate f m x) \rightarrow False.
Lemma Sublist\_iterate':
   \forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x \ y : A),
      Sublist (iterate f n x) (iterate f m y) \rightarrow False.
Lemma Sublist\_tail:
   \forall (A : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : \mathit{list} \ A),
      Sublist l1 l2 \rightarrow l1 = [] \lor last l1 = last l2.
     TODO: insert, remove, take *)
Lemma Sublist\_spec:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \leftrightarrow
      \exists n : nat,
         n < length \ l2 \wedge l1 = drop \ (S \ n) \ l2.
Lemma Sublist\_drop\_r:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow \forall n : nat, Sublist (drop n l1) l2.
Lemma Sublist\_drop:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 \ l2 \rightarrow \forall \ n : nat,
         n < length \ l2 \rightarrow Sublist \ (drop \ n \ l1) \ (drop \ n \ l2).
Lemma Sublist\_replace:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow \forall (l1' l2' : list A) (n : nat) (x : A),
         replace l1 \ n \ x = Some \ l1' \rightarrow replace \ l2 \ (n + length \ l1) \ x = Some \ l2' \rightarrow
            Sublist 11' 12'.
Lemma Sublist\_zip:
   \exists (A B : \mathsf{Type}) (la1 \ la2 : \mathit{list} \ A) (\mathit{lb1} \ \mathit{lb2} : \mathit{list} \ A),
      Sublist la1 la2 \wedge Sublist lb1 lb2 \wedge
         \neg Sublist (zip la1 lb1) (zip la2 lb2).
     TODO: zipWith, unzip, unzipWith *)
```

```
Lemma Sublist\_any\_false:
   \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
      Sublist l1\ l2 \rightarrow any\ p\ l2 = false \rightarrow any\ p\ l1 = false.
Lemma Sublist\_any\_true:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist l1\ l2 \rightarrow any\ p\ l1 = true \rightarrow any\ p\ l2 = true.
     TODO: Sublist_all *)
Lemma Sublist\_findLast:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist 11 12 \rightarrow \forall x : A,
        findLast \ p \ l1 = Some \ x \rightarrow findLast \ p \ l2 = Some \ x.
Lemma Sublist\_removeLast:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist 11 12 \rightarrow
        match removeLast p l1, removeLast p l2 with
                None, None \Rightarrow True
                None, Some (x, l2') \Rightarrow l1 = l2' \vee Sublist l1 l2'
                x, None \Rightarrow False
                Some (x, l1'), Some (y, l2') \Rightarrow x = y \land Sublist l1' l2'
         end.
Lemma Sublist\_findIndex:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist 11 12 \rightarrow \forall n : nat,
        findIndex \ p \ l1 = Some \ n \rightarrow \exists \ m : nat,
            findIndex \ p \ l2 = Some \ m.
Lemma Sublist\_filter:
   \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Sublist (filter p l1) (filter p l2).
Lemma Sublist\_findIndices:
   \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Sublist (findIndices p l1) (findIndices p l2).
Lemma Sublist\_takeWhile:
   \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Sublist (takeWhile p l1) (takeWhile p l2).
Lemma Sublist\_drop While:
   \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Sublist (drop While p l1) (drop While p l2).
Lemma Sublist\_pmap:
   \exists (A B : \mathsf{Type}) (f : A \to option B) (l1 l2 : list A),
      Sublist l1 l2 \land \neg Sublist (pmap f l1) (pmap f l2).
```

```
Lemma Sublist\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Sublist l1\ l2 \rightarrow Sublist\ (intersperse\ x\ l1)\ (intersperse\ x\ l2).
Lemma Sublist\_elem:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow \forall x : A, elem x l1 \rightarrow elem x l2.
Lemma Sublist_In:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow \forall x : A, In x l1 \rightarrow In x l2.
Lemma Sublist\_NoDup:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1\ l2 \rightarrow NoDup\ l2 \rightarrow NoDup\ l1.
Lemma Sublist_Dup:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 \ l2 \rightarrow Dup \ l1 \rightarrow Dup \ l2.
Lemma Sublist\_Rep:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Sublist l1 \ l2 \rightarrow \forall \ n : nat, Rep \ x \ n \ l1 \rightarrow Rep \ x \ n \ l2.
Lemma Sublist\_Exists:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Sublist l1\ l2 \rightarrow Exists\ P\ l1 \rightarrow Exists\ P\ l2.
Lemma Sublist\_Forall:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : \mathit{list} \ A),
      Sublist 11 12 \rightarrow Forall P 12 \rightarrow Forall P 11.
Lemma Sublist\_AtLeast:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 \ l2 \rightarrow \forall \ (P : A \rightarrow Prop) \ (n : nat),
          AtLeast\ P\ n\ l1 \rightarrow AtLeast\ P\ n\ l2.
{\tt Lemma} \ Sublist\_AtMost :
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 \ l2 \rightarrow \forall \ (P : A \rightarrow Prop) \ (n : nat),
          AtMost\ P\ n\ l2 \rightarrow AtMost\ P\ n\ l1.
9.6.2
             Prefiksy
Inductive Prefix \{A : \mathsf{Type}\} : list A \to list A \to \mathsf{Prop} :=
        Prefix\_nil : \forall l : list A, Prefix [] l
       | Prefix\_cons :
             \forall (x : A) (l1 \ l2 : list \ A),
                Prefix l1 l2 \rightarrow Prefix (x :: l1) (x :: l2).
```

```
Lemma Prefix\_spec:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       Prefix l1\ l2 \leftrightarrow \exists\ l3:\ list\ A,\ l2=l1+l3.
Lemma Prefix\_refl:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A), \mathit{Prefix} \ l \ l.
Lemma Prefix\_trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Prefix\ l1\ l2 \rightarrow Prefix\ l2\ l3 \rightarrow Prefix\ l1\ l3.
Lemma Prefix\_wasym:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       Prefix l1\ l2 \rightarrow Prefix\ l2\ l1 \rightarrow l1 = l2.
      TODO: null *)
Lemma Prefix\_length:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       Prefix l1\ l2 \rightarrow length\ l1 \leq length\ l2.
Lemma Prefix\_snoc:
   \exists (A : \mathsf{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 : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
       Prefix l1 l2 \rightarrow Prefix (l3 ++ l1) (l3 ++ l2).
Lemma Prefix\_app\_r:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
       Prefix l1 l2 \rightarrow Prefix l1 (l2 ++ l3).
Lemma Prefix\_rev\_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
       Prefix (rev l) l \rightarrow l = rev l.
Lemma Prefix\_rev\_r:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
       Prefix l (rev \ l) \rightarrow l = rev \ l.
Lemma Prefix_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : \mathit{list} \ A),
       Prefix l1\ l2 \rightarrow Prefix\ (map\ f\ l1)\ (map\ f\ l2).
Lemma Prefix\_join:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
       Prefix l1\ l2 \rightarrow Prefix\ (join\ l1)\ (join\ l2).
Lemma Prefix\_replicate:
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
       Prefix (replicate n x) (replicate m x) \leftrightarrow n \leq m.
```

```
Lemma Prefix\_replicate\_inv\_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      Prefix l (replicate n x) \rightarrow
         \exists m : nat, m \leq n \land l = replicate \ m \ x.
Lemma Prefix\_replicate\_inv\_r:
   \exists (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      Prefix (replicate n x) l \wedge l
         \neg \exists m : nat, n \leq m \land l = replicate m x.
Lemma Prefix\_replicate':
   \forall (A : \mathsf{Type}) (n : nat) (x \ y : A),
      Prefix (replicate n x) (replicate n y) \leftrightarrow n = 0 \lor x = y.
Lemma Prefix\_replicate'':
   \forall (A : \mathsf{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:
   \forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x : A),
      Prefix (iterate f n x) (iterate f m x) \leftrightarrow n \leq m.
Lemma Prefix\_insert:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall\ (n:nat)\ (x:A),
         n \leq length \ l1 \rightarrow Prefix \ (insert \ l1 \ n \ x) \ (insert \ l2 \ n \ x).
Lemma Prefix\_replace:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall\ (n:nat)\ (x:A),
         match replace l1 \ n \ x, replace l2 \ n \ x with
                | Some l1', Some l2' \Rightarrow Prefix l1' l2'
                | \_, \_ \Rightarrow True
         end.
Lemma insert\_length\_ge:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      length l \leq n \rightarrow insert \ l \ n \ x = snoc \ x \ l.
Lemma Prefix\_insert':
   \exists (A : \mathsf{Type}) (l1 \ l2 : \mathit{list} \ A),
      Prefix l1 l2 \land
      \exists (n : nat) (x : A),
         length l1 < n \land \neg Prefix (insert l1 \ n \ x) (insert l2 \ n \ x).
Lemma Prefix_take_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}), \mathit{Prefix} (\mathit{take} \ n \ l) \ l.
Lemma Prefix\_take:
```

```
\forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow \forall n : nat, Prefix (take n l1) (take n l2).
Lemma Prefix\_drop:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall n : nat, Prefix (drop n l1) (drop n l2).
Lemma Prefix\_zip:
   \forall (A B : \mathsf{Type}) (la1 \ la2 : \mathit{list} \ A) (\mathit{lb1} \ \mathit{lb2} : \mathit{list} \ B),
      Prefix\ la1\ la2 \rightarrow Prefix\ lb1\ lb2 \rightarrow
          Prefix (zip \ la1 \ lb1) (zip \ la2 \ lb2).
      TODO: unzip, zipWith, unzipWith *)
Lemma Prefix\_any\_false:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow any\ p\ l2 = false \rightarrow any\ p\ l1 = false.
Lemma Prefix\_any:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow any\ p\ l1 = true \rightarrow any\ p\ l2 = true.
Lemma Prefix\_all\_false:
   \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
      \textit{Prefix l1 l2} \rightarrow \textit{all p l1} = \textit{false} \rightarrow \textit{all p l2} = \textit{false}.
Lemma Prefix\_all\_true:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow all\ p\ l2 = true \rightarrow all\ p\ l1 = true.
Lemma Prefix\_find\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow find\ p\ l2 = None \rightarrow find\ p\ l1 = None.
Lemma Prefix\_find\_Some:
   \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow \forall x : A,
         find p \ l1 = Some \ x \rightarrow find \ p \ l2 = Some \ x.
Lemma Prefix\_findIndex\_None:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow findIndex\ p\ l2 = None \rightarrow findIndex\ p\ l1 = None.
Lemma Prefix\_findIndex\_Some:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall i : nat,
         findIndex \ p \ l1 = Some \ i \rightarrow findIndex \ p \ l2 = Some \ i.
Lemma Prefix\_count:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
```

Prefix $l1\ l2 \rightarrow count\ p\ l1 \leq count\ p\ l2$.

```
Lemma Prefix_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow Prefix (filter p l1) (filter p l2).
Lemma Prefix\_findIndices:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow Prefix\ (findIndices\ p\ l1)\ (findIndices\ p\ l2).
Lemma Prefix_take While:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Prefix (take While p \mid l) l.
Lemma Prefix_takeWhile_pres:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow Prefix (takeWhile p l1) (takeWhile p l2).
Lemma Prefix\_drop While:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow Prefix (drop While p l1) (drop While p l2).
     TODO: findLast, removeFirst i removeLast *)
Lemma Prefix_pmap:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l1 l2 : list A),
      Prefix l1\ l2 \rightarrow Prefix\ (pmap\ f\ l1)\ (pmap\ f\ l2).
Lemma Prefix\_intersperse:
  \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow Prefix (intersperse x l1) (intersperse x l2).
     TODO: groupBy *)
Lemma Prefix\_elem:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall x : A, elem\ x\ l1 \rightarrow elem\ x\ l2.
Lemma Prefix\_elem\_conv:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall x: A, \neg elem\ x\ l2 \rightarrow \neg elem\ x\ l1.
    TODO: In *)
Lemma Prefix_NoDup:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow NoDup\ l2 \rightarrow NoDup\ l1.
Lemma Prefix_Dup:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow Dup\ l1 \rightarrow Dup\ l2.
     TODO: Rep *)
Lemma Prefix\_Exists:
```

```
\forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow Exists\ P\ l1 \rightarrow Exists\ P\ l2.
Lemma Prefix\_Forall:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow Forall\ P\ l2 \rightarrow Forall\ P\ l1.
Lemma Prefix\_AtLeast:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1 l2 \rightarrow \forall (P : A \rightarrow Prop) (n : nat),
          AtLeast\ P\ n\ l1 \rightarrow AtLeast\ P\ n\ l2.
Lemma Prefix\_AtMost:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow \forall (P:A \rightarrow Prop)(n:nat),
          AtMost\ P\ n\ l2 \rightarrow AtMost\ P\ n\ l1.
      TODO: Exactly - raczej nic z tego *)
Lemma Sublist\_Prefix:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Prefix l1 l2.
Lemma Prefix\_spec':
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \leftrightarrow \exists\ n:\ nat,\ l1=take\ n\ l2.
9.6.3
             Sufiksy
Inductive Suffix \{A : \mathsf{Type}\} : list A \to list A \to \mathsf{Prop} :=
      | Suffix\_refl :
             \forall l : list A, Suffix l l
      \mid Suffix\_cons:
             \forall (x : A) (l1 \ l2 : list \ A),
                Suffix l1 l2 \rightarrow Suffix l1 (x :: l2).
Lemma Suffix\_spec:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Suffix l1 l2 \leftrightarrow \exists l3 : list A, l2 = l3 ++ l1.
Lemma Suffix\_cons\_inv:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Suffix (x :: l1) l2 \rightarrow Suffix l1 l2.
Lemma Suffix\_trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Suffix l1\ l2 \rightarrow Suffix\ l2\ l3 \rightarrow Suffix\ l1\ l3.
Lemma Suffix\_wasym:
```

```
\forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Suffix l1\ l2 \rightarrow Suffix\ l2\ l1 \rightarrow l1 = l2.
Lemma Suffix_nil_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A), \mathit{Suffix} [] l.
Lemma Suffix\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Suffix l1 l2 \rightarrow Suffix (snoc x l1) (snoc x l2).
Lemma Suffix\_Sublist:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Suffix l1 l2 \leftrightarrow Sublist l1 l2 \lor l1 = l2.
Lemma Prefix_Suffix:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1 l2 \leftrightarrow Suffix (rev l1) (rev l2).
9.6.4
             Listy jako ciągi
Inductive Subseq \{A : \mathsf{Type}\} : list A \to list A \to \mathsf{Prop} :=
      | Subseq_nil :
            \forall l : list A, Subseq [] l
      | Subseq\_cons :
             \forall (x : A) (l1 \ l2 : list A),
                Subseq l1 l2 \rightarrow Subseq (x :: l1) (x :: l2)
      | Subseq\_skip :
            \forall (x : A) (l1 \ l2 : list \ A),
                Subseq l1\ l2 \rightarrow Subseq\ l1\ (x::l2).
Lemma Subseq\_refl:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A), \mathit{Subseq} \ l \ l.
Lemma Subseq\_cons\_inv:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Subseq\ (x::l1)\ (x::l2) \rightarrow Subseq\ l1\ l2.
Lemma Subseq\_cons\_inv\_l:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Subseq (x :: l1) l2 \rightarrow Subseq l1 l2.
Lemma Subseq\_isEmpty\_l:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      isEmpty\ l1 = true \rightarrow Subseq\ l1\ l2.
Lemma Subseq_nil_r:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Subseq l \mid \mid \rightarrow l = \mid \mid.
{\tt Lemma} \ Subseq\_length:
```

```
\forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow length\ l1 \leq length\ l2.
Lemma Subseq\_trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Subseq l1 l2 \rightarrow Subseq l2 l3 \rightarrow Subseq l1 l3.
Lemma Subseq\_cons\_l\_app:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Subseq(x::l1) l2 \rightarrow
         \exists \ l21 \ l22 : list \ A, \ l2 = l21 \ ++ \ x :: \ l22 \land Subseq \ l1 \ l22.
Lemma Subseq\_wasym:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow Subseq\ l2\ l1 \rightarrow l1 = l2.
Lemma Subseq\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow Subseq\ l1\ (snoc\ x\ l2).
Lemma Subseq\_snoc':
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow Subseq (snoc x l1) (snoc x l2).
Lemma Subseq\_app\_l:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Subseq l1 l2 \rightarrow Subseq l1 (l3 ++ l2).
Lemma Subseq\_app\_r:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Subseq l1 l2 \rightarrow Subseq l1 (l2 ++ l3).
Lemma Subseq\_app\_l':
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Subseq l1\ l2 \rightarrow Subseq\ (l3\ ++\ l1)\ (l3\ ++\ l2).
Lemma Subseq\_app\_r':
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Subseq l1\ l2 \rightarrow Subseq\ (l1\ ++\ l3)\ (l2\ ++\ l3).
Lemma Subseq\_app\_not\_comm:
   \exists (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Subseq l1 (l2 ++ l3) \land \neg Subseq l1 (l3 ++ l2).
Lemma Subseq\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : \mathit{list} \ A),
      Subseq l1\ l2 \rightarrow Subseq\ (map\ f\ l1)\ (map\ f\ l2).
Lemma Subseq\_join:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
```

Subseq $l1 l2 \rightarrow Subseq (join l1) (join l2)$.

```
Lemma Subseq\_replicate:
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      Subseq (replicate n x) (replicate m x) \leftrightarrow n \leq m.
Lemma Subseq_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x : A),
      Subseq (iterate f n x) (iterate f m x) \leftrightarrow n \leq m.
Lemma Subseq_tail:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow \forall l1' l2' : list A,
          tail \ l1 = Some \ l1' \rightarrow tail \ l2 = Some \ l2' \rightarrow Subseq \ l1' \ l2'.
Lemma Subseq\_uncons:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 \ l2 \rightarrow \forall \ (h1 \ h2 : A) \ (t1 \ t2 : list \ A),
         uncons \ l1 = Some \ (h1, \ t1) \rightarrow uncons \ l2 = Some \ (h2, \ t2) \rightarrow
            h1 = h2 \vee Subseq l1 t2.
Lemma Subseq_init:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow \forall l1' l2' : list A,
         init \ l1 = Some \ l1' \rightarrow init \ l2 = Some \ l2' \rightarrow Subseq \ l1' \ l2'.
Lemma Subseq\_unsnoc:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 \ l2 \rightarrow \forall \ (h1 \ h2 : A) \ (t1 \ t2 : list \ A),
         unsnoc\ l1 = Some\ (h1,\ t1) \rightarrow unsnoc\ l2 = Some\ (h2,\ t2) \rightarrow
            h1 = h2 \vee Subseq l1 t2.
Lemma Subseq_nth:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat) (x : A),
      Subseq 11 12 \rightarrow nth n 11 = Some x \rightarrow
         \exists m : nat, nth \ m \ l2 = Some \ x \land n \leq m.
Lemma Subseq\_insert:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat) (x : A),
      Subseq l1\ l2 \rightarrow Subseq\ l1\ (insert\ l2\ n\ x).
Lemma Subseq\_replace:
   \forall (A : \mathsf{Type}) (l1 \ l1' \ l2 : list \ A) (n : nat) (x : A),
      Subseq l1 l2 \rightarrow \text{replace } l1 \ n \ x = Some \ l1' \rightarrow
         \exists (m : nat) (l2' : list A),
            replace l2 m x = Some l2' \wedge Subseq l1' l2'.
Lemma Subseq\_remove':
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n : nat),
      Subseq l1 l2 \rightarrow Subseq (remove' n l1) l2.
```

Lemma $Subseq_take$:

```
\forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      Subseq (take \ n \ l) \ l.
Lemma Subseq\_drop:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      Subseq (drop \ n \ l) \ l.
Lemma Subseq\_zip:
   \exists (A B : \mathsf{Type}) (la1 \ la2 : list \ A) (lb1 \ lb2 : list \ B),
      Subseq la1 la2 \wedge Subseq lb1 lb2 \wedge
         \neg Subseq (zip la1 lb1) (zip la2 lb2).
Lemma Subseq\_all:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow bool\_le (all p l2) (all p l1).
Lemma Subseq\_any:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow bool\_le (any p l1) (any p l2).
Lemma Subseq_all':
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow all\ p\ l2 = true \rightarrow all\ p\ l1 = true.
Lemma Subseq\_any':
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow any\ p\ l2 = false \rightarrow any\ p\ l1 = false.
Lemma Subseq\_findIndex:
   \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A) (n \ m : nat),
      Subseq l1 l2 \land findIndex p l1 = Some n \land
     findIndex \ p \ l2 = Some \ m \wedge m < n.
Lemma Subseq\_count:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow count\ p\ l1 \leq count\ p\ l2.
Lemma Subseq\_filter:
  \forall (A : \mathsf{Type}) (p : A \rightarrow bool) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow Subseq (filter p l1) (filter p l2).
Lemma Subseq\_filter':
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Subseq (filter p \mid l) l.
Lemma Subseq\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Subseq (takeWhile \ p \ l) \ l.
Lemma Subseq_takeWhile':
```

 $\exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),$

```
Subseq l1 l2 \land \neg Subseq (takeWhile p l1) (takeWhile p l2).
Lemma Subseq\_dropWhile:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Subseq (drop While \ p \ l) \ l.
Lemma Subseq\_dropWhile':
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow Subseq\ (dropWhile\ p\ l1)\ (dropWhile\ p\ l2).
Lemma Subseq\_pmap:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (l1 l2 : list A),
      Subseq l1\ l2 \rightarrow Subseq\ (pmap\ f\ l1)\ (pmap\ f\ l2).
Lemma Subseq\_map\_pmap:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (l1 l2 : list A),
      Subseq l1\ l2 \rightarrow Subseq\ (map\ Some\ (pmap\ f\ l1))\ (map\ f\ l2).
Lemma Subseq\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow Subseq (intersperse x\ l1) (intersperse x\ l2).
    TODO *) Fixpoint intercalate \; \{A: {	t Type}\} \; (l: \mathit{list} \; A) \; (\mathit{ll}: \mathit{list} \; (\mathit{list} \; A)) : \mathit{list} \; A :=
match l, ll with
      | [], \_ \Rightarrow join \ ll
      | _{-}, | | \Rightarrow l
      | h :: t, l :: ls \Rightarrow h :: l ++ intercalate t ls
end.
Lemma Subseq\_spec:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq 11 l2 \rightarrow
         \exists ll : list (list A),
            l2 = intercalate l1 ll.
Lemma Subseq_In:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq 11 l2 \rightarrow
         \forall x: A, In x l1 \rightarrow In x l2.
Lemma Subseq\_NoDup:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow NoDup \ l2 \rightarrow NoDup \ l1.
Lemma Subseq_Dup:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq 11 l2 \rightarrow Dup l1 \rightarrow Dup l2.
Lemma Subseq_Rep:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
```

```
Subseq 11 l2 \rightarrow \forall (x : A) (n : nat),
          Rep \ x \ n \ l1 \rightarrow Rep \ x \ n \ l2.
Lemma Subseq\_Exists:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : \mathit{list} \ A),
      Subseq l1 l2 \rightarrow Exists \ P \ l1 \rightarrow Exists \ P \ l2.
Lemma Subseq\_Forall:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Subseq l1 l2 \rightarrow Forall \ P \ l2 \rightarrow Forall \ P \ l1.
Lemma Subseq_AtLeast:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq 11 12 \rightarrow \forall (P : A \rightarrow Prop) (n : nat),
          AtLeast\ P\ n\ l1 \rightarrow AtLeast\ P\ n\ l2.
Lemma Subseq\_AtMost:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq 11 12 \rightarrow \forall (P : A \rightarrow Prop) (n : nat),
          AtMost\ P\ n\ l2 \rightarrow AtMost\ P\ n\ l1.
Lemma Sublist\_Subseq:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \rightarrow Subseq l1 l2.
Lemma Subseq\_Sublist:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 l2 \land \neg Sublist l1 l2.
Lemma Prefix\_Subseq:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow Subseq\ l1\ l2.
Lemma Subseq\_Prefix:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq 11 l2 \land \neg Prefix l1 l2.
```

9.6.5 Zawieranie

```
Definition Incl \{A : \mathsf{Type}\} (l1 \ l2 : list \ A) : \mathsf{Prop} := \forall \ x : A, \ elem \ x \ l1 \rightarrow 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.

```
\forall (A : \mathsf{Type}) (l : \mathit{list} A),
       Incl\ l\ [] \rightarrow l = [].
Lemma Incl\_cons:
   \forall (A : \mathsf{Type}) (h : A) (t1 \ t2 : \mathit{list} \ A),
       Incl\ t1\ t2 \rightarrow Incl\ (h::t1)\ (h::t2).
Lemma Incl\_cons':
   \forall (A : \mathsf{Type}) (h : A) (t : \mathit{list} A),
       Incl\ t\ (h::t).
Lemma Incl\_cons'':
   \forall (A : \mathsf{Type}) (h : A) (t \ l : \mathit{list} \ A),
       Incl\ l\ t \rightarrow Incl\ l\ (h::t).
Lemma Incl\_cons\_conv:
   \exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       Incl\ (x :: l1)\ (x :: l2) \land \neg Incl\ l1\ l2.
Lemma Incl\_refl:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A), \mathit{Incl} l \mathit{l}.
Lemma Incl_{-}trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
       Incl \ l1 \ l2 \rightarrow Incl \ l2 \ l3 \rightarrow Incl \ l1 \ l3.
(* TODO *)
Lemma Incl\_length:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       \neg Dup \ l1 \rightarrow Incl \ l1 \ l2 \rightarrow length \ l1 \leq length \ l2.
Lemma Incl\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       Incl l (snoc x l).
Lemma Incl\_singl\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
       Incl [x] (snoc x l).
Lemma Incl\_snoc\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
       Incl l1 l2 \rightarrow Incl (snoc \ x \ l1) (snoc \ x \ l2).
Lemma Incl\_app\_rl:
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A),
       Incl \ l \ l2 \rightarrow Incl \ l \ (l1 \ ++ \ l2).
{\tt Lemma}\ Incl\_app\_rr\ :
   \forall (A : \mathsf{Type}) (l \ l1 \ l2 : \mathit{list} \ A),
       Incl \ l \ l1 \rightarrow Incl \ l \ (l1 ++ l2).
```

Lemma $Incl_app_l$:

```
\forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
       Incl\ (l1\ ++\ l2)\ l3 \leftrightarrow Incl\ l1\ l3 \land Incl\ l2\ l3.
Lemma Incl\_app\_sym:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l : \mathit{list} \ A),
       Incl\ (l1\ ++\ l2)\ l \rightarrow Incl\ (l2\ ++\ l1)\ l.
Lemma Incl\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A), \mathit{Incl} (\mathit{rev} \ l) \ l.
Lemma Incl_{-}map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : list A),
       Incl l1 l2 \rightarrow Incl (map f l1) (map f l2).
Lemma Incl_{-join}:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
       Incl l1 l2 \rightarrow Incl (join l1) (join l2).
Lemma Incl\_elem\_join:
   \forall (A : \mathsf{Type}) (ll : \mathit{list} (\mathit{list} A)) (l : \mathit{list} A),
       elem l \ ll \rightarrow Incl \ l \ (join \ ll).
Lemma Incl\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A) (l : list A),
       elem \ x \ l \rightarrow Incl \ (replicate \ n \ x) \ l.
Lemma Incl\_replicate':
   \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
       n \neq 0 \rightarrow Incl (replicate \ m \ x) (replicate \ n \ x).
Lemma Incl_nth:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
       Incl l1 l2 \leftrightarrow
      \forall (n1:nat) (x:A), nth \ n1 \ l1 = Some \ x \rightarrow
          \exists n2 : nat, nth n2 l2 = Some x.
Lemma Incl\_remove:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      match remove n l with
              | None \Rightarrow True
             | Some (\_, l') \Rightarrow Incl l' l
       end.
Lemma Incl\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       Incl\ (take\ n\ l)\ l.
Lemma Incl\_drop:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
       Incl\ (drop\ n\ l)\ l.
```

```
Lemma Incl\_insert:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (n \ m : nat) (x : A),
      Incl l1 l2 \rightarrow Incl (insert \ l1 \ n \ x) (insert \ l2 \ m \ x).
Lemma Incl\_replace:
  \forall (A : \mathsf{Type}) (l1 \ l1' \ l2 : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
      Incl l1 l2 \rightarrow replace l1 n x = Some \ l1' \rightarrow
         \exists (m : nat) (l2' : list A),
            replace l2 m x = Some l2' \wedge Incl l1' l2'.
Lemma Incl\_splitAt:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
     match splitAt n l with
            | None \Rightarrow True
            | Some (l1, \_, l2) \Rightarrow Incl l1 l \wedge Incl l2 l
      end.
Lemma Incl_{-}filter:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Incl (filter p l) l.
Lemma Incl_filter\_conv:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Incl l (filter p l) \leftrightarrow filter p l = l.
Lemma Incl\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Incl\ (takeWhile\ p\ l)\ l.
Lemma Incl\_takeWhile\_conv:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Incl l (takeWhile p l) \leftrightarrow takeWhile p l = l.
Lemma Incl\_drop While:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      Incl\ (drop While\ p\ l)\ l.
Lemma Incl\_span:
  \forall (A : \mathsf{Type}) (p : A \to bool) (x : A) (l \ b \ e : list \ A),
      span \ p \ l = Some \ (b, x, e) \rightarrow
         Incl b l \wedge elem \ x \ l \wedge Incl \ e \ l.
     TODO: span i Sublist, palindromy *)
Lemma Incl_pmap:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l : list A),
      Incl\ (map\ Some\ (pmap\ f\ l))\ (map\ f\ l).
Lemma Incl\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
```

```
Incl l1 (intersperse x l2) \rightarrow Incl l1 (x :: l2).
Lemma Incl\_intersperse\_conv:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      2 \leq length \ l2 \rightarrow Incl \ l1 \ (x :: l2) \rightarrow Incl \ l1 \ (intersperse \ x \ l2).
Lemma Incl_{-}In:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Incl l1 l2 \rightarrow \forall x : A, In x l1 \rightarrow In x l2.
Lemma Incl_NoDup:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Incl l1 l2 \wedge NoDup l2 \wedge \neg NoDup l1.
Lemma Incl_Dup:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Incl l1 l2 \wedge Dup l1 \wedge \neg Dup l2.
Lemma Incl\_Rep:
   \exists (A : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Incl 11 12 \rightarrow \forall P : A \rightarrow Prop,
         Forall P l2 \rightarrow Forall P l1.
Lemma Incl\_AtLeast:
   \exists (A : \mathsf{Type}) (P : A \to \mathsf{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 : \mathsf{Type}) (P : A \to \mathsf{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 : \mathsf{Type}) (P : A \to \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 \ l2 \rightarrow Incl \ l1 \ l2.
Lemma Incl\_Sublist:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
```

Incl l1 $l2 \land \neg Sublist l1 l2$.

```
Lemma Prefix\_Incl:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1\ l2 \rightarrow Incl\ l1\ l2.
Lemma Incl_{-}Prefix:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Incl l1 l2 \land \neg Prefix l1 l2.
Lemma Subseq\_Incl:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1\ l2 \rightarrow Incl\ l1\ l2.
Lemma Incl\_Subseq:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Incl l1 l2 \land \neg Subseq l1 l2.
9.6.6
             Listy jako zbiory
Definition SetEquiv \{A : Type\} (l1 \ l2 : list \ A) : Prop :=
   \forall x: A, elem x l1 \leftrightarrow elem x l2.
Lemma SetEquiv\_Incl:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      SetEquiv \ l1 \ l2 \leftrightarrow Incl \ l1 \ l2 \wedge Incl \ l2 \ l1.
Lemma SetEquiv\_refl:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A), \mathit{SetEquiv} \ l \ l.
Lemma SetEquiv\_sym:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      SetEquiv l1 l2 \leftrightarrow SetEquiv l2 l1.
Lemma SetEquiv\_trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ l2 \ l3 \rightarrow SetEquiv \ l1 \ l3.
Lemma SetEquiv\_nil\_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      SetEquiv [] l \rightarrow l = [].
Lemma SetEquiv\_nil\_r:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      SetEquiv \ l \ [] \rightarrow l = [].
Lemma SetEquiv\_cons:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ (x :: l1) \ (x :: l2).
Lemma SetEquiv\_cons\_conv:
   \exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
```

```
SetEquiv (x :: l1) (x :: l2) \land \neg SetEquiv l1 l2.
Lemma SetEquiv\_cons':
   \exists (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      \neg SetEquiv \ l \ (x :: l).
Lemma SetEquiv\_snoc\_cons:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      SetEquiv (snoc x l) (x :: l).
Lemma SetEquiv\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ (snoc \ x \ l1) \ (snoc \ x \ l2).
Lemma SetEquiv\_app\_comm:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      SetEquiv (l1 ++ l2) (l2 ++ l1).
Lemma SetEquiv\_app\_l:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l : \mathit{list} \ A),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ (l ++ l1) \ (l ++ l2).
Lemma SetEquiv\_app\_r:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l : \mathit{list} \ A),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ (l1 \ ++ \ l) \ (l2 \ ++ \ l).
Lemma SetEquiv\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A), \mathit{SetEquiv} (\mathit{rev} \ l) \ l.
Lemma SetEquiv\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : \mathit{list} \ A),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ (map \ f \ l1) \ (map \ f \ l2).
Lemma SetEquiv\_join:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
      SetEquiv \ l1 \ l2 \rightarrow SetEquiv \ (join \ l1) \ (join \ l2).
Lemma SetEquiv\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      SetEquiv \ (replicate \ n \ x) \ (if \ isZero \ n \ then \ [] \ else \ [x]).
Lemma SetEquiv\_replicate':
  \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      m \neq 0 \rightarrow n \neq 0 \rightarrow SetEquiv (replicate m x) (replicate n x).
(* TODO *) Lemma SetEquiv\_nth :
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      SetEquiv l1 l2 \leftrightarrow
      (\forall n : nat, \exists m : nat, nth \ n \ l1 = nth \ m \ l2) \land
      (\forall n : nat, \exists m : nat, nth m l1 = nth n l2).
```

```
\forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}),
      SetEquiv (take \ n \ l) \ l \leftrightarrow Incl (drop \ n \ l) (take \ n \ l).
Lemma SetEquiv\_drop:
   \forall (A : \mathsf{Type}) (n : nat) (l : list A),
      SetEquiv (drop \ n \ l) \ l \leftrightarrow Incl (take \ n \ l) (drop \ n \ l).
Lemma SetEquiv\_filter:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      SetEquiv (filter p l) l \leftrightarrow all p l = true.
Lemma SetEquiv\_takeWhile:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      SetEquiv (takeWhile p l) l \leftrightarrow all p l = true.
Lemma SetEquiv\_drop While:
   \exists (A : \mathsf{Type}) (p : A \to bool) (l : list A),
      SetEquiv (drop While p l) l \wedge any p l = true.
Lemma SetEquiv\_pmap:
   \exists (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (l : list \ A),
      \neg SetEquiv (map Some (pmap f l)) (map f l).
Lemma SetEquiv\_intersperse:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      2 \leq length \ l \rightarrow SetEquiv \ (intersperse \ x \ l) \ (x :: l).
Lemma SetEquiv\_intersperse\_conv:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      SetEquiv (intersperse \ x \ l) (x :: l) \rightarrow
         elem \ x \ l \lor 2 \le length \ l.
Lemma SetEquiv\_singl:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      SetEquiv [x] l \rightarrow \forall y : A, elem y l \rightarrow y = x.
Lemma SetEquiv\_pres\_intersperse:
   \exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      SetEquiv \ l1 \ l2 \land \neg \ SetEquiv \ (intersperse \ x \ l1) \ (intersperse \ x \ l2).
9.6.7
            Listy jako multizbiory
Require Export Coq. Classes. Setoid Class.
Require Import Coq. Classes. Relation Classes.
Inductive Permutation \{A : \mathsf{Type}\} : list A \to list A \to \mathsf{Prop} :=
      | perm_nil : Permutation [] []
      | perm\_skip : \forall (x : A) (l \ l' : list \ A),
            Permutation l \ l' \rightarrow Permutation (x :: l) (x :: l')
```

```
| perm\_swap : \forall (x \ y : A) \ (l : list \ A),
            Permutation (y :: x :: l) (x :: y :: l)
      \mid perm\_trans : \forall l \ l' \ l'' : list \ A,
            Permutation l \ l' \rightarrow Permutation \ l' \ l'' \rightarrow Permutation \ l \ l''.
Hint Constructors Permutation.
Lemma Permutation\_refl:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
     Permutation 1 l.
Lemma Permutation\_trans:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Permutation 11 12 \rightarrow Permutation 12 13 \rightarrow Permutation 11 13.
Lemma Permutation\_sym:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation 11 12 \rightarrow Permutation 12 11.
Instance Permutation_Equivalence:
   \forall A : \mathsf{Type}, RelationClasses.Equivalence (Permutation (A := A)).
Lemma Permutation\_isEmpty:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1\ l2 \rightarrow isEmpty\ l1 = isEmpty\ l2.
Lemma Permutation\_nil\_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      Permutation [] l \rightarrow l = [].
Lemma Permutation\_nil\_r:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Permutation l \parallel \rightarrow l = \parallel.
Lemma Permutation\_nil\_cons\_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      \neg Permutation [] (x :: l).
Lemma Permutation\_nil\_cons\_r:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      \neg Permutation (x :: l) [].
Lemma Permutation\_nil\_app\_cons\_l:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (x : A),
      \neg Permutation [] (l ++ x :: l').
Instance Permutation\_cons:
  \forall A : \mathsf{Type},
      Proper\ (eq ==> @Permutation\ A ==> @Permutation\ A)\ (@cons\ A).
Lemma Permutation\_ind':
  \forall (A : \mathsf{Type}) (P : list A \to list A \to \mathsf{Prop}),
```

```
P \mid \mid \mid \mid \rightarrow
      (\forall x \ l \ l', Permutation \ l \ l' \rightarrow P \ l \ l' \rightarrow P \ (x :: l) \ (x :: l')) \rightarrow
      (\forall x \ y \ l \ l', Permutation \ l \ l' \rightarrow P \ l \ l' \rightarrow
          P(y::x::l)(x::y::l')) \rightarrow
      (\forall l \ l' \ l'', Permutation \ l \ l' \rightarrow P \ l \ l' \rightarrow Permutation \ l' \ l'' \rightarrow
         P l' l'' \rightarrow P l l'') \rightarrow
      \forall l l', Permutation l l' \rightarrow P l l'.
Inductive Elem \{A : \mathsf{Type}\}\ (x : A) : list \ A \to list \ A \to \mathsf{Prop} :=
      \mid es\_here:
            \forall l : list A, Elem x l (x :: l)
      | es\_there :
            \forall (y:A) (l1 \ l2: list \ A),
                Elem \ x \ l1 \ l2 \rightarrow Elem \ x \ (y :: l1) \ (y :: l2).
Lemma Elem\_spec:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Elem x l1 l2 \rightarrow \exists l21 l22 : list A,
         l2 = l21 ++ x :: l22 \wedge l1 = l21 ++ l22.
Lemma Permutation\_Elem:
   \forall (A : \mathsf{Type}) (x : A) (l \ l' : \mathit{list} \ A),
      Elem x \ l \ l' \rightarrow Permutation \ l' \ (x :: l).
Lemma Elem\_Permutation:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l1' : list A),
      Elem x l1 l1' \rightarrow \forall l2': list A,
          Permutation 11' 12' \rightarrow \exists l2 : list A, Elem x l2 l2'.
Lemma Permutation\_Elems:
   \forall (A : \mathsf{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 11' 12'.
Lemma Permutation\_cons\_inv:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (x : A),
      Permutation (x :: l1) (x :: l2) \rightarrow Permutation l1 l2.
Lemma Permutation\_length:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1\ l2 \rightarrow length\ l1 = length\ l2.
Lemma Permutation\_length':
   \forall A : \mathsf{Type},
      Proper (@Permutation A ==> eq) (@length A).
Lemma Permutation\_length\_1:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
```

```
length l1 = 1 \rightarrow Permutation \ l1 \ l2 \rightarrow l1 = l2.
Lemma Permutation\_singl:
  \forall (A : \mathsf{Type}) (a \ b : A),
      Permutation [a] [b] \rightarrow a = b.
Lemma Permutation\_length\_1\_inv:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Permutation [x] l 	o l = [x].
Lemma Permutation\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Permutation l1\ l2 \rightarrow Permutation\ (snoc\ x\ l1)\ (snoc\ x\ l2).
{\tt Lemma}\ Permutation\_cons\_snoc:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      Permutation (x :: l) (snoc x l).
Lemma Permutation\_cons\_snoc':
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A),
      Permutation (x :: l) (l ++ [x]).
Lemma Permutation\_app\_l:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Permutation l1\ l2 \rightarrow Permutation\ (l3\ ++\ l1)\ (l3\ ++\ l2).
Lemma Permutation\_app\_r:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Permutation \ l1 \ l2 \rightarrow Permutation \ (l1 ++ l3) \ (l2 ++ l3).
Lemma Permutation\_app:
  \forall (A : \mathsf{Type}) (l1 \ l1' \ l2 \ l2' : \mathit{list} \ A),
      Permutation \ l1 \ l1' \rightarrow Permutation \ l2 \ l2' \rightarrow
         Permutation (l1 ++ l2) (l1' ++ l2').
Instance Permutation_app':
  \forall A : \mathsf{Type},
      Proper (@Permutation A ==> @Permutation A ==> @Permutation A) (@app A).
Lemma Permutation\_add\_inside:
   \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 \ l1' \ l2' : \mathit{list} \ A),
      Permutation \ l1 \ l1' \rightarrow Permutation \ l2 \ l2' \rightarrow
        Permutation (l1 ++ x :: l2) (l1' ++ x :: l2').
Lemma Permutation\_cons\_middle:
   \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : list \ A) (x : A),
      Permutation \ l1 \ (l2 ++ l3) \rightarrow Permutation \ (x :: l1) \ (l2 ++ x :: l3).
Lemma Permutation\_middle:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (x : A),
      Permutation (l1 ++ x :: l2) (x :: (l1 ++ l2)).
```

```
Lemma Permutation\_app\_comm:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation (l1 ++ l2) (l2 ++ l1).
Lemma Permutation\_app\_inv\_l:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Permutation (l1 ++ l2) (l1 ++ l3) \rightarrow Permutation l2 l3.
Lemma Permutation\_app\_inv\_r:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Permutation (l1 ++ l3) (l2 ++ l3) \rightarrow Permutation l1 l2.
Lemma Permutation\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Permutation (rev l) l.
Instance Permutation_rev':
  \forall A : \mathsf{Type},
      Proper \ (@Permutation \ A ==> @Permutation \ A) \ (@rev \ A).
Lemma Permutation\_map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : \mathit{list} \ A),
      Permutation l1\ l2 \rightarrow Permutation\ (map\ f\ l1)\ (map\ f\ l2).
Lemma Permutation\_map':
  \forall (A B : \mathsf{Type}) (f : A \to B),
      Morphisms. Proper
        (Morphisms.respectful\ (Permutation\ (A:=A))\ (Permutation\ (A:=B)))
        (map\ f).
Lemma Permutation\_join:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
      Permutation l1 l2 \rightarrow Permutation (join l1) (join l2).
Lemma Permutation\_join\_rev:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
     Permutation (join l1) (join l2) \land \neg Permutation l1 l2.
Lemma Permutation\_replicate:
  \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      Permutation (replicate n x) (replicate m x) \leftrightarrow n = m.
Lemma Permutation\_In:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1\ l2 \rightarrow (\forall\ x:A,\ In\ x\ l1 \leftrightarrow In\ x\ l2).
Lemma Permutation_in:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (x : A),
      Permutation l \ l' \rightarrow In \ x \ l \rightarrow In \ x \ l'.
Lemma Permutation_in':
```

```
\forall A : \mathsf{Type},
     Proper (eq ==> @Permutation A ==> iff) (@In A).
Lemma Permutation\_replicate':
  \forall (A : \mathsf{Type}) (n : nat) (x y : A),
     Permutation (replicate n x) (replicate n y) \leftrightarrow n = 0 \lor x = y.
Lemma Permutation\_iterate:
  \forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x : A),
     Permutation (iterate f n x) (iterate f m x) \leftrightarrow n = m.
Lemma Permutation\_iterate':
  \forall (A : \mathsf{Type}) (f : A \to A) (n : nat) (x y : A),
     Permutation (iterate f n x) (iterate f n y) <math>\rightarrow
        n = 0 \lor \exists k : nat, k < n \land iter f k y = x.
Lemma Permutation\_insert:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
     Permutation (insert l n x) (x :: l).
Lemma Permutation\_take:
  \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
     Permutation \ l1 \ l2 \ \land
        \exists n : nat, \neg Permutation (take n l1) (take n l2).
Lemma Permutation\_drop:
  \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
     Permutation l1 l2 \land
        \exists n : nat, \neg Permutation (drop n l1) (drop n l2).
Lemma Permutation\_cycle:
  \forall (A : \mathsf{Type}) (n : nat) (l : list A),
     Permutation (cycle n l) l.
Lemma Permutation_filter_cycle:
  \forall (A : \mathsf{Type}) (p : A \to bool) (n : nat) (l : list A),
     Permutation (filter p (cycle n l)) (filter p l).
Lemma Permutation\_length\_2\_inv:
  \forall (A : \mathsf{Type}) (x \ y : A) (l : \mathit{list} \ A),
     Permutation [x; y] l \rightarrow l = [x; y] \lor l = [y; x].
Lemma Permutation\_length\_2:
  \forall (A : \mathsf{Type}) (x1 \ x2 \ y1 \ y2 : A),
     Permutation [x1; x2] [y1; y2] \rightarrow
        x1 = y1 \land x2 = y2 \lor x1 = y2 \land x2 = y1.
Lemma Permutation\_zip:
  \exists (A B : \mathsf{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 : \mathsf{Type}) \ (f : A \to B \to 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 : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     Permutation l1 l2 \rightarrow any p l1 = any p l2.
Lemma Permutation\_all:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     Permutation l1\ l2 \rightarrow all\ p\ l1 = all\ p\ l2.
Lemma Permutation\_count:
  \forall (A : \mathsf{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 : \mathsf{Type}) (p : A \to 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:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
     (\forall p: A \rightarrow bool, count \ p \ l1 = count \ p \ l2) \rightarrow Permutation \ l1 \ l2.
Lemma Permutation\_filter:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     Permutation l1\ l2 \rightarrow Permutation\ (filter\ p\ l1)\ (filter\ p\ l2).
Lemma Permutation\_takeWhile:
  \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     Permutation \ l1 \ l2 \ \land
      \neg Permutation (takeWhile p l1) (takeWhile p l2).
Lemma Permutation\_drop While:
  \exists (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     Permutation \ l1 \ l2 \ \land
     \neg Permutation (drop While p l1) (drop While p l2).
Lemma Permutation\_span:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l \ b \ e : list \ A) (x : A),
     span \ p \ l = Some \ (b, x, e) \rightarrow Permutation \ l \ (b ++ x :: e).
```

