Lean & ITP

Michał Dobranowski

semestr zimowy 2025
v1.0

Niniejszy skrypt zawiera informacje dotyczące systemu Lean 4 oraz podstawy teoretyczne, które warto poznać, aby dowiedzieć się, na czym *interactive theorem proving* w ogóle polega. Powstawał on przy okazji kursu, który miałem przyjemność przygotować i poprowadzić jako student czwartego roku na AGH w Krakowie.

Spis treści

1. Wstęp teoretyczny

Aby zrozumieć, dlaczego systemowi wspomagającego dowodzenie (ang. *proof assistant*) możemy ufać bardziej niż tuszowi na papierze, należy zrozumieć narzędzia oferowane przez logikę, rachunek lambda oraz teorię typów, na których zbudowany jest każdy znany autorowi tego typu system. Chociaż ten kurs nigdy nie miał być teoretyczny, zdaniem autora formalizmy dotyczące (typowanego) rachunku lambda są niezwykle ciekawe, więc w odpowiednich miejscach Czytelnik jest zachęcany do pogłębienia wiedzy, w tym też przeprowadzenia lub przeczytania dowodów przytaczanych twierdzeń.

1.1. Logika intuicjonistyczna

Typowym przykładem ilustrującym różnicę między logiką klasyczną a intuicjonistyczną (konstruktywną) jest twierdzenie:

> Istnieją takie liczby niewymierne a i b, że a^b jest liczbą wymierną.

oraz jego dowód:

> Jeśli $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$, to $a = b = \sqrt{2}$, w przeciwnym razie niech $a = \sqrt{2}^{\sqrt{2}}$ oraz $b = \sqrt{2}$, wtedy $a^b = 2 \in \mathbb{Q}$.

Dowód ten jest oczywiście słuszny na gruncie logiki klasycznej, ale nie jest konstruktywny, ponieważ dalej nie znamy odpowiednich liczb a i b. W logice konstruktywnej zdanie jest prawdziwe, jeśli można podać jego *konstrukcję* (tzn. intuicyjny dowód), zgodnie z interpretacją Brouwera-Heytinga-Kolmogorowa:

- konstrukcja dla $A \wedge B$ to konstrukcja dla A oraz konstrukcja dla B,
- konstrukcja dla $A \vee B$ to konstrukcja dla A lub konstrukcja dla B wraz z zaznaczeniem, która z nich to jest,
- konstrukcja dla $A \to B$ to przekształcenie każdej konstrukcji dla A w konstrukcję dla B,
- nie istnieje konstrukcja dla fałszu.

Formalnie, logika intuicjonistyczna to pewien system logiczny. Nie różni się od logiki klasycznej składnią, ale regułami wnioskowania. Fałsz oznaczamy przez $\bot$, a *osąd* zapisany w postaci $\Gamma \vdash A$ oznacza, że formuła A jest wywodliwa ze zbioru formuł Γ. Zamiast $\Gamma \cup \{B\}$ będziemy często pisać Γ, B.

$$\frac{}{\Gamma, A \vdash A}\ (\mathrm{Ax}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot \mathrm{E})$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge \mathrm{I}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge \mathrm{E1}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge \mathrm{E2})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee \mathrm{I1}) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash B \vee A}\ (\vee \mathrm{I2}) \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee \mathrm{E})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to \mathrm{I}) \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\to \mathrm{E})$$

Rysunek 1: Reguły wnioskowania w intuicjonistycznym rachunku zdań (IRZ).

Oprócz tego definiujemy negację jako $\neg A := A \to \bot$. Dzięki tej definicji oraz regule $(\to \mathrm{E})$ możemy wywieść

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ (\neg \mathrm{E})$$

Oznaczamy również prawdę przez $\top := \neg\bot = \bot \to \bot$.

Przykład 1.1

Pokaż, że w IRZ zachodzi *słabe prawo podwójnej negacji*, czyli $A \to \neg\neg A$.

Rozwiązanie. Z definicji negacji mamy $\neg\neg A = (A \to \bot) \to \bot$, więc musimy pokazać $A \to ((A \to \bot) \to \bot)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, A \to \bot \vdash A \to \bot}\ (\mathrm{Ax}) \qquad \dfrac{}{A, A \to \bot \vdash A}\ (\mathrm{Ax})}{A, A \to \bot \vdash \bot}\ (\to \mathrm{E})}{A \vdash (A \to \bot) \to \bot}\ (\to \mathrm{I})}{\vdash A \to ((A \to \bot) \to \bot)}\ (\to \mathrm{I})$$

□

Przykład 1.2

Pokaż, że w IRZ zachodzi *prawo kontrapozycji*, czyli $(A \to B) \to (\neg B \to \neg A)$.

Rozwiązanie. Z definicji negacji mamy $\neg B = B \to \bot$ oraz $\neg A = A \to \bot$, więc musimy pokazać $(A \to B) \to ((B \to \bot) \to (A \to \bot))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B, B \to \bot, A \vdash B \to \bot}\ (\mathrm{Ax}) \qquad \dfrac{\dfrac{}{A \to B, B \to \bot, A \vdash A \to B}\ (\mathrm{Ax}) \qquad \dfrac{}{A \to B, B \to \bot, A \vdash A}\ (\mathrm{Ax})}{A \to B, B \to \bot, A \vdash B}\ (\to \mathrm{E})}{A \to B, B \to \bot, A \vdash \bot}\ (\to \mathrm{E})}{A \to B, B \to \bot \vdash A \to \bot}\ (\to \mathrm{I})}{A \to B \vdash (B \to \bot) \to (A \to \bot)}\ (\to \mathrm{I})}{\vdash (A \to B) \to ((B \to \bot) \to (A \to \bot))}\ (\to \mathrm{I})$$