Lemma $Permutation_removeFirst$:

```
\forall (A : \mathsf{Type}) (p : A \to bool) (l \ l' : list \ A) (x : A),
      removeFirst\ p\ l = Some\ (x,\ l') \rightarrow Permutation\ l\ (x::\ l').
Lemma Permutation\_intersperse\_replicate:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Permutation (intersperse x l) (replicate (length l - 1) x ++ l).
Lemma Permutation\_intersperse:
  \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Permutation (intersperse x l1) (intersperse x l2) \leftrightarrow
      Permutation 11 12.
Lemma Permutation\_pmap:
  \forall (A B : \mathsf{Type}) (f : A \to option B) (l1 l2 : list A),
      Permutation l1\ l2 \rightarrow Permutation\ (pmap\ f\ l1)\ (pmap\ f\ l2).
Lemma Permutation\_elem:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation \ l1 \ l2 \rightarrow
        \forall x : A, elem x l1 \leftrightarrow elem x l2.
Lemma Permutation_replicate':
  \forall (A : \mathsf{Type}) (n \ m : nat) (x \ y : A),
      Permutation (replicate n x) (replicate m y) \leftrightarrow
      n = m \wedge (n \neq 0 \rightarrow m \neq 0 \rightarrow x = y).
Lemma NoDup\_Permutation:
  \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1 l2 \rightarrow NoDup l1 \rightarrow NoDup l2.
Lemma Permutation\_NoDup':
  \forall A : \mathsf{Type},
      Morphisms.Proper
         (Morphisms.respectful\ (Permutation\ (A:=A))\ iff)
         (NoDup\ (A:=A)).
Lemma Permutation\_Dup:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1\ l2 \rightarrow Dup\ l1 \leftrightarrow Dup\ l2.
```

Lemma $Permutation_Rep$:

```
\forall (A : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation \ l1 \ l2 \rightarrow
         \forall P: A \rightarrow \texttt{Prop}, Exists P l1 \leftrightarrow Exists P l2.
Lemma Permutation\_Exists\_conv:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      (\forall P: A \rightarrow Prop, Exists P l1 \leftrightarrow Exists P l2) \land
      \neg Permutation l1 l2.
Lemma Permutation\_Forall:
   \forall (A : \mathsf{Type}) (l1 \ l2 : \mathit{list} \ A),
      Permutation \ l1 \ l2 \rightarrow
         \forall P: A \rightarrow \text{Prop}, Forall P l1 \leftrightarrow Forall P l2.
Lemma Permutation\_AtLeast:
   \forall (A : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation \ l1 \ l2 \rightarrow
         \forall (P:A \rightarrow \texttt{Prop}) (n:nat), AtMost P n l1 \leftrightarrow AtMost P n l2.
Lemma Permutation\_Sublist:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1 l2 \land \neg Sublist l1 l2.
{\tt Lemma}\ Sublist\_Permutation:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Permutation l1 l2.
Lemma Permutation\_Prefix:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation 11 12 \wedge \neg Prefix 11 12.
Lemma Prefix_Permutation:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1 l2 \land \neg Permutation l1 l2.
```

Lemma $Permutation_Subseq$:

```
\exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1 l2 \land \neg Subseq l1 l2.
Lemma Subseq\_Permutation:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Subseq l1 l2 \land \neg Permutation l1 l2.
Lemma Permutation\_Incl:
   \forall (A : \mathsf{Type}) (l1 \ l2 : \mathit{list} \ A),
      Permutation l1 l2 \rightarrow Incl l1 l2.
Lemma Permutation\_SetEquiv:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Permutation l1\ l2 \rightarrow SetEquiv\ l1\ l2.
             Listy jako cykle
9.6.8
Inductive Cycle \{A : \mathsf{Type}\} : list A \to list A \to \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \leftrightarrow
      \exists n : nat, n \leq length \ l1 \land l1 = drop \ n \ l2 ++ take \ n \ l2.
Lemma Cycle\_isEmpty:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow isEmpty\ l1 = isEmpty\ l2.
Lemma Cycle\_nil\_l:
   \forall (A : \mathsf{Type}) (l : \mathit{list} \ A),
      Cycle [] l \rightarrow l = [].
Lemma Cycle\_nil\_r:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Cycle l [] \rightarrow l = [].
Lemma Cycle\_length:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow length\ l1 = length\ l2.
```

Lemma $Cycle_cons$:

```
\exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Cycle (x :: l1) (x :: l2).
Lemma Cycle\_cons\_inv:
   \exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Cycle (x :: l1) (x :: l2) \land \neg Cycle l1 l2.
Lemma Cycle\_snoc:
   \exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Cycle (snoc x l1) (snoc x l2).
Lemma Cycle\_sym:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \rightarrow Cycle l2 l1.
Lemma Cycle\_snoc\_cons:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Cycle (snoc \ x \ l) \ (x :: l).
Lemma Cycle\_cons\_snoc:
  \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Cycle\ (x::l)\ (snoc\ x\ l).
Lemma Cycle\_cons\_snoc':
  \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Cycle l1\ (x::l2) \rightarrow Cycle\ l1\ (snoc\ x\ l2).
Lemma Cycle\_trans:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Cycle l1 l2 \rightarrow Cycle l2 l3 \rightarrow Cycle l1 l3.
Lemma Cycle\_app:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : \mathit{list} \ A),
      Cycle l1 (l2 ++ l3) \rightarrow Cycle l1 (l3 ++ l2).
Lemma Cycle\_rev:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \rightarrow Cycle (rev l1) (rev l2).
Lemma Cycle\_map:
  \forall (A B : \mathsf{Type}) (f : A \to B) (l1 \ l2 : \mathit{list} \ A),
      Cycle l1 l2 \rightarrow Cycle (map f l1) (map f l2).
Lemma Cycle\_join:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list (list A)),
      Cycle l1 l2 \rightarrow Cycle (join l1) (join l2).
Lemma Cycle\_replicate:
  \forall (A : \mathsf{Type}) (n \ m : nat) (x : A),
      Cycle (replicate n x) (replicate m x) \leftrightarrow n = m.
```

Lemma $Cycle_iterate$:

```
\forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x : A),
      Cycle (iterate f n x) (iterate f m x) \leftrightarrow n = m.
Lemma Cycle\_iterate':
  \forall (A : \mathsf{Type}) (f : A \to A) (n \ m : nat) (x \ y : A),
      Cycle (iterate f n x) (iterate f m y) \leftrightarrow n = m.
     TODO: head tail etc *)
Lemma Cycle\_nth:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
     Cycle 11 12 \rightarrow \forall (n : nat)(x : A),
        nth \ n \ l1 = Some \ x \rightarrow \exists \ m : nat, nth \ m \ l2 = Some \ x.
Lemma Cycle\_cycle:
  \forall (A : \mathsf{Type}) (n : nat) (l : list A),
      Cycle\ (cycle\ n\ l)\ l.
Lemma cycle\_Cycle:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow \exists\ n:\ nat,\ cycle\ n\ l1=l2.
Lemma Cycle\_insert:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle 11 12 \rightarrow \forall (n : nat)(x : A),
        \exists m : nat, Cycle (insert l1 \ n \ x) (insert l2 \ m \ x).
Lemma Cycle\_zip:
  \exists (A B : \mathsf{Type}) (la1 \ la2 : list A) (lb1 \ lb2 : list B),
      Cycle la1 la2 \wedge Cycle lb1 lb2 \wedge \neg Cycle (zip la1 lb1) (zip la2 lb2).
     TODO: zipW, unzip, unzipW *)
Lemma Cycle\_any:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
     Cycle l1 l2 \rightarrow any p l1 = any p l2.
Lemma Cycle\_all:
  \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Cycle l1 l2 \rightarrow all p l1 = all p l2.
Lemma Cycle\_find:
  \exists (A : \mathsf{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 : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Cycle l1 l2 \rightarrow count p l1 = count p l2.
Lemma Cycle\_filter:
   \forall (A : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Cycle l1 l2 \rightarrow Cycle (filter p l1) (filter p l2).
     TODO: findIndices *)
Lemma Cycle\_takeWhile:
   \exists (A : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Cycle (drop While p l1) (drop While p l2).
Lemma Cycle\_pmap:
   \forall (A B : \mathsf{Type}) (f : A \to option B) (l1 l2 : list A),
      Cycle l1 l2 \rightarrow Cycle (pmap f l1) (pmap f l2).
Lemma Cycle\_intersperse:
   \exists (A : \mathsf{Type}) (x : A) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Cycle (intersperse x l1) (intersperse x l2).
Lemma Cycle\_Permutation:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow Permutation\ l1\ l2.
Lemma Cycle\_elem:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow \forall x: A, elem\ x\ l1 \leftrightarrow elem\ x\ l2.
Lemma Cycle\_replicate':
   \forall (A : \mathsf{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 : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow \forall x : A, In x l1 \leftrightarrow In x l2.
Lemma Cycle\_NoDup:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow NoDup\ l1 \rightarrow NoDup\ l2.
Lemma Cycle\_Dup:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow Dup\ l1 \rightarrow Dup\ l2.
Lemma Cycle\_Rep:
```

 $\forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),$

```
Cycle 11 12 \rightarrow \forall (x : A) (n : nat),
          Rep \ x \ n \ l1 \rightarrow Rep \ x \ n \ l2.
Lemma Cycle\_Exists:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \rightarrow Exists \ P \ l1 \rightarrow Exists \ P \ l2.
Lemma Cycle\_Forall:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (l1 \ l2 : list \ A),
      Cycle l1\ l2 \rightarrow Forall\ P\ l1 \rightarrow Forall\ P\ l2.
Lemma Cycle\_AtLeast:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle 11 l2 \rightarrow \forall (P : A \rightarrow Prop) (n : nat),
          AtLeast\ P\ n\ l1 \rightarrow AtLeast\ P\ n\ l2.
Lemma Cycle\_Exactly:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle 11 12 \rightarrow \forall (P : A \rightarrow Prop) (n : nat),
          Exactly P \ n \ l1 \rightarrow Exactly \ P \ n \ l2.
Lemma Cycle\_AtMost:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle 11 12 \rightarrow \forall (P : A \rightarrow Prop) (n : nat),
          AtMost\ P\ n\ l1 \rightarrow AtMost\ P\ n\ l2.
Lemma Cycle\_Sublist:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Sublist l1 l2.
Lemma Sublist\_Cycle:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1 l2 \land \neg Cycle l1 l2.
Lemma Cycle\_Prefix:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Prefix l1 l2.
Lemma Prefix\_Cycle:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Prefix l1 l2 \land \neg Cycle l1 l2.
Lemma Cycle\_Subseq:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Cycle l1 l2 \land \neg Subseq l1 l2.
Lemma Subseq\_Cycle:
   \exists (A : \mathsf{Type}) (l1 \ l2 : list \ A),
```

Lemma $Cycle_Incl$:

Subseq l1 $l2 \land \neg Cycle l1 l2$.

```
\forall \; (A: {\tt Type}) \; (l1 \; l2 : list \; A), \\ Cycle \; l1 \; l2 \to Incl \; l1 \; l2. Lemma Incl\_Cycle: \exists \; (A: {\tt Type}) \; (l1 \; l2 : list \; A), \\ Incl \; l1 \; l2 \land \neg \; Cycle \; l1 \; l2. Lemma Cycle\_SetEquiv: \forall \; (A: {\tt Type}) \; (l1 \; l2 : list \; A), \\ Cycle \; l1 \; l2 \to SetEquiv \; l1 \; l2. Lemma SetEquiv\_Cycle: \exists \; (A: {\tt Type}) \; (l1 \; l2 : list \; A), \\ SetEquiv \; l1 \; l2 \land \neg \; Cycle \; l1 \; l2.
```

9.7 Niestandardowe reguły indukcyjne

Wyjaśnienia nadejdą już wkrótce.

```
Fixpoint list_ind_2
   (A : \mathsf{Type}) (P : list A \to \mathsf{Prop})
   (H\theta:P]
   (H1: \forall x: A, P[x])
   (H2: \forall (x y: A) (l: list A), P l \rightarrow P (x:: y:: l))
      (l: list A): P l.
Lemma list\_ind\_rev:
   \forall (A : \mathsf{Type}) (P : \mathit{list} A \to \mathsf{Prop})
      (Hnil: P[])
      (Hsnoc: \forall (h: A) (t: list A), P t \rightarrow P (t ++ [h]))
         (l: list A), P l.
Lemma list\_ind\_app\_l:
   \forall (A : \mathsf{Type}) (P : list A \to \mathsf{Prop})
   (Hnil: P \parallel) (IH: \forall l \ l': list \ A, P \ l \rightarrow P \ (l'++l))
      (l: list A), P l.
Lemma list\_ind\_app\_r:
   \forall (A : \mathsf{Type}) (P : \mathit{list} A \to \mathsf{Prop})
   (Hnil: P \parallel) (IH: \forall l \ l': list \ A, P \ l \rightarrow P \ (l ++ l'))
      (l: list A), P l.
Lemma list\_ind\_app:
   \forall (A : \mathsf{Type}) (P : list A \to \mathsf{Prop})
   (Hnil: P \parallel) (Hsingl: \forall x: A, P \mid x \mid)
   (IH: \forall l \ l': list \ A, P \ l \rightarrow P \ l' \rightarrow P \ (l ++ l'))
      (l: list A), P l.
```

```
Lemma list\_app\_ind:
\forall (A: \texttt{Type}) (P: list A \rightarrow \texttt{Prop}),
P [] \rightarrow \\ (\forall (l \ l1 \ l2: list \ A), P \ l \rightarrow P \ (l1 \ ++ \ l \ ++ \ l2)) \rightarrow \\ \forall \ l: list \ A, P \ l.
```

9.7.1 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:
  \forall (A : \mathsf{Type}) (x \ y : A),
      Palindrome [x; y] \rightarrow x = y.
Lemma Palindrome\_inv\_3:
   \forall (A : \mathsf{Type}) (x \ y : A) (l : \mathit{list} \ A),
      Palindrome\ (x::l++[y]) \rightarrow x = y.
Lemma nat_{-}ind_{-}2:
   \forall P : nat \rightarrow \mathsf{Prop},
      P \ 0 \rightarrow P \ 1 \rightarrow (\forall \ n : nat, P \ n \rightarrow P \ (S \ (S \ n))) \rightarrow
         \forall n : nat, P n.
Lemma Palindrome\_length:
  \forall (A : \mathsf{Type}) (x : A) (n : nat),
      \exists l : list A, Palindrome l \land n \leq length l.
Lemma Palindrome\_cons\_snoc:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Palindrome \ l \rightarrow Palindrome \ (x :: snoc \ x \ l).
Lemma Palindrome\_app:
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Palindrome \ l1 \rightarrow Palindrome \ l2 \rightarrow Palindrome \ (l1 ++ l2 ++ rev \ l1).
Lemma Palindrome\_app':
   \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Palindrome \ l2 \rightarrow Palindrome \ (l1 ++ l2 ++ rev \ l1).
Lemma Palindrome\_rev:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Palindrome \ l \leftrightarrow Palindrome \ (rev \ l).
Definition lengthOrder \{A : Type\} (l1 \ l2 : list \ A) : Prop :=
   length l1 < length l2.
Lemma lengthOrder\_wf:
```

```
\forall A : \mathsf{Type}, well\_founded (@lengthOrder A).
(* TODO: spec bez używania indukcji dobrze ufundowanej *)
Lemma Palindrome\_spec:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Palindrome \ l \leftrightarrow l = rev \ l.
Lemma Palindrome\_spec':
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Palindrome \ l \rightarrow \exists \ l1 \ l2 : list \ A,
         l = l1 ++ l2 ++ rev l1 \wedge length l2 \leq 1.
Lemma Palindrome_{-}map:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : \mathit{list} A),
      Palindrome \ l \rightarrow Palindrome \ (map \ f \ l).
Lemma replicate\_S:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      replicate (S \ n) \ x = x :: replicate \ n \ x.
Lemma Palindrome\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x : A),
      Palindrome (replicate \ n \ x).
Lemma Palindrome\_cons\_replicate:
   \forall (A : \mathsf{Type}) (n : nat) (x \ y : A),
      Palindrome (x :: replicate \ n \ y) \rightarrow n = 0 \lor x = y.
Lemma Palindrome\_iterate:
   \forall (A : \mathsf{Type}) (f : A \to A),
      (\forall (n: nat) (x: A), Palindrome (iterate f n x)) \rightarrow
         \forall x: A, f x = x.
Lemma Palindrome\_nth:
   \forall (A : \mathsf{Type}) (l : \mathit{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 : \mathsf{Type}) (l : \mathit{list} A),
      (\forall n : nat, Palindrome (drop n l)) \rightarrow
         l = [] \lor \exists (n : nat) (x : A), l = replicate n x.
Lemma Palindrome\_take:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      (\forall n : nat, Palindrome (take n l)) \rightarrow
         l = [] \lor \exists (n : nat) (x : A), l = replicate n x.
Lemma replace\_Palindrome:
   \forall (A : \mathsf{Type}) (l \ l' : \mathit{list} \ A) (n : \mathit{nat}) (x : A),
```

```
replace l \ n \ x = Some \ l' \rightarrow Palindrome \ l \rightarrow
         Palindrome l' \leftrightarrow length \ l = 1 \land n = 0 \lor nth \ n \ l = Some \ x.
Lemma Palindrome\_zip:
   \exists (A B : \mathsf{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 : \mathsf{Type}) (p : A \to bool) (l : list A),
      Palindrome \ l \rightarrow find \ p \ l = findLast \ p \ l.
Lemma Palindrome\_pmap:
   \forall (A \ B : \mathsf{Type}) \ (f : A \to option \ B) \ (l : list \ A),
      Palindrome \ l \rightarrow Palindrome \ (pmap \ f \ l).
Lemma Palindrome\_intersperse:
   \forall (A : \mathsf{Type}) (x : A) (l : \mathit{list} A),
      Palindrome \ l \rightarrow Palindrome \ (intersperse \ x \ l).
      TODO: groupBy *)
Lemma Palindrome_{-}Dup:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Palindrome \ l \rightarrow length \ l \leq 1 \vee Dup \ l.
     TODO: Incl, Sublist, subseq *)
Lemma Sublist\_Palindrome:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A),
      Sublist l1\ l2 \rightarrow Palindrome\ l1 \rightarrow Palindrome\ l2 \rightarrow l1 = [].
Lemma Prefix_Palindrome:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Prefix (rev \ l) \ l \leftrightarrow Palindrome \ l.
Lemma Subseq\_rev\_l:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Subseq (rev l) l \leftrightarrow Palindrome l.
Lemma Subseq\_rev\_r:
   \forall (A : \mathsf{Type}) (l : \mathit{list} A),
      Subseq l (rev l) \leftrightarrow Palindrome l.
```

Rozdział 10

E1: Rekordy, klasy i moduły - TODO

UWAGA: ten rozdział został naprędce posklejany z fragmentów innych, więc może nie mieć większego sensu.

10.1 Rekordy (TODO)

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 \neq 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

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...

10.2 Klasy (TODO)

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: {\tt Type}): {\tt Type}:= \{ \\ eq\_dec: A \rightarrow A \rightarrow bool; \\ eq\_dec\_spec: \forall\ x\ y: A,\ eq\_dec\ x\ y = true \leftrightarrow 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".

```
 \begin{split} \#[\texttt{refine}] \\ & \texttt{Instance} \ EqDec\_bool : EqDec \ bool := \\ \{ & eq\_dec := \texttt{fun} \ b \ b' : bool \Rightarrow \\ & \texttt{match} \ b, \ b' \ \texttt{with} \\ & | \ true, \ true \Rightarrow true \\ & | \ false, \ false \Rightarrow true \\ & | \ \_, \ \_ \Rightarrow false \\ & \texttt{end} \\ \}. \\ & \texttt{Proof.} \\ & \texttt{destruct} \ x, \ y; \ \texttt{split}; \ \texttt{trivial}; \ \texttt{inversion} \ 1. \\ & \texttt{Defined.} \end{split}
```

Instancje klas definiujemy przy pomocy słowa kluczowego #[refine] 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:

• $bool \rightarrow bool$

```
• bool \rightarrow nat
```

- $nat \rightarrow bool$
- \bullet $nat \rightarrow nat$
- Prop

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

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).

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 $A \rightarrow B$.

10.3 Moduły (TODO)

Rozdział 11

E2: Funkcje

Require Import Arith.

W tym rozdziale zapoznamy się z najważniejszymi rodzajami funkcji. Trzeba przyznać na wstępie, że rozdział będzie raczej matematyczny (co wcale nie powinno cię odstraszać - matematyka jest świetna, a najbardziej praktyczną rzeczą w kosmosie jest dobra teoria).

11.1 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: 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.

```
Notation "f $ x":= (f\ x) (left associativity, at level 110, only\ parsing). Notation "x |> f":= (f\ x) (right associativity, at level 60, only\ parsing). Check plus\ (2+2)\ (3+3). Check plus\ $2+2\ $3+3. Check (fun n:\ nat\Rightarrow n+n) 21. Check 21\ |> fun n:\ nat\Rightarrow 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.

Notacja \$ (pożyczona z języka Haskell) 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 |> (pożyczona z języka F#) umożliwi nam pisanie aplikacji w odwrotnej kolejności. Dzięki temu będziemy mogli np. pomijać nawiasy w abstrakcji. Jako, że nie da się zrobić notacji w stylu "x f", jest to najlepsze dostępne nam rozwiązanie.

Definition comp

```
 \{A \ B \ C : \mathtt{Type}\} \ (f : A \to B) \ (g : B \to C) : A \to C := \mathtt{fun} \ x : A \Rightarrow g \ (f \ x).
```

Notation "f .> g":= $(comp \ f \ g)$ (left associativity, at level 40).

Drugą najważniejszą operacją, jaką możemy wykonywać na funkcjach, jest składanie. Jedynym warunkiem jest aby przeciwdziedzina pierwszej funkcji była taka sama, jak dziedzina drugiej funkcji.

Lemma $comp_assoc$:

```
\forall \; (A \; B \; C \; D : \mathtt{Type}) \; (f : A \to B) \; (g : B \to C) \; (h : C \to D), \\ (f .> g) .> h = f .> (g .> h).
```

Składanie funkcji jest łączne. Zagadka: czy jest przemienne?

Uwaga techniczna: jeżeli prezentuję jakieś twierdzenie bez dowodu, to znaczy, że dowód jest ćwiczeniem.

```
Definition id \{A : \mathsf{Type}\} : A \to A := \mathsf{fun} \ x : A \Rightarrow 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.

Lemma $id_{-}left$:

```
\forall \; (A \; B \; \colon \mathtt{Type}) \; (f \; \colon A \to B), \; id \; .> f = f.
```

Lemma id_right :

$$\forall \ (A \ B : \mathtt{Type}) \ (f : A \rightarrow B), f > id = f.$$

$$\texttt{Definition} \ const \ \{A \ B : \texttt{Type}\} \ (b : B) : A \to B := \texttt{fun} \ _ \Rightarrow b.$$

Funkcja stała to funkcja, która ignoruje swój drugi argument i zawsze zwraca pierwszy argument.

Definition flip

```
 \{A\ B\ C: \mathtt{Type}\}\ (f:A\to B\to C): B\to A\to C:=\mathtt{fun}\ (b:B)\ (a:A)\Rightarrow f\ a\ b.
```

flip to całkiem przydatny kombinator (funkcja wyższego rzędu), który zamienia miejscami argumenty funkcji dwuargumentowej.

```
Fixpoint iter \{A: \mathtt{Type}\}\ (n:nat)\ (f:A\to A):A\to A:= match n with |0\Rightarrow id
```

```
\mid S \mid n' \Rightarrow f > iter \mid n' \mid 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.

11.2 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

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.

```
Lemma eta\_expansion: \forall (A \ B : \mathsf{Type}) \ (f : A \to B), f = \mathsf{fun} \ x : A \Rightarrow f \ x. Proof. reflexivity. Qed. Print Assumptions eta\_expansion. (* ===> Closed under the global context *)
```

-ekspansja 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.

```
Lemma plus\_1\_eq:   (fun n:nat\Rightarrow 1+n)= (fun n:nat\Rightarrow 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.

 ${\tt Check} @ functional_extensionality.$

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.

```
Lemma plus\_1\_eq:   (fun n: nat \Rightarrow 1+n) = (fun n: nat \Rightarrow 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 $plus_comm$.

Cwiczenie Użyj aksjomatu ekstensjonalności, żeby pokazać, że dwie funkcje binarne są równe wtedy i tylko wtedy, gdy ich wartości dla wszystkich argumentów są równe.

Lemma $binary_funext$:

```
\forall (A \ B \ C : \mathtt{Type}) \ (f \ g : A \to B \to C), f = g \leftrightarrow \forall \ (a : A) \ (b : B), f \ a \ b = g \ a \ b.
```

11.3 Odwrotności i izomorfizmy (TODO)

W tym podrozdziale zajmiemy się pojęciem funkcji odwrotnej i płynącą z niego mądrością.

```
Definition has\_preinverse\ \{A\ B: \mathtt{Type}\}\ (f:A\to B): \mathtt{Type}:=\{g:B\to A\mid \forall\ b:B,f\ (g\ b)=b\}. Definition has\_postinverse\ \{A\ B: \mathtt{Type}\}\ (f:A\to B): \mathtt{Type}:=\{g:B\to A\mid \forall\ a:A,\ g\ (f\ a)=a\}.
```

Intuicja jest dość prosta: wiemy ze szkoły, że na liczbach całkowitych odejmowanie jest odwrotnością dodawania (np. a + b - b = a), tzn. jeżeli do a dodamy b, a potem odejmiemy b, to znowu mamy a. Podobnie w liczbach rzeczywistych mnożenie przez liczbę niezerową ma odwrotność w postaci dzielenia, np. (x * y) / y = x.

Oczywiście pojęcie odwrotności dotyczy nie tylko działań na liczbach, ale także dowolnych funkcji - g jest odwrotnością f, gdy odwraca ono działanie f dla dowolnego argumentu a, tzn. najpierw mamy a, potem aplikujemy f i mamy f a, zaś na koniec aplikujemy g i znów mamy a, czyli g (f a) = a. To właśnie jest napisane w definicji $has_postinverse$.

No właśnie - powyższy opis jest opisem postodwrotności. Nazwa wynika z kolejności - g jest postodwrotnością f, gdy najpierw aplikujemy f, a potem odwracamy jego działanie za pomocą g (po łacinie "post" znaczy "po", np. "post meridiem" znaczy "po południu").

Analogicznie, choć może nieco mniej intuicyjnie, prezentuje się definicja preodwrotności (po łacinie "prae" znaczy "przed"). g jest preodwrotnością f, gdy f jest postodwrotnością g. Innymi słowy: f ma preodwrotność, jeżeli odwraca ono działanie jakiejś funkcji.

Dobra, wystarczy gadania. Czas na ćwiczenia.

Ćwiczenie Pokaż, że odjęcie n jest postodwrotnością dodania n. Czy jest także preodwrotnością?

```
Lemma plus_n-has_postinverse\_sub_n: \forall n: nat, \\ has_postinverse (plus n).
```

Zauważ, że sortem has_preinverse i has_postinverse jest Type, nie zaś Prop. Jest tak dlatego, że o ile stwierdzenie "f jest pre/post odwrotnością g" jest zdaniem, to posiadanie odwrotności już nie, gdyż dana funkcja może mieć wiele różnych odwrotności.

Ćwiczenie Rozważmy funkcję *app* [1; 2; 3], która dokleja na początek listy liczb naturalnych listę [1; 2; 3]. Znajdź dwie różne jej postodwrotności (nie musisz formalnie dowodzić, że są różne - wystarczy nieformalny argument). Czy funkcja ta ma preodwrotność?

Require Import D5.

Ćwiczenie Czasem funkcja może mieć naprawdę dużo odwrotności. Pokaż, że funkcja cons x dla x: A ma ich nieskończenie wiele. Nie musisz dowodzić, że odwrotności są różne (ani że jest ich dużo), jeżeli widać to na pierwszy rzut oka.

```
rotności. Może więc preodwrotności nie istnieją? Otóż nie tym razem!
    Dla jakich n funkcja cycle n ma (pre/post)odwrotność?
Definition uncycle \{A : \mathsf{Type}\}\ (n : nat)\ (l : list\ A) : list\ A :=
  cycle\ (length\ l-n)\ l.
Lemma cycle\_has\_preinverse:
  \forall (A : \mathsf{Type}) (n : nat),
     has\_preinverse (@cycle A n).
Proof.
  intros. red.
  \exists (uncycle \ n).
  unfold uncycle, cycle.
Abort.
(*
   Lemma cycle'_has_preinverse :
  forall (A : Type) (n : nat),
     has_preinverse (@cycle' A n).
   Proof.
  intros. red.
  exists (fun 1 : list A => cycle' (length 1 - n) 1).
  induction n as | n'; cbn.
   Abort.
    *)
(* end hide *)
Definition isomorphism \{A \mid B : \mathsf{Type}\}\ (f : A \to B) : \mathsf{Type} :=
  \{g: B \to A \mid (\forall a: A, g (f a) = a) \land \}
                    (\forall b: B, f(g b) = b)\}.
Lemma iso\_has\_preinverse:
  \forall \{A \ B : \mathsf{Type}\} \{f : A \to B\},\
     isomorphism \ f \rightarrow has\_preinverse \ f.
Lemma iso\_has\_postinverse:
  \forall \{A \ B : \mathsf{Type}\} \{f : A \to B\},\
     isomorphism\ f \rightarrow has\_postinverse\ f.
Lemma both\_inverses\_isomorphism:
  \forall \{A \ B : \mathsf{Type}\} \ \{f : A \to B\},\
```

Cwiczenie Dla listowych funkcji widzieliśmy postodwrotności, ale nie widzieliśmy preodw-

11.4 Skracalność (TODO)

```
Definition precancellable \{A B : \mathsf{Type}\} (f : A \to B) : \mathsf{Prop} :=
```

 $has_preinverse\ f \rightarrow has_postinverse\ f \rightarrow isomorphism\ f.$

```
\forall \; (X : {\tt Type}) \; (g \; h : B \to X), f \; .> g = f \; .> h \to g = h. Definition postcancellable \; \{A \; B : {\tt Type}\} \; (f : A \to B) : {\tt Prop} := \forall \; (X : {\tt Type}) \; (g \; h : X \to A), \; g \; .> f = h \; .> f \to g = h. Lemma has\_preinverse\_precancellable : \forall \; \{A \; B : {\tt Type}\} \; \{f : A \to B\}, \\ has\_preinverse \; f \to precancellable \; f. Lemma has\_postinverse\_postcancellable : \\ \forall \; \{A \; B : {\tt Type}\} \; \{f : A \to B\}, \\ has\_postinverse \; f \to postcancellable \; f.
```

11.5 Injekcje

```
Definition injective \{A \ B : \mathtt{Type}\}\ (f : A \to B) : \mathtt{Prop} := \forall x \ x' : A, f \ x = f \ x' \to 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". Na wiki można zapoznać się z obrazkami poglądowymi:

https://en.wikipedia.org/wiki/Bijection,%20injection%20and%20surjection

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 \Rightarrow 2+n). 
 Proof. 
 unfold injective; intros. destruct x, x'; cbn in *. 
 trivial. 
 inversion H. 
 inversion H. 
 inversion H. trivial. 
 Qed.
```

Funkcja fun $n: nat \Rightarrow 2+n$, 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 \neg injective (fun n: nat \Rightarrow n \times n - n).