□

1.1.1. Związek z logiką klasyczną

Dokładając do reguł wnioskowania IRZ *silne prawo podwójnej negacji* $(\neg\neg A \to A)$ lub *prawo wyłączonego środka* $(A \vee \neg A)$, otrzymujemy logikę klasyczną. W przypadku prawa podwójnej negacji jest to oczywiste. W przypadku prawa wyłączonego środka (EM) można to pokazać następująco:

$$\dfrac{\dfrac{\dfrac{}{(A \to \bot) \to \bot \vdash A \vee (A \to \bot)}\ (\mathrm{EM}) \qquad \dfrac{}{(A \to \bot) \to \bot, A \vdash A}\ (\mathrm{Ax}) \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash (A \to \bot) \to \bot}\ (\mathrm{Ax}) \qquad \dfrac{}{\Gamma \vdash A \to \bot}\ (\mathrm{Ax})}{\Gamma \vdash \bot}\ (\to \mathrm{E})}{\Gamma \vdash A}\ (\bot \mathrm{E})}{(A \to \bot) \to \bot \vdash A}\ (\vee \mathrm{E})}{\vdash (A \to \bot) \to \bot \to A}\ (\to \mathrm{I})$$

gdzie $\Gamma = \{(A \to \bot) \to \bot, A \to \bot\}$.

Problem 1.3. Pokazać, że prawo wyłączonego środka nie jest dowodliwe w IRZ.

Uwaga (ciekawostka)

Istnieją tautologie KRZ, które nie są dowodliwe w IRZ, ale po dodaniu do IRZ jako aksjomaty nie prowadzą do logiki klasycznej, tworząc logiki „pomiędzy" intuicjonistyczną i klasyczną. Przykłady:

- IRZ + $(\neg A \vee \neg\neg A)$[a] — logika Jankova (de Morgana), w której zachodzą wszystkie cztery prawa de Morgana,
- IRZ + $((A \to B) \vee (B \to A))$ — logika Gödla-Dummeta, w której wartościowania formuł można interpretować jako liczby z przedziału $[0, 1]$.

[a] słabe prawo wyłączonego środka

Skoro zbiór reguł wnioskowania IRZ jest podzbiorem zbioru reguł wnioskowania KRZ, to każda formuła dowodliwa IRZ jest również dowodliwa w KRZ.

1.1.2. Semantyka

Trochę zaniedbując formalizmy, skupimy się przez chwilę na wartościowaniach formuł logicznych. Możemy określić *semantykę* dla logiki klasycznej, przypisując formułom prawdziwym wartość 1, a formułom fałszywym wartość 0 (definiując przy okazji funkcje $\wedge, \vee, \to$). Dla logiki intuicjonistycznej jest to trudniejsze, ale i ciekawsze. Pokażemy dwa z (nieskończenie) wielu możliwych sposobów. Oba z nich są *algebrami Heytinga*, których nie będziemy tutaj definiować. Warto jednak wiedzieć, że formuła logiczna jest tautologią IRZ wtedy i tylko wtedy, gdy jest prawdziwa w każdej algebrze Heytinga.

Semantyka topologiczna Możemy zdefiniować semantykę za pomocą topologii na $\mathbb{R}$:

$$\begin{aligned}
[\![\bot]\!] &= \varnothing, \\
[\![\top]\!] &= \mathbb{R}, \\
[\![A \wedge B]\!] &= [\![A]\!] \cap [\![B]\!], \\
[\![A \vee B]\!] &= [\![A]\!] \cup [\![B]\!], \\
[\![A \to B]\!] &= \operatorname{int}\left([\![A]\!]^{\mathsf{C}} \cup [\![B]\!]\right), \\
[\![A]\!] &= \text{dowolny otwarty podzbiór } \mathbb{R}.
\end{aligned}$$

Wtedy

$$[\![\neg A]\!] = [\![A \to \bot]\!] = \operatorname{int}\left([\![A]\!]^{\mathsf{C}} \cup \varnothing\right) = \operatorname{int}\left([\![A]\!]^{\mathsf{C}}\right)$$

Można przy pomocy takiej semantyki pokazać, że prawo wyłączonego środka nie jest dowodliwe w IRZ. Pod $[\![A]\!]$ możemy podstawić np. zbiór $(0, \infty)$, wtedy

$$[\![A \vee \neg A]\!] = [\![A]\!] \cup [\![\neg A]\!] = [\![A]\!] \cup \operatorname{int}\left([\![A]\!]^{\mathsf{C}}\right) = (0, \infty) \cup (-\infty, 0) \neq \mathbb{R}$$

Semantyka kraty dystrybutywnej Krata to zbiór częściowo uporządkowany, w którym istnieją kresy dolne i górne dowolnych par elementów. Będziemy je przedstawiać za pomocą *diagramów Hassego*[1]. Definiujemy działania

$$\begin{aligned}
a \wedge b &:= \inf\{a, b\}, \\
a \vee b &:= \sup\{a, b\}.
\end{aligned}$$

[1] Czym dokładnie jest diagram Hassego można dowiedzieć się na Wikipedii.

Krata dystrybutywna to krata, w której zachodzą prawa rozdzielności:

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c),$$
$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c).$$

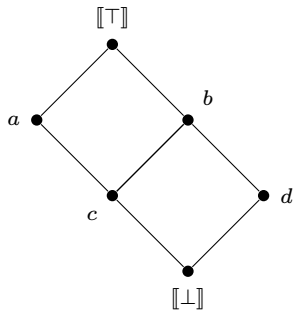
Można udowodnić, że każda niepusta i skończona krata jest ograniczona, czyli posiada elementy najmniejszy i największy. Krata, która jest niepusta, skończona i dystrybutywna posłuży nam do zdefiniowania semantyki:

$$\begin{aligned}
[\![\bot]\!] &= \text{element najmniejszy},\\
[\![\top]\!] &= \text{element największy},\\
[\![A \wedge B]\!] &= [\![A]\!] \wedge [\![B]\!],\\
[\![A \vee B]\!] &= [\![A]\!] \vee [\![B]\!],\\
[\![A \to B]\!] &= \sup\{c : c \wedge [\![A]\!] \leqslant [\![B]\!]\},\\
[\![A]\!] &= \text{dowolny element kraty}.
\end{aligned}$$

Wtedy

$$[\![\neg A]\!] = [\![A \to \bot]\!] = \sup\{c : c \wedge [\![A]\!] \leqslant [\![\bot]\!]\} = \sup\{c : c \wedge [\![A]\!] = [\![\bot]\!]\}$$

Biorąc przykładową kratę



możemy udowodnić, że prawo wyłączonego środka nie jest dowodliwe w IRZ. Jeśli weźmiemy $[\![A]\!] = c$, to $[\![\neg A]\!] = d$, więc $[\![A \vee \neg A]\!] = c \vee d = b \neq [\![\top]\!]$.

Problem 1.4. Pokazać, że silne prawo podwójnej negacji nie jest dowodliwe w IRZ na podstawie dwóch powyższych semantyk.

Problem 1.5. Stwierdzić, które z czterech praw de Morgana są dowodliwe w IRZ.

1.2. Rachunek lambda

Rachunek lambda to język złożony z termów, z których każdy to:

- zmienna (zazwyczaj oznaczana małą literą, np. x),
- λ-abstrakcja postaci $\lambda x.\, M$, gdzie x jest zmienną, a M jest termem,
- aplikacja postaci MN, gdzie M i N są termami.

Formalnie termy można więc zdefiniować następująco:

$$M ::= x \mid (\lambda x.\, M) \mid (MM),$$

gdzie x reprezentuje dowolną zmienną.

Aby uprościć zapis, będziemy pisać MNP zamiast $(MN)P$ – to znaczy stwierdzamy, że aplikacja jest łączna lewostronnie. Ponadto, wiele λ-abstrakcji zapisujemy jako $\lambda x_1 \dots x_n.\, M$, co jest równoważne termowi $\lambda x_1.\, (\lambda x_2.\, (\dots (\lambda x_n.\, M) \dots))$. Kropka w tym zapisie jest bardzo istotna, np.

$$\lambda xyz.\, M = \lambda x.\, (\lambda y.\, (\lambda z.\, M)),$$
$$\lambda xy.\, zM = \lambda x.\, (\lambda y.\, (zM)).$$

W termie $\lambda x.\, M$ zmienna x jest *zmienną związaną* w M. Zmienne występujące w M, które nie są związane przez żadną λ-abstrakcję, nazywamy *zmiennymi wolnymi*. Na przykład w termie $\lambda x.\, (xy)$ zmienna x jest zmienną związaną, a y jest zmienną wolną. W termie $xz(\lambda xy.\, (xyz))$ zmienna x raz występuje jako zmienna wolna, a raz jako związana. Takich sytuacji będziemy unikać ze względów czysto estetycznych.

Zbiór zmiennych wolnych termu M oznaczamy jako $\mathrm{FV}(M)$. Term nazywamy *termem zamkniętym* lub *kombinatorem*, jeśli $\mathrm{FV}(M) = \varnothing$.

Uwaga 1.6

Formalnie definiujemy $\mathrm{FV}(M)$ rekurencyjnie:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}, \\ \mathrm{FV}(\lambda x.\, M) &= \mathrm{FV}(M) \setminus \{x\}, \\ \mathrm{FV}(MN) &= \mathrm{FV}(M) \cup FV(N). \end{aligned}$$

1.2.1. Alfa-konwersja

α-konwersja to przekształcenie termu, które polega na zmianie nazwy zmiennej (unikając kolizji oznaczeń zmiennych). Wprowadzamy relację równoważności $\equiv_\alpha$ w zbiorze termów Λ w ten sposób, że dane dwa termy M i N są równoważne, jeśli można otrzymać jeden z drugiego poprzez (wielokrotne) α-konwersje. Na przykład:

$$a(\lambda b.\, bc) \equiv_\alpha a(\lambda d.\, dc),$$

natomiast

$$\lambda a.\, ab \not\equiv_\alpha \lambda b.\, bb,$$

ponieważ zamiana zmiennej a na b prowadzi do kolizji oznaczeń zmiennych.

Od tej pory utożsamiamy ze sobą termy różniące się jedynie nazwami zmiennych, czyli jeśli $M \equiv_\alpha N$, to M i N są tym samym termem.

Uwaga 1.7

Bardziej formalnie, od tej pory będziemy operować na klasach abstrakcji relacji $\equiv_\alpha$ (elementach zbioru $\Lambda/\equiv_\alpha$), podobnie jak przy działaniach modulo operujemy na klasach abstrakcji relacji przystawania modulo p (elementach zbioru $\mathbb{Z}/\equiv_p$).

Będziemy stosować dosyć uniwersalny zapis $M[x := N]$ na term, który powstał z termu M poprzez zastąpienie wszystkich *wolnych* wystąpień zmiennej x termem N. Zakładamy przy tym, że żadna zmienna wolna w N nie zacznie być związana w $M[x := N]$ (ponownie unikamy kolizji oznaczeń).

1.2.2. Beta-redukcja

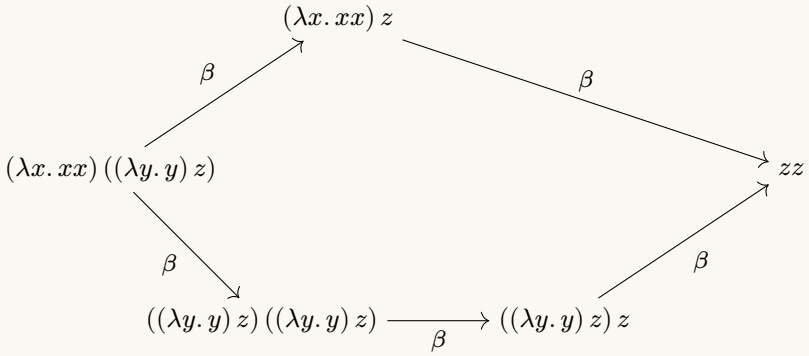
β-redukcja to taka relacja $\to_\beta$ w zbiorze termów Λ, że $M \to_\beta N$, gdy *β-redeks* postaci $(\lambda x.\, P)Q$ w termie M zostaje zastąpiony przez $P[x := Q]$ w termie N. Bardziej formalnie, jest to najmniejsza relacja w zbiorze Λ spełniająca następujące warunki:

1. $(\lambda x.\, P)Q \to_\beta P[x := Q]$,
2. jeśli $M \to_\beta M'$, to
   - $MN \to_\beta M'N$,
   - $NM \to_\beta NM'$,
   - $\lambda x.\, M \to_\beta \lambda x.\, M'$.