Proof.

unfold injective, not; intros.

specialize (H\ 0\ 1). cbn 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 : \mathtt{Type}\}\ (f : A \to B) : \mathtt{Prop} := \forall x \ x' : A, \ x \neq x' \to f \ x \neq f \ x'.
```

Głosi ona, że funkcja injektywna to funkcja, która dla różnych argumentów przyjmuje różne wartości. Innymi słowy, injekcja to funkcja, która zachowuje relację \neq . 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 lepsza od drugiej.

Ćwiczenie Pokaż, że *injective* jest mocniejsze od *injective*. Pokaż też, że w logice klasycznej są one równoważne.

```
Lemma injective_injective':
   \forall (A B : \mathsf{Type}) (f : A \to B),
      injective f \rightarrow injective' f.
Lemma injective'_-injective:
   (\forall P : \mathtt{Prop}, \neg \neg P \rightarrow P) \rightarrow
   \forall (A B : \mathsf{Type}) (f : A \to B),
      injective' f \rightarrow injective f.
    Udowodnij, że różne funkcje są lub nie są injektywne.
Lemma id_{-}injective:
   \forall A : \mathsf{Type}, injective (@id A).
Lemma S_{-}injective : injective S.
Lemma const\_unit\_inj:
   \forall (A : \mathsf{Type}) (a : A),
      injective (fun \_: unit \Rightarrow a).
Lemma add_k_left_inj:
   \forall k : nat, injective (fun n : nat \Rightarrow k + n).
Lemma mul_{-}k_{-}inj:
   \forall k : nat, k \neq 0 \rightarrow injective (fun n : nat \Rightarrow k \times n).
Lemma const\_2elem\_not\_inj:
   \forall (A B : \mathsf{Type}) (b : B),
      (\exists \ a \ a' : A, \ a \neq a') \rightarrow \neg \ injective \ (fun \ \_ : A \Rightarrow b).
Lemma mul_{-}k_{-}\theta_{-}not_{-}inj:
   \neg injective (fun \ n : nat \Rightarrow 0 \times n).
Lemma pred\_not\_injective : \neg injective pred.
```

Jedną z ważnych właściwości injekcji jest to, że są składalne: złożenie dwóch injekcji daje injekcję.

Lemma inj_comp :

```
\forall (A \ B \ C : \mathtt{Type}) \ (f : A \to B) \ (g : B \to C), injective f \to injective \ g \to injective \ (f .> g).
```

Ta właściwość jest dziwna. Być może kiedyś wymyślę dla niej jakaś bajkę.

Lemma LOLWUT:

```
\forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B) \ (g : B \to C), injective (f .> g) \to injective \ f.
```

Na zakończenie należy dodać do naszej interpretacji pojęcia "injekcja" jeszcze jedną ideę. Mianowicie jeżeli istnieje injekcja $f:A\to B$, 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*.

```
Lemma 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).

```
Lemma inj\_Empty\_set\_A: \forall A : \texttt{Type}, \exists f : Empty\_set \rightarrow A, injective f.
```

11.6 Surjekcje

Drugim ważnym rodzajem funkcji są surjekcje.

```
Definition surjective\ \{A\ B: {\tt Type}\}\ (f:A\to B): {\tt 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.

```
Lemma pred\_surjective: surjective \ pred. Proof. unfold surjective; intros. \exists \ (S \ b). \ cbn. trivial. Qed.
```

TODO Uwaga techniczna: od teraz do upraszczania zamiast taktyki cbn używać będziemy taktyki cbn. Różni się ona nieznacznie od cbn, ale jej główną zaletą jest nazwa — cbn to trzy litery, a cbn 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 $(S \ n) = n$.

```
Lemma S\_not\_surjective: \neg 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.

```
 \begin{array}{l} \texttt{Lemma} \ sur\_comp : \\ \forall \ (A \ B \ C : \texttt{Type}) \ (f : A \to B) \ (g : B \to C), \\ surjective \ f \to surjective \ g \to surjective \ (f .> g). \\ \texttt{Lemma} \ LOLWUT\_sur : \\ \forall \ (A \ B \ C : \texttt{Type}) \ (f : A \to B) \ (g : B \to C), \\ surjective \ (f .> g) \to surjective \ g. \end{array}
```

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*.

```
Lemma no\_sur\_unit\_bool:

\neg \exists f : unit \rightarrow 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.

```
Lemma sur\_A\_unit: \forall (A : \texttt{Type}) \ (nonempty : A), \exists f : A \rightarrow unit, surjective f.
```

11.7 Bijekcje

```
Definition bijective \{A \ B : \mathsf{Type}\}\ (f : A \to B) : \mathsf{Prop} := injective \ f \land surjective \ f.
```

Po łacinie przedrostek "bi-" oznacza "dwa". Bijekcja to funkcja, która jest zarówno injekcją, jak i surjekcją.

```
Lemma id\_bij: \forall A: \mathsf{Type}, \ bijective \ (@id\ A). Proof.

split; intros.

apply id\_injective.

apply id\_sur.
Qed.

Lemma S\_not\_bij: \neg \ 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 : \mathtt{Type}\}\ (f : A \to B) : \mathtt{Prop} := \forall \ b : B, \exists ! \ a : A, f \ a = b.
```

Ćwiczenie Udowodnij, że obie definicje są równoważne.

```
Lemma bijective_bijective': \forall (A \ B : \texttt{Type}) \ (f : A \rightarrow B), bijective f \leftrightarrow bijective' f.
```

Ćwiczenie Require Import D5.

```
Fixpoint unary\ (n:nat): list\ unit:= match n with  \mid 0 \Rightarrow [] \\ \mid S\ n' \Rightarrow tt:: unary\ n'  end.
```

Funkcja unary reprezentuje liczbę naturalną n za pomocą listy zawierającej n kopii termu tt. Udowodnij, że unary jest bijekcją.

Lemma $unary_bij$: $bijective\ unary$.

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.

Lemma bij_comp :

```
\forall \ (A \ B \ C : \mathtt{Type}) \ (f : A \to B) \ (g : B \to C), bijective \ f \to bijective \ g \to 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
- ullet 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 A ~ B":= (equipotent A B) (at level 10).
```

Równoliczność A i B będziemy oznaczać przez $A \neg B$. Nie należy notacji \neg mylić z notacją \neg oznaczającej negację logiczną. Ciężko jednak jest je pomylić, gdyż pierwsza zawsze bierze dwa argumenty, a druga tylko jeden.

```
Lemma equipotent\_refl: \forall A: \mathtt{Type}, A \overset{\sim}{-} A. Lemma equipotent\_sym: \forall A B: \mathtt{Type}, A \overset{\sim}{-} B \to B \overset{\sim}{-} A. Lemma equipotent\_trans: \forall A B C: \mathtt{Type}, A \overset{\sim}{-} B \to B \overset{\sim}{-} C \to A \overset{\sim}{-} C.
```

11.8 Inwolucje

```
Definition involutive\ \{A: {\tt Type}\}\ (f:A\to A): {\tt 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.

```
 \begin{array}{l} \text{Lemma } id\_inv: \\ \forall \ A: \texttt{Type}, \ involutive \ (@id \ A). \\ \\ \text{Lemma } rev\_inv: \\ \forall \ A: \texttt{Type}, \ involutive \ (@rev \ A). \\ \\ \text{Lemma } negb\_inv: \ involutive \ negb. \\ \\ \dot{\textbf{Z}} \text{eby nie odgrzewać starych kotletów, przyjrzyjmy się funkcji } weird. \\ \\ \textbf{Fixpoint } weird \ \{A: \texttt{Type}\} \ (l: list \ A): list \ A:= \\ \\ \text{match } l \ \text{with} \\ & |\ \|\Rightarrow \| \\ & |\ [x] \Rightarrow [x] \\ \end{array}
```

```
\mid x :: y :: t \Rightarrow y :: x :: weird t end.
```

Lemma $weird_inv$:

```
\forall A : \mathsf{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.

Lemma $flip_inv$:

```
\forall 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 \neg 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.

Lemma $comp_inv$:

```
\forall (A: \mathtt{Type}) (f \ g: A \to A), involutive f \to involutive \ g \to f .> g = g .> f \to 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.

Lemma inv_bij :

```
\forall (A : \mathsf{Type}) (f : A \to A), involutive f \to 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?
```

11.9 Uogólnione inwolucje

Pojęcie inwolucji można nieco uogólnić. Żeby to zrobić, przeformułujmy najpierw definicję inwolucji.

```
\begin{array}{ll} {\tt Definition} \ involutive' \ \{A: {\tt Type}\} \ (f: A \rightarrow A): {\tt Prop} := \\ f: > f = id. \end{array}
```

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.

Lemma $involutive_involutive$ ':

```
\forall (A : \mathsf{Type}) (f : A \to A), involutive f \leftrightarrow involutive' f.
```

Pójdźmy o krok dalej. Zamiast składania .> użyjmy kombinatora *iter* 2, który ma taki sam efekt.

```
\begin{array}{l} {\tt Definition} \ involutive'' \ \{A: {\tt Type}\} \ (f:A \to A): {\tt Prop} := \\ iter \ f \ 2 = id. \\ {\tt Lemma} \ involutive'\_involutive'' : \\ {\tt } \forall \ (A: {\tt Type}) \ (f:A \to A), \\ involutive' \ f \ \leftrightarrow involutive'' \ f. \end{array}
```

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.

```
 \begin{array}{l} {\tt Definition} \ gen\_involutive \\ \{A: {\tt Type}\} \ (n: nat) \ (f: A \rightarrow A) : {\tt Prop} := \\ iter \ f \ n = id. \end{array}
```

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 : \texttt{Type}\}\ (l : list\ A) : list\ A := \texttt{match}\ l \ \texttt{with}
|\ \| \Rightarrow \| \\ |\ [x] \Rightarrow [x] \\ |\ [x;\ y] \Rightarrow [x;\ y] \\ |\ x :: \ y :: \ z :: \ t \Rightarrow y :: \ z :: \ x :: \ weirder\ t \ \texttt{end}.
Compute weirder [1;\ 2;\ 3;\ 4;\ 5].
(* \ ===>\ =\ 2;\ 3;\ 1;\ 4;\ 5 : \ \texttt{list}\ \texttt{nat}\ *)
Compute iter weirder [1;\ 2;\ 3;\ 4;\ 5].
(* \ ===>\ =\ 1;\ 2;\ 3;\ 4;\ 5 : \ \texttt{list}\ \texttt{nat}\ *)
Lemma weirder_inv_3:
\forall\ A : \ \texttt{Type},\ gen_involutive}\ 3\ (@weirder\ A).
```

11.10 Idempotencja

```
Definition idempotent \{A : \mathtt{Type}\}\ (f : A \to A) : \mathtt{Prop} := \forall \ 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.

```
Lemma id\_idem:
\forall A : \texttt{Type}, idempotent (@id A).
Lemma const\_idem:
\forall (A B : \texttt{Type}) (b : B), idempotent (const b).
Lemma take\_idem:
\forall (A : \texttt{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.

```
Lemma comp\_idem:
```

```
\forall \ (A: \mathtt{Type}) \ (f \ g: A \rightarrow A), \ idempotent \ f \rightarrow idempotent \ g \rightarrow f \ .> g = g \ .> f \rightarrow idempotent \ (f \ .> g).
```

Jeżeli chodzi o składanie funkcji idempotentnych, sytuacja jest podobna do tej, jaka jest udziałem inwolucji.

Rozdział 12

E3: Relacje

```
Require Import E2.
Require Import Functional Extensionality.
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.

12.1 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 → 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 → 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 ++ l') (l' ++ '). 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 dopasowanie do wzorca 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: \mathsf{Type}): \mathsf{Type} := A \to B \to \mathsf{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).

12.2 Identyczność relacji

```
Definition subrelation \ \{A \ B : {\tt Type}\} \ (R \ S : hrel \ A \ B) : {\tt Prop} := \ \forall \ (a : A) \ (b : B), \ R \ a \ b \to S \ a \ b. Notation \dot{\tt Z} -> {\tt S}" := (subrelation \ R \ S) \ ({\tt at level} \ 40). Definition same\_hrel \ \{A \ B : {\tt Type}\} \ (R \ S : hrel \ A \ B) : {\tt Prop} := \ subrelation \ R \ S \wedge subrelation \ S \ R. Notation \dot{\tt Z} <-> {\tt S}" := (same\_hrel \ R \ S) \ ({\tt 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.

Udowodnij, że powyższa definicja le' porządku "mniejszy lub równy" na liczbach naturalnych jest ta 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.

12.3 Operacje na relacjach

```
 \begin{array}{l} \texttt{Definition} \ Rcomp \\ \{A \ B \ C : \texttt{Type}\} \ (R : hrel \ A \ B) \ (S : hrel \ B \ C) : hrel \ A \ C := \\ \texttt{fun} \ (a : A) \ (c : C) \Rightarrow \exists \ b : B, \ R \ a \ b \land S \ b \ c. \\ \texttt{Definition} \ Rid \ \{A : \texttt{Type}\} : hrel \ A \ A := @eq \ A. \end{array}
```

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.

```
 \begin{array}{l} \texttt{Lemma} \ Rcomp\_assoc: \\ \forall \\ \qquad (A \ B \ C \ D : \texttt{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C) \ (T : hrel \ C \ D), \\ \qquad Rcomp \ R \ (Rcomp \ S \ T) <-> \ Rcomp \ (Rcomp \ R \ S) \ T. \\ \\ \texttt{Lemma} \ Rid\_left: \\ \forall \ (A \ B : \texttt{Type}) \ (R : hrel \ A \ B), \\ \qquad Rcomp \ (@Rid \ A) \ R <-> \ R. \\ \\ \texttt{Lemma} \ Rid\_right: \\ \forall \ (A \ B : \texttt{Type}) \ (R : hrel \ A \ B), \\ \qquad Rcomp \ R \ (@Rid \ B) <-> \ R. \end{array}
```

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 : \mathtt{Type}\}\ (R : hrel\ A\ B) : hrel\ B\ A := \mathtt{fun}\ (b : B)\ (a : A) \Rightarrow 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 : \mathtt{Type}) \ (R : \mathit{hrel} \ A \ B) \ (S : \mathit{hrel} \ B \ C), Rinv \ (Rcomp \ R \ S) <-> Rcomp \ (Rinv \ S) \ (Rinv \ R).
```

```
Lemma Rinv_Rid:
```

```
\forall A : \texttt{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.

```
 \begin{array}{l} \texttt{Definition} \ Rnot \ \{A \ B : \texttt{Type}\} \ (R : hrel \ A \ B) : hrel \ A \ B := \\ \texttt{fun} \ (a : A) \ (b : B) \Rightarrow \neg \ R \ a \ b. \\ \\ \texttt{Definition} \ Rand \ \{A \ B : \texttt{Type}\} \ (R \ S : hrel \ A \ B) : hrel \ A \ B := \\ \texttt{fun} \ (a : A) \ (b : B) \Rightarrow R \ a \ b \land S \ a \ b. \\ \\ \texttt{Definition} \ Ror \ \{A \ B : \texttt{Type}\} \ (R \ S : hrel \ A \ B) : hrel \ A \ B := \\ \texttt{fun} \ (a : A) \ (b : B) \Rightarrow R \ a \ b \lor S \ a \ b. \\ \end{aligned}
```

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: \mathtt{Type}\}: hrel\ A\ B:=\mathtt{fun}\ (a:A)\ (b:B)\Rightarrow True. Definition RFalse\ \{A\ B: \mathtt{Type}\}: hrel\ A\ B:=\mathtt{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 : \mathtt{Type}) \ (R : hrel \ A \ B), \ R \longrightarrow Rnot \ (Rnot \ R).
```

Lemma $Rand_assoc$:

```
\forall \ (A\ B: {\tt Type})\ (R\ S\ T: hrel\ A\ B), \ Rand\ R\ (Rand\ S\ T) <-> Rand\ (Rand\ R\ S)\ T.
```

Lemma $Rand_comm$:

```
orall \; (A \; B : {\tt Type}) \; (R \; S : hrel \; A \; B), \ Rand \; R \; S <-> Rand \; S \; R.
```

Lemma $Rand_RTrue_l$:

```
\forall (A \ B : \mathtt{Type}) \ (R : hrel \ A \ B), Rand RTrue R < -> R.
```

Lemma $Rand_RTrue_r$:

```
\forall \ (A \ B : \mathtt{Type}) \ (R : hrel \ A \ B), \\ Rand \ R \ RTrue <-> R.
```

Lemma $Rand_RFalse_l$:

```
\forall (A B : \mathsf{Type}) (R : hrel A B),
     Rand RFalse R \iff RFalse.
Lemma Rand\_RFalse\_r:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     Rand\ R\ RFalse <->\ RFalse.
Lemma Ror\_assoc:
  \forall (A B : \mathsf{Type}) (R S T : hrel A B),
     Ror R (Ror S T) <-> Ror (Ror R S) T.
Lemma Ror\_comm:
  \forall (A B : \mathsf{Type}) (R S : hrel A B),
     Ror R S < -> Ror S R.
Lemma Ror_RTrue_l:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     Ror RTrue R < -> RTrue.
Lemma Ror_RTrue_r:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     Ror \ R \ RTrue <-> RTrue.
Lemma Ror_RFalse_l:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     Ror RFalse R < -> R.
Lemma Ror_RFalse_r:
  \forall (A B : \mathsf{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.

12.4 Rodzaje relacji heterogenicznych

```
\label{eq:class_left_unique} \begin{array}{l} \texttt{Class} \ \textit{LeftUnique} \ \{A \ B : \texttt{Type}\} \ (R : \textit{hrel} \ A \ B) : \texttt{Prop} := \\ \{ & \textit{left\_unique} : \\ & \forall \ (a \ a' : A) \ (b : B), \ R \ a \ b \rightarrow R \ a' \ b \rightarrow a = a' \\ \}. \\ \texttt{Class} \ \textit{RightUnique} \ \{A \ B : \texttt{Type}\} \ (R : \textit{hrel} \ A \ B) : \texttt{Prop} := \\ \{ \end{array}
```

```
 \begin{array}{c} right\_unique: \\ \forall \ (a:A) \ (b\ b':B), \ R\ a\ b \rightarrow R\ a\ b' \rightarrow b = b' \end{array}  }.
```

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 : \mathtt{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 : \mathsf{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C),
RightUnique \ R \to RightUnique \ S \to RightUnique \ (Rcomp \ R \ S).
```

Składanie zachowuje oba rodzaje relacji unikalnych. Nie ma tu co za dużo filozofować — żeby się przekonać, narysuj obrazek. TODO.

```
\begin{split} & \text{Instance } LeftUnique\_Rinv: \\ & \forall \; (A \; B : \texttt{Type}) \; (R : hrel \; A \; B), \\ & \quad RightUnique \; R \rightarrow LeftUnique \; (Rinv \; R). \\ & \text{Instance } RightUnique\_Rinv: \\ & \forall \; (A \; B : \texttt{Type}) \; (R : hrel \; A \; B), \\ & \quad LeftUnique \; R \rightarrow RightUnique \; (Rinv \; R). \end{split}
```

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 : \texttt{Type}) \ (R \ S : hrel \ A \ B),
LeftUnique \ R \to LeftUnique \ (Rand \ R \ S).
Instance RightUnique\_Rand:
\forall (A \ B : \texttt{Type}) \ (R \ S : hrel \ A \ B),
RightUnique \ R \to RightUnique \ (Rand \ R \ S).
Lemma Ror\_not\_LeftUnique:
\exists (A \ B : \texttt{Type}) \ (R \ S : hrel \ A \ B),
LeftUnique \ R \land LeftUnique \ S \land \neg LeftUnique \ (Ror \ R \ S).
```

```
Lemma Ror\_not\_RightUnique:

\exists (A B : \texttt{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 : \texttt{Type}) \ (R : hrel \ A \ B),
LeftUnique \ R \land \neg \ LeftUnique \ (Rnot \ R).
Lemma Rnot\_not\_RightUnique:
\exists \ (A \ B : \texttt{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

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_{-}eg:
  \forall A : \mathsf{Type}, \ LeftTotal \ (@eq \ A).
Instance RightTotal\_eq:
  \forall A : \mathsf{Type}, RightTotal (@eq A).
   Równość jest relacją totalną, gdyż każdy term x:A jest równy samemu sobie.
Instance LeftTotal\_Rcomp:
  \forall (A \ B \ C : \mathsf{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C),
     LeftTotal\ R \to LeftTotal\ S \to LeftTotal\ (Rcomp\ R\ S).
Instance RightTotal\_Rcomp:
  \forall (A \ B \ C : \mathsf{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C),
     RightTotal\ R 	o RightTotal\ S 	o RightTotal\ (Rcomp\ R\ S).
Instance RightTotal\_Rinv:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     LeftTotal R \rightarrow RightTotal (Rinv R).
Instance LeftTotal\_Rinv:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     RightTotal\ R \rightarrow 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:
\exists \ (A \ B : \texttt{Type}) \ (R \ S : hrel \ A \ B),
LeftTotal \ R \land LeftTotal \ S \land \neg LeftTotal \ (Rand \ R \ S).
Lemma Rand\_not\_RightTotal:
\exists \ (A \ B : \texttt{Type}) \ (R \ S : hrel \ A \ B),
RightTotal \ R \land RightTotal \ S \land \neg RightTotal \ (Rand \ R \ S).
Lemma LeftTotal\_Ror:
\forall \ (A \ B : \texttt{Type}) \ (R \ S : hrel \ A \ B),
LeftTotal \ R \rightarrow LeftTotal \ (Ror \ R \ S).
Lemma RightTotal\_Ror:
\forall \ (A \ B : \texttt{Type}) \ (R \ S : hrel \ A \ B),
Lemma \ RightTotal\_Ror:
```

```
RightTotal\ R \rightarrow 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:
\exists \ (A \ B : \texttt{Type}) \ (R : hrel \ A \ B),
RightTotal \ R \land \neg \ RightTotal \ (Rnot \ R).
Lemma Rnot\_not\_RightTotal:
\exists \ (A \ B : \texttt{Type}) \ (R : hrel \ A \ B),
RightTotal \ R \land \neg \ 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.

12.5 Rodzaje relacji heterogenicznych v2

Poznawszy cztery właściwości, jakie relacje mogą posiadać, rozważymy teraz relacje, które posiadają dwie lub więcej z tych właściwości.

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: \mathtt{Type}\}\ (f:A\to 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 : \mathsf{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 : \mathtt{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C), Functional R \to Functional \ S \to 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 : \mathsf{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 : \mathsf{Type}) \ (R \ S : \mathit{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 : \mathtt{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.

Injective $R \wedge \neg Injective (Rnot R)$.

```
Class Injective \{A B : Type\} (R : hrel A B) : Prop :=
     I_{-}Fun :> Functional R;
     I_{-}LU :> LeftUnique R;
}.
Instance inj_to_Injective :
  \forall (A B : \mathsf{Type}) (f : A \to B),
     injective f \to Injective (fun (a : A) (b : B) \Rightarrow f (a = b)).
    Relacje funkcyjne, które są lewostronnie unikalne, odpowiadają funkcjom injektywnym.
Instance Injective_eq:
  \forall A : \mathsf{Type}, \ \mathit{Injective} \ (@eq \ A).
Instance Injective\_Rcomp:
  \forall (A \ B \ C : \mathsf{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C),
     Injective R \to Injective S \to Injective (Rcomp R S).
Lemma Rinv_not_Injective:
  \exists (A B : \mathsf{Type}) (R : hrel A B),
     Injective R \wedge \neg Injective (Rinv R).
Lemma Rand\_not\_Injective:
  \exists (A B : \mathsf{Type}) (R S : hrel A B),
     Injective R \wedge Injective S \wedge \neg Injective (Rand R S).
Lemma Ror\_not\_Injective:
  \exists (A B : \mathsf{Type}) (R S : hrel A B),
     Injective R \wedge Injective S \wedge \neg Injective (Ror R S).
Lemma Rnot\_not\_Injective:
  \exists (A B : \mathsf{Type}) (R : hrel A B),
```

Właściwości relacji injektywnych są takie, jak funkcji injektywnych, gdyż te pojęcia ściśle sobie odpowiadają.

```
Cwiczenie Udowodnij, że powyższe zdanie nie kłamie.
Lemma injective\_Injective:
  \forall (A B : \mathsf{Type}) (f : A \to B),
     injective f \leftrightarrow Injective (fun (a : A) (b : B) \Rightarrow f (a = b)).
Proof.
  split.
     compute; intros. repeat split; intros.
       \exists (f \ a). reflexivity.
       rewrite \leftarrow H0, \leftarrow H1. reflexivity.
        apply H. rewrite H\theta, H1. reflexivity.
     destruct 1 as [[[]]] []]. red. intros.
        apply left\_unique\theta with (f x').
          assumption.
          reflexivity.
Qed.
    end hide *)
Class Surjective \{A \mid B : \mathsf{Type}\}\ (R : hrel \mid A \mid B) : \mathsf{Prop} :=
     S_Fun :> Functional R;
     S_{-}RT :> RightTotal R;
}.
Instance sur\_to\_Surjective:
  \forall (A B : \mathsf{Type}) (f : A \to B),
     surjective f \to Surjective (fun (a : A) (b : B) \Rightarrow f (a = b)).
   Relacje funkcyjne, które są prawostronnie totalne, odpowiadają funkcjom surjektywnym.
Instance Surjective\_eq:
  \forall A : \mathsf{Type}, Surjective (@eq A).
Instance Surjective\_Rcomp:
  \forall (A \ B \ C : \mathsf{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C),
     Surjective R \to Surjective \ S \to Surjective \ (Rcomp \ R \ S).
Lemma Rinv\_not\_Surjective:
  \exists (A B : \mathsf{Type}) (R : hrel A B),
     Surjective R \wedge \neg Surjective (Rinv R).
Lemma Rand\_not\_Surjective:
  \exists (A B : \mathsf{Type}) (R S : hrel A B),
     Surjective R \wedge Surjective S \wedge \neg Surjective (Rand R S).
Lemma Ror\_not\_Surjective:
  \exists (A B : \mathsf{Type}) (R S : hrel A B),
     Surjective R \wedge Surjective S \wedge \neg Surjective (Ror R S).
Lemma Rnot\_not\_Surjective:
```

```
\exists (A B : \mathsf{Type}) (R : hrel A B),
     Surjective R \wedge \neg 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 :
  \forall (A B : \mathsf{Type}) (f : A \to B),
     bijective f \to Bijective (fun (a:A) (b:B) \Rightarrow f \ a=b).
   Relacje funkcyjne, które są lewostronnie totalne (czyli injektywne) oraz prawostronnie
totalne (czyli surjektywne), odpowiadają bijekcjom.
Instance Bijective\_eq:
  \forall A : \mathsf{Type}, Bijective (@eq A).
Instance Bijective\_Rcomp:
  \forall (A \ B \ C : \mathsf{Type}) \ (R : hrel \ A \ B) \ (S : hrel \ B \ C),
     Bijective R \to Bijective S \to Bijective (Rcomp R S).
Instance Bijective\_Rinv:
  \forall (A B : \mathsf{Type}) (R : hrel A B),
     Bijective R \rightarrow Bijective (Rinv R).
Lemma Rand\_not\_Bijective:
  \exists (A B : \mathsf{Type}) (R S : hrel A B),
     Bijective R \wedge Bijective \ S \wedge \neg Bijective \ (Rand \ R \ S).
Lemma Ror\_not\_Bijective:
  \exists (A B : \mathsf{Type}) (R S : hrel A B),
     Bijective R \wedge Bijective S \wedge \neg Bijective (Ror R S).
Lemma Rnot\_not\_Bijective:
  \exists (A B : \mathsf{Type}) (R : hrel A B),
     Bijective R \wedge \neg 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.

12.6 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"

12.6.1 Zwrotność

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: \forall 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 : \mathtt{Type}\} : Reflexive (@eq A).
Instance Reflexive\_RTrue :
\forall A : \mathtt{Type}, Reflexive (@RTrue A A).
```

```
Lemma RFalse\_nonempty\_not\_Reflexive: \forall A : Type, A \rightarrow \neg 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 $eq_subrelation_Reflexive$:

```
\forall (A: \mathtt{Type}) (R: rel \ A), Reflexive \ R \rightarrow 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: \forall (A: \texttt{Type}) \ (R \ S: rel \ A), Reflexive \ R \to Reflexive \ S \to Reflexive \ (Rcomp \ R \ S). Instance Reflexive\_Rinv: \forall (A: \texttt{Type}) \ (R: rel \ A), Reflexive \ R \to Reflexive \ (Rinv \ R). Instance Reflexive\_Rand: \forall (A: \texttt{Type}) \ (R \ S: rel \ A), Reflexive \ R \to Reflexive \ S \to Reflexive \ (Rand \ R \ S). Instance Reflexive\_Ror: \forall (A: \texttt{Type}) \ (R \ S: rel \ A), Reflexive \ R \to 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: \forall (A : Type), Antireflexive (fun <math>x y : A \Rightarrow x \neq 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:
\forall R: rel\ Empty\_set,\ Antireflexive\ R.

Lemma eq\_nonempty\_not\_Antireflexive:
\forall\ A: {\tt Type},\ A \to \neg\ Antireflexive\ (@eq\ A).

Lemma RTrue\_nonempty\_not\_Antireflexive:
\forall\ A: {\tt Type},\ A \to \neg\ Antireflexive\ (@RTrue\ A\ A).

Instance Antireflexive\_RFalse:
\forall\ A: {\tt 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:
\exists (A: \mathsf{Type}) (R \ S: rel \ A),
Antireflexive \ R \land Antireflexive \ S \land
\neg Antireflexive \ (Rcomp \ R \ S).

Instance Antireflexive\_Rinv:
\forall (A: \mathsf{Type}) \ (R: rel \ A),
Antireflexive \ R \rightarrow Antireflexive \ (Rinv \ R).

Instance Antireflexive\_Rand:
\forall (A: \mathsf{Type}) \ (R \ S: rel \ A),
Antireflexive \ R \rightarrow Antireflexive \ (Rand \ R \ S).

Instance Antireflexive\_Ror:
\forall (A: \mathsf{Type}) \ (R \ S: rel \ A),
Antireflexive \ R \rightarrow Antireflexive \ S \rightarrow 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: \forall A: Type, A \rightarrow 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:
\forall R: rel\ Empty\_set, \neg\ Irreflexive\ R.
Lemma eq\_empty\_not\_Irreflexive:
\neg\ Irreflexive\ (@eq\ Empty\_set).
Lemma eq\_nonempty\_not\_Irreflexive:
\forall\ A: {\tt Type},\ A \rightarrow \neg\ 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:
\neg Irreflexive (@RTrue Empty\_set Empty\_set).
Lemma RTrue\_nonempty\_not\_Irreflexive:
\forall A: \texttt{Type}, A \rightarrow \neg Irreflexive (@RTrue A A).
Lemma RFalse\_empty\_not\_Irreflexive:
\neg Irreflexive (@RFalse Empty\_set Empty\_set).
Instance Irreflexive\_RFalse\_nonempty:
\forall A: \texttt{Type}, A \rightarrow 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:
\exists (A : \mathsf{Type}) \ (R \ S : rel \ A),
Irreflexive \ R \land Irreflexive \ S \land \neg 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: \forall (A: \mathsf{Type}) \ (R: rel \ A), Irreflexive \ R \to Irreflexive \ (Rinv \ R). Instance Irreflexive\_Rand: \forall (A: \mathsf{Type}) \ (R \ S: rel \ A), Irreflexive \ R \to Irreflexive \ (Rand \ R \ S). Lemma Ror\_not\_Irreflexive: \exists \ (A: \mathsf{Type}) \ (R \ S: rel \ A), Irreflexive \ R \land Irreflexive \ S \land \neg 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, anty-zwrotnością i niezwrotnością.

```
\begin{split} & \text{Instance } Reflexive\_Rnot: \\ & \forall \; (A: \texttt{Type}) \; (R: rel \; A), \\ & \quad Antireflexive \; R \rightarrow Reflexive \; (Rnot \; R). \\ & \text{Instance } Antireflexive\_Rnot: \\ & \forall \; (A: \texttt{Type}) \; (R: rel \; A), \\ & \quad Reflexive \; R \rightarrow Antireflexive \; (Rnot \; R). \end{split}
```

Podstawowa zależność między nimi jest taka, że negacja relacji zwrotnej jest antyzwrotna, zaś negacja relacji antyzwrotnej jest zwrotna.

```
Lemma Reflexive\_Antireflexive\_empty:

\forall R : rel \ Empty\_set, \ Reflexive \ R \land Antireflexive \ R.

Lemma Reflexive\_Antireflexive\_nonempty:

\forall (A : \texttt{Type}) \ (R : rel \ A),
```

 $A \to Reflexive R \to Antireflexive R \to False.$

Każda relacja na typie pustym jest jednocześnie zwrotna i antyzwrotna, ale nie może taka być żadna relacja na typie niepustym.

```
Instance Irreflexive\_nonempty\_Antireflexive: \forall (A: Type) (R: rel A), A \rightarrow Antireflexive R \rightarrow Irreflexive R.
```

Związek między niezwrotnością i antyzwrotnością jest nadzwyczaj prosty: każda relacja antyzwrotna na typie niepustym jest też niezwrotna.

12.6.2 Symetria

```
 \begin{array}{l} \texttt{Class} \ Symmetric \ \{A : \texttt{Type}\} \ (R : rel \ A) : \texttt{Prop} := \\ \{ \\ symmetric : \forall \ x \ y : A, \ R \ x \ y \rightarrow R \ y \ x \\ \}. \\ \\ \texttt{Class} \ Antisymmetric \ \{A : \texttt{Type}\} \ (R : rel \ A) : \texttt{Prop} := \\ \{ \\ antisymmetric : \forall \ x \ y : A, \ R \ x \ y \rightarrow \neg R \ y \ x \\ \}. \\ \\ \texttt{Class} \ Asymmetric : \exists \ x \ y : A, \ R \ x \ y \land \neg R \ y \ x \\ \}. \\ \end{aligned}
```

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:
  \forall (A : \mathsf{Type}) (R : rel A),
     Symmetric\ R \leftrightarrow same\_hrel\ (Rinv\ R)\ R.
    Alterntywną charakteryzacją symetrii może być stwierdzenie, że relacja symetryczna to
taka, która jest swoją własną odwrotnością.
Instance Symmetric\_eq:
  \forall A : \mathsf{Type}, \, Symmetric \, (@eq A).
Instance Symmetric\_RTrue:
  \forall A : \mathsf{Type}, \, Symmetric \, (@RTrue \, A \, A).
Instance Symmetric\_RFalse:
  \forall A : \mathsf{Type}, \, Symmetric \, (@RFalse \, A \, A).
   Równość, relacja pełna i pusta są symetryczne.
Lemma Rcomp\_not\_Symmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     Symmetric R \wedge Symmetric S \wedge \neg Symmetric (Rcomp R S).
   Złożenie relacji symetrycznych nie musi być symetryczne.
Instance Symmetric\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
     Symmetric R \to Symmetric (Rinv R).
Instance Symmetric\_Rand:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     Symmetric R \to Symmetric S \to Symmetric (Rand R S).
Instance Symmetric\_Ror:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     Symmetric\ R \rightarrow Symmetric\ S \rightarrow Symmetric\ (Ror\ R\ S).
Instance Symmetric\_Rnot:
  \forall (A : \mathsf{Type}) (R : rel A),
     Symmetric\ R \rightarrow 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. Spora klasę przykładów stanowią różne relacje służące
do porównywania: "x jest wyższy od y", "x jest silniejszy od y", "x jest bogatszy od y".
```

Lemma $Antisymmetric_empty$:

 $\forall R : rel \ Empty_set, \ Antisymmetric \ R.$

Lemma $eq_nonempty_not_Antisymmetric$:

```
\forall A : \mathsf{Type}, A \to \neg Antisymmetric (@eq A).
Lemma RTrue\_nonempty\_not\_Antisymmetric:
  \forall A : \mathsf{Type}, A \to \neg Antisymmetric (@RTrue A A).
Instance RFalse\_Antiymmetric:
  \forall A : \mathsf{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 antysyme-
tryczna, gdyż przesłanka R x y występująca w definicji antysymetrii jest zawsze fałszywa.
Lemma Rcomp\_not\_Antisymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     Antisymmetric \ R \land Antisymmetric \ S \land
     \neg Antisymmetric (Rcomp R S).
Instance Antisymmetric\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
     Antisymmetric\ R \to Antisymmetric\ (Rinv\ R).
Instance Antisymmetric\_Rand:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     Antisymmetric R \to Antisymmetric (Rand R S).
Lemma Ror\_not\_Antisymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     Antisymmetric\ R\ \land\ Antisymmetric\ S\ \land
     \neg Antisymmetric (Ror R S).
Lemma Rnot\_not\_Antisymmetric:
  \exists (A : \mathsf{Type}) (R : rel A),
     Antisymmetric\ R \land \neg\ Antisymmetric\ (Rnot\ R).
Lemma empty\_not\_Asymmetric:
  \forall R : rel \ Empty\_set, \neg \ Asymmetric \ R.
Lemma Rcomp\_not\_Asymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     Asymmetric R \wedge Asymmetric S \wedge \neg Asymmetric (Rcomp R S).
Instance Asymmetric\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
     Asymmetric\ R \rightarrow Asymmetric\ (Rinv\ R).
Lemma Rand\_not\_Asymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     Asymmetric R \wedge Asymmetric S \wedge \neg Asymmetric (Rand R S).
Lemma Ror\_not\_Asymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
```

Asymmetric $R \wedge Asymmetric S \wedge \neg Asymmetric (Ror R S)$.

12.6.3 Przechodniość

```
Class Transitive \{A : Type\} (R : rel A) : Prop :=
      transitive: \forall x \ y \ z: A, R \ x \ y \rightarrow R \ y \ z \rightarrow R \ x \ z
}.
Instance Transitive\_eq:
  \forall A : \mathsf{Type}, \ \mathit{Transitive} \ (@\mathit{eq} \ A).
Lemma Rcomp\_not\_Transitive:
   \exists (A : \mathsf{Type}) (R \ S : rel \ A),
      Transitive R \wedge Transitive S \wedge \neg Transitive (Rcomp R S).
Instance Transitive\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
      Transitive R \to Transitive (Rinv R).
Instance Transitive\_Rand:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
      Transitive R \to Transitive S \to Transitive (Rand R S).
Lemma Ror\_not\_Transitive:
   \exists (A : \mathsf{Type}) (R \ S : rel \ A),
      Transitive R \wedge Transitive S \wedge \neg Transitive (Ror R S).
Lemma Rnot\_not\_Transitive:
   \exists (A : \mathsf{Type}) (R : rel A),
      Transitive R \wedge \neg Transitive (Rnot R).
12.6.4
             Inne
Class Total \{A : \mathsf{Type}\} (R : rel A) : \mathsf{Prop} :=
      total: \forall x y: A, R x y \lor R y x
}.
Instance Total\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
      Total \ R \rightarrow Total \ (Rinv \ R).
Instance Total\_Ror:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
      Total \ R \rightarrow Total \ S \rightarrow Total \ (Ror \ R \ S).
Lemma Rnot\_not\_Total:
   \exists (A : \mathsf{Type}) (R : rel A),
      Total R \wedge \neg Total (Rnot R).
```

```
Instance Total\_Reflexive:
  \forall (A : \mathsf{Type}) (R : rel A),
      Total R \to Reflexive R.
Class Trichotomous \{A : \mathtt{Type}\} (R : rel A) : \mathtt{Prop} :=
     trichotomous: \forall x y: A, R x y \lor x = y \lor R y x
}.
Instance Trichotomous_empty :
  \forall R : rel \ Empty\_set, \ Trichotomous \ R.
Instance Trichotomous_eq_singleton :
  \forall A : \mathsf{Type}, (\forall x \ y : A, x = y) \rightarrow \mathit{Trichotomous} \ (@eq \ A).
Instance Total\_Trichotomous:
  \forall (A : \mathsf{Type}) (R : rel A),
      Total R \rightarrow Trichotomous R.
Lemma eq_not_Trichotomous:
  \exists A : \mathsf{Type}, \neg \mathit{Trichotomous} \ (@\mathit{eq} \ A).
Instance Trichotomous\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
      Trichotomous R \rightarrow Trichotomous (Rinv R).
Lemma Rnot\_not\_Trichotomous:
  \exists (A : \mathsf{Type}) (R : rel A),
      Trichotomous\ R \land \neg\ Trichotomous\ (Rnot\ R).
Class Dense \{A : Type\} (R : rel A) : Prop :=
     dense: \forall x y: A, R x y \rightarrow \exists z: A, R x z \land R z y
}.
Instance Dense\_eq:
  \forall A : \mathsf{Type}, \ Dense \ (@eq \ A).
Instance Dense\_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
     Dense R \to Dense (Rinv R).
Instance Dense\_Ror:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     Dense R \to Dense S \to Dense (Ror R S).
```

12.7 Relacje równoważności

```
Class Equivalence \{A : \mathtt{Type}\} (R : rel A) : \mathtt{Prop} :=
```

12.8 Słabe relacje homogeniczne

```
Class WeakAntisymmetric \{A : Type\} (R : rel A) : Prop :=
     wantisymmetric: \forall x y : A, R x y \rightarrow R y x \rightarrow x = y
}.
Instance WeakAntisymmetric_eq :
  \forall A : \mathsf{Type}, \ WeakAntisymmetric \ (@eq A).
Lemma Rcomp\_not\_WeakAntisymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     WeakAntisymmetric\ R\ \land\ WeakAntisymmetric\ S\ \land
       \neg WeakAntisymmetric (Rcomp R S).
Instance WeakAntisymmetric_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
     WeakAntisymmetric\ R \rightarrow WeakAntisymmetric\ (Rinv\ R).
Instance WeakAntisymmetric_Rand :
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     WeakAntisymmetric R \rightarrow WeakAntisymmetric S \rightarrow
        WeakAntisymmetric (Rand R S).
{\tt Lemma}\ Ror\_not\_WeakAntisymmetric:
  \exists (A : \mathsf{Type}) (R \ S : rel \ A),
     WeakAntisymmetric\ R \land WeakAntisymmetric\ S \land
       \neg WeakAntisymmetric (Ror R S).
Lemma Rnot\_not\_WeakAntisymmetric:
```

```
\exists (A : \mathsf{Type}) (R : rel A),
     WeakAntisymmetric\ R \land \neg\ WeakAntisymmetric\ (Rnot\ R).
Class WeakAntisymmetric' \{A : Type\} \{E : rel A\}
  (H:Equivalence\ E)\ (R:rel\ A): \texttt{Prop}:=
     wasym: \forall x y: A, R x y \rightarrow R y x \rightarrow E x y
}.
Instance WeakAntisymmetric\_equiv:
  \forall (A : \mathsf{Type}) (E : rel A) (H : Equivalence E),
     WeakAntisymmetric' H E.
Lemma Rcomp\_not\_WeakAntisymmetric':
  \exists (A : \mathsf{Type}) (E \ R \ S : rel \ A), \forall H : Equivalence \ E,
     WeakAntisymmetric' \ H \ R \land WeakAntisymmetric' \ H \ S \land
        \neg WeakAntisymmetric' H (Rcomp R S).
Instance WeakAntisymmetric'\_Rinv:
  \forall (A : \mathsf{Type}) (E : rel \ A) (H : Equivalence \ E) (R : rel \ A),
     WeakAntisymmetric' \ H \ R \rightarrow WeakAntisymmetric' \ H \ (Rinv \ R).
Instance WeakAntisymmetric'_Rand:
  \forall (A : \mathsf{Type}) (E : rel A) (H : Equivalence E) (R S : rel A),
     WeakAntisymmetric' \ H \ R \rightarrow WeakAntisymmetric' \ H \ S \rightarrow
        WeakAntisymmetric' H (Rand R S).
Lemma Ror\_not\_WeakAntisymmetric':
  \exists (A : \mathsf{Type}) (E \ R \ S : rel \ A), \ \forall \ H : Equivalence \ E,
     WeakAntisymmetric' H R \wedge WeakAntisymmetric' H S \wedge
        \neg WeakAntisymmetric' H (Ror R S).
Lemma Rnot\_not\_WeakAntisymmetric':
  \exists (A : \mathsf{Type}) (E \ R : rel \ A), \forall H : Equivalence \ E,
     WeakAntisymmetric' \ H \ R \land \neg \ WeakAntisymmetric' \ H \ (Rnot \ R).
```

12.9 Złożone relacje homogeniczne

```
PartialOrder\_WeakAntisymmetric :> WeakAntisymmetric R;
}.
Class TotalOrder \{A : \mathsf{Type}\} (R : rel A) : \mathsf{Prop} :=
     TotalOrder\_PartialOrder :> PartialOrder R;
     TotalOrder\_Total : Total R;
}.
Class StrictPreorder \{A : \mathsf{Type}\}\ (R : rel\ A) : \mathsf{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: {\tt Type}\}\ (R:rel\ A): {\tt Prop}:=
    StrictTotalOrder\_PartialOrder:>StrictPartialOrder\ R;
    StrictTotalOrder\_Total : Total R;
}.
            Domknięcia
12.10
```

```
Inductive refl\_clos\ \{A: \mathtt{Type}\}\ (R: rel\ A): rel\ A:= |\ rc\_step: \forall\ x\ y:\ A,\ R\ x\ y \rightarrow refl\_clos\ R\ x\ y |\ rc\_refl: \forall\ x:\ A,\ refl\_clos\ R\ x\ x. Instance Reflexive\_rc: \ \forall\ (A: \mathtt{Type})\ (R: rel\ A),\ Reflexive\ (refl\_clos\ R). Lemma subrelation\_rc: \ \forall\ (A: \mathtt{Type})\ (R: rel\ A),\ subrelation\ R\ (refl\_clos\ R). Lemma rc\_smallest: \ \forall\ (A: \mathtt{Type})\ (R\ S: rel\ A),\ subrelation\ R\ S \rightarrow Reflexive\ S \rightarrow subrelation\ (refl\_clos\ R)\ S. Lemma rc\_idempotent: \ \forall\ (A: \mathtt{Type})\ (R: rel\ A),\ refl\_clos\ (refl\_clos\ R) <-> refl\_clos\ R. Lemma rc\_Rinv:
```

```
\forall (A : \mathsf{Type}) (R : rel A),
      Rinv \ (refl\_clos \ (Rinv \ R)) <-> \ refl\_clos \ R.
Inductive symm\_clos \{A : Type\} (R : rel A) : rel A :=
      |sc\_step:
           \forall x y : A, R x y \rightarrow symm\_clos R x y
     |sc\_symm:
           \forall x \ y : A, symm\_clos \ R \ x \ y \rightarrow symm\_clos \ R \ y \ x.
Instance Symmetric\_sc:
  \forall (A : \mathsf{Type}) (R : rel A), Symmetric (symm_clos R).
Lemma subrelation\_sc:
  \forall (A : \mathsf{Type}) (R : rel A), subrelation R (symm\_clos R).
Lemma sc\_smallest:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     subrelation R S \to Symmetric S \to subrelation (symm_clos R) S.
Lemma sc\_idempotent:
  \forall (A : \mathsf{Type}) (R : rel A),
     symm\_clos (symm\_clos R) <-> symm\_clos R.
Lemma sc_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
      Rinv \; (symm\_clos \; (Rinv \; R)) < -> symm\_clos \; R.
Inductive trans\_clos \{A : Type\} (R : rel A) : rel A :=
      | tc\_step :
           \forall x y : A, R x y \rightarrow trans\_clos R x y
     | tc\_trans :
           \forall x \ y \ z : A,
              trans\_clos \ R \ x \ y \rightarrow trans\_clos \ R \ y \ z \rightarrow trans\_clos \ R \ x \ z.
Instance Transitive\_tc:
  \forall (A : \mathsf{Type}) (R : rel A), Transitive (trans\_clos R).
Lemma subrelation\_tc:
  \forall (A : \mathsf{Type}) (R : rel A), subrelation R (trans\_clos R).
Lemma tc\_smallest:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     subrelation R S \rightarrow Transitive S \rightarrow
        subrelation (trans\_clos\ R) S.
Lemma tc\_idempotent:
  \forall (A : \mathsf{Type}) (R : rel A),
     trans\_clos (trans\_clos R) <-> trans\_clos R.
Lemma tc-Rinv:
```

```
\forall (A : \mathsf{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
  \mid ec\_reft:
        \forall x: A, equiv\_clos R x x
  \mid 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 : \mathsf{Type}) (R : rel A), Equivalence (equiv\_clos R).
Lemma subrelation\_ec:
  \forall (A : \mathsf{Type}) (R : rel A), subrelation R (equiv\_clos R).
Lemma ec\_smallest:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
     subrelation R S \rightarrow Equivalence S \rightarrow
        subrelation (equiv_clos R) S.
Lemma ec_{-}idempotent:
  \forall (A : \mathsf{Type}) (R : rel A),
      equiv\_clos \ (equiv\_clos \ R) <-> equiv\_clos \ R.
Lemma ec_Rinv:
  \forall (A : \mathsf{Type}) (R : rel A),
     Rinv (equiv\_clos (Rinv R)) <-> equiv\_clos R.
Domknięcie zwrotnosymetryczne
Definition rsc \{A : \mathsf{Type}\} (R : rel A) : rel A :=
  refl\_clos\ (symm\_clos\ R).
Instance Reflexive\_rsc:
  \forall (A : \mathsf{Type}) (R : rel A), Reflexive (rsc R).
Instance Symmetric\_rsc:
  \forall (A : \mathsf{Type}) (R : rel \ A), Symmetric (rsc \ R).
{\tt Lemma}\ subrelation\_rsc:
  \forall (A : \mathsf{Type}) (R : rel \ A), subrelation \ R (rsc \ R).
Lemma rsc\_smallest:
  \forall (A : \mathsf{Type}) (R \ S : rel \ A),
```

```
subrelation \ R \ S \rightarrow Reflexive \ S \rightarrow Symmetric \ S \rightarrow subrelation \ (rsc \ R) \ S. Lemma rsc\_idempotent: \forall \ (A: \texttt{Type}) \ (R: rel \ A), \\ rsc \ (rsc \ R) <-> \ rsc \ R. Lemma rsc\_Rinv: \forall \ (A: \texttt{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 :
\forall (A : Type) (R : rel A),
Equivalence (trans\_clos (symm\_clos (refl\_clos R))).

Lemma subrelation\_rstc :
\forall (A : Type) (R : rel A), subrelation R (rstc R).

Lemma rstc\_smallest :
\forall (A : Type) (R S : rel A),
subrelation R S \rightarrow Equivalence S \rightarrow subrelation (rstc R) S.
```

12.11 Redukcje

```
 \begin{array}{l} \text{Definition } rr \; \{A: \texttt{Type}\} \; (R: rel \; A): rel \; A:= \\ \text{fun } x \; y: A \Rightarrow R \; x \; y \wedge x \neq y. \\ \\ \text{Instance } Antireflexive\_rr: \\ \forall \; (A: \texttt{Type}) \; (R: rel \; A), \; Antireflexive \; (rr \; R). \\ \\ \text{Lemma } rr\_rc: \\ \forall \; (A: \texttt{Type}) \; (R: rel \; A), \\ & \; Reflexive \; R \rightarrow refl\_clos \; (rr \; R) <-> \; R. \\ \end{array}
```

Rozdział 13

F1: Koindukcja i korekursja

```
Require Import List.

Import ListNotations.

Require Import FunctionalExtensionality.

Require Import Setoid.

Require Import FinFun.

Require Import NArith.

Require Import Div2.

Require Import ZArith.
```

13.1 Koindukcja (TODO)

13.1.1 Strumienie (TODO)

```
CoInductive Stream\ (A: {\tt Type}): {\tt Type}:= \{ \\ hd: A; \\ tl: Stream\ A; \}. \\ Arguments\ hd\ \{A\}. \\ Arguments\ tl\ \{A\}. \\ {\tt CoInductive}\ bisim\ \{A: {\tt Type}\}\ (s1\ s2: Stream\ A): {\tt Prop}:= \{ \\ hds: hd\ s1=hd\ s2; \\ tls: bisim\ (tl\ s1)\ (tl\ s2); \}. \\ {\tt Lemma}\ bisim\_refl: \\ \forall\ (A: {\tt Type})\ (s: Stream\ A),\ bisim\ s\ s. \\ \\ {\tt Stream}\ A) = (s)
```

```
Proof.
  cofix CH. constructor; auto.
Qed.
Lemma bisim\_sym:
  \forall (A : \mathsf{Type}) (s1 \ s2 : Stream \ A),
     bisim\ s1\ s2 \rightarrow bisim\ s2\ s1.
Proof.
  cofix CH.
  destruct 1 as [hds \ tls]. constructor; auto.
Qed.
Lemma bisim\_trans:
  \forall (A : \mathsf{Type}) (s1 \ s2 \ s3 : Stream \ A),
     bisim\ s1\ s2 \rightarrow bisim\ s2\ s3 \rightarrow bisim\ s1\ s3.
Proof.
  cofix \ CH.
  destruct 1 as [hds1 \ tls1], 1 as [hds2 \ tls2].
  constructor; eauto. rewrite hds1. assumption.
Qed.
Jakieś pierdoły
CoFixpoint from'(n:nat): Stream\ nat:=
\{ \mid
     hd := n;
     tl := from'(S n);
|}.
{	t CoFixpoint}\ facts'\ (r\ n:nat):Stream\ nat:=
\{ \mid
     hd := r;
     tl := facts' (r \times S \ n) (S \ n);
|}.
Definition facts: Stream \ nat := facts' \ 1 \ 0.
(* Compute stake 9 facts.*)
Z manuala Agdy
hd (evens s) := hd s; tl (evens s) := evens (tl (tl s));
CoFixpoint evens \{A : \mathsf{Type}\}\ (s : Stream\ A) : Stream\ A :=
\{ \mid
     hd := hd s;
     tl := evens (tl (tl s));
```

```
|}.
CoFixpoint odds \{A : Type\} (s : Stream A) : Stream A :=
\{ \mid
     hd := hd (tl s);
     tl := odds (tl (tl s));
]}.
Definition split \{A : \mathsf{Type}\}\ (s : \mathit{Stream}\ A) : \mathit{Stream}\ A \times \mathit{Stream}\ A :=
  (evens s, odds s).
CoFixpoint merge \{A : \mathsf{Type}\}\ (ss : Stream\ A \times Stream\ A) : Stream\ A :=
     hd := hd (fst ss);
     tl := merge (snd ss, tl (fst ss));
{\tt Lemma}\ merge\_split:
  \forall (A : \mathsf{Type}) (s : Stream \ A),
     bisim (merge (split s)) s.
Proof.
  cofix \ CH.
  intros. constructor.
     cbn. reflexivity.
     cbn. constructor.
        cbn. reflexivity.
        cbn. apply CH.
Qed.
Bijekcja między strumieniami i funkcjami
Instance Equiv\_bisim (A: Type): Equivalence (@bisim A).
Proof.
  split; red.
     apply bisim_refl.
     apply bisim\_sym.
     apply bisim\_trans.
Defined.
{\tt CoFixpoint}\ the Chosen One: Stream\ unit:=
\{ \mid
     hd := tt;
     tl := theChosenOne;
|}.
Lemma all\_chosen\_unit\_aux:
  \forall s : Stream \ unit, \ bisim \ s \ the Chosen One.
```

```
Proof.
  cofix \ CH.
  constructor.
     destruct (hd \ s). cbn. reflexivity.
     cbn. apply CH.
Qed.
Theorem all\_chosen\_unit:
  \forall x \ y : Stream \ unit, \ bisim \ x \ y.
Proof.
  intros.
  rewrite (all\_chosen\_unit\_aux\ x), (all\_chosen\_unit\_aux\ y).
  reflexivity.
Qed.
Axiom bisim_{-}eq:
  \forall (A : \mathsf{Type}) (x \ y : \mathit{Stream} \ A), \ bisim \ x \ y \to x = y.
Theorem all_{-}eq:
  \forall x \ y : Stream \ unit, x = y.
Proof.
  intros. apply bisim_eq. apply all_chosen_unit.
Definition unit\_to\_stream\ (u:unit):Stream\ unit:=theChosenOne.
Definition stream\_to\_unit (s:Stream\ unit): unit:=tt.
Theorem unit\_is\_Stream\_unit:
  Bijective\ unit\_to\_stream.
Proof.
  red. \exists stream\_to\_unit.
  split; intros.
    destruct x; trivial.
     apply all_{-}eq.
Qed.
Trochę losowości
CoFixpoint rand (seed \ n1 \ n2 : Z) : Stream Z :=
\{ \mid
     hd := Zmod \ seed \ n2;
     tl := rand (Zmod seed n2 \times n1) n1 n2;
}.
CoFixpoint rand' (seed n1 n2 : Z) : Stream Z :=
\{ \mid
     hd := Zmod \ seed \ n2;
```

```
tl := rand (Zmod (seed \times n1) n2) n1 n2;
|}.
Fixpoint stake \{A : \mathsf{Type}\}\ (n : nat)\ (s : Stream\ A) : list\ A :=
match n with
     | 0 \Rightarrow []
     \mid S \mid n' \Rightarrow hd \mid s :: stake \mid n' \mid (tl \mid s)
end.
Compute stake 10 (rand 1 123456789 987654321).
Compute stake 10 (rand' 1235 234567890 6652).
            Kolisty
13.1.2
CoInductive Conat: Type :=
     pred : option Conat;
CoInductive coList (A : Type) : Type :=
     uncons: option (A \times coList A);
}.
Arguments uncons \{A\}.
Fixpoint tocoList \{A : Type\} (l : list A) : coList A :=
\{ \mid
     uncons :=
     match l with
          | | | \Rightarrow None
          | h :: t \Rightarrow Some (h, tocoList t)
     end
| \}.
Lemma tocoList_inj:
  \forall \{A : \mathtt{Set}\} (l1 \ l2 : list \ A),
     tocoList\ l1 = tocoList\ l2 \rightarrow l1 = l2.
Proof.
  induction l1 as [|h1 \ t1]; destruct l2 as [|h2 \ t2]; cbn; inversion 1.
     reflexivity.
     f_{equal.} apply IHt1. assumption.
Defined.
CoFixpoint from (n : nat) : coList nat :=
     uncons := Some (n, from (S n));
```

```
|}.
Definition lhead \{A : Type\} (l : coList A) : option A :=
match uncons l with
     | Some (a, \bot) \Rightarrow Some a
     | \_ \Rightarrow None
end.
Definition ltail \{A : Type\} (l : coList A) : option (coList A) :=
match uncons l with
      Some(_-, t) \Rightarrow Some t
     | \_ \Rightarrow None
end.
Fixpoint lnth \{A : Type\} (n : nat) (l : coList A) : option A :=
match n, uncons l with
     | \_, None \Rightarrow None
     \mid 0, Some (x, \bot) \Rightarrow Some x
     \mid S \mid n', Some (\_, l') \Rightarrow lnth \mid n' \mid l'
end.
Eval compute in lnth 511 (from 0).
Definition nats := from 0.
CoFixpoint repeat \{A : Type\}\ (x : A) : coList\ A :=
\{ \mid
     uncons := Some(x, \mathtt{repeat}\ x);
|}.
Eval cbn in lnth 123 (repeat 5).
CoFixpoint lapp \{A : \mathsf{Type}\} (l1 \ l2 : coList \ A) : coList \ A :=
match uncons 11 with
      None \Rightarrow l2
      Some (h, t) \Rightarrow \{ | uncons := Some (h, lapp t l2) | \}
end.
(*
   CoFixpoint general_omega {A : Set} (11 12 : coList A) : coList A :=
   match 11, 12 with
     | _, LNil => 11
     | LNil, LCons h' t' => LCons h' (general_omega t' 12)
     | LCons h t, _ => LCons h (general_omega t 12)
   end.
   *)
CoFixpoint lmap \{A \ B : \mathsf{Type}\}\ (f : A \to B)\ (l : coList\ A) : coList\ B :=
\{ \mid
     uncons :=
```

```
{\tt match}\ uncons\ l\ {\tt with}
                                        | None \Rightarrow None
                                        | Some (h, t) \Rightarrow Some (f h, lmap f t)
                    end
|}.
\texttt{Inductive} \ \textit{Finite} \ \{\textit{A}: \texttt{Type}\}: \ \textit{coList} \ \textit{A} \rightarrow \texttt{Prop}:=
                     | Finite\_nil : Finite \{ | uncons := None | \}
                    | Finite\_cons :
                                        \forall (h : A) (t : coList A),
                                                   Finite t \to Finite \{ | uncons := Some (h, t) | \}.
CoInductive Infinite \{A : \mathtt{Type}\} (l : coList A) : \mathtt{Prop} :=
                    h:A;
                    t: coList A;
                    p: uncons \ l = Some \ (h, t);
                    inf': Infinite t;
}.
Lemma empty\_not\_Infinite:
          \forall A : \mathsf{Type}, \neg Infinite \{ | uncons := @None (A \times coList A) | \}.
Proof.
          intros A \mid \mid \cdot \mid cbn \text{ in } p \cdot \mid \text{inversion } p \cdot \mid \cdot \mid \text{inversion } p \cdot \mid \cdot \mid \text{inversion } p \cdot \mid \text
Qed.
Lemma lmap\_Infinite:
         \forall (A B : \mathsf{Type}) (f : A \to B) (l : coList A),
                    Infinite l \to Infinite (lmap f l).
Proof.
          cofix \ CH.
          destruct 1. econstructor.
                     cbn. rewrite p. reflexivity.
                    apply CH. assumption.
Qed.
Lemma lapp\_Infinite\_l:
         \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
                    Infinite l1 \rightarrow Infinite (lapp l1 l2).
Proof.
          cofix CH.
          destruct 1. econstructor.
                    destruct l1; cbn in *; inversion p; cbn. reflexivity.
                    apply CH. assumption.
Qed.
Lemma lapp_Infinite_r:
```

```
\forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
     Infinite l2 \rightarrow Infinite (lapp l1 l2).
Proof.
  cofix CH.
  destruct l1 as [[[h \ t] \ ]]; intros.
     econstructor.
       cbn. reflexivity.
       apply CH. assumption.
    destruct H. econstructor.
       lazy. destruct l2; cbn in *. rewrite p. reflexivity.
       assumption.
Qed.
Lemma Finite\_not\_Infinite:
  \forall (A : \mathsf{Type}) (l : coList A),
     Finite l \rightarrow \neg Infinite l.
Proof.
  induction 1; intro.
     inversion H.\ cbn in p. inversion p.
     apply IHFinite. inversion H\theta; inversion p; subst. assumption.
Qed.
(*
   Lemma Infinite_not_Finite :
  forall (A : Type) (1 : coList A),
     Infinite 1 -> ~ Finite 1.
   Proof.
  induction 2.
     inversion H. inversion p.
     apply IHFinite. inversion H; inversion p; subst. assumption.
   Qed.
   *)
{\tt CoInductive}\,\,bisim2\,\,\{A:{\tt Type}\}\,\,(l1\ l2:\,coList\,\,A):{\tt Prop}:=
    bisim2':
       uncons \ l1 = None \land uncons \ l2 = None \lor
       \exists (h1:A) (t1:coList\ A) (h2:A) (t2:coList\ A),
         uncons l1 = Some(h1, t1) \land
         uncons \ l2 = Some \ (h2, t2) \land
            h1 = h2 \wedge bisim2 \ t1 \ t2
}.
Hint Constructors bisim2.
Lemma bisim2\_refl:
```

```
\forall (A : \mathsf{Type}) (l : coList A), bisim2 l l.
Proof.
  cofix CH.
  destruct l as [[[h \ t]]].
     constructor. right. \exists h, t, h, t; auto.
     constructor. left. cbn. split; reflexivity.
Qed.
Lemma bisim2\_symm:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
     bisim2 l1 l2 \rightarrow bisim2 l2 l1.
Proof.
  cofix CH.
  destruct 1 as [[[] | (h1 \& t1 \& h2 \& t2 \& p1 \& p2 \& p3 \& H)]].
     constructor. left. split; assumption.
     constructor. right. \exists h2, t2, h1, t1. auto.
Qed.
Lemma bisim2\_trans:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : coList \ A),
     bisim2 \ l1 \ l2 \rightarrow bisim2 \ l2 \ l3 \rightarrow bisim2 \ l1 \ l3.
Proof.
  cofix CH.
  destruct 1 as [[[] | (h11 \& t11 \& h12 \& t12 \& p11 \& p12 \& p13 \& H1)]],
              1 as [[[] | (h21 \& t21 \& h22 \& t22 \& p21 \& p22 \& p23 \& H2)]];
  subst.
     auto.
     rewrite H0 in p21. inversion p21.
    rewrite H in p12. inversion p12.
     rewrite p12 in p21; inversion p21; subst.
       econstructor. right. \exists h22, t11, h22, t22.
          do 3 try split; auto. eapply CH; eauto.
Qed.
Lemma lmap\_compose:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B) \ (g : B \to C) \ (l : coList \ A),
     bisim2 \ (lmap \ g \ (lmap \ f \ l)) \ (lmap \ (fun \ x \Rightarrow g \ (f \ x)) \ l).
Proof.
  cofix CH.
  constructor. destruct l as [[[h \ t]]]; [right | left]; cbn.
     \exists (g (f h)), (lmap g (lmap f t)),
              (g(f h)), (lmap (fun x \Rightarrow g(f x)) t).
       repeat (split; |reflexivity | idtac|). apply CH.
     do 2 split.
Qed.
```

```
Lemma bisim2\_Infinite:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
     bisim2\ l1\ l2 \rightarrow Infinite\ l1 \rightarrow Infinite\ l2.
Proof.
  \operatorname{cofix} \operatorname{CH}.
  destruct 1 as [[]] | (h1 \& t1 \& h2 \& t2 \& p1 \& p2 \& p3 \& H)]], 1.
     rewrite H in p. inversion p.
     econstructor.
        exact p2.
        rewrite p1 in p. inversion p; subst. eapply CH; eauto.
Qed.
13.1.3
            Drzewka
CoInductive coBTree (A : Type) : Type :=
     root: option (coBTree A \times A \times coBTree A)
Arguments \ root \{A\} _.
CoFixpoint fmap \{A \ B : Type\} \ (f : A \rightarrow B) \ (t : coBTree \ A) : coBTree \ B :=
\{ \mid
     root :=
     match root t with
          | None \Rightarrow None
          | Some (l, v, r) \Rightarrow Some (fmap f l, f v, fmap f r)
     end
|}.
CoFixpoint ns(n:nat): coBTree \ nat :=
\{ \mid
     root := Some (ns (1 + 2 \times n), n, ns (2 + 2 \times n))
|}.
Inductive BTree\ (A: {\tt Type}): {\tt Type} :=
      Empty: BTree A
     | Node : A \rightarrow BTree \ A \rightarrow BTree \ A \rightarrow BTree \ A.
Arguments Empty \{A\}.
Arguments Node \{A\} _ _ _.
Fixpoint ttake\ (n:nat)\ \{A: Type\}\ (t:coBTree\ A):BTree\ A:=
match n with
     \mid 0 \Rightarrow Empty
     \mid S \mid n' \Rightarrow
          match root t with
```

```
| None \Rightarrow Empty
                | Some (l, v, r) \Rightarrow Node v (ttake n' l) (ttake n' r)
          end
end.
Compute ttake \ 5 \ (ns \ 0).
CoInductive tsim \{A : Type\} (t1 \ t2 : coBTree \ A) : Prop :=
{
     tsim':
        root \ t1 = None \land root \ t2 = None \lor
       \exists (v1 \ v2 : A) (l1 \ l2 \ r1 \ r2 : coBTree A),
          root \ t1 = Some \ (l1, v1, r1) \land
          root \ t2 = Some \ (l2, v2, r2) \land
             tsim \ l1 \ l2 \wedge tsim \ r1 \ r2
}.
CoFixpoint mirror \{A : Type\} (t : coBTree A) : coBTree A :=
\{ \mid
     root :=
       match \ root \ t \ with
              None \Rightarrow None
              | Some (l, v, r) \Rightarrow Some (mirror r, v, mirror l) |
        end
|}.
Lemma tsim\_mirror\_inv:
  \forall (A : \mathsf{Type}) (t : coBTree \ A),
     tsim (mirror (mirror t)) t.
Proof.
  cofix \ CH.
  destruct t as [[[l \ v] \ r]]]; constructor; [right \ | \ left]. cbn.
     \exists v, v, (mirror (mirror l)), l, (mirror (mirror r)), r. auto.
     auto.
Qed.
            Rekursja ogólna
13.1.4
CoInductive Div(A : Type) : Type :=
     call: A + Div A
Fixpoint even(n:nat):bool:=
match n with
     \mid 0 \Rightarrow true
```

```
1 \Rightarrow false
      \mid S(S(n')) \Rightarrow even(n')
end.
(* The name is very unfortunate. *)
{	t CoFixpoint}\ collatz\ (n:nat):\ Div\ unit:=
      call :=
     match n with
            \mid 0 \mid 1 \Rightarrow inl \ tt
            \mid n' \Rightarrow
                  if even n'
                  then inr (collatz (div2 n'))
                  else inr (collatz (1 + 3 \times n'))
      end
]}.
Print Div.
Fixpoint fuel\ (n:nat)\ \{A: \mathsf{Type}\}\ (d:Div\ A):option\ A:=
match n, d with
      \mid 0, \bot \Rightarrow None
      | \_, Build\_Div \_ (inl \ a) \Rightarrow Some \ a
      \mid S \mid n', Build\_Div \_ (inr \mid d') \Rightarrow fuel \mid n' \mid d'
end.
Compute fuel \ 5 \ (collatz \ 4).
Arguments uncons \{A\} _.
{	t CoFixpoint}\ collatz'\ (n:nat):\ coList\ nat:=
match n with
      \mid 0 \Rightarrow \{\mid uncons := None \mid \}
       1 \Rightarrow \{ \mid uncons := Some \ (1, \{ \mid uncons := None \ | \}) \ | \}
      \mid n' \Rightarrow
            if even n'
            then \{ | uncons := Some (n', collatz' (div2 n')) | \}
            \texttt{else} \; \{ \mid \, uncons := Some \; (n', \, collatz' \; (1 + 3 \times n')) \mid \}
end.
Fixpoint take (n : nat) \{A : Type\} (l : coList A) : list A :=
match n, uncons l with
      \mid 0, \bot \Rightarrow []
      | \_, None \Rightarrow []
       S \ n', \ Some \ (h, \ t) \Rightarrow h :: take \ n' \ t
end.
Compute map (fun n: nat \Rightarrow take\ 200\ (collatz'\ n)) [30; 31; 32; 33].
```

```
Compute take 150 (collatz' 12344).
   insertion sort na kolistach
CoFixpoint ins (n : nat) (s : coList nat) : coList nat :=
\{ \mid
     uncons :=
       match uncons \ s with
             None \Rightarrow None
            | Some (h, t) \Rightarrow
                 if n <= ? h
                 then
                    Some (n, \{ | uncons := Some (h, t) | \})
                   Some (h, ins \ n \ t)
       end
|}.
(*
   CoFixpoint ss (s : coList nat) : coList nat :=
   \{ \mid
     uncons :=
       match uncons s with
             | None => None
             | Some (h, t) =>
                  match uncons (ss t) with
                       | None => None
                       | Some (h', t') => Some (h', ins h t')
                  end
       end
   |}.
   *)
   Relacja dobrze ufundowana nie ma nieskończonych łańcuchów malejących
CoInductive DecChain \{A : \mathtt{Type}\}\ (R : A \to A \to \mathtt{Prop})\ (x : A) : \mathtt{Prop} :=
     hd': A;
     R_-hd'_-x : R hd' x;
     tl': DecChain R hd';
}.
Lemma wf_no_DecChain:
  \forall (A : \mathsf{Type}) (R : A \to A \to \mathsf{Prop}) (x : A),
     well\_founded\ R \rightarrow DecChain\ R\ x \rightarrow False.
Proof.
  unfold well_founded.
```

```
intros A R x H C.
  specialize (H x).
  revert C.
  induction H; intro.
  inversion C.
  apply H\theta with hd'; assumption.
Qed.
   wut - końcówka kolistowego burdla
Ltac inv H := inversion H; subst; clear H.
   CoInductive Rev {A : Type} (1 r : coList A) : Prop :=
    Rev':
      (uncons l = None / uncons r = None)
      exists (h : A) (tl tr : coList A),
        uncons l = Some (h, tl) / r = snoc tr h / Rev tl tr
  }.
  Lemma Rev_wut :
  forall (A : Type) (l r : coList A),
    Infinite 1 -> Infinite r -> Rev 1 r.
   (* begin hide *)
  Proof.
  cofix CH.
  constructor. right. destruct H, HO.
  exists h, t, r. split.
    assumption.
    split.
      Focus 2. apply CH; auto; econstructor; eauto.
      apply eq_lsim. apply lsim_symm. apply Infinite_snoc. econstructor; eauto.
   (* end hide *)
  Lemma Rev_Finite :
  forall (A : Type) (l r : coList A),
    Rev l r \rightarrow Finite r \rightarrow Finite 1.
   (* begin hide *)
  Proof.
  intros A 1 r HRev H. revert 1 HRev.
  induction H as r H \mid h t r' H IH; intros.
    destruct HRev as [[H1 H2] \mid (h \& tl \& tr \& H1 \& H2 \& H3)].
```

```
left. assumption.
    subst. cbn in H. destruct (uncons tr) as []|; inv H.
  destruct HRev as [[H1 H2] \mid (h' \& tl \& tr \& H1 \& H2 \& H3)].
    congruence.
    subst.
 Abort.
 (* end hide *)
Lemma Rev_Finite:
forall (A : Type) (1 r : coList A),
  Rev l r \rightarrow Finite l \rightarrow Finite r.
 (* begin hide *)
Proof.
intros A 1 r HRev H. revert r HRev.
induction H; intros.
  destruct HRev as [[H1 H2] \mid (h \& tl \& tr \& H1 \& H2 \& H3)].
    left. assumption.
    subst. congruence.
  destruct HRev as [[H1 H2] \mid (h' \& tl \& tr \& H1 \& H2 \& H3)].
    subst. rewrite H1 in H. inv H. apply Finite_snoc, IHFinite.
      assumption.
 Qed.
 (* end hide *)
Lemma Infinite_Rev :
forall (A : Type) (l r : coList A),
  Rev 1 r -> Infinite 1 -> Infinite r.
 (* begin hide *)
Proof.
cofix CH.
destruct 1. (* decompose ex or and Revc0; clear Revc0.
  destruct 1. congruence.
  intro. subst. destruct x1 as [[h t]]; cbn.
    econstructor.
      cbn. reflexivity.
      apply (CH_{-}(cocons x t)).
        constructor. cbn. right. exists x, t, t. auto.
    congruence.
    subst. cbn in *. inversion p; subst. *)
 Abort.
 (* end hide *)
```

```
Lemma Finite_cocons :
forall (A : Type) (x : A) (1 : coList A),
  Finite 1 -> Finite (cocons x 1).
 (* begin hide *)
Proof.
intros. apply (Finite_Some x 1); auto.
 (* end hide *)
Fixpoint fromList {A : Type} (1 : list A) : coList A :=
 {|
  uncons :=
  match 1 with
      | => None
      | h :: t => Some (h, fromList t)
  end
 | } .
Lemma fromList_inj :
forall {A : Set} (11 12 : list A),
  fromList 11 = fromList 12 -> 11 = 12.
 (* begin hide *)
Proof.
induction 11 as \mid h1 \mid t1; destruct 12 as \mid h2 \mid t2; cbn; inversion 1.
  reflexivity.
  f_equal. apply IHt1. assumption.
Defined.
 *)
```

13.2 Ćwiczenia

Ćwiczenie Zdefiniuj typ potencjalnie nieskończonych drzew binarnych trzymających wartości typu A w węzłach, jego relację bipodobieństwa, predykaty "skończony" i "nieskończony" oraz funkcję mirror, która zwraca lustrzane odbicie drzewa. Udowodnij, że bipodobieństwo jest relacją równoważności i że funkcja mirror jest inwolucją (tzn. mirror (mirror t) jest bipodobne do t), która zachowuje właściwości bycia drzewem skończonym/nieskończonym. Pokaż, że drzewo nie może być jednocześnie skończone i nieskończone.

Ćwiczenie Znajdź taką rodzinę typów koinduktywnych C, że dla dowolnego typu A, C A jest w bijekcji z typem funkcji $nat \to A$. Przez bijekcję będziemy tu rozumieć funkcję, która ma odwrotność, z którą w obie strony składa się do identyczności.

Uwaga: nie da się tego udowodnić bez użycia dodatkowych aksjomatów, które na szczęście są bardzo oczywiste i same się narzucają.

Ćwiczenie Zdefiniuj pojęcie "nieskończonego łańcucha malejącego" (oczywiście za pomocą koindukcji) i udowodnij, że jeżeli relacja jest dobrze ufundowana, to nie ma nieskończonych łańcuchów malejących.

Rozdział 14

TODO: coś tu napisać.

 $\neg sim zero$ omega.