Jeśli w termie występuje więcej niż jeden β-redeks, to β-redukcja nie jest deterministyczna – możemy wybrać dowolny z nich i wykonać redukcję. Przez $\twoheadrightarrow_\beta$ oznaczamy domknięcie przechodnio-zwrotne relacji $\to_\beta$ (czyli najmniejsza relacja przechodnia i zwrotna, która zawiera relację $\to_\beta$), a przez $=_\beta$ jej domknięcie równoważnościowe.

Przykład 1.8

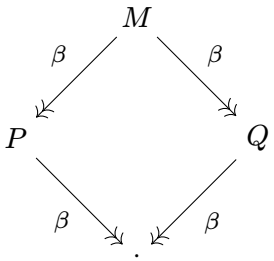
Pokazane poniżej są różne ścieżki redukcyjne dla podanego termu.



Twierdzenie 1.9 (Churcha-Rossera)

Jeśli $M \twoheadrightarrow_\beta P$ oraz $M \twoheadrightarrow_\beta Q$, to istnieje takie M', że $P \twoheadrightarrow_\beta M'$ i $Q \twoheadrightarrow_\beta M'$.

Jest to jedno z ważniejszych twierdzeń rachunku lambda, mówiące o tym, że relacja $\twoheadrightarrow_\beta$ ma *własność rombu* (ang. *diamond property*), której notabene nie ma relacja $\to_\beta$ (czego dowodzi przykład 1.8). Jeśli dopełnienie przechodnio-zwrotne relacji ma własność rombu, to mówimy, że relacja ta jest *konfluentna*. Powyższe twierdzenie mówi więc, że relacja $\to_\beta$ jest konfluentna.



Rysunek 2: Własność rombu relacji $\twoheadrightarrow_\beta$.

Prostym wnioskiem z tego twierdzenia jest fakt, że każdy term ma co najwyżej jedną postać normalną, to znaczy pozbawioną β-redeksów. Dlaczego *co najwyżej*, a nie *dokładnie*? Na przykład term

$$\mathbf{\Omega} := (\lambda x.\, xx)\,(\lambda x.\, xx)$$

posiada tylko jeden β-redeks, a jego redukcja prowadzi do termu $\mathbf{\Omega}$ (co Czytelnik raczy sprawdzić), czyli $\mathbf{\Omega} \to_\beta \mathbf{\Omega}$. Term $\mathbf{\Omega}$ nie ma więc postaci normalnej.

1.2.3. Kombinator punktu stałego

Kombinator $\mathbf{Y} := \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$ nazywamy *kombinatorem punktu stałego*. Ma on tę ciekawą własność, że dla każdego termu $F \in \Lambda$, zachodzi

$$F(\mathbf{Y}(F)) =_\beta \mathbf{Y}(F),$$

ponieważ

$$F(\mathbf{Y}(F)) \to_\beta F\,((\lambda x.\, F(xx))(\lambda x.\, F(xx)))$$

oraz

$$\begin{aligned}\mathbf{Y}(F) &\to_\beta (\lambda x.\, F(xx))(\lambda x.\, F(xx)) \\ &\to_\beta F\,((\lambda x.\, F(xx))(\lambda x.\, F(xx)))\,.\end{aligned}$$

Problem 1.10. Znajdź term P taki, że $Px =_\beta P$.

Problem 1.11. Znajdź term P taki, że $Px =_\beta xP$.

1.2.4. Eta-redukcja

η-redukcja to taka relacja $\to_\eta$ w zbiorze termów Λ, że $M \to_\eta N$, gdy *η-redeks* postaci $\lambda x.\, Px$ w termie M zostaje zastąpiony przez P w termie N, zakładając $x \notin \mathrm{FV}(P)$. Bardziej formalnie, jest to najmniejsza relacja w zbiorze Λ spełniająca następujące warunki:

1. jeśli $x \notin \mathrm{FV}(P)$, to $\lambda x.\, Px \to_\eta P$,
2. jeśli $M \to_\eta M'$, to
   - $MN \to_\eta M'N$,
   - $NM \to_\eta NM'$,
   - $\lambda x.\, M \to_\eta \lambda x.\, M'$.

Tak jak poprzednio, domknięcie przechodnio-zwrotne relacji $\to_\eta$ oznaczamy przez $\twoheadrightarrow_\eta$. Ponadto, sumę relacji $\to_\beta$ i $\to_\eta$ oznaczamy przez $\to_{\beta\eta}$, a jej domknięcie przechodnio-zwrotne przez $\twoheadrightarrow_{\beta\eta}$.

Twierdzenie 1.12 (Churcha-Rossera dla η-redukcji i $\beta\eta$-redukcji)
Relacje $\to_\eta$ oraz $\to_{\beta\eta}$ są konfluentne.