F2: Liczby konaturalne

Zdefiniuj liczby konaturalne oraz ich relację bipodobieństwa. Pokaż, że jest to relacja równoważności. Lemma $sim_{-}refl$: $\forall n : conat, sim n n.$ Lemma sim_-sym : $\forall n \ m : conat, sim \ n \ m \rightarrow sim \ m \ n.$ Lemma $sim_{-}trans$: $\forall a \ b \ c : conat, sim \ a \ b \rightarrow sim \ b \ c \rightarrow sim \ a \ c.$ 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 $succ_pred$: $\forall n \ m : conat$, $n = succ \ m \leftrightarrow pred \ n = Some \ m.$ Lemma $zero_not_omega$:

```
Lemma sim\_succ\_omega:
  \forall n : conat, sim \ n \ (succ \ n) \rightarrow sim \ n \ omega.
Lemma succ\_omega:
  omega = succ omega.
Lemma sim\_succ:
  \forall n \ m : conat, sim \ n \ m \rightarrow sim \ (succ \ n) \ (succ \ m).
Lemma sim\_succ\_inv:
  \forall n \ m : conat, sim (succ \ n) (succ \ m) \rightarrow 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 \rightarrow sim \ n \ zero.
Lemma sim_{-}add_{-}zero_{-}r :
  \forall n \ m : conat
     sim (add \ n \ m) \ zero \rightarrow sim \ m \ zero.
    Zdefiniuj relację ≤ na liczbach konaturalnych i udowodnij jej podstawowe właściwości.
Lemma le_refl:
```

```
\forall n : conat, le n n.
Lemma le\_trans:
   \forall a \ b \ c : conat, le \ a \ b \rightarrow le \ b \ c \rightarrow le \ a \ c.
Lemma le\_sim :
   \forall n1 \ n2 \ m1 \ m2 : conat,
      sim\ n1\ n2 \rightarrow sim\ m1\ m2 \rightarrow le\ n1\ m1 \rightarrow le\ n2\ m2.
Lemma le_-\theta_-l :
   \forall n : conat, le zero n.
Lemma le_-\theta_-r :
   \forall n : conat, le \ n \ zero \rightarrow n = zero.
Lemma le\_omega\_r:
   \forall n : conat, le n \text{ omega.}
Lemma le\_omega\_l:
   \forall n : conat, le \text{ omega } n \rightarrow sim \ n \text{ omega.}
Lemma le\_succ\_r:
   \forall n \ m : conat, le \ n \ m \rightarrow le \ n \ (succ \ m).
Lemma le\_succ:
   \forall n \ m : conat, le \ n \ m \rightarrow le \ (succ \ n) \ (succ \ m).
Lemma le_{-}add_{-}l:
   \forall a b c : conat,
      le \ a \ b \rightarrow le \ a \ (add \ b \ c).
Lemma le_add_r:
   \forall a b c : conat,
      le\ a\ c \rightarrow le\ a\ (add\ b\ c).
Lemma le_{-}add:
   \forall n1 \ n2 \ m1 \ m2 : conat,
      le \ n1 \ n2 \rightarrow le \ m1 \ m2 \rightarrow le \ (add \ n1 \ m1) \ (add \ n2 \ m2).
Lemma le_add_l':
   \forall n \ m : conat, le \ n \ (add \ n \ m).
Lemma le_add_r':
   \forall n \ m : conat, le \ m \ (add \ n \ m).
Lemma le_add_l'':
   \forall n n' m : conat,
      le \ n \ n' \rightarrow le \ (add \ n \ m) \ (add \ n' \ m).
Lemma le_add_r;:
   \forall n \ m \ m' : conat,
      le m m' \rightarrow le (add n m) (add n m').
     Zdefiniuj funkcje min i max i udowodnij ich właściwości.
```

```
Lemma min\_zero\_l:
  \forall n : conat. min zero n = zero.
Lemma min\_zero\_r:
  \forall n : conat, min \ n \ zero = zero.
Lemma min\_omega\_l:
  \forall n : conat, sim (min omega n) n.
Lemma min\_omega\_r :
  \forall n : conat, sim (min \ n \ omega) \ n.
Lemma min\_succ:
  \forall n \ m : conat
     sim (min (succ n) (succ m)) (succ (min n m)).
Lemma max\_zero\_l:
  \forall n : conat, sim (max zero n) n.
Lemma max\_zero\_r:
  \forall n : conat, sim (max \ n \ zero) \ n.
Lemma max\_omega\_l:
  \forall n : conat, sim (max omega n) omega.
Lemma max\_omega\_r:
  \forall n : conat, sim (max \ n \text{ omega}) \text{ omega}.
Lemma max\_succ:
  \forall n \ m : conat.
     sim (max (succ n) (succ m)) (succ (max n m)).
Lemma min\_assoc:
  \forall a \ b \ c : conat, sim (min (min a b) c) (min a (min b c)).
Lemma max\_assoc:
  \forall a \ b \ c : conat, sim (max (max a b) c) (max a (max b c)).
Lemma min\_comm:
  \forall n \ m : conat, sim (min \ n \ m) (min \ m \ n).
Lemma max\_comm:
  \forall n \ m : conat, sim (max \ n \ m) (max \ m \ n).
Lemma min_-add_-l:
  \forall a b c : conat.
     sim (min (add \ a \ b) (add \ a \ c)) (add \ a \ (min \ b \ c)).
Lemma min\_add\_r:
  \forall a b c : conat,
     sim (min (add \ a \ c) (add \ b \ c)) (add (min \ a \ b) \ c).
Lemma max\_add\_l:
  \forall 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\_reft:
   \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 \text{ omega}) \text{ 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 \rightarrow 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 \rightarrow sim n \text{ omega.}
Lemma Finite\_Infinite:
  \forall n : conat, Finite n \rightarrow Infinite n \rightarrow False.
nieparzysta.
Lemma Even\_zero:
```

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

```
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 \rightarrow Odd n \rightarrow Infinite n.
Lemma Even\_succ:
  \forall n : conat, Odd \ n \rightarrow Even (succ \ n).
Lemma Odd\_succ:
  \forall n : conat, Even n \rightarrow Odd (succ n).
Lemma Even\_succ\_inv:
  \forall n : conat, Odd (succ n) \rightarrow Even n.
Lemma Odd\_succ\_inv:
  \forall n : conat, Even (succ n) \rightarrow 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.

W tym celu definiujemy relację Sub, która ma reprezentować wykres funkcji odejmującej, tzn. specyfikować, jaki jest związek argumentów funkcji z jej wynikiem.

```
 \begin{array}{l} \texttt{Definition} \ sub : \texttt{Type} := \\ \{f : conat \rightarrow conat \rightarrow conat \mid \\ \forall \ n \ m \ r : conat, f \ n \ m = r \leftrightarrow Sub \ n \ m \ r \}. \end{array}
```

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 \text{ 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 \rightarrow 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
- \bullet 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): {	t Prop}:=
      Mul':
         (n = zero \land r = zero) \lor
         (m = zero \land r = zero) \lor
         \exists n' m' r' : conat,
            n = succ \ n' \wedge m = succ \ m' \wedge Mul \ n' \ m \ r' \wedge sim \ r \ (add \ r' \ m);
}.
Ltac unmul\ H :=
   destruct H as [H]; decompose [or and ex] H; clear H; subst.
Lemma Mul\_zero\_l:
   \forall n \ r : conat, Mul \ zero \ n \ r \rightarrow sim \ r \ zero.
Lemma Mul\_zero\_r:
   \forall n \ r : conat, Mul \ n \ zero \ r \rightarrow sim \ r \ zero.
Lemma Mul\_one\_l:
   \forall n \ r : conat, Mul (succ zero) \ n \ r \rightarrow sim \ n \ r.
Lemma Mul\_one\_r:
   \forall n \ r : conat, Mul \ n \ (succ \ zero) \ r \rightarrow sim \ n \ r.
Lemma Mul_{-}det:
   \forall n \ m \ r1 \ r2 : conat, Mul \ n \ m \ r1 \rightarrow Mul \ n \ m \ r2 \rightarrow sim \ r1 \ r2.
Lemma Mul\_comm:
  \forall n \ m \ r : conat, Mul \ n \ m \ r \rightarrow Mul \ m \ n \ r.
Lemma Mul\_assoc:
   \forall a b c d r1 r2 s1 s2 : conat,
      Mul \ a \ b \ r1 \rightarrow Mul \ r1 \ c \ r2 \rightarrow
      Mul b c s1 \rightarrow Mul a s1 s2 \rightarrow sim r2 s2.
```

F3: Strumienie

TODO: w tym rozdziale będą ćwiczenia dotyczące strumieni, czyli ogólnie wesołe koinduktywne zabawy, o których jeszcze nic nie napisałem.

15.1 Bipodobieństwo

```
CoInductive sim\ \{A: \mathtt{Type}\}\ (s1\ s2: Stream\ A): \mathtt{Prop}:= \{ \\ hds: hd\ s1 = hd\ s2; \\ tls: sim\ (tl\ s1)\ (tl\ s2); \}.
Lemma sim\_refl:
\forall\ (A: \mathtt{Type})\ (s: Stream\ A),\ sim\ s\ s.
Lemma sim\_sym:
\forall\ (A: \mathtt{Type})\ (s1\ s2: Stream\ A), \\ sim\ s1\ s2 \to sim\ s2\ s1.
Lemma sim\_trans:
\forall\ (A: \mathtt{Type})\ (s1\ s2\ s3: Stream\ A), \\ sim\ s1\ s2 \to sim\ s2\ s3 \to sim\ s1\ s3.
```

15.2 sapp

Zdefiniuj funkcję sapp, która konkatenuje dwa strumienie. Czy taka funkcja w ogóle ma sens? CoFixpoint $sapp \{A : Type\} (s1 \ s2 : Stream \ A) : Stream \ A :=$ $\{ \mid$ $hd := hd \ s1;$ $tl := sapp (tl \ s1) \ s2;$ |}. Lemma $sapp_pointless$: $\forall (A : \mathsf{Type}) (s1 \ s2 : Stream \ A),$ sim (sapp s1 s2) s1.Lemma $map_{-}id$: $\forall (A : \mathsf{Type}) (s : Stream \ A), sim (map (@id \ A) \ s) \ s.$ Lemma $map_compose$: $\forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B) \ (g : B \to C) \ (s : \mathit{Stream} \ A),$ $sim (map \ g \ (map \ f \ s)) \ (map \ (fun \ x \Rightarrow g \ (f \ x)) \ s).$ (* 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 $intersperse_merge_repeat$: $\forall (A : \mathsf{Type}) (x : A) (s : \mathit{Stream} A),$ $sim (intersperse \ x \ s) (merge \ s \ (repeat \ x)).$ (* Dlaczego s nie musi tu być indeksem? *) Inductive $Elem \{A : Type\} (x : A) (s : Stream A) : Prop :=$ $| Elem_hd : x = hd \ s \rightarrow Elem \ x \ s$ $| Elem_tl : Elem x (tl s) \rightarrow Elem x s.$ Hint Constructors *Elem*. Inductive $Dup \{A : Type\} (s : Stream A) : Prop :=$ $|Dup_hd : Elem (hd s) (tl s) \rightarrow Dup s$ $|Dup_{-}tl:Dup(tls)\rightarrow Dups.$ Ltac inv H := inversion H; subst; clear H. Require Import Arith. Lemma $Elem_nth$: $\forall (A : \mathsf{Type}) (x : A) (s : \mathit{Stream} A),$

Elem $x \ s \leftrightarrow \exists \ n : nat, nth \ n \ s = x.$

```
Lemma nth\_from:
   \forall n \ m : nat,
      nth \ n \ (from \ m) = n + m.
Lemma Elem\_from\_add:
  \forall n \ m : nat, Elem \ n \ (from \ m) \rightarrow
      \forall k : nat, Elem (k + n) (from m).
Lemma Elem\_from:
   \forall n \ m : nat, Elem \ n \ (from \ m) \leftrightarrow m \leq n.
Lemma Dup\_spec:
   \forall (A : \mathsf{Type}) (s : Stream \ A),
      Dup s \leftrightarrow \exists n \ m : nat, \ n \neq m \land nth \ n \ s = nth \ m \ s.
Lemma NoDup\_from:
   \forall n : nat, \neg Dup (from n).
     To samo: dlaczego s nie musi być indeksem? *)
Inductive Exists \{A : \mathsf{Type}\}\ (P : A \to \mathsf{Prop})\ (s : Stream\ A) : \mathsf{Prop} :=
       Exists\_hd: P(hd s) \rightarrow Exists P s
      \mid Exists\_tl : Exists P (tl s) \rightarrow Exists P s.
Lemma Exists\_spec:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (s : Stream A),
      Exists P \ s \leftrightarrow \exists \ n : nat, P \ (nth \ n \ s).
CoInductive Forall \{A : \mathsf{Type}\}\ (s : Stream\ A)\ (P : A \to \mathsf{Prop}) : \mathsf{Prop} :=
      Forall\_hd : P (hd s);
      Forall\_tl : Forall (tl s) P;
}.
Lemma Forall\_spec:
   \forall (A : \mathsf{Type}) (s : Stream \ A) (P : A \to \mathsf{Prop}),
      Forall s \ P \leftrightarrow \forall n : nat, P (nth \ n \ s).
Lemma Forall\_spec':
   \forall (A : \mathsf{Type}) (s : Stream \ A) (P : A \to \mathsf{Prop}),
      Forall s \ P \leftrightarrow \forall \ x : A, Elem \ x \ s \rightarrow P \ x.
Lemma Forall\_Exists:
   \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}) (s : Stream A),
      Forall s P \to Exists P s.
CoInductive Substream \{A : \mathsf{Type}\}\ (s1\ s2 : Stream\ A) : \mathsf{Prop} :=
      n: nat;
      p: hd\ s1 = nth\ n\ s2;
      Substream': Substream (tl s1) (drop (S n) s2);
```

```
}.
Lemma drop_{-}tl:
  \forall (A : \mathsf{Type}) (n : nat) (s : Stream A),
      drop \ n \ (tl \ s) = drop \ (S \ n) \ s.
Lemma tl\_drop:
  \forall (A : \mathsf{Type}) (n : nat) (s : Stream A),
      tl(drop \ n \ s) = drop \ n \ (tl \ s).
Lemma nth\_drop:
  \forall (A : \mathsf{Type}) (n \ m : nat) (s : Stream \ A),
      nth \ n \ (drop \ m \ s) = nth \ (n + m) \ s.
Lemma drop\_drop:
  \forall (A : \mathsf{Type}) (n \ m : nat) (s : Stream \ A),
      drop \ m \ (drop \ n \ s) = drop \ (n + m) \ s.
Lemma Substream_{-}tl:
  \forall (A : \mathsf{Type}) (s1 \ s2 : Stream \ A),
      Substream s1 s2 \rightarrow Substream (tl s1) (tl s2).
Lemma Substream\_drop:
  \forall (A : \mathsf{Type}) (n : nat) (s1 \ s2 : Stream \ A),
      Substream s1 \ s2 \rightarrow Substream \ (drop \ n \ s1) \ (drop \ n \ s2).
Lemma hd_{-}drop:
   \forall (A : \mathsf{Type}) (n : nat) (s : Stream A),
     hd (drop \ n \ s) = nth \ n \ s.
Lemma Substream\_drop\_add:
  \forall (A : \mathsf{Type}) (n \ m : nat) (s1 \ s2 : Stream \ A),
      Substream s1 (drop n s2) \rightarrow Substream s1 (drop (n + m) s2).
Lemma Substream\_trans:
  \forall (A : \mathsf{Type}) (s1 \ s2 \ s3 : Stream \ A),
      Substream s1 s2 \rightarrow Substream s2 s3 \rightarrow Substream s1 s3.
Lemma Substream\_not\_antisymm:
   \exists (A : \mathsf{Type}) (s1 \ s2 : Stream \ A),
      Substream s1 s2 \wedge Substream s2 s1 \wedge \neg sim s1 s2.
Inductive Suffix \{A : \mathsf{Type}\} : Stream \ A \to Stream \ A \to \mathsf{Prop} :=
      | Suffix\_refl :
           \forall s : Stream \ A, Suffix \ s \ s
      | Suffix_tl :
           \forall s1 \ s2 : Stream \ A,
              Suffix (tl s1) s2 \rightarrow Suffix s1 s2.
Fixpoint snoc \{A : \mathsf{Type}\}\ (x : A)\ (l : list\ A) : list\ A :=
match l with
```

```
nil \Rightarrow cons \ x \ nil
       cons \ h \ t \Rightarrow cons \ h \ (snoc \ x \ t)
end.
Lemma Suffix\_spec:
  \forall (A : \mathsf{Type}) (s1 \ s2 : Stream \ A),
     Suffix s1 \ s2 \leftrightarrow \exists \ l : list \ A, \ s1 = lsapp \ l \ s2.
\forall x: A, Elem \ x \ s1 \rightarrow Elem \ x \ s2.
Definition SetEquiv \{A : \texttt{Type}\} (s1 \ s2 : Stream \ A) : \texttt{Prop} :=
  Incl s1 s2 \wedge Incl s2 s1.
Lemma sim_-Elem:
  \forall (A : \mathsf{Type}) (x : A) (s1 \ s2 : Stream \ A),
     sim\ s1\ s2 \rightarrow Elem\ x\ s1 \rightarrow Elem\ x\ s2.
Lemma sim_{-}Incl:
  \forall (A : \mathsf{Type}) (s1 \ s1' \ s2 \ s2' : Stream \ A),
     sim\ s1\ s1' \rightarrow sim\ s2\ s2' \rightarrow Incl\ s1\ s2 \rightarrow Incl\ s1'\ s2'.
Lemma sim\_SetEquiv:
  \forall (A : \mathsf{Type}) (s1 \ s1' \ s2 \ s2' : Stream \ A),
     sim\ s1\ s1' \rightarrow sim\ s2\ s2' \rightarrow SetEquiv\ s1\ s2 \rightarrow SetEquiv\ s1'\ s2'.
Definition scons \{A : Type\} (x : A) (s : Stream A) : Stream A :=
\{ \mid
     hd := x:
     tl := s;
}.
Inductive SPermutation \{A : \mathsf{Type}\} : Stream A \to Stream A \to \mathsf{Prop} :=
      |SPerm\_refl:
           \forall s : Stream \ A, SPermutation \ s \ s
     |SPerm\_skip|:
           \forall (x : A) (s1 \ s2 : Stream \ A),
              SPermutation \ s1 \ s2 \rightarrow SPermutation \ (scons \ x \ s1) \ (scons \ x \ s2)
     |SPerm\_swap:
           \forall (x \ y : A) (s1 \ s2 : Stream \ A),
              SPermutation \ s1 \ s2 \rightarrow
                 SPermutation (scons \ x \ (scons \ y \ s1)) (scons \ y \ (scons \ x \ s2))
     |SPerm\_trans|:
           \forall s1 \ s2 \ s3 : Stream \ A,
              SPermutation \ s1 \ s2 \rightarrow SPermutation \ s2 \ s3 \rightarrow SPermutation \ s1 \ s3.
Hint Constructors SPermutation.
(* TODO *)
Require Import Permutation.
```

```
Lemma lsapp\_scons:
  \forall (A : \mathsf{Type}) (l : \mathit{list} \ A) (x : A) (s : \mathit{Stream} \ A),
     lsapp \ l \ (scons \ x \ s) = lsapp \ (snoc \ x \ l) \ s.
Lemma SPermutation\_Permutation\_lsapp:
  \forall (A : \mathsf{Type}) (l1 \ l2 : list \ A) (s1 \ s2 : Stream \ A),
     Permutation \ l1 \ l2 \rightarrow SPermutation \ s1 \ s2 \rightarrow
        SPermutation (lsapp l1 s1) (lsapp l2 s2).
Lemma take\_drop:
  \forall (A : \mathsf{Type}) (n : nat) (s : Stream A),
     s = lsapp (take \ n \ s) (drop \ n \ s).
Lemma take\_add:
  \forall (A : \mathsf{Type}) (n \ m : nat) (s : Stream \ A),
      take (n + m) s = List.app (take n s) (take m (drop n s)).
Lemma SPermutation\_spec:
  \forall (A : \mathsf{Type}) (s1 \ s2 : Stream \ A),
     SPermutation \ s1 \ s2 \leftrightarrow
     \exists n : nat,
        Permutation (take n s1) (take n s2) \land
        drop \ n \ s1 = drop \ n \ s2.
    Strumienie za pomocą przybliżeń.
Module approx.
Print take.
Inductive Vec\ (A: \mathsf{Type}): nat \to \mathsf{Type} :=
       vnil: Vec A 0
      | vcons : \forall n : nat, A \rightarrow Vec A n \rightarrow Vec A (S n).
Arguments \ vnil \{A\}.
Arguments vcons \{A\} \{n\}.
Definition vhd \{A : \mathsf{Type}\} \{n : nat\} (v : Vec \ A (S \ n)) : A :=
match v with
     \mid vcons \ h \ \_ \Rightarrow h
end.
Definition vtl \{A : \mathsf{Type}\} \{n : nat\} (v : Vec A (S n)) : Vec A n :=
match v with
     | vcons _t \Rightarrow t
end.
Require Import Program. Equality.
Lemma vhd_-vtl:
  \forall (A : \mathsf{Type}) (n : nat) (v : Vec A (S n)),
     v = v cons \ (vhd \ v) \ (vtl \ v).
```

```
Fixpoint vtake \{A : Type\} (s : Stream A) (n : nat) : Vec A n :=
match n with
      0 \Rightarrow vnil
     |S| n' \Rightarrow vcons (hd s) (vtake (tl s) n')
end.
Fixpoint vtake' {A : Type} (s : Stream \ A) \ (n : nat) : Vec \ A \ (S \ n) :=
match n with
      \mid 0 \Rightarrow vcons \ (hd \ s) \ vnil
     \mid S \mid n' \Rightarrow v cons \ (hd \mid s) \ (v take' \ (tl \mid s) \mid n')
end.
CoFixpoint unvtake \{A : Type\} (f : \forall n : nat, Vec A (S n)) : Stream A :=
\{ \mid
     hd := vhd (f 0);
     tl :=
        unvtake (fun \ n : nat \Rightarrow vtl \ (f \ (S \ n)))
|}.
Fixpoint vnth \{A : Type\} \{n : nat\} (v : Vec A n) (k : nat) : option A :=
{\tt match}\ v,\ k\ {\tt with}
       vnil, \_ \Rightarrow None
       vcons h t, 0 \Rightarrow Some h
       vcons\ h\ t,\ S\ k' \Rightarrow vnth\ t\ k'
end.
Ltac depdestr x :=
  let x' := fresh "x"in remember x as x'; dependent destruction x'.
Lemma unvtake\_vtake':
  \forall (A : \mathsf{Type}) (n : nat) (f : \forall n : nat, Vec A (S n)),
     (\forall m1 \ m2 \ k : nat, k \leq m1 \rightarrow k \leq m2 \rightarrow
        vnth (f m1) k = vnth (f m2) k) \rightarrow
         vtake' (unvtake f) n = f n.
Lemma vtake\_unvtake:
  \forall (A : \mathsf{Type}) (s : Stream \ A),
     sim (unvtake (vtake' s)) s.
End approx.
    Pomysł dawno zapomniany: induktywne specyfikacje funkcji.
Inductive Filter \{A: \mathsf{Type}\}\ (f: A \to bool): Stream\ A \to Stream\ A \to \mathsf{Prop}:=
     | Filter\_true :
           \forall s \ r \ r' : Stream \ A,
              f(hd s) = true \rightarrow Filter f(tl s) r \rightarrow
                 hd\ r' = hd\ s \rightarrow tl\ r' = r \rightarrow Filter\ f\ s\ r'
      | Filter\_false :
```

```
\forall s \ r : Stream \ A,
               f (hd \ s) = false \rightarrow Filter \ f \ (tl \ s) \ r \rightarrow Filter \ f \ s \ r.
Lemma\ Filter\_bad:
   \forall (A : \mathsf{Type}) (f : A \to bool) (s \ r : Stream \ A),
      Filter f \ s \ r \to (\forall \ x : A, f \ x = false) \to False.
CoInductive Filter' \{A: \mathsf{Type}\}\ (f:A \to bool)\ (s\ r:Stream\ A): \mathsf{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 : \mathsf{Type}) (s \ r : \mathit{Stream} \ A),
      Filter' (fun \_ \Rightarrow false) s r.
Lemma Filter'\_const\_true:
   \forall (A : \mathsf{Type}) (s \ r : \mathit{Stream} \ A),
      Filter' (fun \_ \Rightarrow true) s \ r \rightarrow sim \ s \ r.
```

F4: Kolisty

```
Require Import book.D5.
```

Ten rozdział będzie o kolistach, czyli koinduktywnych odpowiednikach list różniących się od nich tym, że mogą być potencjalnie nieskończone.

```
\begin{tabular}{ll} {\tt CoInductive} & coList & (A: {\tt Type}): {\tt Type} := \\ \{ & uncons: option & (A\times coList A); \\ \}. \end{tabular}
```

Arguments uncons $\{A\}$.

Przydatny będzie następujący, dość oczywisty fakt dotyczący równości kolist.

```
Lemma eq\_uncons:
```

```
\forall (A: \mathtt{Type}) \ (l1 \ l2: coList \ A), \ uncons \ l1 = uncons \ l2 \rightarrow 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: \forall (A: \mathsf{Type})\ (l: coList\ A),\ lsim\ l\ l. Lemma lsim\_symm: \forall (A: \mathsf{Type})\ (l1\ l2: coList\ A),\ lsim\ l1\ l2 \to lsim\ l2\ l1. Lemma lsim\_trans: \forall (A: \mathsf{Type})\ (l1\ l2\ l3: coList\ A),\ lsim\ l1\ l2 \to lsim\ l2\ l3 \to 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: {\tt 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.

```
\forall (A: \mathtt{Type}) \ (x \ y: A) \ (l1 \ l2: coList \ A), \ x = y \rightarrow lsim \ l1 \ l2 \rightarrow lsim \ (cocons \ x \ l1) \ (cocons \ y \ l2).
```

Lemma $lsim_cocons_inv$:

```
\forall (A: \mathsf{Type}) (x \ y: A) (l1 \ l2: coList \ A), lsim (cocons \ x \ l1) (cocons \ y \ l2) \rightarrow x = y \land 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 coconsa powinny być oczywiste.

Require Import F2.

```
Lemma sim_{-}len:
```

```
\forall (A: \mathtt{Type}) (l1 \ l2: coList \ A), lsim \ l1 \ l2 \rightarrow sim \ (len \ l1) \ (len \ l2).
```

Lemma len_-conil :

```
\forall A : \mathsf{Type}, \\ len (@conil A) = zero.
```

Lemma len_cocons :

```
\forall (A : \mathsf{Type}) (x : A) (l : coList A),

len (cocons x l) = succ (len l).
```

Zdefiniuj funkcję snoc, która dostawia element na koniec kolisty.

Lemma $snoc_cocons$:

```
\forall (A: \mathsf{Type}) \ (l: coList\ A) \ (x\ y: A), \\ lsim \ (snoc\ (cocons\ x\ l)\ y) \ (cocons\ x\ (snoc\ l\ y)).
```

Lemma len_snoc :

```
\forall (A : \mathsf{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 : \mathsf{Type}) (l : coList A),
     app\ conil\ l=l.
Lemma app\_conil\_r:
  \forall (A : \mathsf{Type}) (l : coList A),
     lsim (app \ l \ conil) \ l.
Lemma app\_cocons\_l:
  \forall (A : \mathsf{Type}) (x : A) (l1 \ l2 : coList \ A),
     lsim (app (cocons x l1) l2) (cocons x (app l1 l2)).
Lemma len_-app:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
     sim (len (app l1 l2)) (add (len l1) (len l2)).
Lemma snoc\_app:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A) (x : A),
     lsim (snoc (app l1 l2) x) (app l1 (snoc l2 x)).
Lemma app\_snoc\_l:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A) (x : A),
     lsim (app (snoc l1 x) l2) (app l1 (cocons x l2)).
Lemma app\_assoc:
  \forall (A : \mathsf{Type}) (l1 \ l2 \ l3 : coList \ A),
     lsim (app (app l1 l2) l3) (app l1 (app l2 l3)).
    Zdefiniuj funkcje lmap, która aplikuje funkcje f:A\to B do każdego elementu kolisty.
    TODO: wyklarować, dlaczego niektóre rzeczy mają "l" na początku nazwy
Lemma lmap\_conil:
  \forall (A B : \mathsf{Type}) (f : A \to B),
     lmap \ f \ conil = conil.
Lemma lmap\_cocons:
  \forall (A B : \mathsf{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 : \mathsf{Type}) (f : A \to B) (l : coList A),
     sim (len (lmap f l)) (len l).
Lemma lmap\_snoc:
  \forall (A B : \mathsf{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 : \mathsf{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 : \mathtt{Type}) (l : coList \ A), lsim (lmap id \ l) \ l.
```

Lemma $lmap_compose$:

```
\forall (A \ B \ C : \mathtt{Type}) \ (f : A \to B) \ (g : B \to C) \ (l : coList \ A), lsim \ (lmap \ g \ (lmap \ f \ l)) \ (lmap \ (\mathtt{fun} \ x \Rightarrow g \ (f \ x)) \ l).
```

Lemma $lmap_ext$:

$$\forall \ (A \ B : \mathtt{Type}) \ (f \ g : A \to B) \ (l : \mathit{coList} \ A), \ (\forall \ x : A, f \ x = g \ x) \to \mathit{lsim} \ (\mathit{lmap} \ f \ l) \ (\mathit{lmap} \ g \ l).$$

Zdefiniuj funkcję iterate, która tworzy nieskończoną kolistę przez iterowanie funkcji f poczynając od pewnego ustalonego elementu.

Lemma $len_-iterate$:

```
\forall (A: \mathtt{Type}) (f: A \rightarrow A) (x: A), sim (len (iterate f x)) omega.
```

Zdefiniuj funkcję piterate, która tworzy kolistę przez iterowanie funkcji częściowej $f:A \to option\ B$ poczynając od pewnego ustalonego elementu.

Zdefiniuj funkcję zipW, która bierze funkcję $f:A\to B\to C$ oraz dwie kolisty l1 i l2 i zwraca kolistę, której elementy powstają z połączenia odpowiadających sobie elementów l1 i l2 za pomocą funkcji f.

 $\operatorname{Lemma} zip W_conil_l$:

```
\forall \; (A \; B \; C : \mathtt{Type}) \; (f : A \to B \to C) \; (l : \mathit{coList} \; B), \\ lsim \; (\mathit{zipW} \; f \; \mathit{conil} \; l) \; \mathit{conil}.
```

Lemma $zip W_-conil_-r$:

```
\forall (A \ B \ C : \mathtt{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 :

$$\forall (A \ B \ C : \mathtt{Type}) \ (f : A \rightarrow B \rightarrow C) \ (l1 : coList \ A) \ (l2 : coList \ B), sim \ (len \ (zipW \ f \ l1 \ l2)) \ (min \ (len \ l1) \ (len \ l2)).$$

Napisz funkcję scan, która przekształca $l: coList\ A$ w kolistę sum częściowych działania $f: B \to A \to B$.

Lemma $scan_conil$:

```
\forall \ (A \ B : \mathtt{Type}) \ (f : B \to A \to B) \ (b : B), \\ lsim \ (scan \ conil \ f \ b) \ conil.
```

Lemma $scan_cocons$:

```
\forall (A \ B : \mathtt{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 :

$$\forall (A \ B : \texttt{Type}) \ (l : coList \ A) \ (f : B \rightarrow A \rightarrow B) \ (b : B),$$

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 \Rightarrow None
     | Some (\_, x, \_) \Rightarrow Some x
end.
{\tt Definition}\ take\ \{A:{\tt Type}\}\ (l:\ coList\ A)\ (n:\ nat):\ option\ (\mathit{list}\ A):=
match splitAt \ l \ n with
     | None \Rightarrow None
     | Some (l, \_, \_) \Rightarrow Some l
end.
Definition drop \{A : \mathsf{Type}\}\ (l : coList\ A)\ (n : nat) : option\ (coList\ A) :=
match splitAt \ l \ n with
       None \Rightarrow None
     | Some (\_, \_, l) \Rightarrow Some l
Fixpoint fromList \{A : \mathsf{Type}\}\ (l : list A) : coList A :=
match l with
     | | | \Rightarrow conil
     | h :: t \Rightarrow cocons \ h \ (fromList \ t)
end.
Definition insert \{A : Type\} (l : coList A) (n : nat) (x : A)
  : option (coList A) :=
match splitAt l n with
```

```
 \mid None \Rightarrow None \\ \mid Some \; (start, \; mid, \; rest) \Rightarrow \\ Some \; (app \; (fromList \; start) \; (cocons \; x \; (cocons \; mid \; rest))) \\ \text{end.} \\  \text{Definition } remove \; \{A: \mathsf{Type}\} \; (l: coList \; A) \; (n: nat) \\ : option \; (coList \; A) := \\ \mathsf{match} \; splitAt \; l \; n \; \mathsf{with} \\ \mid None \; \Rightarrow None \\ \mid Some \; (start, \; \_, \; rest) \; \Rightarrow Some \; (app \; (fromList \; start) \; rest) \\ \mathsf{end.} \\ \end{aligned}
```

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:
  \forall (A : \mathsf{Type}) (l : coList A),
      Finite l \to Infinite \ l \to False.
Lemma sim_{-}Infinite:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
      lsim \ l1 \ l2 \rightarrow Infinite \ l1 \rightarrow Infinite \ l2.
Lemma len_Finite:
  \forall (A : \mathsf{Type}) (l : coList A),
      Finite l \to len \ l \neq omega.
Lemma len_Infinite:
  \forall (A : \mathsf{Type}) (l : coList A),
      len \ l = omega \rightarrow Infinite \ l.
Lemma Finite\_snoc:
   \forall (A : \mathsf{Type}) (l : coList A) (x : A),
      Finite l \to Finite (snoc l x).
Lemma Infinite\_snoc:
  \forall (A : \mathsf{Type}) (l : coList \ A) (x : A),
      Infinite l \to lsim (snoc \ l \ x) \ l.
Lemma Infinite\_app\_l:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
      Infinite l1 \rightarrow Infinite (app l1 l2).
Lemma Infinite\_app\_r:
  \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
      Infinite l2 \rightarrow Infinite (app l1 l2).
Lemma Finite\_app:
```

```
\forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
      Finite l1 \rightarrow Finite \ l2 \rightarrow Finite \ (app \ l1 \ l2).
Lemma Finite\_app\_conv:
   \forall (A : \mathsf{Type}) (l1 \ l2 : coList \ A),
      Finite (app l1 l2) \rightarrow Finite l1 \vee Finite l2.
Lemma Finite\_lmap:
   \forall (A B : \mathsf{Type}) (f : A \to B) (l : coList A),
      Finite l \to Finite (lmap f l).
Lemma Infinite\_lmap:
  \forall (A B : \mathsf{Type}) (f : A \to B) (l : coList A),
      Infinite l \to Infinite (lmap f l).
Lemma Infinite\_iterate:
  \forall (A : \mathsf{Type}) (f : A \to A) (x : A),
      Infinite (iterate f(x)).
Lemma piterate\_Finite:
  \forall (A : \mathsf{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 : \mathsf{Type}) \ (f : A \to B \to C) \ (l1 : coList \ A) \ (l2 : coList \ B),
      Finite l1 \rightarrow Finite (zip W f l1 l2).
Lemma Finite\_zipW\_r:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (l1 : coList \ A) \ (l2 : coList \ B),
      Finite l2 \rightarrow Finite \ (zip W \ f \ l1 \ l2).
Lemma Infinite\_zip W\_l:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (l1 : coList \ A) \ (l2 : coList \ B),
      Infinite (zipW \ f \ l1 \ l2) \rightarrow Infinite \ l1.
Lemma Infinite\_zipW\_r:
  \forall (A \ B \ C : \mathsf{Type}) \ (f : A \to B \to C) \ (l1 : coList \ A) \ (l2 : coList \ B),
      Infinite (zipW \ f \ l1 \ l2) \rightarrow Infinite \ l1.
Lemma Infinite\_splitAt:
   \forall (A : \mathsf{Type}) (n : nat) (l : coList A),
      Infinite l \rightarrow
         \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 $\{A : \mathsf{Type}\}\ (P : A \to \mathsf{Prop}) : coList\ A \to \mathsf{Prop} :=$

```
 | Exists\_hd: \\ \forall \ (l: coList \ A) \ (h: A) \ (t: coList \ A), \\ uncons \ l = Some \ (h, \ t) \rightarrow P \ h \rightarrow Exists \ P \ l \\ | Exists\_tl: \\ \forall \ (l: coList \ A) \ (h: A) \ (t: coList \ A), \\ uncons \ l = Some \ (h, \ t) \rightarrow Exists \ P \ t \rightarrow Exists \ P \ l. \\ \text{CoInductive All } \{A: \texttt{Type}\} \ (P: A \rightarrow \texttt{Prop}) \ (l: coList \ A) : \texttt{Prop} := \\ \{ All': \\ uncons \ l = None \ \lor \\ \exists \ (h: A) \ (t: coList \ A), \\ uncons \ l = Some \ (h, \ t) \land P \ h \land \texttt{All} \ P \ t; \\ \}. \\ \texttt{Lemma } Exists\_not\_All: \\ \forall \ (A: \texttt{Type}) \ (P: A \rightarrow \texttt{Prop}) \ (l: coList \ A), \\ Exists \ P \ l \rightarrow \neg \ \texttt{All} \ (\texttt{fun} \ x: A \Rightarrow \neg P \ x) \ l.
```

G: Inne spojrzenia na typy induktywne i koinduktywne

17.1 W-typy (TODO)

```
Inductive W (A: \mathsf{Type}) (B: A \to \mathsf{Type}): \mathsf{Type} := | sup: \forall x: A, (B x \to W A B) \to W A B.

Arguments sup \{A B\} _ _.
```

W-typy (ang. W-types) to typy dobrze ufundowanych drzew (ang. well-founded trees - W to skrót od well-founded), tzn. skończonych drzew o niemal dowolnych kształtach wyznaczanych przez parametry A i B.

Nie są one zbyt przydatne w praktyce, gdyż wszystko, co można za ich pomocą osiągnąć, można też osiągnąć bez nich zwykłymi typami induktywnymi i będzie to dużo bardziej czytelne oraz prostsze w implementacji. Ba! W-typy są nawet nieco słabsze, gdyż go udowodnienia reguły indukcji wymagają aksjomatu ekstensjonalności dla funkcji.

Jednak z tego samego powodu są bardzo ciekawe pod względem teoretycznym - wszystko, co można zrobić za pomocą parametryzowanych typów induktywnych, można też zrobić za pomocą samych W-typów. Dzięki temu możemy badanie parametryzowanych typów induktywnych, których jest mniej więcej nieskończoność i jeszcze trochę, sprowadzić do badania jednego tylko W (o ile godzimy się na aksjomat ekstensjonalności dla funkcji).

Zanim jednak zobaczymy przykłady ich wykorzystania, zastanówmy się przez kilka chwil, dlaczego są one tak ogólne.

Sprawa jest dość prosta. Rozważmy typ induktywny T i dowolny z jego konstruktorów $c: X1 \to ... \to Xn \to T$. Argumenty Xi możemy podzielić na dwie grupy: argumenty nieindukcyjne (oznaczmy je literą A) oraz indukcyjne (które są postaci T). Wobec tego typ c możemy zapisać jako $c: A1 \to ... \to Ak \to T \to ... \to T \to T$.

W kolejnym kroku łączymy argumenty za pomocą produktu: niech $A:=A1\times...\times Ak$. Wtedy typ c wygląda tak: $c:A\to T\times...\times T\to T$. Zauważmy, że $T\times...\times T$ możemy zapisać równoważnie jako $B\to T$, gdzie B to typ mający tyle elementów, ile razy

T występuje w produkcie $T \times ... \times T$. Zatem typ c przedstawia się tak: $c: A \to (B \to T) \to T$.

Teraz poczynimy kilka uogólnień. Po pierwsze, na początku założyliśmy, że c ma skończenie wiele argumentów indukcyjnych, ale postać $B \to T$ uwzględnia także przypadek, gdy jest ich nieskończenie wiele (tzn. gdy c miał oryginalnie jakiś argument postaci $Y \to T$ dla nieskończonego Y).

Po drugie, założyliśmy, że c jest funkcją niezależną. Przypadek, gdy c jest funkcją zależną możemy pokryć, pozwalając naszemu B zależeć od A, tzn. $B:A\to {\tt Type}$. Typ konstruktora c zyskuje wtedy postać sumy zależnej $\{x:A\&Bx\to T\}\to T$. W ostatnim kroku odpakowujemy sumę i c zyskuje postać $c:\forall x:A,Bx\to T$.

Jak więc widać, typ każdego konstruktora można przekształcić tak, żeby móc zapisać go jako $\forall x: A, B x \to T$. Zauważmy też, że jeżeli mamy dwa konstruktory $c1: \forall x: A1$, $B1 x \to T$ oraz $c2: \forall x: A2, B2 x \to T$, to możemy równie dobrze zapisać je za pomocą jednego konstruktora c: niech A:=A1+A2 i niech B (inl a1) := B1 a1, B (inl a2) := B2 a2. Wtedy konstruktory c1 i c2 są równoważne konstruktorowi c.

Stosując powyższe przekształcenia możemy sprowadzić każdy typ induktywny do równoważnej postaci z jednym konstruktorem o typie $\forall x: A, B x \to T$. Skoro tak, to definiujemy nowy typ, w którym A i B są parametrami... i bum, tak właśnie powstało W!

Podejrzewam, że powyższy opis przyprawia cię o niemały ból głowy. Rzućmy więc okiem na przykład, który powinien być wystarczająco ogólny, żeby wszystko stało się jasne.

Print list.

Spróbujmy zastosować powyższe przekształcenia na typie $list\ X,$ żeby otrzymać reprezentację list za pomocą W.

Zajmijmy się najpierw konstruktorem nil. Nie ma on ani argumentów indukcyjnych, ani nieindukcyjnych, co zdaje się nie pasować do naszej ogólnej metody. Jest to jednak jedynie złudzenie: brak argumentów nieindukcyjnych możemy zareprezentować za pomocą argumenu o typie unit, zaś brak argumentów indukcyjnych możemy zareprezentować argumentem o typie $False \rightarrow list \ X$. Wobec tego typ konstruktora nil możemy zapisać jako $unit \rightarrow (False \rightarrow list \ X) \rightarrow list \ X$.

Dla consa jest już prościej: argument nie
indukcyjny to po prostu X, zaś jeden argument indukcyjny możemy przedstawić jako $unit \to list\ X$. Nowy typ
 consa możemy zapisać jako $X \to (unit \to list\ X) \to list\ X$.

Pozostaje nam skleić obydwa konstruktory w jeden. Niech A := unit + X i niech B ($inl\ tt$) := False, B ($inr\ x$) := unit. W ten sposób dostajemy poniższe kodowanie list za pomocą W (oczywiście nie jest to jedyne możliwe kodowanie - równie dobrze zamiast unit + X moglibyśmy użyć typu $option\ X$).

Module listW.

```
\mathtt{Definition}\ list W\ (X: \mathtt{Type}): \mathtt{Type} :=
```

Qed.

Wartą zauważenia różnicą konceptualną jest to, że jeżeli myślimy Coqowymi typami induktywnymi, to list ma dwa konstruktory - nil i cons, ale gdy myślimy za pomocą W, to sprawa ma się inaczej. Formalnie listW ma jeden konstruktor sup, ale w praktyce jest aż 1+|X| konstruktorów, gdzie |X| oznacza liczbę elementów typu X. Jeden z nich opdowiada nil, a każdy z pozostałych |X| konstruktorów odpowiada $cons\ x$ dla pewnego x:X. Jedyną pozostałością po oryginalnej liczbie konstruktorów jest liczba składników, które pojawiają się w sumie unit+X.

Oczywiście posługiwanie się nil i cons jest dużo wygodniejsze niż używanie sup, więc czas odzyskać utracone konstruktory!

```
Definition nilW (X: {\tt Type}): listW X:=sup\ (inl\ tt)\ ({\tt fun}\ e: False \Rightarrow {\tt match}\ e\ {\tt with}\ {\tt end}). Definition consW \{X: {\tt Type}\}\ (h:X)\ (t: listW\ X): listW\ X:=sup\ (inr\ h)\ ({\tt fun}\ \_: unit \Rightarrow t).
```

Zauważ, że consW jest jedynie jednym z wielu możliwych kodowań konstruktora cons. Inaczej możemy go zakodować np. tak:

```
Definition consW' \{X: \mathtt{Type}\} (h:X) (t:listW|X):listW|X:=sup\ (inr\ h) (fun u:unit\Rightarrow \mathtt{match}\ u with |\ tt\Rightarrow t end).
```

Kodowania te nie są konwertowalne, ale jeżeli użyjemy aksjomatu ekstensjonalności dla funkcji, to możemy pokazać, że są równe.

Podobnym mykiem musimy posłużyć się, żeby udowodnić regułę indukcji. Dowód zaczynamy od indukcji po l (musimy pamiętać, że nasze W jest typem induktywnym, więc

ma regułę indukcji), ale nie możemy bezpośrednio użyć hipotez PnilW ani PconsW, gdyż dotyczą one innych kodowań nil i cons niż te, które pojawiają się w celu. Żeby uporać się z problemem, używamy taktyki replace, a następnie dowodzimy, że obydwa kodowania są ekstensjoalnie równe.

```
Lemma listW_{-}ind:
  \forall
     (X : \mathsf{Type}) \ (P : listW \ X \to \mathsf{Type})
     (PnilW : P (nilW X))
     (PconsW : \forall (h : X) (t : listW X), P t \rightarrow P (consW h t)),
       \forall l : listW X, P l.
Proof.
  induction l as [[[] \mid x] \mid b \mid IH].
    replace (P (sup (inl \ tt) \ b)) with (P (nilW \ X)).
       assumption.
       unfold nilW. do 2 f_equal. extensionality e. destruct e.
    replace with (P (consW \ x \ (b \ tt))).
       apply Pcons W. apply IH.
       unfold cons W. do 2 f_equal.
         extensionality u. destruct u. reflexivity.
Defined.
Check W_{-}ind.
Lemma listW_{-}ind':
     (X : \mathsf{Type}) \ (P : listW \ X \to \mathsf{Type})
     (PnilW : P (nilW X))
     (PconsW: \forall (h:X) (t:listW|X), P|t \rightarrow P|(consW|h|t)),
       \{f: \forall l: listW X, P l \mid
         f(nilW|X) = PnilW \wedge
         \forall (h:X) (t: listW|X), f (consW|h|t) = PconsW|h|t (f|t).
Proof.
  esplit. Unshelve. Focus 2.
     induction l as |||| \mid x \mid b \mid IH|.
       Print nilW.
       replace (P(sup(inl\ tt)\ b)) with (P(nilW\ X)).
         assumption.
         unfold nilW. do 2 f_equal. extensionality e. destruct e.
       replace _ with (P (consW \ x \ (b \ tt))).
         apply PconsW. apply IH.
         unfold cons W. do 2 f_equal.
            extensionality u. destruct u. reflexivity.
     cbn. split.
       compute.
```

Admitted.

Skoro mamy regulę indukcji, to bez problemu jesteśmy w stanie pokazać, że typy list X oraz listW X są izomorficzne, tzn. istnieją funkcje f: list $X \to listW$ X oraz g: listW $X \to list$ X, które są swoimi odwrotnościami.

```
Fixpoint f \{X : \mathsf{Type}\} (l : list X) : list W X :=
match l with
     \mid nil \Rightarrow nilW X
     | cons h t \Rightarrow cons W h (f t)
end.
Definition g\{X: \mathsf{Type}\}: \mathit{list}W\ X \to \mathit{list}\ X.
Proof.
  apply listW_ind.
     exact nil.
     intros h = t. exact (cons \ h \ t).
Defined.
Lemma fq:
  \forall \{X : \mathsf{Type}\} (l : \mathit{list}\ X),
     g(f l) = l.
Proof.
  induction l as [|h|t].
     unfold g in *. destruct (listW_{-}ind'_{-}) as (g \& eq1 \& eq2).
        cbn. apply eq1.
     unfold g in *; destruct (listW_{-}ind'_{-}) as (g \& eq1 \& eq2).
        cbn. rewrite eq2, IHt. reflexivity.
Qed.
Lemma gf:
  \forall \{X : \mathsf{Type}\} (l : \mathit{list}W \ X),
    f(g l) = l.
Proof.
  intro.
  apply listW_ind;
  unfold g; destruct (list W_{-}ind'_{-}) as (g \& eq1 \& eq2).
     rewrite eq1. cbn. reflexivity.
     intros. rewrite eq2. cbn. rewrite H. reflexivity.
Qed.
Definition boolW: Type :=
   W \ bool \ (fun \ \_ \Rightarrow Empty\_set).
Definition trueW : boolW :=
  sup\ true\ (fun\ e: Empty\_set \Rightarrow match\ e\ with\ end).
Definition \ falseW:boolW:=
```

```
sup\ false\ (fun\ e: Empty\_set \Rightarrow match\ e\ with\ end).
Definition notW:boolW\to boolW:=
   W\_rect\ bool\ (fun\ \_\Rightarrow Empty\_set)\ (fun\ \_\Rightarrow boolW)
            (fun b = \Rightarrow if b then falseW else trueW).
Definition bool\_boolW (b:bool):boolW :=
  if b then trueW else falseW.
Definition boolW\_bool:boolW \rightarrow bool:=
   W_{-}rect\ bool\ (fun\ \_\Rightarrow Empty\_set)\ (fun\ \_\Rightarrow bool)\ (fun\ b\ \_\ \_\Rightarrow b).
Lemma boolW\_bool\_notW:
  \forall b : boolW,
     boolW\_bool\ (notW\ b) = negb\ (boolW\_bool\ b).
Lemma boolW\_bool\_\_bool\_boolW:
  \forall b : bool,
     boolW\_bool\ (bool\_boolW\ b) = b.
Lemma bool\_boolW\_\_bool\_boolW:
  \forall b : boolW,
     bool\_boolW (boolW\_bool b) = b.
Definition natW: Type :=
   W \ bool \ (fun \ b : bool \Rightarrow if \ b \ then \ Empty\_set \ else \ unit).
Definition zero W : nat W :=
  sup\ true\ (fun\ e: Empty\_set \Rightarrow match\ e\ with\ end).
Definition succW (n: natW): natW :=
  sup \ false \ (fun \ u : unit \Rightarrow n).
Definition doubleW : natW \rightarrow natW :=
   W_{-}rect (fun b:bool \Rightarrow if b then Empty_{-}set else unit) (fun _{-} \Rightarrow natW)
     (fun a \Rightarrow
        match a with
             \mid true \Rightarrow fun \_ \_ \Rightarrow zeroW
             | false \Rightarrow fun _g \Rightarrow succ W (succ W (g tt))
        end).
Definition natW_{-}nat :=
   W_{-}rect _ (fun b:bool \Rightarrow if b then Empty_{-}set else unit) (fun _ \Rightarrow nat)
     (fun a \Rightarrow
        match a with
             | true \Rightarrow fun \_ \_ \Rightarrow 0
             | false \Rightarrow fun \ \_g \Rightarrow S \ (g \ tt)
        end).
Fixpoint nat_natW (n:nat):natW:=
match n with
```

```
 \mid 0 \Rightarrow zeroW \\ \mid S \ n' \Rightarrow succW \ (nat\_natW \ n') \\ \text{end.}  
 Lemma natW\_nat\_doubleW : \\ \forall \ n : natW, \\ natW\_nat \ (doubleW \ n) = 2 \times natW\_nat \ n. \\ \text{Lemma } natW\_nat\_nat\_natW : \\ \forall \ n : nat, \\ natW\_nat \ (nat\_natW \ n) = n. \\ \text{Lemma } nat\_natW\_nat\_nat\_natW : \\ \forall \ n : natW, \\ nat\_natW \ (natW\_nat \ n) = n. \\ \text{End } listW.
```

Ćwiczenie Napisałem we wstępie, że W-typy umożliwiają reprezentowanie dowolnych typów induktywnych, ale czy to prawda? Przekonajmy się!

Zdefiniuj za pomocą W następujące typy i udowodnij, że są one izomorficzne z ich odpowiednikami:

- False (czyli Empty_set)
- unit
- bool
- \bullet typ o n elementach
- liczby naturalne
- produkt
- sume

Załóżmy teraz, że żyjemy w świecie, w którym nie ma typów induktywnych. Jakich typów, oprócz W, potrzebujemy, by móc zdefiniować wszystkie powyższe typy?

17.2 Indeksowane W-typy

Jak głosi pewna stara książka z Palestyny, nie samymi W-typami żyje człowiek. W szczególności, W-typy mogą uchwycić jedynie dość proste typy induktywne, czyli takie, które wspierają jedynie parametry oraz argumenty indukcyjne. Na chwilę obecną wydaje mi się też, że W nie jest w stanie reprezentować typów wzajemnie induktywnych, lecz pewny nie jest jestem.

Trochę to smutne, gdyż naszą główną motywacją ku poznawaniu W-typów jest teoretyczne zrozumienie mechanizmu działania typów induktywnych, a skoro W jest biedne, to nie możemy za jego pomocą zrozumieć wszystkich typów induktywnych. Jednak uszy do góry, gdyż na ratunek w naszej misji przychodzą nam indeksowane W-typy!

Co to za zwierzę, te indeksowane W-typy? Ano coś prawie jak oryginalne W, ale trochę na sterydach. Definicja wygląda tak:

```
Inductive IW
(I: \mathsf{Type})
(S: I \to \mathsf{Type})
(P: \forall (i:I), S: i \to \mathsf{Type})
(r: \forall (i:I) (s:S:i), P: s \to I)
: I \to \mathsf{Type} :=
| \mathit{isup} :
\forall (i:I) (s:S:i),
(\forall p:P:s, IW:I:S:P:r(r:s:p)) \to IW:I:S:P:r:i.
Arguments \; \mathit{isup} \; \{I:S:P:r\} = --.
```

Prawdopodobnie odczuwasz w tej chwili wielką grozę, co najmniej jakbyś zobaczył samego Cthulhu. Nie martw się - zaraz dokładnie wyjasnimy, co tu się dzieje, a potem rozpracujemy indeksowane W typy na przykładach rodzin typów induktywnych, które powinieneś już dobrze znać.

Objaśnienia:

- *I* : Type to typ indeksów
- S i to typ kształtów o indeksie i. Kształt to konstruktor wraz ze swoimi argumentami nieindukcyjnymi.
- P s to typ mówiący, ile jest argumentów indukcyjnych o kształcie s.
- r p mówi, jak jest indeks argumentu indukcyjnego p.

Konstruktor isup mówi, że jeżeli mamy indeks i kształt dla tego indeksu, to jeżeli uda nam się zapchać wszystkie argumenty indukcyjne rzeczami o odpowiednim indeksie, to dostajemy element IW ... o takim indeksie jak chcieliśmy.

Czas na przykład:

```
\begin{array}{l} \textbf{Inductive } \textit{Vec } (\textit{A}: \texttt{Type}): \textit{nat} \rightarrow \texttt{Type} := \\ \mid \textit{vnil}: \textit{Vec } \textit{A} \; 0 \\ \mid \textit{vcons}: \; \forall \; n: \; \textit{nat}, \; \textit{A} \rightarrow \textit{Vec } \textit{A} \; n \rightarrow \textit{Vec } \textit{A} \; (\textit{S} \; n). \\ \textit{Arguments } \textit{vcons} \; \{\textit{A} \; n\} \; \_ \; \_. \end{array}
```

Na pierwszy ogień idzie Vec, czyli listy indeksowane długością. Jak wygląda jego reprezentacja za pomoca IW?

```
Definition I_{-}Vec (A : Type) : Type := nat.
```

Przede wszystkim musimy zauważyć, że typem indeksów I jest nat.

```
Definition S\_Vec \{A: \mathtt{Type}\} (i:I\_Vec\ A): \mathtt{Type}:= match i with  \mid 0 \Rightarrow unit \\ \mid S \ \_ \Rightarrow A  end.
```

Typ kształtów definiujemy przez dopasowanie indeksu do wzorca, bo dla różnych indeksów mamy różne możliwe kształty. Konstruktor vnil jest jedynym konstruktorem o indeksie 0 i nie bierze on żadnych argumentów nieindukcyjnych, stąd w powyższej definicji klauzula $| 0 \Rightarrow unit$. Konstruktor vcons jest jedynym konstruktorem o indeksie niezerowym i niezależnie od indeksu bierze on jeden argument nieindukcyjny typu A, stąd klauzula $| S |_{-} \Rightarrow A$.

```
Definition P\_Vec \{A: \mathtt{Type}\} \{i: I\_Vec\ A\} (s: S\_Vec\ i): \mathtt{Type}:= match i with  \mid 0 \Rightarrow False \\ \mid S \ \_ \Rightarrow unit end.
```

Typ pozycji różwnież definiujemy przez dopasowanie indeksu do wzorca, bo różne kształty będą miały różne pozycje, a przecież kształty też są zdefiniowane przez dopasowanie indeksu. Konstruktor vnil nie bierze argumentów indukcyjnych i stąd klauzula $\mid 0 \Rightarrow False$. Konstruktor vcons bierze jeden argument indukcyjny i stąd klauzula $\mid S \implies unit$.

Zauważmy, że niebranie argumentu nieindukcyjnego reprezentujemy inaczej niż niebranie argumentu indukcyjnego.

vnil nie bierze argumentów nie
indukcyjnych, co w typie kształtów S_Vec reprezentujemy za pomocą typu
 unit. Możemy myśleć o tym tak, że vnil bierze jeden argument typu unit.
 Ponieważ unit ma tylko jeden element, to i tak z góry wiadomo, że będziemy musieli jako ten argument wstawić tt.

vnil nie bierze też argumentów indukcyjnych, co w typue pozycji P_Vec reprezentujemy za pomocą typu False. Możemy myśleć o tym tak, że jest tyle samo argumentów indukcyjnych, co dowodów False, czyli zero.

Podsumowując, różnica jest taka, że dla argumentów nieindukcyjnych typ X oznacza "weź element typu X", zaś dla argumentów indukcyjnych typ X oznacza "weź tyle elementów typu, który właśnie definiujemy, ile jest elementów typu X".

```
 \begin{array}{l} {\rm Definition}\ r_- Vec \\ {\{A: {\rm Type}\}}\ \{i: I_- Vec\ A\}\ \{s: S_- Vec\ i\}\ (p: P_- Vec\ s): I_- Vec\ A. \\ {\rm Proof.} \\ {\rm destruct}\ i\ {\rm as}\ [|\ i'];\ cbn\ {\rm in}\ p. \\ {\rm destruct}\ p. \\ {\rm exact}\ i'. \\ {\rm Defined.} \end{array}
```

Pozostaje nam tylko zdefiniować funkcję, która pozycjom argumentów indukcyjnym w

poszczególnych kształtach przyporządkowuje ich indeksy. Definiujemy tę funkcję przez dowód, gdyż Coq dość słabo rodzi sobie z dopasowaniem do wzorca przy typach zależnych.

Ponieważ kształty są zdefiniowane przez dopasowanie indeksu do wzorca, to zaczynamy od rozbicia indeksu na przypadki. Gdy indeks wynosi zero, to mamy do czynienia z reprezentacją konstruktora vnil, który nie bierze żadnych argumentów indukcyjnych, co ujawnia się pod postacią sprzeczności. Gdy indeks wynosi S i', mamy do czynienia z konstruktorem vcons i', który tworzy element typu Vec A (S i'), zaś bierze element typu Vec A i'. Wobec tego w tym przypadku zwracamy i'.

```
Definition Vec'(A: \mathsf{Type}): nat \to \mathsf{Type} := IW(I\_Vec\ A)(@S\_Vec\ A)(@P\_Vec\ A)(@r\_Vec\ A).
```

Tak wygląda ostateczna definicja Vec' - wrzucamy do IW odpowiednie indeksy, kształty, pozycje oraz funkcję przypisującą indeksy pozycjom.

Spróbujmy przekonać się, że typy Vec~A~n oraz Vec'~A~n są izomorficzne. W tym celu musimy zdefiniować funkcje $f:Vec~A~n\to Vec'~A~n$ oraz $g:Vec'~A~n\to Vec~A~n$, które są swoimi odwrotnościami.

```
Fixpoint f \{A: \mathsf{Type}\} \{n: nat\} (v: Vec\ A\ n): Vec'\ A\ n. Proof. destruct v. apply (isup\ 0\ tt).\ cbn. destruct p. apply (@isup\ \_\ \_\ \_\ (@r\_Vec\ A)\ (S\ n)\ a).\ cbn. intros \_. exact (f\ \_\ \_\ v). Defined.
```

Najłatwiej definiować nam będzie za pomocą taktyk. Definicja f idzie przez rekursję strukturalną po v. isup~0~tt to reprezentacja vnil, zaś @ $isup~_-~_-$ (@ $r_Vec~A$) (S n) a to reprezentacja vcons~a. Dość nieczytelne, prawda? Dlatego właśnie nikt w praktyce nie używa indeksowanych W-typów.

```
Fixpoint g \{A: \mathsf{Type}\} \{n: nat\} (v: Vec' A n): Vec A n. Proof.

destruct v as [[|i'] s p]; cbn in s.

apply vnil.

apply (vcons s). apply g. apply (p tt).

Defined.
```

W drugą stronę jest łatwiej. Definicja idzie oczywiście przez rekursję po v (pamiętajmy, że Vec' A n to tylko specjalny przypadek IW, zaś IW jest induktywne). Po rozbiciu v na komponenty sprawdzamy, jaki ma ono indeks. Jeżeli 0, zwracamy vnil. Jeżeli niezerowy, to zwracamy vcons z głową s rekurencyjnie obliczonym ogonem.

```
 \begin{array}{l} \texttt{Lemma} \ f_-g : \\ \forall \ \{A: \texttt{Type}\} \ \{n: \ nat\} \ (v: \ Vec \ A \ n), \\ g \ (f \ v) = v. \\ \\ \texttt{Proof.} \\ \texttt{induction} \ v; \ cb \, n. \end{array}
```

```
reflexivity. rewrite \mathit{IHv}. reflexivity. Qed.
```

Dowód odwrotności w jedną stronę jest banalny - indukcja po v idzie gładko, bo v jest typu $Vec\ A\ n.$

Require Import FunctionalExtensionality.

```
Lemma g\_f: \forall \ \{A: \mathtt{Type}\} \ \{n: nat\} \ (v: \mathit{Vec'}\ A\ n), \\ f\ (g\ v) = v. \mathsf{Proof}. \mathsf{induction}\ v\ \mathsf{as}\ [[[\ i']\ s\ p\ IH]; \ \mathsf{unfold}\ I\_\mathit{Vec}\ \mathsf{in}\ ^*; \ \mathit{cbn}\ \mathsf{in}\ ^*. \\ \mathsf{destruct}\ s.\ \mathsf{f\_equal}.\ \mathsf{extensionality}\ x.\ \mathsf{destruct}\ x. \\ \mathsf{f\_equal}.\ \mathsf{extensionality}\ u.\ \mathsf{destruct}\ u.\ \mathsf{apply}\ \mathit{IH}. \mathsf{Qed}.
```

W drugę stronę dowód jest nieco trudniejszy. Przede wszystkim, musimy posłużyć się aksjomatem ekstensjonalności dla funkcji. Wynika to z faktu, że w *IW* reprezentujemy argumenty indukcyjne wszystkie na raz za pomocą pojedynczej funkcji.

Zaczynamy przez indukcję po v i rozbijamy indeks żeby sprawdzić, z którym kształtem mamy do czynienia. Kluczowym krokime jest odwinięcie definicji $I_{-}Vec$ - bez tego taktyka f_{-} equal nie zadziała tak jak powinna. W obu przypadkach kończymy przez użycie ekstensjonalności do udowodnienia, że argumenty indukcyjne sa równe.

Ćwiczenie Zdefiniuj za pomocą *IW* następujące predykaty/relacje/rodziny typów:

- even i odd
- typ drzew binarnych trzymających wartości w węzłach, indeksowany wysokością
- to samo co wyżej, ale indeksowany ilością elementów
- porządek < dla liczb naturalnych
- ullet relację $Perm: list \ A o list \ A o$ Prop mówiącą, że l1 i l2 są swoimi permutacjami

17.3 M-typy (TODO)

M-typy to to samo co W-typy, tylko że dualne. Pozdro dla kumatych.

Require Import F1.

Naszą motywacją do badania W-typów było to, że są one jedynym pierścieniem (w sensie Władcy Pierścieni, a nie algebry abstrakcyjnej), tj. pozwalają uchwycić wszystkie typy induktywne za pomocą jednego (oczywiście o ile mamy też False, unit, bool, prod i sum).

Podobnie możemy postawić sobie zadanie zbadania wszystkich typów koinduktywnych. Rozwiązanie zadania jest (zupełnie nieprzypadkowo) analogiczne do tego dla typów induktywnych, a są nim M-typy. Skąd nazwa? Zauważ, że M to nic innego jak W postawione na głowie - podobnie esencja M-typów jest taka sama jak W-typów, ale interpretacja jest postawiona na głowie.

```
 \begin{array}{l} \texttt{CoInductive} \ M \ (S : \texttt{Type}) \ (P : S \to \texttt{Type}) : \texttt{Type} := \\ \{ \\ shape : S; \\ position : P \ shape \to M \ S \ P \\ \}. \\ Arguments \ shape \ \{S \ P\}. \\ Arguments \ position \ \{S \ P\} \ \_ \ \_. \end{array}
```

Zastanówmy się przez chwilę, dlaczego definicja M wygląda właśnie tak. W tym celu rozważmy dowolny typ koinduktywny C i przyjmijmy, że ma on pola f1:X1,...,fn:Xn. Argumenty możemy podzielić na dwie grupy: koindukcyjne (których typem jest C lub funkcje postaci $B \to C$) oraz niekoindukcyjne (oznaczmy ich typy przez A).

Oczywiście wszystkie niekoindukcyjne pola o typach A1, ..., Ak możemy połączyć w jedno wielgachne pole o typie $A1 \times ... \times Ak$ i tym właśnie jest występujące w M pole shape. Podobnie jak w przypadku W-typów, typ S będziemy nazywać typem kształtów.

Pozostała nam jeszcze garść pól typu C (lub w nieco ogólniejszym przypadku, o typach $B1 \to C$, ..., $Bn \to C$. Nie trudno zauważyć, że można połączyć je w typ $B1 + ... + Bn \to C$. Nie tłumaczy to jednak tego, że typ pozycji zależy od konkretnego kształtu.

Źródeł można doszukiwać się w jeszcze jednej, nieco bardziej złożonej postaci destruktorów. Żeby za dużo nie mącić, rozważmy przykład. Niech C ma destruktor postaci $nat \to C + nat$. Jak dokodować ten destruktor do shape i position? Otóż dorzucamy do shape nowy komponent, czyli $shape' := shape \times nat \to option nat$.

A psu w dupę z tym wszystkim. TODO

```
Definition transport \{A: \mathsf{Type}\}\ \{P: A \to \mathsf{Type}\}\ \{x\ y: A\}\ (p: x=y)\ (u: P\ x): P\ y:= \mathsf{match}\ p\ \mathsf{with} |\ eq\_refl \Rightarrow u end. \mathsf{CoInductive}\ siM\ \{S: \mathsf{Type}\}\ \{P: S \to \mathsf{Type}\}\ (m1\ m2: M\ S\ P): \mathsf{Prop}:= \{ shapes: shape\ m1 = shape\ m2; positions: \forall\ p: P\ (shape\ m1), siM\ (position\ m1\ p)\ (position\ m2\ (transport\ shapes\ p)) \}. \mathsf{Definition}\ P\_Stream\ (A: \mathsf{Type}): A \to \mathsf{Type}:= \mathsf{fun}\ \_\Rightarrow unit.
```

```
Definition Stream'(A : Type) : Type := M \ A \ (P_Stream \ A).
CoFixpoint ff \{A : Type\} (s : Stream A) : Stream' A :=
\{ \mid
     shape := hd s;
     position = := ff(tl s);
|}.
CoFixpoint gg \{A : \mathsf{Type}\}\ (s : \mathit{Stream'}\ A) : \mathit{Stream}\ A :=
     hd := shape s;
     tl := gg \ (position \ s \ tt);
|}.
Lemma ff_{-}gg:
  \forall \{A : \mathsf{Type}\} \ (s : \mathit{Stream}\ A),
     bisim (gg (ff s)) s.
Proof.
  cofix \ CH.
  constructor; cbn.
     reflexivity.
     apply CH.
Qed.
Lemma gg_{-}ff:
  \forall \{A : \mathsf{Type}\} \ (s : \mathit{Stream}' \ A),
     siM (ff (gg\ s)) s.
Proof.
  cofix CH.
  econstructor. Unshelve.
     Focus 2. cbn. reflexivity.
     destruct p. cbn. apply CH.
Qed.
Definition coListM (A: Type): Type :=
  M \ (option \ A) \ (fun \ x : option \ A \Rightarrow
                     match x with
                         | None \Rightarrow False
                        | Some \_ \Rightarrow unit
                      end).
CoFixpoint fff \{A: \mathsf{Type}\}\ (l: coList\ A): coListM\ A:=
match uncons l with
       None \Rightarrow \{ | shape := None; position := fun \ e : False \Rightarrow match \ e \ with \ end \ | \}
     | Some (h, t) \Rightarrow \{ | shape := Some h; position := fun \_ \Rightarrow @fff \_ t | \}
end.
Print coBTree.
```

```
Definition coBTreeM (A: Type): Type:= M \ (option \ A) \ (fun \ x: option \ A \Rightarrow  match x with |\ None \Rightarrow False \ |\ Some \ \_ \Rightarrow bool end).
```

17.4 Indeksowane M-typy?

Nie dla psa kiełbasa.

17.5 Kodowanie Churcha (TODO)

Achtung: póki co wisi tu kod roboczy

```
Definition clist (A : Type) : Type :=
  \forall X : \mathsf{Type}, X \to (A \to X \to X) \to X.
Definition cnil \{A : Type\} : clist A :=
  fun X nil cons \Rightarrow nil.
Definition ccons \{A : \mathsf{Type}\} : A \to clist A \to clist A :=
  fun h \ t \Rightarrow fun X \ nil \ cons \Rightarrow cons \ h \ (t \ X \ nil \ cons).
Notation \dot{\mathbf{c}}[]":= cnil.
Notation "x :c: y":= (ccons \ x \ y) (at level 60, right associativity).
Notation c|x; ...; y| := (ccons x ... (ccons y cnil) ..).
Definition head \{A : Type\} (l : clist A) : option A :=
  l (option A) None (fun h <math> \Rightarrow Some h).
(*
   Definition tail {A : Type} (1 : clist A) : option (clist A) :=
  1 _ None
     (fun h t => t).
    Compute tail c1; 2; 3.
    *)
Definition null \{A : \mathsf{Type}\} (l : clist A) : bool :=
  l - true (fun - \Rightarrow false).
Definition clen \{A : \mathsf{Type}\} (l : clist A) : nat :=
  l \ nat \ 0 \ (fun \ \_ \Rightarrow S).
Definition snoc \{A : \mathsf{Type}\}\ (l : clist \ A)\ (x : A) : clist \ A :=
  fun X nil cons \Rightarrow l \ (c[x] \ nil \ cons) cons.
```

```
Definition rev \{A : Type\} (l : clist A) : clist A.
Proof.
  unfold clist in *.
  intros X nil cons.
Abort.
Definition capp \{A : \mathsf{Type}\} (l1 \ l2 : clist \ A) : clist \ A :=
  fun X nil cons \Rightarrow l1 X (l2 X nil cons) cons.
Fixpoint fromList \{A : \mathsf{Type}\}\ (l : list A) : clist A :=
match l with
     | | | \Rightarrow cnil
     | h :: t \Rightarrow ccons \ h \ (from List \ t)
end.
{\tt Definition}\ toList\ \{A:{\tt Type}\}\ (l:\ clist\ A):list\ A:=
  l (list A) [] (@cons A).
Lemma toList\_fromList:
  \forall (A : \mathsf{Type}) (l : \mathit{list} A),
     toList\ (fromList\ l) = l.
Proof.
  induction l as [|h|t]; compute in *; rewrite ?IHt; reflexivity.
Qed.
Lemma fromList\_toList:
  \forall (A : \mathsf{Type}) (cl : clist A),
     fromList\ (toList\ cl) = cl.
  intros. unfold clist in *. compute.
Abort.
Definition wut : Type :=
  \forall X : \mathsf{Type}, (X \to X) \to X.
```

H1: Uniwersa - pusty

Chwilowo nic tu nie ma.

 ${\tt Set}\ {\it Universe}\ {\it Polymorphism}.$

Require Import Arith.

Require Import Bool.

Ćwiczenie Miło by było pamiętać, że Coq to nie jest jakiś tam biedajęzyk programowania, tylko pełnoprawny system podstaw matematyki (no, prawie...). W związku pokaż, że $nat \neq$ Type.

Ćwiczenie To samo co wyżej, ale tym razem dla dowolnego typu, który ma rozstrzygalną równość oraz spełnia aksjomat K.

Ćwiczenie Dobrze wiemy, że Prop to nie Type... a może jednak? Rozstrzygnij, czy Prop = Type zachodzi, czy nie.

H2: Równość - pusty

Chwilowo nic tu nie ma.

I1: 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 meke.

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/proofengine/ltac.html

20.1 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.

```
\begin{array}{l} \texttt{Goal} \ \forall \ P \ Q \ R : \texttt{Prop}, \ P \land \ Q \land R \rightarrow R \land \ Q \land P. \\ \texttt{Proof.} \\ \texttt{repeat split.} \\ Focus \ 3. \\ Unfocus. \\ Focus \ 2. \\ \texttt{Abort.} \end{array}
```

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:

- \bullet n: t użyj taktyki t na n-tym celu. 1: t jest równoważne t.
- $\bullet \ a\text{-}b$: 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 $> t_-1 \mid \dots \mid t_-n$, które aplikuje taktykę t_-i do i-tego celu zaznaczonego danym selektorem

```
Goal \forall \ P \ Q \ R : {\tt Prop}, \ P \land \ Q \land R \to R \land \ Q \land 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: [H\ [H'\ H'']].
```

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 \forall \ P \ Q \ R : {\tt Prop}, \ P \land \ Q \land R \to R \land \ Q \land P. Proof.
destruct 1 as [H \ [H' \ H'']].
repeat split; only \ 1\text{-}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.

```
\begin{array}{l} \texttt{Goal} \ \forall \ P \ Q \ R \ S : \texttt{Prop}, \ P \rightarrow P \ \land \ Q \ \land \ R \ \land \ S. \\ \texttt{Proof.} \\ \texttt{repeat split.} \\ \texttt{revgoals.} \ all: \ revgoals. \ all: \ revgoals. \\ \texttt{swap 1 3.} \ all: \ swap 1 3. \ all: \ swap 1 3. \\ \texttt{cycle 42.} \ all: \ cycle 3. \ all: \ cycle -3. \\ \texttt{Abort.} \end{array}
```

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 nie (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.

20.2 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.

```
\begin{array}{l} \texttt{Goal} \  \, True. \\ \texttt{Proof.} \\ \texttt{pose} \  \, true. \\ \texttt{pose} \  \, (nazwa := 123). \\ \texttt{Abort.} \end{array}
```

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 := \text{constr}:(42) in pose x.
let x := \text{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 fixowi 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 \Rightarrow 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 $name\ := fun\ arg_1\ ...\ arg_n \Rightarrow body$. Jest to uwaga mocno techniczna, gdyż pierwszy zapis jest prawie zawsze preferowany wobec drugiego.

20.3 Backtracking

Poznałeś już kombinator alternatywy ||. Nie jest to jednak jedyny kombinator służący do wyrażania tej idei — są jeszcze kombinatory + oraz tryjf 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 moga 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:=\exists\ n\mid\mid existsNatFrom\ (S\ n).
Ltac existsNat:=existsNatFrom\ O.
Goal \exists\ n:\ nat,\ n=42.
Proof.
Fail (existsNat;\ reflexivity).
Abort.
Ltac existsNatFrom'\ n:=\exists\ n+existsNatFrom'\ (S\ n).
Ltac existsNat':=existsNatFrom'\ O.
Goal \exists\ 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 porażka.

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:

- ∃ 0; reflexivity porażka
- ∃ 1; reflexivity porażka
- ∃ 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 maja 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 ∃ 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 \exists 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 \exists \ 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:
\exists \ 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 : {\tt Prop.}
```

Theorem $search_-\theta$:

$$I \to A \vee B \vee C \vee D \vee E \vee F \vee G \vee I \vee J.$$

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 \vee J \vee ((A \vee A \vee (C \vee D \vee (E \vee E))))).
Proof. search. Qed.
Theorem search_4:
   A \rightarrow 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 ((\bar{B} \lor (I \to I) \lor (J \to J)) \lor (D \lor (\bar{D} \to \bar{D}) \lor B \lor B)).
Proof. search. Qed.
Theorem search_{-}6:
   C \to ({}^{\sim}C \wedge {}^{\sim}{}^{\sim}C) \vee ((C \wedge \neg C) \vee ({}^{\sim}C \wedge C) \vee (C \to C) \vee (C \vee \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 $[t_-1 \mid \dots \mid t_-N]$
- \bullet solve $[t_1 \mid ... \mid t_N]$

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 +.

20.4 Dopasowanie kontekstu i celu

Chyba najważniejszym konstruktem Ltaca jest match goal, który próbuje dopasować kontekst oraz cel do podanych wzorców. Mają one postać | $kontekst \vdash cel \Rightarrow taktyka$.

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 \Rightarrow następuje taktyka, której chcemy użyć, jeżeli dany wzorzec zostanie dopasowany.

Zamiast wzorców postaci | $kontekst \vdash cel \Rightarrow taktyka$ możemy też używać wzorców postaci | $\vdash cel \Rightarrow taktyka$, które dopasowują jedynie cel, zaś kontekst ignorują; wzorców postaci |

 $kontekst \vdash _ \Rightarrow taktyka$, które dopasowują jedynie kontekst, a cel ignorują; oraz wzorca $_$, który oznacza "dopasuj cokolwiek".