1.2.5. Wyrażalność algorytmów w rachunku lambda

Liczby naturalne możemy reprezentować w rachunku lambda za pomocą tzw. *liczebników Churcha*:

$$\mathbf{n} = \lambda fx.\, f^n(x),$$

na przykład

$$\begin{aligned}\mathbf{0} &= \lambda fx.\, x,\\ \mathbf{1} &= \lambda fx.\, f(x),\\ \mathbf{2} &= \lambda fx.\, f(f(x)),\\ \mathbf{3} &= \lambda fx.\, f(f(f(x))).\end{aligned}$$

Wtedy operacja następnika jest reprezentowana przez term

$$\mathbf{succ} = \lambda nfx.\, f(nfx),$$

a dodawanie przez term

$$\mathbf{add} = \lambda mnfx.\, mf(nfx).$$

Instrukcje warunkowe można zaimplementować za pomocą termów

$$\mathbf{true} = \lambda xy.\, x,$$

$$\mathbf{false} = \lambda xy.\, y,$$

wtedy instrukcja warunkowa `if B then M else N` jest reprezentowana przez term

$$\mathbf{if} = \lambda BMN.\, BMN.$$

Istotnie,

$$\begin{aligned}\mathbf{if\ true}\ M\ N &\to_\beta (\lambda xy.\, x)MN \to_\beta M,\\ \mathbf{if\ false}\ M\ N &\to_\beta (\lambda xy.\, y)MN \to_\beta N.\end{aligned}$$

Porównanie liczby n do zera można zaimplementować za pomocą termu

$$\mathbf{iszero} = \lambda n.\, n(\lambda x.\, \mathbf{false})\ \mathbf{true}.$$

Problem 1.13. Zdefiniuj w rachunku lambda negację, koniunkcję, alternatywę oraz implikację.

Problem 1.14. Zdefiniuj w rachunku lambda funkcje mnożenia oraz potęgowania liczb naturalnych.

Problem 1.15. Zdefiniuj w rachunku lambda funkcję poprzednika liczby naturalnej. Poprzednik zera powinien zwracać zero.

Problem 1.16. Zdefiniuj w rachunku lambda funkcję obliczającą $n!$.

Twierdzenie 1.17 (Kleene'a)
Każda funkcja częściowo rekurencyjna jest reprezentowalna w rachunku lambda.

Twierdzenie odwrotne również jest prawdziwe. Funkcje częściowo rekurencyjne to dokładnie te funkcje, które są obliczalne przez maszyny Turinga. Z powyższego twierdzenia wynika więc, że rachunek lambda jest modelem obliczalności równoważnym maszynom Turinga.

1.3. Typowany rachunek lambda

Do rachunku lambda będziemy chcieli dodać typy w następujący sposób. Każdy term rachunku lambda będzie miał przypisany pewien typ, co będziemy zapisywać jako $M : \tau$, gdzie M jest termem, a τ jest typem. Niech $\mathbb{A}$ będzie zbiorem *typów atomowych*. Zbiór wszystkich typów $\mathbb{T}$ definiujemy rekurencyjnie:

$$\begin{aligned} \alpha \in \mathbb{A} &\implies \alpha \in \mathbb{T}, \\ \sigma, \tau \in \mathbb{T} &\implies (\sigma \to \tau) \in \mathbb{T}, \end{aligned}$$

gdzie druga reguła jest *konstruktorem typów funkcyjnych*. Zakładamy przy tym, że nie istnieją typy $\alpha, \beta, \gamma \in \mathbb{A}$ takie, że $\alpha \to \beta = \gamma$. Zamiast pisać $\sigma \to (\tau \to \rho)$, będziemy pisać $\sigma \to \tau \to \rho$ (operator $\to$ jest łączny prawostronnie).

$$\frac{}{\Gamma, x : A \vdash x : A}\ (\text{Ax})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : (A \to B)}\ (\lambda \text{ abs}) \qquad \frac{\Gamma \vdash M : (A \to B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\lambda \text{ app})$$

Rysunek 3: Reguły przypisania typów.

Przykład 1.18

Niech $\mathbb{A} = \{\alpha\}$. Wtedy term $\mathbf{true} = \lambda x.\, \lambda y.\, x$ ma typ $\alpha \to \alpha \to \alpha$, ponieważ

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ (\text{Ax})}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ (\lambda \text{ abs})}{\vdash \lambda x.\, \lambda y.\, x : \alpha \to \alpha \to \alpha}\ (\lambda \text{ abs})$$

Oznaczając $\alpha \to \alpha \to \alpha$ przez $\mathbb{B}$ (typ boolowski), możemy pokazać, że term **neg true** również ma typ $\mathbb{B}$.

1.3.1. Izomorfizm Curry'ego-Howarda

Pokazany powyżej system typów możemy rozszerzyć o konstruktory typów $\wedge$ i $\vee$ oraz dodać do $\mathbb{A}$ specjalny typ $\bot$, który nie będzie przypisany żadnemu termowi. Reguły przypisania typów należałoby wtedy rozszerzyć o odpowiednie zasady konstrukcji i destrukcji dla tych nowych typów.

Izomorfizm Curry'ego-Howarda to twierdzenie, które łączy logikę i tak zbudowaną teorię typów. Mówi ono, że jeśli formułom logicznym przypiszemy odpowiednie typy, a dowodom tych formuł przypiszemy odpowiednie termy, to otrzymamy izomorfizm postaci:

$$\text{formuła ma dowód} \iff \text{typ ma term.}$$

W ten sposób możemy przepisać formułę logiczną do postaci typu i udowodnić ją, konstruując term tego typu. W tym właśnie procesie przydaje się oprogramowanie takie jak Lean.

logika intuicjonistyczna	typowany rachunek lambda
formuła A	typ $A \in \mathbb{T}$
dowód formuły A	term typu A
implikacja	typ funkcyjny $(\to)$
formuła niedowodliwa	typ nieposiadający termu
tautologia	typ kombinatora

Tabela 1: Odpowiadające sobie pojęcia w logice i w typowanym rachunku lambda.

2. Podstawy dowodzenia w Lean

Polecam używać VS Code z rozszerzeniem **Lean 4** lub Neovim z rozszerzeniem **lean.nvim**. Aby rozpocząć pracę, należy stworzyć projekt za pomocą

```
lake new project-name math-lax
```

który od razu dodaje Mathlib (de facto bibliotekę standardową, zawierającą wiele twierdzeń z różnych działów matematyki) do wymaganych zależności.

2.1. Składnia – bardzo krótki wstęp

Póki co nie będziemy się zagłębiać w składnię, a jedynie wymienimy kilka podstawowych elementów, które posłużą nam do w miarę swobodnego poruszania się w Leanie (a przynajmniej jego części dotyczącej dowodzenia twierdzeń matematycznych).

Funkcje i stałe Funkcje (oraz stałe, które są funkcjami stałej wartości) definiujemy za pomocą słowa kluczowego `def`, na przykład:

```
def add (a b : Nat) : Nat := a + b
```

definiuje funkcję `add`, która przyjmuje dwie liczby naturalne (typ `Nat`) i zwraca ich sumę.

Twierdzenia i przykłady Twierdzenia definiujemy za pomocą słowa kluczowego `theorem` lub `lemma`, na przykład:

```
theorem add_zero (a : Nat) : add a 0 = a := -- dowód
```

przy czym używanie `lemma` wymaga dodania Mathliba (`import Mathlib.Tactic`). Jeśli mamy pewne stwierdzenie (z dowodem), którego nie będziemy używać, więc nie potrzebuje nazwy, możemy użyć słowa kluczowego `example`:

```
example (a : Nat) : add 0 a = a := -- dowód
```

Samo sformułowanie twierdzeń, lematów i przykładów jest często pewną trudnością. Możemy być więc w sytuacji, gdy chcemy najpierw sformułować tezę, a jej dowód zostawić na później. W takim przypadku nieodzowne będzie słowo kluczowe `sorry`, dzięki któremu program się skompiluje (z ostrzeżeniem), mimo braku dowodu.

```
theorem pow_comm (a b : Nat) : a ^ b = b ^ a :=
  sorry
```

Oczywiście, taka teza może być fałszywa (jak powyżej).

> **Uwaga 2.1** (Plausible)
> Istnieje paczka Leana o nazwie **Plausible**, która pomaga w szybkim szukaniu kontrprzykładów do hipotez. Jej działania opiera się po prostu na losowym próbkowaniu argumentów i sprawdzaniu, czy teza zachodzi. Jeśli nie, to zwraca znaleziony kontrprzykład (przez informację diagnostyczną błędu kompilacji).
>
> ```
> import Plausible
>
> theorem pow_comm (a b : Nat) : a ^ b = b ^ a := by
>   plausible
> ```

Aksjomaty Czasami będziemy chcieli wprowadzić pewne aksjomaty, czyli stwierdzenia przyjmowane bez dowodu. W tym celu używamy słowa kluczowego `axiom`:

```
axiom add_comm (a b : Nat) : a + b = b + a
```

Komentarze Komentować kod można i należy następująco:

```
def foo : Nat := 42 -- to jest komentarz jednowierszowy
/- to jest
komentarz
wielowierszowy -/
```

Sprawdzenie typu Sprawdzić typ termu (czyli, przez izomorfizm Curry'ego-Howarda, również wypowiedź twierdzenia), możemy następująco:

```
#check Nat -- Nat : Type
#check Nat.add -- Nat.add : Nat → Nat → Nat
#check Nat.add 1 -- Nat.add 1 : Nat → Nat
#check Nat.add 1 2 -- Nat.add 1 2 : Nat
#check Nat.add_comm -- Nat.add_comm (n m : Nat) : n + m = m + n
```

Problem 2.2. Skoro `Nat` jest typu `Type`, to jaki jest typ `Type`? Jaki jest typ tego typu?

Ewaluacja funkcji Czasami, zamiast dowodzić twierdzeń, będziemy chcieli po prostu obliczyć wartość pewnej funkcji. Możemy to zrobić tak:

```
#eval Nat.add 2 3 -- 5
```

Operatory potoku i odwróconego potoku Jak na porządny język programowania przystało, Lean posiada operatory umożliwiające kontrolę kolejności ewaluacji wyrażeń. Operatory `<|` i `|>` zachowują się tak samo jak w F# czy Elm, sam operator `|>` znajdziemy również w Elixirze czy OCamlu. W Haskellu odpowiadają one odpowiednio operatorom `$` i `&`. Przykłady użycia:

```
#eval Nat.add 1 (Nat.mul 2 3)   -- 7
#eval Nat.add 1 <| Nat.mul 2 3  -- 7
#eval Nat.mul 2 3 |> Nat.add 1  -- 7
```

2.1.1. Tryby dowodzenia

Dowodząc twierdzenie, lemat, czy przykład, możemy wybrać jeden z dwóch trybów:

- *tryb termowy* (ang. *term mode*), w którym dowód jest pojedynczym (potencjalnie długim) termem,
- *tryb taktyczny* (ang. *tactic mode*), w którym dowód jest ciągiem *taktyk* modyfikujących stan dowodu.

Pierwszy jest zbliżony do rachunku lambda, drugi bardziej przypomina tradycyjne dowodzenie matematyczne. Często proste fakty łatwiej udowodnić w trybie termowym, a bardziej złożone w trybie taktycznym. Oto przykład dowodu (a właściwie zastosowania twierdzenia) w obu trybach:

```
-- term mode (aplikacja 2a i 2b do twierdzenia Nat.add_comm)
example (a b : Nat) : 2 * a + 2 * b = 2 * b + 2 * a := Nat.add_comm (2 * a) (2 * b)
-- tactic mode (po by następuje taktyka rw)
example (a b : Nat) : 2 * a + 2 * b = 2 * b + 2 * a := by rw [Nat.add_comm]
```

2.1.2. Argumenty i ich rodzaje

Czytelnik już zapewne zdążył zauważyć, że twierdzenia w Leanie zazwyczaj przyjmują argumenty. Na przykład w twierdzeniu `Nat.add_comm` mamy argumenty `(n : Nat)` i `(m : Nat)`, zapisywane jako `(n m : Nat)`. Takimi argumentami będą również założenia twierdzenia, na przykład w:

```
theorem tw (a : Nat) (b : Nat) (c : Nat) (h₁ : a < b) (h₂ : b < c) : a < c := sorry
```