Zobaczmy, jak to wygląda na przykładach.

```
\begin{array}{l} \texttt{Goal} \\ \forall \ P \ Q \ R \ S : \texttt{Prop}, \ P \rightarrow Q \rightarrow R \rightarrow S. \\ \texttt{Proof.} \\ \texttt{intros.} \\ \texttt{match goal with} \\ \mid x : \texttt{Prop} \vdash \_ \Rightarrow \texttt{idtac} \ x \\ \texttt{end.} \\ \texttt{Abort.} \end{array}
```

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 \Rightarrow 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 \forall \ P \ Q \ R \ S : {\tt Prop}, \ P \to Q \to R \to S. Proof. intros. Fail match goal with \mid x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x; {\tt 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.
- ullet 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 \forall \ P \ Q \ R \ S : {\tt Prop}, \ P \to Q \to R \to S. Proof. intros. Fail match reverse goal with \mid x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x; {\tt 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 matcha wynika stąd, że jeżeli taktyka po prawej od \Rightarrow zawiedzie, to następuje próba znalezienia jakiegoś innego dopasowania wzorca x: Prop. Dopiero gdy taktyka na prawo od \Rightarrow zawiedzie dla wszystkich możliwych takich dopasowań, cały match zawodzi.

```
\begin{array}{l} \texttt{Goal} \\ \forall \ P \ Q \ R \ S : \texttt{Prop}, \ P \rightarrow \ Q \rightarrow R \rightarrow S. \\ \\ \texttt{Proof.} \\ \texttt{intros.} \\ Fail \\ \texttt{match goal with} \\ | \ x : \texttt{Prop} \vdash \_ \Rightarrow \texttt{idtac} \ x \\ \texttt{end; fail.} \\ \\ \texttt{Abort.} \end{array}
```

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
```

```
orall P \ Q \ R \ S : {\tt Prop}, \ P 	o Q 	o R 	o S. Proof. intros. Fail lazymatch goal with | \ x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x; \ {\tt fail}
```

end.

Abort.

Konstrukt lazymatch różni się od matcha tym, że jeżeli taktyka na prawo od \Rightarrow 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
```

```
\begin{array}{l} \forall \ P \ Q \ R \ S : {\tt Prop}, \ P \to Q \to R \to S. \\ {\tt Proof.} \\ & {\tt intros.} \\ & Fail \\ & multimatch \ {\tt goal \ with} \\ & \mid x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x \\ & {\tt end; \ fail.} \\ {\tt Abort.} \end{array}
```

multimatch to wariant matcha, 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.
- \bullet Znajdujemy dopasowanie S: Prop i cały multimatchkończy 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
```

```
orall \ P \ Q \ R \ S : {\tt Prop}, \ P 	o Q 	o R 	o S. Proof.
intros.
match goal with
 \mid x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x
end.
 multimatch \ {\tt goal} \ {\tt with}
 \mid x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x
end.
 {\tt repeat} \ {\tt match} \ {\tt goal} \ {\tt with}
 \mid x : {\tt Prop} \vdash \_ \Rightarrow {\tt idtac} \ x
```

```
end. repeat multimatch goal with \mid x: \texttt{Prop} \vdash \_ \Rightarrow \texttt{idtac} \ x end. Abort.
```

Przyjrzyjmy się jeszcze różnicy w zachowaniach matcha 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 matcha 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
```

```
\forall \; (P \; Q \; R \; S : \texttt{Prop}) \; (a \; b \; c : \; nat), \\ 42 = 43 \; \land \; (P \rightarrow Q). \texttt{Proof.} \texttt{intros. split}; \texttt{match goal with} \mid X : \texttt{Prop} \vdash P \rightarrow Q \Rightarrow \texttt{idtac} \; X \mid n : \; nat \vdash 42 = 43 \Rightarrow \texttt{idtac} \; n \texttt{end.} \texttt{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 \forall \ (P \ Q \ R \ S : \mathsf{Prop}) \ (a \ b \ c : nat), \ P. Proof. intros. match goal with |\ \_ \Rightarrow \mathsf{idtac} \ -\_-" end. match goal with |\ \_ \Rightarrow \mathsf{fail} \ |\ X : \mathsf{Prop} \vdash \_ \Rightarrow \mathsf{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 matchu 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 $destruct_{-}\theta$:

$$A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G \wedge H \wedge I \wedge J \rightarrow D.$$

Proof. $destr_and$. Qed.

Theorem $destruct_1$:

Proof. $destr_and$. Qed.

Theorem $destruct_2$:

$$A \wedge \neg B \wedge (C \vee C \vee C \vee C) \wedge ((((D \wedge I) \wedge I) \wedge I) \wedge I) \wedge J) \rightarrow I.$$

Proof. destr_and. Qed.

End destr_and.

Ćwiczenie (solve_and_perm) Napisz taktykę $solve_and_perm$, która będzie potrafiła rozwiązywać cele postaci $P_1 \land P_2 \land \dots \land P_n \rightarrow P_i1 \land P_i2 \land \dots \land P_iN$, 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_θ :

$$\begin{array}{c} A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G \wedge H \wedge I \wedge J \rightarrow \\ J \wedge I \wedge H \wedge G \wedge F \wedge E \wedge D \wedge C \wedge B \wedge A. \end{array}$$

Proof. $solve_and_perm$. Qed.

Theorem and_perm_1 :

$$\begin{array}{l} A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G \wedge H \wedge I \wedge J \rightarrow \\ (((((((((A \wedge B) \wedge C) \wedge D) \wedge E) \wedge F) \wedge G) \wedge H) \wedge I) \wedge J). \end{array}$$

Proof. $solve_and_perm$. Qed.

Theorem and_perm_2 :

$$(A \wedge B) \wedge (C \wedge (D \wedge E)) \wedge (((F \wedge G) \wedge H) \wedge I) \wedge J \rightarrow (I \wedge I \wedge J) \wedge ((A \wedge B \wedge (A \wedge B)) \wedge J) \wedge (C \wedge (E \wedge (D \wedge F \wedge F))).$$

Proof. $solve_and_perm$. Qed.

End $solve_and_perm$.

Cwiczenie (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 :

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 :

$$\begin{array}{l} A \vee B \vee C \vee D \vee E \vee F \vee G \vee H \vee I \vee J \rightarrow \\ ((((((((A \vee B) \vee C) \vee D) \vee E) \vee F) \vee G) \vee H) \vee I) \vee J). \end{array}$$

Proof. $solve_or_perm$. Qed.

End $solve_or_perm$.

Ćwiczenie (negn) Section negn.

Require Import Arith.

Napisz funkcję $negn: nat \to \mathtt{Prop} \to \mathtt{Prop},$ gdzie $negn \ n \ P$ zwraca zdanie P zanegowane n razy.

```
Eval cbn in negn 10 True.

(* ===> = ~ ~ ~ ~ ~ ~ ~ True : Prop *)
```

Udowodnij poniższe lematy.

Lemma dbl_neg :

```
\forall P : \mathtt{Prop}, P \rightarrow \neg \neg P.
```

Lemma $double_n$:

$$\forall n : nat, 2 \times n = n + n.$$

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:
```

```
\forall (n : nat) (P : Prop), P \rightarrow negn (2 \times n) P.
```

Theorem $even_neg'$:

```
\forall (n \ k : nat) \ (P : Prop),
negn \ (2 \times n) \ P \rightarrow negn \ (2 \times (n+k)) \ P.
```

Napisz taktykę negtac, która będzie potrafiła udowadniać cele postaci $\forall P$: Prop, negn $(2 \times n)$ $P \rightarrow negn$ $(2 \times (n+k))$ P, gdzie n oraz k są stałymi. Nie używaj twierdzeń, które udowodniłeś wyżej.

Wskazówka: przydatny może byc konstrukt match reverse goal.

Theorem neg_22_14 :

```
\forall P : \mathsf{Prop}, \ negn \ 2 \ P \rightarrow negn \ 14 \ P.
```

Proof. negtac. Qed.

Theorem neg_100_200 :

 $\forall P : \text{Prop}, negn \ 100 \ P \rightarrow negn \ 200 \ P.$

Proof. negtac. Qed.

Theorem $neg_{-4}2_{-1}000$:

```
\forall \ P : \texttt{Prop}, \ negn \ 42 \ P \rightarrow negn \ 200 \ P. \texttt{Proof}. \ negtac. \ \texttt{Qed}. \texttt{End} \ negn.
```

20.5 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.

```
\begin{array}{l} \texttt{Goal} \\ \forall \ P \ Q : \texttt{Prop}, \ P \to P \ \lor \ Q. \\ \texttt{Proof.} \\ \texttt{intros.} \\ \texttt{match goal with} \\ \mid \ p : \ P \vdash P \ \lor \ Q \Rightarrow \texttt{left}; \texttt{assumption} \\ \texttt{end.} \\ \texttt{Qed.} \end{array}
```

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 $P \vee Q$, 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.

```
\begin{array}{l} \texttt{Goal} \\ \forall \ A \ B : \texttt{Prop}, \ A \rightarrow A \lor B. \\ \texttt{Proof.} \\ \texttt{intros.} \\ Fail \ \texttt{match goal with} \\ \mid p : P \vdash P \lor Q \Rightarrow \texttt{left}; \ \texttt{assumption} \\ \texttt{end.} \\ \texttt{match goal with} \end{array}
```

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
```

```
\forall~A~B~: \texttt{Prop},~A \to A \lor B. \texttt{Proof}. \texttt{intros}. \texttt{match goal with} \mid~p~:~?P \vdash ?P \lor ?Q \Rightarrow \texttt{idtac}~P;~\texttt{idtac}~Q;~\texttt{left};~\texttt{assumption}~\texttt{end}. \texttt{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 \Rightarrow 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 matcha 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
        |\vdash ?x = \_ \Rightarrow idtac x
  end.
  match goal with
        \mid \vdash cons ?h \_ = nil \Rightarrow idtac h
  end.
  match goal with
        |\vdash 2 :: \_ = ?l \Rightarrow idtac l
  end.
  match goal with
        |\vdash[?x]=[]\Rightarrow 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? $h_{\perp} = nil$ 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 matcha.

```
\begin{array}{l} \texttt{Goal} \\ \forall \ P : \texttt{Prop}, \ P \rightarrow P. \\ \\ \texttt{Proof.} \\ \texttt{intros.} \ my\_assumption. \\ \\ \texttt{Qed.} \end{array}
```

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.

```
\begin{array}{l} {\tt Example} \ forward\_1 : \\ \forall \ P \ Q \ R : {\tt Prop}, \ (P \to Q) \to (Q \to R) \to P \to R. \\ {\tt Proof.} \ forward. \ {\tt Qed}. \\ {\tt Example} \ forward\_2 : \\ \forall \ P \ Q \ R \ S : {\tt Prop}, \ (P \to Q \to R) \to (S \to Q \to P \to R). \\ {\tt Proof.} \ forward. \ {\tt Qed}. \end{array}
```

20.6 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.

20.6.1 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 \forall \ a \ b \ c: \ nat, \ a=b \rightarrow b=c \rightarrow a=c. Proof. intros a \ b \ c.
```

```
match goal with |\vdash \mathsf{context} \ G \ [?x = ?y] \Rightarrow \mathsf{idtac} \ G \ x \ y end. \mathsf{repeat} \ multimatch \ \mathsf{goal} \ \mathsf{with} |\vdash \mathsf{context} \ G \ [?x = ?y] \Rightarrow \mathsf{idtac} \ G \ x \ y end. \mathsf{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:

- ullet 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.

20.6.2 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 \forall \ x \ y : \ nat, \ \{x = y\} + \{x \neq 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:=\operatorname{fresh} x in intro x. Restart. intro x. let x:=\operatorname{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 matcha, nie znamy oryginalnych nazw dopasowanych termów — możemy odwoływać się do nich tylko za pomocą nazw lokalnych, wprowadzonych na potrzeby danego wzorca.

Z odsięczą 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".

20.6.3 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 matcha i przejście do następnej. Użycie taktyki fail n, gdzie n nie jest równe 0, powoduje opuszczenie całego obecnego matcha (tj. wszystkich gałęzi) lub bloku do/repeat i wywołanie fail (n - 1).

Przyjrzyjmy się temu zachowaniu na przykładzie.

```
Goal False.
Proof.
  match goal with
        | \_ \Rightarrow idtac "first branch"; fail
        | \_ \Rightarrow idtac \text{ second branch}''
  end.
  Fail match goal with
        | \_ \Rightarrow idtac "first branch"; fail 1
        | _ ⇒ idtac second branch"
  end.
  try match goal with
        | \_ \Rightarrow idtac "first branch"; fail 1
        | \_ \Rightarrow idtac \text{ second branch}''
  end.
  Fail try match goal with
        | \_ \Rightarrow idtac "first branch"; fail 2
        | _ ⇒ idtac second 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.

20.7 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 \wedge False \wedge 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).

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 \rightarrow \operatorname{Prop} := | even 0 : even 0 : even SS: \forall n: nat, even n \rightarrow even (S(Sn)).

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.

20.8 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 21

12: 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.

21.1 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 przyjał się i od tej

pory Coqowi bogowie poczęli używać jej do tworzenia coraz to innych taktyk. Tak refine stała się matka 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_-\theta: 42=42. Proof. refine eq_-refl. Qed.
```

W powyższym przykładzie używamy refine tak jak użylibyśmy exacta. 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: \forall P \ Q : \operatorname{Prop}, P \land Q \to Q \land P. Proof. refine (fun P \ Q : \operatorname{Prop} \Rightarrow \_). refine (fun H \Rightarrow \operatorname{match} H \text{ with } | \ conj \ p \ q \Rightarrow \_ \ \operatorname{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 \Rightarrow _). 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 $P \land Q \rightarrow Q \land P$, 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 $H\Rightarrow$ match H with | conj p $q\Rightarrow$ _ 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 $H\Rightarrow$..., zaś p i q znajdą się w kontekście dzięki temu, że

zostały związane podczas dopasowania conj p q.

Teraz naszym celem jest $Q \wedge P$. 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: \forall P: \texttt{Prop}, P \to 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: \forall P: \texttt{Prop}, P \to P. Proof. my\_intro1.\ my\_intro2\ H.\ my\_exact\ H. Restart. my\_intro.\ my\_intro\ H.\ my\_exact\ H. Qed.
```

Čwiczenie ($\mathbf{my}_{-}\mathbf{apply}$) Napisz taktykę $my_{-}apply~H$, która działa tak jak $\mathbf{apply}~H$. Użyj taktyki refine.

```
Example my\_apply\_0: \forall \ P \ Q : \texttt{Prop}, \ P \to (P \to Q) \to 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\_\theta:
  \forall P \ Q : \mathsf{Prop}, P \to Q \to P \land Q.
   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\_\theta:
  \forall P : \mathtt{Prop}, P \rightarrow P \vee P.
   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_-\theta:
   \exists n : nat, n = 42.
Proof.
   my_-exists 42. reflexivity.
Qed.
```

21.2 Drobne taktyki

21.2.1 clear

```
Goal \forall \ x \ y : nat, \ x = y \rightarrow y = x \rightarrow 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 $a,\,b,\,c:\,\mathit{True},\,p:P\vdash _\le p$: $P\vdash _$.

```
Section tru.
```

```
Example tru_-0: \forall P: \texttt{Prop}, \ True \to True \to P. Proof. tru. \ (* \ \texttt{Kontekst}: P: \texttt{Prop} \vdash P \ *) Abort. Example tru_-1: \ True. Proof. tru. \ \texttt{Qed}. End tru.
```

```
Cwiczenie (satans_neighbour_not_even) Inductive even: nat \rightarrow Prop := | even 0 : even 0 | even SS : <math>\forall n : nat, even n \rightarrow even (S(S(n))).
```

Napisz taktykę even, która potrafi udowodnić poniższy cel.

Theorem $satans_neighbour_not_even : \neg even 667$.

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 $\operatorname{destruct} x$, gdzie x jest dowolnego typu induktywnego.

```
\begin{array}{l} {\tt Example} \ my\_destruct\_and\_0: \\ \forall \ P \ Q: {\tt Prop}, \ P \land Q \rightarrow P. \\ {\tt Proof}. \\ my\_intro \ P; \ my\_intro \ Q; \ my\_intro \ H. \\ my\_destruct\_and \ H \ p \ q. \ my\_exact \ p. \\ {\tt Restart}. \\ my\_intro \ P; \ my\_intro \ Q; \ my\_intro \ H. \\ my\_destruct\_and \ H. \ my\_exact \ H0. \\ {\tt Qed}. \end{array}
```

21.2.2 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.

Čwiczenie (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_-\theta: \forall \ n \ m: \ nat, \ n+m=m+n. Proof. intros. unfold plus. fold plus. Restart. intros. unfold plus. my\_fold \ plus. Abort.
```

21.2.3 move

```
Example move_{-}\theta: \forall~P~Q~R~S~T: Prop, P \wedge Q \wedge R \wedge S \wedge T \rightarrow T. Proof. destruct 1 as [p~[q~[r~[s~t]]]]. move p after t. move p before s. move p at top.
```

```
\label{eq:move_p} \text{move } p \text{ 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.

21.2.4 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.

21.2.5 rename

```
\label{eq:coal_prop} \begin{array}{l} \operatorname{Goal} \ \forall \ P : \operatorname{Prop}, \ P \to P. \\ \operatorname{Proof.} \\ \operatorname{intros. rename} \ H \ into \ wut. \\ \operatorname{Abort.} \end{array}
```

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 $satans_neighbour_not_even'$: $\neg even 667$.

21.2.6 *admit*

```
Module admit.

Lemma forgery:
\forall \ P \ Q: \operatorname{Prop}, \ P \to Q \land P.

Proof.
intros. split.
admit.
assumption.

Admitted.

Print forgery.
(* ===> *** [ forgery: \forall \ P: \operatorname{Prop}, \ P \to \neg \ P \land 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 wczesniejszego użycia taktyki admit. Różnica między tymi dwoma bytami jest taka, że taktyka admit służy do "udowodnienia" pojedynczego celu, a komenda Admitted — całego twierdzenia.

21.3 Średnie taktyki

21.3.1 $case_eq$

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
```

```
orall \ 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 orall \ 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.
```

21.3.2 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:
\forall P : \texttt{Prop}, False \rightarrow P.

Proof.
contradiction.

Restart.
my\_contradiction.

Qed.

Example my\_contradiction\_1:
\forall P : \texttt{Prop}, \neg True \rightarrow P.

Proof.
contradiction.

Restart.
my\_contradiction.

Restart.
my\_contradiction.
```

```
Qed.
Example my\_contradiction\_2:
  \forall (P : \mathsf{Prop}) (n : nat), n \neq n \rightarrow P.
Proof.
  contradiction.
Restart.
  my\_contradiction.
Qed.
Example my\_contradiction\_3:
  \forall P \ Q : \mathsf{Prop}, P \to \neg P \to Q.
Proof.
  contradiction.
Restart.
  my\_contradiction.
Qed.
Example my\_contradiction\_4:
  \forall P : \texttt{Prop}, true = false \rightarrow P.
Proof.
  try contradiction.
Restart.
  my\_contradiction.
Qed.
Example my\_contradiction\_5:
  \forall P : \texttt{Prop}, false = true \rightarrow 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.

21.3.3 constructor

```
\begin{array}{l} \texttt{Example} \ constructor\_\theta : \\ \forall \ P \ Q : \texttt{Prop}, \ P \to Q \ \lor \ P. \\ \texttt{Proof}. \\ \texttt{intros}. \ \texttt{constructor} \ 2. \ \texttt{assumption}. \end{array}
```

```
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.
```

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 ∃?

21.3.4 decompose

```
 \begin{array}{l} \texttt{Example} \ decompose\_0: \\ \forall \ P \ Q \ R \ S: nat \to \texttt{Prop}, \\ (\exists \ n: nat, \ P \ n) \ \land (\exists \ n: nat, \ Q \ n) \ \land \\ (\exists \ n: nat, \ R \ n) \ \land (\exists \ n: nat, \ S \ n) \to \\ \exists \ n: nat, \ P \ n \lor Q \ n \lor R \ n \lor S \ n. \\ \\ \texttt{Proof}. \\ \texttt{intros}. \ decompose \ [and \ ex] \ H. \ \texttt{clear} \ H. \ \exists \ x. \ \texttt{left.} \ \texttt{assumption}. \\ \\ \texttt{Qed}. \end{array}
```

decompose to bardzo użyteczna taktyka, która potrafi za jednym zamachem rozbić bardzo skomplikowane hipotezy. $decompose [t_-1 \dots t_-n] H$ 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.

21.3.5 intros

Dotychczas używałeś taktyk intro i intros jedynie z nazwami lub wzorcami do rozbijania elementów typów induktywnych. Taktyki te potrafią jednak dużo więcej.

```
\begin{array}{l} {\rm Example} \ intros\_\theta: \\ \forall \ P \ Q \ R \ S: {\rm Prop}, \ P \land Q \land R \to S. \\ {\rm Proof.} \\ \quad {\rm intros} \ P \ Q \ R \ S \ [p \ [q \ r]]. \\ {\rm Restart.} \\ \quad {\rm intros} \ ? P \ Q \ R. \ {\rm intros} \ (p, \ (p\theta, \ q)). \\ {\rm Restart.} \\ \quad {\rm intros} \ \times. \\ {\rm Restart.} \\ \quad {\rm intros} \ A \ B \ **. \\ {\rm Restart.} \\ \quad {\rm intros} \ \times \_. \\ {\rm Restart.} \\ \quad {\rm intros} \ \times \_. \\ {\rm Restart.} \\ \quad {\rm Intros} \ \times \_. \\ {\rm Restart.} \\ \quad {\rm Intros} \ \times \_. \\ {\rm Abort.} \end{array}
```

Pierwszy przykład to standardowe użycie intros — wprowadzamy cztery zmienne, która nazywamy kolejno P, Q, R i S, po czym wprowadzamy bezimienną hipotezę typu $P \wedge Q \wedge R$, która natychmiast rozbijamy za pomocą wzorca $p \mid q \mid r \mid$.

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 $P \wedge Q \wedge R$ 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 $P \wedge Q \wedge R$, gdyż żaden term od niej nie zależy. Jednak w ostatnim przykładzie nie możemy usunąć P, gdyż zależy od niego hipoteza $P \wedge Q \wedge R$.

```
Example intros\_1: \forall \ P0 \ P1 \ P2 \ P3 \ P4 \ P5: Prop, P0 \ \land P1 \ \land P2 \ \land P3 \ \land P4 \ \land P5 \ \rightarrow P3. Proof. intros \times \ [p0 \ [p1 \ [p2 \ [p3 \ [p4 \ p5]]]]]. Restart. intros \times \ (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_{-}\mathcal{Z}: \forall x \ y: nat, \ x=y \rightarrow y=x. Proof. intros \times \rightarrow . Restart. intros \times \leftarrow . Abort.
```

Wzorców \rightarrow oraz \leftarrow możemy użyć, gdy chcemy wprowadzić do kontekstu równanie, przepisać je i natychmiast się go pozbyć. Wobec tego taktyka intros \rightarrow jest równoważna czemuś w stylu intro H; rewrite H in *; clear H (oczywiście pod warunkiem, że nazwa H nie jest zajęta).

```
Example intros\_3: \forall~a~b~c~d:~nat,~(a,~b)=(c,~d)\rightarrow a=c. Proof. Fail intros \times~[p1~p2]. Restart. intros \times~[=~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 koniunkjcę wywnioskować. Taki właśnie efekt ma wzorzec = p1 p2 — dodaje on nam do kontekstu hipotezy p1: a = c oraz p2: b = d.

```
Example intros\_4: \forall~P~Q~R: \texttt{Prop},~(P \to Q) \to (Q \to R) \to P \to R. Proof.
```

```
intros until 2. intro p. apply H in p. apply H\theta in p. Restart. intros until 2. intros p \%H \%H\theta. 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 $x \% H_- 1 \dots \% H_- n$ 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.

21.3.6 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!).

```
\begin{array}{l} \text{Definition } nat\_ind\_fix\_term \\ (P:nat \rightarrow \texttt{Prop}) \; (H0:P\;0) \\ (HS:\forall\; n:nat,P\;n \rightarrow P\;(S\;n)) \\ : \; \forall\; n:nat,P\;n := \\ \quad \text{fix}\; f\;(n:nat):P\;n := \\ \quad \text{match } n\; \text{with} \\ \quad \mid 0 \Rightarrow H0 \\ \quad \mid S\;n' \Rightarrow HS\;n'\;(f\;n') \\ \text{end.} \end{array}
```

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 $\forall n: nat, P n$, a następnie powtarzamy niemal to samo, pisząc fix f(n:nat): P n.

```
Fixpoint nat\_ind\_Fixpoint\_term

(P: nat \rightarrow Prop) (H0: P 0)

(HS: \forall n: nat, P n \rightarrow P (S n))

(n: nat): P n:=
```

```
\begin{array}{c} {\rm match}\ n\ {\rm with} \\ {\rm |}\ 0 \Rightarrow H0 \\ {\rm |}\ S\ n' \Rightarrow HS\ n'\ (nat\_ind\_Fixpoint\_term\ P\ H0\ HS\ n') \\ {\rm end.} \end{array}
```

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 \rightarrow Prop) (H0: P 0)
  (HS: \forall n: nat, P n \rightarrow P (S n))
  (n:nat):Pn.
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 \rightarrow P (S n))
                   (n : nat) {struct n} : P n :=
                     nat_ind_Fixpoint_tac P HO HS n) *)
Restart.
  destruct n as [\mid n'].
    apply H\theta.
    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 $nat_ind_Fixpoint_tac\ P\ H0\ HS\ n$, 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:
  \forall (P: nat \rightarrow Prop) (H0: P 0)
    (HS: \forall n: nat, P n \rightarrow 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 \rightarrow P (S n))
                   (n : nat) {struct n} : P n := ... *)
 destruct n as [\mid n' \mid].
    apply H\theta.
    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.

21.3.7 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.

21.3.8 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: \forall \ n \ m: nat, \ n=m \to 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:
\forall \ n: nat, \ n+1=S \ n.

Proof.
rec n.
reflexivity.
cbn. f_equal. rewrite IH. reflexivity.
Qed.
```

21.4 Taktyki dla równości i równoważności

21.4.1 reflexivity, symmetry i transitivity

```
Require Import Arith.

Example reflexivity_0: \forall n: nat, n \leq 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 : \mathsf{Type}\}\ (f \ g : A \to B) : \mathsf{Prop} := \forall \ 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 Relation Classes.

Moduł Relation Classes 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:
```

```
\forall A B : \text{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 #[refine] Instance zamiast z Theorem lub Lemma.

```
Example reflexivity_1:
```

```
eq_-ext (fun _: nat \Rightarrow 42) (fun _: nat \Rightarrow 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:
```

```
\forall \ (A: \mathtt{Type}) \ (x \ y \ z: \ nat), \ x = y \rightarrow y = z \rightarrow 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 po-

staci 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: \forall 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: \forall 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: \forall (A \ B : \mathsf{Type}) \ (f \ g \ h : A \to B), \\ eq\_ext \ f \ g \to eq\_ext \ g \ h \to 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.

21.4.2 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 matcha. 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_-\theta:
  \forall (A : \mathsf{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:
  \forall (A : \mathsf{Type}) (x \ y : A), x = y \rightarrow x = y.
Proof.
  intros. f_equal.
Restart.
  my_f_equal.
Qed.
Example f_{-}equal_{-}2:
  \forall (A \ B \ C \ D \ E : \mathsf{Type}) \ (f \ f' : A \to B \to C \to D \to E)
     (a \ a' : A) (b \ b' : B) (c \ c' : C) (d \ d' : D),
        f=f' 
ightarrow a=a' 
ightarrow b=b' 
ightarrow c=c' 
ightarrow d=d' 
ightarrow
          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 \to B) (g:B \to 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.
```

Ostatnia taktyka, która 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 : \mathtt{Type}\} (l1 \ l2 : list \ A) : \mathtt{Prop} :=
   length l1 = length l2.
```

W naszym przykładzie posłużymy się relacją len_eq, która głosi, że dwie listy są w relacji gdy maja taka sama 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 $A ==> @<math>len_eq$ $A ==> @<math>len_eq$ A).

Należy też zauważyć, że strzałka ==> jest jedynie notacją dla tworu zwanego respectful, co możemy łatwo sprawdzić komendą Locate.

```
\begin{array}{l} \texttt{Example} \ f_-equiv_-\theta : \\ \forall \ (A \ B : \texttt{Type}) \ (f : A \to B) \ (l1 \ l1' \ l2 \ l2' : list \ A), \\ len_-eq \ l1 \ l1' \to len_-eq \ l2 \ l2' \to \\ len_-eq \ (l1 \ ++ \ l2) \ (l1' \ ++ \ l2'). \\ \\ \texttt{Proof.} \\ \texttt{intros.} \ f_-equiv. \\ \texttt{assumption.} \\ \texttt{assumption.} \\ \texttt{Qed.} \end{array}
```

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.

```
Cwiczenie (f_equiv_filter) Require Import List. Import ListNotations.

Definition stupid\_id \{A: Type\} (l: list A): list A:= filter (fun \_\Rightarrow 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 wpowadzania zmian.

```
Example f_-equiv_-1: \forall (A : Type) (l \ l' : list \ A),
```

```
len\_eq\ l\ l' \rightarrow len\_eq\ (stupid\_id\ l)\ (stupid\_id\ l'). Proof.  intros.\ f\_equiv.\ assumption. Qed.
```

21.4.3 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 \rightarrow 4 + 4 = 42.
Proof.
  intro.
  rewrite \leftarrow plus\_n\_Sm.
  rewrite plus_n-Sm.
  rewrite \leftarrow plus\_n\_Sm.
  rewrite \rightarrow plus_{-}n_{-}Sm.
  rewrite \leftarrow !plus_n\_Sm.
  Fail rewrite \leftarrow !plus_n\_Sm.
  rewrite \leftarrow ?plus_n_Sm.
  rewrite 4!plus_n_Sm.
  rewrite \leftarrow 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 $\rightarrow H$ oznacza to samo, co rewrite H
- rewrite $\leftarrow H$ 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 \leftarrow można łączyć z modyfikatorami określającymi ilość przepisań.

```
Lemma rewrite\_ex\_1: \forall n \ m: nat, 42 = 42 \rightarrow S \ (n+m) = n+S \ m. Proof. intros. apply plus\_n\_Sm. Qed. Goal 2+3=6 \rightarrow 5+5=12 \rightarrow (4+4)+((5+5)+(6+6))=42. Proof. intros. rewrite \leftarrow plus\_n\_Sm, \leftarrow plus\_n\_Sm. rewrite \leftarrow plus\_n\_Sm in H. rewrite \leftarrow plus\_n\_Sm in H. rewrite \leftarrow plus\_n\_Sm in *. rewrite !plus\_n\_Sm in *. rewrite !plus\_n\_Sm in *. rewrite + rewrite\_ex\_1. + 2: 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 * \vdash przepisuje H we wszystkich hipotezach różnych od H
- \bullet 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.

21.5 Taktyki dla redukcji i obliczeń (TODO)

21.6 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.

21.6.1 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:
\forall 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:
  \forall 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.
21.6.2
            congruence
Example congruence_{-}\theta:
  \forall P : Prop, true \neq false.
Proof. congruence. Qed.
Example congruence_1 :
  \forall (A : \mathsf{Type}) (f : A \to A) (g : A \to A \to A) (a b : A),
     a = f \ a \rightarrow g \ b \ (f \ a) = f \ (f \ a) \rightarrow g \ a \ b = f \ (g \ b \ a) \rightarrow
        g \ a \ b = a.
Proof.
  congruence.
Qed.
Example congruence_2:
  \forall (A : \mathsf{Type}) (f : A \to A \times A) (a \ c \ d : A),
```

```
f = pair \ a \rightarrow Some \ (f \ c) = Some \ (f \ d) \rightarrow 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.

21.6.3 decide equality

```
\begin{array}{c|c} \text{Inductive } C: \texttt{Type} := \\ & \mid c\theta : C \\ & \mid c1 : C \rightarrow C \\ & \mid c2 : C \rightarrow C \rightarrow C \\ & \mid c3 : C \rightarrow C \rightarrow C \rightarrow C. \end{array}
```

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.

21.6.4 omega

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 \wedge , \vee , \rightarrow oraz \neg , ale nie mogą być kwantyfikowane — omega nie umie wprowadzać zmiennych do kontekstu.

Uwaga: ta taktyka jest przestarzała, a jej opis znajduje się tutaj tylko dlatego, że jak go pisałem, to jeszcze nie była. Nie używaj jej! Zamiast omega używaj lia!

Require Import Arith Omega.

```
Example omega_-\theta: \forall n: nat, n+n=2\times n. Proof. intro. omega. Qed. Example omega_-1: \forall n\ m: nat, 2\times n+1\neq 2\times m. Proof. intros. omega. Qed. Example omega_-2: \forall n\ m: nat, n\times m=m\times 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:

\forall (A: Type) (f: A \rightarrow bool) (l: list A),
length (filter f l) \leq 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': \forall (A: \texttt{Type}) (f: A \rightarrow bool) (l: list A), \\ length (filter f l) \leq 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 linijek 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.

21.6.5 Procedury decyzyjne dla logiki

```
Example tauto_-0: \forall A \ B \ C \ D: Prop, \neg \ A \ \lor \neg \ B \ \lor \neg \ C \ \lor \neg \ D \ \to \neg \ (A \land B \land C \land D). Proof. tauto. Qed. Example tauto_-1: \forall \ (P: nat \to \operatorname{Prop}) \ (n: nat), n=0 \ \lor P \ n \to n \neq 0 \to 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.

```
 \begin{array}{l} \mathtt{Example} \ intuition\_\theta : \\ \forall \ (A : \mathtt{Prop}) \ (P : nat \to \mathtt{Prop}), \\ A \lor (\forall \ n : nat, \neg \ A \to P \ n) \to \forall \ n : nat, \neg \ \neg \ (A \lor P \ n). \\ \mathtt{Proof}. \\ Fail \ \mathtt{tauto.} \ \mathtt{intuition.} \\ \mathtt{Qed.} \end{array}
```

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 \times , co też tłumaczy, dlaczego intuition potrafi więcej niż tauto.

```
Record and 3 (P Q R : Prop) : Prop :=
     left: P;
     mid:Q;
     right: R;
}.
Example firstorder_{-}\theta:
  \forall (B : \mathsf{Prop}) (P : nat \to \mathsf{Prop}),
     and3 (\forall x : nat, P x) B B \rightarrow
        and 3 \ (\forall y : nat, P y) \ (P 0) \ (P 0) \lor B \land P 0.
Proof.
  Fail tauto.
  intuition.
Restart.
  firstorder.
Qed.
Example firstorder_1:
  \forall (A : \mathsf{Type}) (P : A \to \mathsf{Prop}),
     (\exists x : A, \neg P x) \rightarrow \neg \forall 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.

Čwiczenie (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.

21.7 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.

21.7.1 auto i trivial

auto jest najbardziej ogólną taktyką służącą do automatyzacji.

```
Example auto\_ex0: \forall (P: \texttt{Prop}), P \to P. Proof. auto. Qed. Example auto\_ex1: \forall A \ B \ C \ D \ E : \texttt{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: \texttt{Type}) \ (x:A), \ x=x. Proof. auto. Qed. Example auto\_ex3: \forall (A: \texttt{Type}) \ (xy: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_name 1 ... db_n widzi wszystkie lematy dodane do baz db_n 1, ..., db_n 1. Jeżeli to dla ciebie wciąż zbyt wiele pisania, uszy do góry!

Example $auto_ex4$ '': P.

Proof. auto with \times . Qed.

Taktyka auto with \times 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 $even\theta$, evenSS.

Qed.

Kolejną słabością auto jest fakt, że taktyka ta nie potrafi budować wartości typów induktywnych. Na szczęście możemy temu zaradzić używając klauzuli using $c_1 \ldots c_n$, gdzie c_1, \ldots, c_n są konstruktorami naszego typu, lub dodając je jako podpowiedzi za pomocą komendy Hint Resolve $c_1 \ldots c_n : db_name$.

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).

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: \forall \ (A: {\tt Type}) \ (x\ y\ z: A), \ x=y \to y=z \to 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 us ing 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 \Rightarrow match goal with  \mid H:?x=?y,\ H':?y=?z\vdash ?x=?z \Rightarrow \text{apply } (@eq\_trans\_x\ y\ z) \\ \text{end}: extern\_db. \\ \text{Example } auto\_ex7: \\ \forall\ (A:\text{Type})\ (x\ y\ z:A),\ x=y\rightarrow y=z\rightarrow x=z. \\ \text{Proof. auto with } extern\_db.\ \text{Qed.}
```

Jest jednak sposób, żeby uporać się i z tym problemem: jest nim komenda Hint Extern. Jej ogólna postać to Hint Extern n pattern $\Rightarrow tactic: db$. 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 @ $eq_{-}trans_{-}xy_{-}xy_{-}$, 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) \Rightarrow match goal with  \mid H:?x=?y,\ H':?y=?z\vdash \_\Rightarrow \text{apply } (@\mathit{eq\_trans}\ \_x\ y\ z)  end. Example \mathit{auto\_ex7'}:  \forall\ (A: \mathsf{Type})\ (x\ y\ z:A),\ x=y\to y=z\to 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\_ex\theta:
  \forall (P : \mathsf{Prop}), P \to P.
Proof. trivial. Qed.
Example trivial_-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. trivial. Abort.
Example trivial_{-}ex2:
  \forall (A : \mathsf{Type}) (x : A), x = x.
Proof. trivial. Qed.
Example trivial_{-}ex3:
  \forall (A : \mathsf{Type}) (x \ y : A), x = y \rightarrow 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/proof-engine/tactics.html

21.7.2 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 \left(rev \left(l1 \ ++ \ rev \left(rev \ l2 \ ++ \ rev \ l3 \right) \ ++ \ rev \ l4 \right) \ ++ \ rev \left(rev \ l5 \right) 
ight) = \left(rev \left(rev \ l5 \ ++ \ l1 
ight) 
ight) ++ \left(l3 \ ++ \ rev \left(rev \ l2 
ight) 
ight) ++ \ rev \ l4 
ight)
```

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 \leftarrow 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 [<-] $ident_{-}\theta$... $ident_{-}n$: $db_{-}name$ dodaje podpowiedzi $ident_{-}\theta$, ..., $ident_{-}n$ 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 $db_-\theta$... db_-n przepisuje lematy ze wszystkich baz podpowiedzi $db_-\theta$, ..., 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: coq.inria.fr/refman/proofengine/tactics.html

```
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 $db_-\theta$, ..., db_-n potrafi odwinąć wszystkie definicje z baz $db_-\theta$, ..., 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*: coq.inria.fr/refman/proofengine/tactics.html

Ćwiczenie (bazy podpowiedzi) Przeczytaj w manualu dokładny opis działania systemu baz podpowiedzi oraz komend pozwalających go kontrolować: coq.inria.fr/refman/proofengine/tactics.html

21.8 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/addendum/ring.html

21.9 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.