Chcąc użyć takiego twierdzenia, musimy podać wszystkie argumenty, np.

```
axiom one_lt_two : 1 < 2
axiom two_lt_three : 2 < 3

#check tw 1 2 3 one_lt_two two_lt_three
```

Widać tutaj więc pewną redundancję – argumenty `a`, `b` i `c` są w pełni określone przez założenia `h₁` i `h₂`. Aby tego uniknąć, Lean pozwala na definiowanie argumentów *implicit*, czyli takich, które mogą być pominięte przy wywoływaniu danej funkcji lub twierdzenia. W tym celu używamy nawiasów klamrowych zamiast okrągłych:

```
theorem tw {a : Nat} {b : Nat} {c : Nat} (h₁ : a < b) (h₂ : b < c) : a < c := sorry
```

Teraz możemy użyć tego twierdzenia jako

```
#check tw one_lt_two two_lt_three
```

Oprócz argumentów *explicit* i *implicit* Lean pozwala na definiowanie argumentów *instance implicit* oraz *strict implicit*, oznaczanych odpowiednio przez [...] oraz ⦃...⦄. Zajmiemy się nimi później.

2.1.3. Typy Sort, Type i Prop

W ramach problemu 2.2 Czytelnik miał okazję zastanowić się nad typami `Type`, `Type 1`, `Type 2` itd. Są one częścią nieskończonej hierarchii typów `Sort u`, gdzie `u` jest zmienną uniwersalną oznaczającą poziom uniwersum. Typ `Sort u` jest więc typu `Sort (u + 1)`. `Type u` jest synonimem dla `Sort (u + 1)`, a `Prop` jest synonimem dla `Sort 0`.

Skąd takie przesunięcie? Typy typu `Type u` (a więc sorty poziomu co najmniej 1) „trzymają" pewne obiekty matematyczne, np. term (liczba) 1 jest „trzymana" przez typ `Nat : Type 0`. Typy typu `Prop` (czyli sorty poziomu 0) nie „trzymają" nic. Tego typu będą więc twierdzenia, od których nie wymagamy (albo nawet nie chcemy), żeby trzymały swój dowód (*proof irrelevance*).

2.2. Rachunek zdań i pierwsze taktyki

Od tego miejsca będziemy stosować logikę klasyczną. Jak Lean radzi sobie z rozbieżnościami między logiką intuicjonistyczną a klasyczną? Definiuje aksjomat wyboru:

```
axiom Classical.choice {α : Sort u} : Nonempty α → α,
```

z którego następnie, dzięki twierdzeniu Diaconescu, wyprowadza prawo wyłączonego środka `Classical.em` oraz wiele innych praw logiki klasycznej, jak chociażby silne prawo podwójnego przeczenia `Classical.not_not`. Niżej omawiane taktyki Leana korzystają z tych aksjomatów, gdy jest to konieczne.

Taktyki `intro`, `exact` i `apply` Jeśli cel dowodzenia znajduje się wśród założeń, to wystarczy go wskazać za pomocą taktyki `exact`. Jeśli cel jest w postaci `P → Q`, to możemy użyć taktyki `intro`, aby wprowadzić do dowodu założenie `P` i zmienić cel na `Q`.

```
example {P : Prop} (h : P) : P := by
  exact h

example {P : Prop} : P → P := by
  intro h
  exact h
```

Jeśli celem jest `Q`, a mamy wśród założeń `P → Q`, to możemy użyć taktyki `apply`, aby zmienić cel na `P`. Możemy stosować również aplikację znaną z typowanego rachunku lambda.

```
example {P Q : Prop} (h₁ : P → Q) (h₂ : P) : Q := by
  apply h₁
  exact h₂

example {P Q : Prop} (h₁ : P → Q) (h₂ : P) : Q := by
  exact h₁ h₂
```

Koniunkcja W Lean istnieje struktura `And`, której jedynym konstrukturem jest

```
And.intro {a b : Prop} (left : a) (right : b) : a ∧ b.
```

Jako że `And.intro` ma typ `a → b → a ∧ b`, to świetnie nadaje się do użycia z `apply`, które wprowadzi dwa cele (do rozwiązania jeden po drugim).

Jeśli chcemy w ten sposób użyć konstruktora dowolnej struktury, to możemy użyć taktyki `constructor`.

```
example {P Q : Prop} (hp : P) (hq : Q) : P ∧ Q := by
  apply And.intro
  · exact hp
  · exact hq

example {P Q : Prop} (hp : P) (hq : Q) : P ∧ Q := by
  constructor
  · exact hp
  · exact hq

example {P Q : Prop} (h : P ∧ Q) : P := by
  exact h.left
```

Równoważność Struktura `Iff`, która reprezentuje równoważność logiczną, ma konstruktor

```
Iff.intro {a b : Prop} (mp : a → b) (mpr : b → a) : a ↔ b,
```

gdzie pierwszy argument to *modus ponens* ($a \implies b$), a drugi to *modus ponens reversed* ($b \implies a$). Sposób użycia jest analogiczny jak w przypadku koniunkcji. Na przykład

```
example {P Q : Prop} (h : P ↔ Q) : Q → P := by
  exact h.mpr
```

Alternatywa Istnieje również struktura `Or`, której konstruktorami są

```
inl {a b : Prop} (h : a) : a ∨ b
```

oraz

```
inr {a b : Prop} (h : b) : a ∨ b.
```

Aby ich użyć, skorzystamy z taktyki `cases` lub `rcases` (które rekurencyjnie wywołuje `cases` zgodnie z podanym wzorem).

```
example {P Q R : Prop} (hpq : P ∨ Q) (hpr : P → R) (hqr : Q → R) : R := by
  cases hpq with
  | inl hp =>
    apply hpr
    exact hp
  | inr hq =>
    apply hqr
    apply hq

example {P Q R : Prop} (hpq : P ∨ Q) (hpr : P → R) (hqr : Q → R) : R := by
  rcases hpq with hp | hq
  · apply hpr
    exact hp
  · apply hqr
    apply hq
```