21.10 Taktyki do radzenia sobie z typami zależnymi (TODO)

Opisać taktyki dependent induction, dependent inversion, dependent destruction, dependent rewrite etc.

21.11 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.

Cwiczenie (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. (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 (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 (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 ogarnać, co one robia: https://coq.inria.fr/refman/addendum/micromega.html

Ćwiczenie (wyższa magia) Spróbuj ogarnąć, co robią taktyki nsatz, psatz i fourier.

21.12 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: https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html oraz https://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, cbn 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.

21.13 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/
- \bullet cs.cornell.edu/courses/cs3110/2017fa/a5/coq-tactics-cheatsheet.html
- typesofnote.com/posts/cog-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 22

13: Reflekcja w dużej skali, czyli jak odbijać z rozmachem

Chwilowo nic tu nie ma.

22.1 Ltac: manipulowanie termami

TODO:

- match expr
- \bullet lazymatch expr
- multimatch expr
- type of term
- eval redexpr
- constr/uconstr/ltac
- type_term?

22.2 Taktyki dla unifikacji

TODO:

- \bullet has_evar, is_evar, is_var
- unify
- \bullet constr_eq

- instantiate
- quote

22.3 Programowanie funkcyjne w Ltacu

Wstawić tutaj przykłady podobne do tych, które opisuje Chlipala. Być może jakiś większy development, tzn. zaprogramować listy w dwóch wersjach (zwykłe i zrobione produktem i unitem).

22.4 Big scale reflection

Przykłady:

- logika boolowska, czyli legitne btauto
- permutacje
- formuly logiczne
- upraszczanie w monoidzie

(* end hide *)

Rozdział 23

J: Kim jesteśmy i dokąd zmierzamy pusty

Chilowo nic tu nie ma (ale będzie).

Rozdział 24

W1: Konstruktywny rachunek zdań [schowany na końcu dla niepoznaki]

24.1 Zdania i spójniki logiczne

24.1.1 Implikacja

Rozumowanie w przód

Rozumowanie w tył

```
(* rozumowanie od tyłu jest lepsze, logika jest bezmyślna *)
```

- 24.1.2 Koniunkcja
- 24.1.3 Dysjunkcja
- 24.1.4 Prawda i fałsz
- 24.1.5 Równoważność
- 24.1.6 Negacja

24.1.7 Silna negacja

Poznaliśmy uprzednio pewien spójnik, zapisywany wdzięcznym wygibaskiem ¬, a zwany górnolotnie negacją. Powinniśmy się jednak zastanowić: czy spójnik ten jest dla nas zadowalający? Czy pozwala on nam wyrażać nasze przemyślenia w najlepszy możliwy sposób?

Jeżeli twoja odpowiedź brzmi "tak", to uroczyście oświadczam, że wcale nie masz racji. Wyobraźmy sobie następującą sytuację: jesteśmy psycho patusem, próbującym pod pozorem podrywu poobrażać przeróżne panienki.

Podbijamy do pierwszej z brzegu, która akurat jest normalną dziewczyną, i mówimy: "Hej mała, jesteś gruba i mądra". Nasza oburzona rozmówczyni, jako że jest szczupła, odpowiada nam: "Wcale nie jestem gruba. Spadaj frajerze".

Teraz na cel bierzemy kolejną, która siedzi sobie samotnie przy stoliku w Starbuniu, popija kawkę z papierowego kubka i z uśmiechem na ustach próbuje udowodnić w Coqu jakieś bardzo skomplikowane twierdzenie. Podbijamy do niej i mówimy: "Hej mała, jesteś gruba i mądra". Jako, że ona też jest szczupła, oburza się i odpowiada nam tak:

"Czekaj, czekaj, Romeo. Załóżmy, że twój tani podryw jest zgodny z prawdą. Gdybym była gruba i mądra, to byłabym w szczególności mądra, bo P i Q implikuje Q. Ale gdybym była mądra, to wiedziałabym, żeby tyle nie żreć, a skoro tak, to bym nie żarła, więc nie byłabym gruba, ale na mocy założenia jestem, więc twój podryw jest sprzeczny. Jeżeli nie umiesz logiki, nie ide z toba do łóżka."

Widzisz różnicę w tych dwóch odpowiedziach? Pierwsza z nich wydaje nam się bardzo naturalna, bo przypomina zaprzeczenia, jakich zwykli ludzie używają w codziennych rozmowach. Druga wydaje się zawoalowana i bardziej przypomina dowód w Coqu niż codzienne rozmowy. Między oboma odpowiedziami jest łatwo zauważalna przepaść.

Żeby zrozumieć tę przepaść, wprowadzimy pojęcia silnej i słabej negacji. W powyższym przykładzie silną negacją posłużyła się pierwsza dziewczyna - silną negacją zdania "jesteś gruba i mądra" jest tutaj zdanie "wcale nie jestem gruba". Oczywiście jest też druga możliwość silnego zaprzeczenia temu zdaniu - "nie jestem mądra" - ale z jakichś powodów to zaprzeczenie nie padło. Ciekawe dlaczego? Druga dziewczyna natomiast posłużyła się słabą negacją, odpowiadając "gdybym była gruba i mądra, to... (tutaj długaśne rozumowanie)... więc sprzeczność".

Słaba negacja to ta, którą już znamy, czyli Coqowe *not*. Ma ona charakter hipotetyczny, gdyż jest po prostu implikacją, której konkluzją jest *False*. W rozumowaniach słownych sprowadza się ona do schematu "gdyby tak było, to wtedy...".

Silna negacja to najbardziej bezpośredni sposób zaprzeczenia danemu zdaniu. W Coqu nie ma żadnego spójnika, który ją wyraża, bo ma ona charakter dość ad hoc - dla każdego zdania musimy sami sobie wymyślić, jak brzmi zdanie, który najsilniej mu przeczy. W rozumowaniach słownych silna negacja sprowadza się zazwyczaj do schematu "nie jest tak".

Spróbujmy przetłumaczyć powyższe rozważania na język logiki. Niech P oznacza "gruba", zaś Q - "mądra". Silną negacją zdania $P \wedge Q$ jest zdanie $\neg P \vee \neg Q$ ("nie gruba lub nie mądra"), zaś jego słabą negacją jest $\neg (P \wedge Q)$, czyli $P \wedge Q \rightarrow False$ ("jeżeli gruba i mądra, to sprzeczność").

Zauważmy, że o ile słaba negacja jest uniwersalna, tj. słabą negacją $P \wedge Q$ zawsze jest $\neg (P \wedge Q)$, to silna negacja jest ad hoc w tym sensie, że gdyby P było postaci $P1 \wedge P2$, to wtedy silną negacją $P \wedge Q$ nie jest już $\neg P \vee \neg Q$, a $\neg P1 \vee \neg P2 \vee \neg Q$ - żeby uzyskać silną negację, musimy zanegować P silnie, a nie słabo.

Dlaczego silna negacja jest silna, a słaba jest słaba, tzn. dlaczego nazwaliśmy je tak a nie inaczej? Wyjaśnia to poniższe twierdzenie oraz następująca po nim beznadziejna próba udowodnienia analogicznego twierdzenia z implikacją idącą w drugą stronę.

Lemma $stronq_to_weak_and$:

```
\begin{array}{l} \forall \ P \ Q : {\tt Prop}, \neg \ P \lor \neg \ Q \to \neg \ (P \land \ Q). \\ {\tt Proof}. \\ & {\tt intros} \ P \ Q \ Hor \ Hand. \\ & {\tt destruct} \ Hand \ {\tt as} \ [p \ q]. \\ & {\tt destruct} \ Hor \ {\tt as} \ [notp \mid notq]. \\ & {\tt apply} \ notp. \ {\tt assumption}. \\ & {\tt apply} \ notq. \ {\tt assumption}. \\ & {\tt Qed}. \end{array}
```

Jak widać, silna negacja koniunkcji pociąga za sobą jej słabą negację. Powód tego jest prosty: jeżeli jeden z koniunktów nie zachodzi, ale założymy, że oba zachodzą, to w szczególności każdy z nich zachodzi osobno i mamy sprzeczność.

A czy implikacja w drugą stronę zachodzi?

```
Lemma weak\_to\_strong\_and: \forall~P~Q: \texttt{Prop}, \lnot~(P~\land~Q) \rightarrow \lnot~P~\lor \lnot~Q. Proof. intros P~Q~notpq. left. intro p. apply notpq. split. assumption. Abort.
```

Jak widać, nie udało nam się udowodnić odwrotnej implikacji i to wcale nie dlatego, że jesteśmy mało zdolni - po prostu konstruktywnie nie da się tego zrobić.

Powód tego jest prosty: jeżeli wiemy, że P i Q razem prowadzą do sprzeczności, to wiemy zdecydowanie za mało. Mogą być dwa powody:

- P i Q moga bez problemu zachodzić osobno, ale być sprzeczne razem
- nawet jeżeli któryś z koniunktów prowadzi do sprzeczności, to nie wiemy, który

Žeby zrozumieć pierwszą możliwość, niech P oznacza "siedzę", a Q - "stoję". Rozważmy zdanie $P \wedge Q$, czyli "siedzę i stoję". Żeby nie było za łatwo załóżmy też, że znajdujesz się podrugiej stronie kosmosu i mnie nie widzisz.

Oczywiście nie mogę jednocześnie siedzieć i stać, gdyż czynności te się wykluczają, więc możesz skonkludować, że $\neg (P \land Q)$. Czy możesz jednak wywnioskować stąd, że $\neg P \lor \neg Q$, czyli że "nie siedzę lub nie stoję"? Konstruktywnie nie, bo będąc po drugiej stronie kosmosu nie wiesz, której z tych dwóch czynności nie wykonuję.

Z drugim przypadkiem jest tak samo, jak z końcówką powyższego przykładu: nawet jeżeli zdania P i Q się wzajemnie nie wykluczają i niesłuszność $P \wedge Q$ wynika z tego, że któryś z koniunktów nie zachodzi, to możemy po prostu nie wiedzieć, o który z nich chodzi.

Żeby jeszcze wzmocnić nasze zrozumienie, spróbujmy w zaskakujący sposób rozwinąć definicję (słabej) negacji dla koniunkcji:

```
Lemma not\_and\_surprising: \forall \ P \ Q: \texttt{Prop}, \ \neg \ (P \land Q) \leftrightarrow (P \rightarrow \neg \ Q). Proof.
```

```
split. intros npq\ p\ q. apply npq. split. assumption. assumption. intros pnq\ pq. destruct pq as [p\ q]. apply pnq. assumption. assumption. Qed. I jeszcze raz... Lemma not\_and\_surprising: \forall\ P\ Q: {\tt Prop}, \ \neg\ (P\ \land\ Q) \leftrightarrow (Q\ \rightarrow\ \neg\ P).
```

Jak (mam nadzieję) widać, słaba negacja koniunkcji nie jest niczym innym niż stwierdzeniem, że oba koniunkty nie mogą zachodzić razem. Jest to więc coś znacznie słabszego, niż stwierdzenie, że któryś z koniunktów nie zachodzi z osobna.

Lemma mid_neg_conv :

$$\forall \ P \ Q : \mathtt{Prop}, \ \neg \ (P \ \land \ Q) \rightarrow ((P \rightarrow \neg \ Q) \ \land \ (Q \rightarrow \neg \ P)).$$

Proof.

firstorder.

Qed.

Jak napisano w Piśmie, nie samą koniunkcją żyje człowiek. Podumajmy więc, jak wygląda silna negacja dysjunkcji. Jeżeli chcemy dosadnie powiedzieć, że $P \vee Q$ nie zachodzi, to najprościej powiedzieć: $\neg P \wedge \neg Q$. Słaba negacja dysjunkcji ma zaś rzecz jasna postać $\neg (P \vee Q)$.

```
Lemma strong\_to\_weak\_or: \forall \ P \ Q : \texttt{Prop}, \ \neg \ P \ \land \ \neg \ Q \ \rightarrow \ \neg \ (P \lor Q). Proof. do 2 destruct 1; contradiction. Qed.
```

Co jednak dość ciekawe, silna negacja nie zawsze jest silniejsza od słabej (ale z pewnością nie może być od niej słabsza - gdyby mogła, to nazywałaby się inaczej). W przypadku dysjunkcji obie negacje są równoważne, co obrazuje poniższe twierdzenie, które głosi, że słaba negacja implikuje silną (a to razem z powyższym daje równoważność):

 $\begin{array}{l} \operatorname{Lemma} \ weak_to_strong_or: \\ \forall \ P \ Q: \operatorname{Prop}, \ \neg \ (P \lor Q) \to \neg \ P \land \neg \ Q. \\ \operatorname{Proof.} \\ \operatorname{split}; \ \operatorname{intro}; \ \operatorname{apply} \ H; \ [\operatorname{left} \mid \operatorname{right}]; \ \operatorname{assumption}. \\ \operatorname{Qed.} \end{array}$

Wynika to z faktu, że $\neg P \land \neg Q$ to tak naprawdę para implikacji $P \to False$ i $Q \to False$, zaś $\neg (P \lor Q)$ to... gdy pomyślimy nad tym odpowiednio mocno... ta sama para

implikacji. Jest tak dlatego, że jeżeli $P \vee Q$ implikuje R, to umieć wyprodukować R musimy zarówno z samego P, jak i z samego Q.

```
 \begin{array}{l} \operatorname{Lemma} \ deMorgan\_dbl\_neg : \\ (\forall \ P \ Q : \operatorname{Prop}, \neg \ (P \land Q) \to \neg \ P \lor \neg \ Q) \leftrightarrow \\ (\forall \ P : \operatorname{Prop}, \neg \neg \ P \to P). \\ \\ \operatorname{Proof.} \\ \operatorname{split.} \\ \operatorname{intros} \ deMorgan \ P \ H. \\ \\ \operatorname{Abort.} \end{array}
```

24.1.8 Czy Bozia dała inne spójniki logiczne?

24.2 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".

```
Lemma copy: \forall P : \texttt{Prop}, P \rightarrow 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 $andn: nat \to \mathsf{Prop} \to \mathsf{Prop},$ 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!

24.3 Zadania

- na koniec dać tylko te zadania, które łączą wiele spójników
 - dodać zadanie dotyczące czytania twierdzeń i dowodów
 - dodać zadania dotyczące czytania formuł (precedencja etc.)

24.4 Ściąga

Rozdział 25

W2: Konstruktywny rachunek kwantyfikatorów [schowany na końcu dla niepoznaki]

25.1 Typy i ich elementy

Tu zestawić ze sobą P: Prop, A: Type, p: P, x: A

- 25.2 Predykaty i relacje
- 25.3 Kwantyfikatory
- 25.3.1 Kwantyfikator uniwersalny
- 25.3.2 Kwantyfikator egzystencjalny
- 25.4 Zmienne związane
- 25.5 Predykatywizm

25.6 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 kwantyfikato-

rów logicznych, które mieliśmy okazję poznać w tym rozdziale. Żeby to sobie uświadomić, zapoznamy się z pewnego rodzaju "paradoksem".

Theorem $barbers_paradox$:

```
\forall \ (man: \mathtt{Type}) \ (barber: man) \ (shaves: man \to man \to \mathtt{Prop}), \ (\forall \ x: man, \ shaves \ barber \ x \leftrightarrow \neg \ shaves \ x \ x) \to False.
```

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

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 $\forall x : man$, shaves barber $x \leftrightarrow \neg$ 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 ($\forall\ x:man$), którzy sami siebie nie golą ($\neg\ shaves\ x\ x$), to podstawiając $barber\ za\ x$ otrzymujemy sprzeczność: $shaves\ barber\ barber\ z$ achodzi 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, przeformułowywujac powyższe twierdzenie z użyciem innych nazw:

Lemma $barbers_paradox'$:

```
\forall (A : \mathtt{Type})(x : A) (P : A \rightarrow A \rightarrow \mathtt{Prop}), \\ (\forall y : A, P \ x \ y \leftrightarrow \neg P \ y \ y) \rightarrow 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.

25.7 Zadania

- modelowanie różnych sytuacji za pomocą zdań i predykatów
 - rozwiązywanie zagadek logicznych
 - więcej zadań z exists

25.8 Ściąga

Rozdział 26

W3: Logika klasyczna [schowana na końcu dla niepoznaki]

26.1 Prawa logiki klasycznej

```
Definition LEM: Prop :=
  \forall P : \mathsf{Prop}, P \vee \neg P.
Definition MI: Prop :=
  \forall P \ Q : \mathsf{Prop}, (P \to Q) \to \neg P \lor Q.
Definition ME: Prop :=
  \forall \ P \ Q : \mathtt{Prop}, \ (P \leftrightarrow Q) \rightarrow (P \land Q) \lor (\tilde{\ } P \land \neg \ Q).
Definition DNE: Prop :=
  \forall P : \mathsf{Prop}, \neg \neg P \to P.
Definition CM: Prop :=
  \forall P : \mathsf{Prop}, (\ P \to P) \to P.
Definition Peirce : Prop :=
  \forall P \ Q : \mathsf{Prop}, ((P \to Q) \to P) \to P.
Definition Contra : Prop :=
  \forall P \ Q : \mathsf{Prop}, (\ Q \to \neg P) \to (P \to Q).
Ltac u :=
  unfold LEM, DNE, CM, MI, ME, Peirce, Contra.
    Tu o prawie zachowania informacji.
    A o paradoksach implikacji materialnej?
```

26.2 Logika klasyczna jako logika Boga

Lemma $LEM_-hard: \forall P: \texttt{Prop}, P \vee \neg P.$

```
Proof.
   intro P. left.
Restart.
   intro P. right. intro p.
Abort.
Lemma LEM\_irrefutable:
   \forall P : \mathtt{Prop}, \neg \neg (P \lor \neg P).
Proof.
   intros P H.
   apply H. right. intro p.
   apply H. left. assumption.
Qed.
Lemma LEM_-DNE :
   (\forall P : \mathtt{Prop}, P \lor \neg P) \rightarrow
       (\forall P : \mathsf{Prop}, \neg \neg P \rightarrow P).
Lemma LEM_-MI :
   (\forall P : \mathsf{Prop}, P \lor \neg P) \to
       (\forall P \ Q : \mathtt{Prop}, (P \rightarrow Q) \rightarrow \neg P \lor Q).
\mathtt{Lemma}\ LEM\_ME\ :
   (\forall P : \mathsf{Prop}, P \lor \neg P) \to
      (\forall P \ Q : \mathtt{Prop}, (P \leftrightarrow Q) \rightarrow (P \land Q) \lor (\tilde{\ } P \land \neg Q)).
Lemma LEM_-Peirce:
   (\forall P : \mathsf{Prop}, P \lor \neg P) \to
       (\forall P \ Q : \mathtt{Prop}, ((P \rightarrow Q) \rightarrow P) \rightarrow P).
Lemma LEM_-CM:
   (\forall P : \mathsf{Prop}, P \lor \neg P) \to
      (\forall P : \mathsf{Prop}, (\ P \to P) \to P).
Lemma LEM_-Contra:
   (\forall P : \mathsf{Prop}, P \lor \neg P) \rightarrow
      (\forall P \ Q : Prop, (\ Q \rightarrow \neg P) \rightarrow (P \rightarrow Q)).
```

26.3 Logika klasyczna jako logika materialnej implikacji i równoważności

```
Lemma material\_implication\_conv: \forall \ P \ Q: \texttt{Prop}, \ \neg \ P \ \lor \ Q \ \rightarrow \ (P \ \rightarrow \ Q). Proof. intros P \ Q \ H. destruct H as [np \ | \ q]. intro p. contradiction.
```

```
intro p. assumption.
Qed.
Lemma material\_implication':
  \forall P \ Q : Prop, (P \rightarrow Q) \rightarrow \neg P \lor Q.
Proof.
   intros P Q H. left. intro p. specialize (H p).
Restart.
   intros P Q H. right. apply H.
Abort.
Lemma material\_implication\_irrefutable:
  \forall P \ Q : \mathtt{Prop}, \neg \neg ((P \rightarrow Q) \rightarrow \neg P \lor Q).
Proof.
   intros P Q H.
   apply H. intro pq.
   left. intro.
   apply H. intros _.
  \verb"right. apply" $pq$.
   assumption.
Qed.
Lemma MI\_LEM :
   MI \rightarrow LEM.
Lemma MI_-DNE :
   MI \rightarrow DNE.
Lemma MI\_CM:
   MI \rightarrow CM.
Lemma MI_-ME :
   MI \rightarrow ME.
Lemma MI\_Peirce:
   MI \rightarrow Peirce.
Lemma MI\_Contra:
   MI \rightarrow Contra.
Lemma material\_equivalence\_conv:
  \forall \ P \ Q : \mathtt{Prop}, \ (P \ \land \ Q) \ \lor \ (\tilde{\ } P \ \land \ \neg \ Q) \ \rightarrow \ (P \ \leftrightarrow \ Q).
Proof.
   intros P Q H. destruct H as [pq \mid npnq].
     destruct pq as |p|q|. split.
        intro p'. assumption.
        intro q'. assumption.
     destruct npnq as [np nq]. split.
        intro p. contradiction.
```

```
intro q. contradiction.
Qed.
Lemma material\_equivalence:
  \forall \ P \ Q : \mathtt{Prop}, \ (P \ \leftrightarrow \ Q) \rightarrow (P \ \land \ Q) \lor (\ \tilde{} \ P \ \land \ \neg \ Q).
Proof.
  intros P Q [pq qp]. left. split.
     apply qp. apply pq.
Restart.
  intros P Q [pq qp]. right. split.
     intro p.
Abort.
Lemma material\_equivalence\_irrefutable:
  \forall \ P \ Q : \mathtt{Prop}, \neg \neg ((P \leftrightarrow Q) \rightarrow (P \land Q) \lor (\tilde{\ }P \land \neg \ Q)).
Proof.
  intros P \ Q \ nme.
  apply nme. intros [pq \ qp].
  right. split.
     intro p. apply nme. intros \_. left. split.
        assumption.
        apply pq. assumption.
     intro q. apply nme. intros \_. left. split.
        apply qp. assumption.
        assumption.
Qed.
Lemma ME\_LEM :
  ME \rightarrow LEM.
Lemma ME\_DNE :
  ME \rightarrow DNE.
Lemma ME_-MI :
  ME \rightarrow MI.
Lemma ME\_CM:
  ME \rightarrow CM.
Lemma ME\_Peirce:
  ME \rightarrow Peirce.
Lemma ME_{-}Contra:
```

 $ME \rightarrow Contra.$

26.4 Logika klasyczna jako logika diabła

Dawno dawno temu w odległej galaktyce, a konkretniej w ZSRR, był sobie pewien rusek. Pewnego razu do ruska przyszedł diaboł (a to, jak wiadomo, coś dużo gorszego niż diabeł) i zaoferował mu taki dil: "dam ci miliard dolarów albo jeżeli dasz mi miliard dolarów, to spełnie dowolne twoje życzenie".

Rusek trochę skonsternowany, nie bardzo widzi mu się podpisywanie cyrografu krwią. "Nie nie, żadnych cyrografów, ani innych takich kruczków prawnych", zapewnia go diaboł. Rusek myśli sobie tak: "pewnie hajsu nie dostanę, ale przecież nic nie tracę", a mówi: "No dobra, biere".

"Świetnie!" - mówi diaboł - "Jeżeli dasz mi miliard dolarów, to spełnie dowolne twoje życzenie". Cóż, rusek był zawiedziony, ale nie zaskoczony - przecież dokładnie tego się spodziewał. Niedługo później diaboł zniknął, a rusek wrócił do pracy w kołchozie.

Jako, że był przodownikiem pracy i to na dodatek bardzo oszczędnym, bo nie miał dzieci ani baby, szybko udało mu się odłożyć miliard dolarów i jeszcze kilka rubli na walizkę. Wtedy znów pojawił się diaboł.

"O, cóż za spotkanie. Trzym hajs i spełnij moje życzenie, tak jak się umawialiśmy" - powiedział rusek i podał diabołowi walizkę. "Wisz co" - odpowiedział mu diaboł - "zmieniłem zdanie. Życzenia nie spełnię, ale za to dam ci miliard dolarów. Łapaj" - i diaboł oddał ruskowi walizkę.

Jaki morał płynie z tej bajki? Diaboł to bydle złe i przeokrutne, gdyż w logice, którą posługuje się przy robieniu dili (względnie podpisywaniu cyrografów) obowiązuje prawo eliminacji podwójnej negacji.

Prawo to prezentuje się podobnie jak prawo wyłączonego środka:

```
Lemma DNE\_hard:
  \forall P : \mathsf{Prop}, \neg \neg P \to P.
Proof.
  intros P nnp.
Abort.
   Po pierwsze, nie da się go konstruktywnie udowodnić.
Lemma DNE\_irrefutable:
  \forall P : \mathsf{Prop}, \neg \neg (\tilde{\ } \neg P \rightarrow P).
Proof.
  intros P H.
  apply H.
  intro nnp.
  cut False.
     contradiction.
     apply nnp. intro p. apply H. intros \_. assumption.
Qed.
   Po drugie, jest ono niezaprzeczalne.
```

Po trzecie, jest równoważne prawu wyłączonego środka.

```
\label{eq:lemma_DNE_LEM} \begin{tabular}{ll} Lemma $DNE\_MI:$\\ $DNE \to MI.$\\ Lemma $DNE\_ME:$\\ $DNE \to ME.$\\ Lemma $DNE\_CM:$\\ $DNE \to CM.$\\ Lemma $DNE\_Peirce:$\\ $DNE \to Peirce.$\\ Lemma $DNE\_Contra:$\\ $DNE \to Contra.$\\ \end{tabular}
```

26.5 Logika klasyczna jako logika kontrapozycji

```
Lemma contraposition':
  \forall \ P \ Q : \stackrel{-}{\operatorname{Prop}}, \left( {^{\sim}} \ Q \to \neg \ P \right) \to (P \to Q).
Proof.
   intros P Q H p.
Abort.
Lemma contraposition\_irrefutable:
  \forall P \ Q : \mathtt{Prop}, \neg \neg ((\ Q \rightarrow \neg P) \rightarrow (P \rightarrow Q)).
Proof.
   intros P Q H. apply H.
   intros nqnp p. cut False.
      contradiction.
      apply nqnp.
        intro. apply H. intros \_ .. assumption.
        assumption.
Qed.
Lemma Contra\_LEM:
   Contra \rightarrow LEM.
Lemma Contra\_MI:
   Contra \rightarrow MI.
Lemma Contra\_ME:
   Contra \rightarrow ME.
Lemma Contra\_DNE:
   Contra \rightarrow DNE.
Lemma Contra\_CM:
```

```
Contra 	o CM.
Lemma Contra\_Peirce:
Contra 	o Peirce.
```

26.6 Logika klasyczna jako logika Peirce'a

26.6.1 Logika cudownych konsekwencji

```
Lemma consequentia\_mirabilis:
  \forall P : \mathsf{Prop}, (\ P \to P) \to P.
Proof.
  intros P H. apply H. intro p.
Abort.
Lemma consequentia\_mirabilis\_irrefutable:
  \forall \ P : \mathtt{Prop}, \, \neg \, \neg \, ((\ ^{\sim} P \rightarrow P) \rightarrow P).
Proof.
  intros P H. apply H.
  intro npp. apply npp.
  intro p. apply H.
  intros _. assumption.
Qed.
Lemma CM\_LEM :
  CM \rightarrow LEM.
Lemma CM_-MI:
  CM \rightarrow MI.
Lemma CM_-ME :
  CM \rightarrow ME.
Lemma CM_-DNE :
  CM \rightarrow DNE.
Lemma CM_{-}Peirce:
  CM \rightarrow Peirce.
Lemma CM_-Contra:
  CM \rightarrow Contra.
```

26.6.2 Logika Peirce'a

```
Lemma Peirce\_hard: \forall \ P \ Q: \texttt{Prop}, \ ((P \to Q) \to P) \to P. Proof.
```

```
intros P \ Q \ H.
  apply H. intro p.
Abort.
Lemma Peirce\_irrefutable:
  \forall P \ Q : \mathsf{Prop}, \neg \neg (((P \to Q) \to P) \to P).
Proof.
  intros P Q H.
  apply H. intro pqp.
  apply pqp. intro p.
  cut False.
     contradiction.
     apply H. intros _. assumption.
Qed.
Lemma Peirce\_LEM:
  Peirce \rightarrow LEM.
Lemma Peirce\_MI:
  Peirce \rightarrow MI.
Lemma Peirce\_ME:
  Peirce \rightarrow ME.
Lemma Peirce\_DNE :
  Peirce \rightarrow DNE.
Lemma Peirce\_CM:
  Peirce \rightarrow CM.
Lemma Peirce\_Contra:
  Peirce \rightarrow Contra.
```

26.7 Paradoks pijoka

```
Theorem drinkers\_paradox:
\forall (man: Type) (drinks: man \rightarrow Prop) (random\_guy: man),
\exists drinker: man, drinks drinker \rightarrow
\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 $P \to Q \leftrightarrow \neg P \lor Q$, 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:
```

26.8 Paradoks Curry'ego

"Jeżeli niniejsze zdanie jest prawdziwe, to Niemcy graniczą z Chinami."

Inne paradoksy autoreferencji: paradoks Richarda, słowa heterologiczne

26.9 Zadania

wyrzucić zadania mącące (mieszające typy i zdania)

26.10 Ściąga

Rozdział 27

W4: Inne logiki [schowane na końcu dla niepoznaki]

- 27.1 Porównanie logiki konstruktywnej i klasycznej
- 27.2 Pluralizm logiczny
- 27.3 Inne logiki?

27.4 Logika de Morgana

jako coś pomiędzy logiką konstruktywną i klasyczną

27.5 Inne logiki - podsumowanie

krótkie, acz realistyczne (logiki parakonsystentne to guwno)

27.6 Kombinatory taktyk

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

```
 \begin{array}{l} \operatorname{Lemma} \ iff\_intro': \\ \forall \ P \ Q: \operatorname{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \operatorname{Proof}. \\ \operatorname{intros. \ split}. \\ \operatorname{intro. \ apply} \ H. \ \operatorname{assumption}. \\ \operatorname{intro. \ apply} \ H\theta. \ \operatorname{assumption}. \\ \end{array}
```

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.

27.6.1 ; (średnik)

```
 \begin{array}{l} \operatorname{Lemma} \ iff\_intro'': \\ \forall \ P \ Q: \operatorname{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \\ \operatorname{Proof}. \\ \operatorname{split}; \operatorname{intros}; [\operatorname{apply} H \mid \operatorname{apply} H\theta]; \operatorname{assumption}. \\ \\ \operatorname{Qed}. \end{array}
```

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 splitem, 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; $[t1 \mid t2 \mid ... \mid tn]$ 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; $[t2 \mid t2 \mid ... \mid t2]$.

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.

```
 \begin{array}{l} \operatorname{Lemma} \ or\_comm\_ex: \\ \forall \ P \ Q: \operatorname{Prop}, \ P \lor \ Q \to Q \lor P. \\ \\ \operatorname{Lemma} \ diamond: \\ \forall \ P \ Q \ R \ S: \operatorname{Prop}, \\ (P \to Q) \lor (P \to R) \to (Q \to S) \to (R \to S) \to P \to S. \end{array}
```

$27.6.2 \quad || \text{ (alternatywa)}$

```
\label{eq:lemma_lemma} \begin{array}{l} \operatorname{Lemma} \ iff\_intro"" : \\ \forall \ P \ Q : \operatorname{Prop}, \\ (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \\ \\ \operatorname{Proof}. \\ \operatorname{split}; \operatorname{intros}; \operatorname{apply} \ H\theta \ || \operatorname{apply} \ H; \operatorname{assumption}. \\ \\ \operatorname{Qed}. \end{array}
```

Innym przydatnym kombinatorem jest ||, który będziemy nazywać alternatywą. Zeby 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 $t1 \mid\mid t2$ używa na obecnym celu t1. Jeżeli t1 dokona postępu, to $t1 \mid\mid t2$ 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 $t1 \mid\mid t2$ będzie miało taki efekt jak t2 i skończy się sukcesem. Jeżeli t2 nie dokona postępu, to $t1 \mid\mid t2$ zawiedzie i cel się nie zmieni.

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

27.6.3 idtac, do oraz repeat

```
\label{eq:lemma_dec} \begin{array}{l} \text{Lemma } idtac\_do\_example: \\ \forall \ P \ Q \ R \ S: \texttt{Prop}, \\ P \to S \lor R \lor Q \lor P. \\ \\ \text{Proof.} \\ \text{idtac. intros. do } 3 \ \texttt{right. assumption.} \\ \\ \text{Qed.} \end{array}
```

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.

```
\label{eq:lemma_repeat_example} \begin{split} &\forall~P~A~B~C~D~E~F: \texttt{Prop}, \\ &P \to A \lor B \lor C \lor D \lor E \lor F \lor P. \end{split} \label{eq:lemma_repeat_example} \\ &\text{Proof.} \\ &\text{intros. repeat right. assumption.} \\ \\ &\text{Qed.} \end{split}
```

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.

27.6.4 try i fail

```
\label{eq:lemma_lemma} \begin{split} \text{Lemma } & \textit{iff\_intro4} : \\ & \forall \ P \ Q : \texttt{Prop}, \\ & (P \to Q) \to (Q \to P) \to (P \leftrightarrow Q). \end{split} \label{eq:lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma_lemma
```

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; $t1 \mid ... \mid tn$, 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/coq-tacindex.html

27.7 Zadania

rozwiąż wszystkie zadania jeszcze raz, ale tym razem bez używania Module/Section/Hypothesis oraz z jak najkrótszymi dowodami

27.8 Jakieś podsumowanie

- taktyka firstorder
 - -zrobić test diagnostyczny tak/nie
 - fiszki do nauki nazw praw

Rozdział 28

Z: Złożoność obliczeniowa

Prerekwizyty:

- rekursja strukturalna
- dowodzenie przez indukcję
- listy
- teoria relacji

```
Require Import D5.
Require Import Lia.
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.

28.1 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.

28.2 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.

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 jest dopasowanie do wzorca. 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.

28.3 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.

28.4 Duże O

28.4.1 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 \rightarrow nat) : Prop := \exists \ c \ n : nat, \ \forall \ n' : nat, \ n \leq n' \rightarrow f \ n' \leq c \times 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:

- ullet 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 $g(n) = 42n^4$, to nie interesują nas stałe 4 i 42. Najbardziej znaczącym komponentem f jest n^2 , zaś g n^4 2.

Poszukaj w Internecie wizualizacji tej idei — ja niestety mam bardzo ograniczone możliwości osadzania multimediów w niniejszej książce (TODO: postaram się coś na to poradzić).

28.4.2 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.

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).

Cwiczenie Udowodnij, że *O* jest relacją zwrotną i przechodnią. Pokaż też, że nie jest ani symetryczna, ani słabo antysymetryczna.

```
Lemma O\_refl:

\forall f : nat \rightarrow nat, O f f.

Lemma O\_trans:
```

```
\begin{array}{c} \forall \ f \ g \ h : \ nat \rightarrow nat, \\ O \ f \ g \rightarrow O \ g \ h \rightarrow O \ f \ h. \end{array} Lemma O\_asym: \exists \ f \ g : \ nat \rightarrow nat, \ O \ f \ g \land \neg O \ g \ f. Lemma O\_not\_weak\_antisym: \exists \ f \ g : \ nat \rightarrow nat, \ O \ f \ g \land O \ g \ f \land f \neq g.
```

28.4.3 Duże Omega

```
Definition Omega\ (f\ g: nat \rightarrow nat): \texttt{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 \rightarrow nat, Omega f f.

Lemma Omega\_trans:
\forall f g h: nat \rightarrow nat,
Omega f g \rightarrow Omega g h \rightarrow Omega f h.

Lemma Omega\_not\_weak\_antisym:
\exists f g: nat \rightarrow nat, Of g \land Og f \land f \neq g.
```

28.5 Duże Theta

```
Definition Theta\ (f\ g:\ nat\ 
ightarrow\ nat): {\tt 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:

- \bullet O fg znaczy tyle, co "najbardziej znaczący komponent fjest mniejszy lub równy najbardziej znaczącemu komponentowig"
- \bullet $Theta \ f \ g$ znaczy "najbardziej znaczące komponenty fi gsą sobie równe".

```
Ćwiczenie Theorem Theta\_refl: \forall f: nat \rightarrow nat, Theta f f.
Theorem Theta\_trans:
```

```
orall f \ g \ h : nat 
ightarrow nat, Theta \ f \ g 
ightarrow Theta \ g \ h 
ightarrow Theta \ f \ h. Theorem Theta\_sym : orall f \ g : nat 
ightarrow nat, Theta \ f \ g 
ightarrow Theta \ g \ f.
```

28.6 Złożoność typowych funkcji na listach

28.6.1 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.
```

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).

28.6.2 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\_\theta: T\ 0=1. Theorem T\_spec\_S: \forall\ n: nat,\ T\ (S\ n)=1+T\ n. Theorem T\_sum: \forall\ n: nat,\ T\ n=n+1. Theorem O\_T\_n: O\ T\ (\text{fun } n\Rightarrow n).
```

Prześledźmy jeszcze raz cała analize, krok po kroku:

- oznaczamy złożoność analizowanej funkcji przez T
- \bullet patrząc na definicję analizowanej funkcji definiujemy T za pomocą równań T(0)=1 i T(n+1)=T(n)+1
- ullet 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: {\tt Type}\} (fuel: nat) (l: list A): option nat:= {\tt match} \ fuel, \ l \ {\tt with} |\ 0,\ \_\Rightarrow None |\ \_,\ |] \Rightarrow Some \ 0 |\ S\ fuel',\ \_:: \ t \Rightarrow {\tt match} \ length'\ fuel'\ t \ {\tt with} |\ None \Rightarrow None |\ Some\ n \Rightarrow 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: \forall (A: \texttt{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.

Žeby 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.

28.7 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.

28.8 Przyspieszanie funkcji rekurencyjnych

28.8.1 Złożoność rev

Przyjrzyjmy się złożoności funkcji rev.

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.

28.8.2 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 <math>A): list A:= match l with  | \ | \ | \Rightarrow acc \\  | \ h:: t \Rightarrow 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: \forall (A: \texttt{Type}) \ (l \ acc: list \ A), \\ rev\_aux \ l \ acc = rev \ l \ ++ \ acc. Theorem rev'\_spec: \forall (A: \texttt{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 $to\theta$, gdzie $to\theta$ n jest listą liczb od n do 0. Udowodnij poprawność zdefiniowanej funkcji.

```
Theorem to\theta\_spec: \forall n \ k: nat, \ k \leq n \rightarrow elem \ k \ (to\theta \ n). Time Eval compute in rev \ (to\theta \ 2000). (* ===> (...) Finished transaction in 7. secs (7.730824u,0.s) *) Time Eval compute in rev' \ (to\theta \ 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).

28.9 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.