Taktyka `rcases` może się przydać również przy koniunkcjach i bardziej złożonych zdaniach, na przykład:

```
example {P Q R S : Prop} (h : S ∧ R ∧ P ∨ S ∧ (Q ∧ R)) : R := by
  rcases h with ⟨_, hr, _⟩ | ⟨_, ⟨_, hr⟩⟩
  · exact hr
  · exact hr
```

Jeśli alternatywa jest naszym celem, a nie założeniem, to przydatne będą taktyki `left` i `right`.

```
example {P Q : Prop} (h : P) : P ∨ Q := by
  left
  exact h
```

> **Uwaga 2.3**
> Dokładniej, taktyka `constructor` używa pierwszego pasującego konstruktora struktury, a taktyki `left` i `right` używają odpowiednio pierwszego i drugiego konstruktora struktury, która ma dokładnie dwa konstruktory. Nie są one przypisane do struktur `And` i `Or`. W szczególności, zamiast `left` zawsze można używać `constructor`, ale sensowność takiego działania pozostawiamy do oceny Czytelnikowi.

Negacja Podobnie jak w IRZ, `¬P` definiujemy jako `P → False`, gdzie `False` jest typem bez termów (i bez konstruktora). Bardzo łatwo jest więc dowieść, że $\neg(P \wedge \neg P)$.

```
example (P : Prop) : ¬(P ∧ ¬P) := by
  intro h
  exact h.right h.left
```

Z fałszu możemy oczywiście dowieść wszystko. W Leanie *ex falso quodlibet* jest realizowane przez `False.elim` lub `absurd`.

```
example {P Q : Prop} (h : P) (nh : ¬P) : Q := by
  exact False.elim (nh h)
```

```
example {P Q : Prop} (h : P) (nh : ¬P) : Q := by
  exact absurd h nh
```

Taktyki `assumption`, `contradiction`, `trivial` oraz operator `<;>` Poniższy przykład (reguła *modus tollendo ponens*) Czytelnik powinien już być w stanie zrozumieć. Pokażemy, jak udowodnić go trochę prościej.

```
example {P Q : Prop} (h : P ∨ Q) (not_p : ¬P) : Q := by
  rcases h with hp | hq
  · exact False.elim (not_p hp)
  · exact hq
```

Zamiast `exact hq` możemy użyć taktyki `assumption`, która sprawdza, czy cel jest wśród założeń (nie musimy go wskazywać). Zamiast `False.elim` możemy użyć taktyki `contradiction`, która sprawdza, czy wśród założeń znajdują się dwa trywialne sprzeczne stwierdzenia.

```
example {P Q : Prop} (h : P ∨ Q) (not_p : ¬P) : Q := by
  rcases h with hp | hq
  · contradiction
  · assumption
```

Jeśli nie używamy wprowadzonych założeń (w przykładzie `hp` i `hq`), to nie musimy ich nazywać – dalej będą one dostępne dla taktyk `assumption` i `contradiction`.

```
example {P Q : Prop} (h : P ∨ Q) (not_p : ¬P) : Q := by
  cases h
  · contradiction
  · assumption
```

Istnieje również taktyka `trivial`, która próbuje rozwiązać cel używając kilku prostych taktyk, jak `assumption`, `contradiction` czy `rfl` (użyteczna przy równościach).

```
example {P Q : Prop} (h : P ∨ Q) (not_p : ¬P) : Q := by
  cases h
  · trivial
  · trivial
```

Jeśli chcemy użyć tej samej taktyki dla różnych przypadków, to możemy użyć `<;>` w następujący sposób:

```
example {P Q : Prop} (h : P ∨ Q) (not_p : ¬P) : Q := by
  cases h <;> trivial
```

Problem 2.4. Zrobić pierwszy zestaw zadań:

```
-- Zadanie 1
example {A B C D : Prop} (h₁ : A → B) (h₂ : B → C) (h₃ : C → D) : A → D := sorry

-- Zadanie 2
example (P Q R : Prop) (h₁ : P ∧ Q) (h₂ : P → R) : R := sorry

-- Zadanie 3
theorem modus_tollens {P Q : Prop} (h₁ : P → Q) (h₂ : ¬Q) : ¬P := sorry

-- Zadanie 4
example {P Q : Prop} (h : ¬(P ∨ Q)) : ¬P ∧ ¬Q := sorry

-- Zadanie 5
example {P Q : Prop} (h : ¬P ∧ ¬Q) : ¬(P ∨ Q) := sorry

-- Zadanie 6
example (A B C D E : Prop) (h₁ : A ∧ B) (h₂ : B → ¬C ∧ D) (h₃ : E → C) : ¬E := sorry

-- Zadanie 7
example {P : Prop} : ((P → P) → P) → P := sorry

-- Zadanie 8
-- Pokaż, że system logiczny oparty na logice klasycznej z aksjomatem A jest trywialny.
axiom A {P : Prop} : (((P → P) → P) → P) → P

example {Q : Prop} : Q := sorry
```

2.3. Przekształcenia algebraiczne i równości

Jeśli nasz cel jest trywialną równością, możemy użyć taktyki `rfl` (od ang. *reflexivity*).

```
example (a : Nat) : a = a := by
  rfl

example : 1 + 1 = 2 := by
  rfl
```

Jeśli nasz cel nie jest równością, ale wymaga jedynie *obliczenia*, a nie dowodu (jak w drugim przykładzie powyżej), wystarczy użyć taktyki `decide`.

```
example : 2 * 3 + 1 ≥ 4 := by
  decide
```

Często jednak będziemy chcieli użyć jakiejś równości do zmiany fragmentu bardziej skomplikowanej formuły. W tym celu możemy użyć taktyki `rewrite`, lub `rw`, która działa jak `rewrite`, ale próbuje użyć jeszcze `rfl`.

```
example (h : a = b) : a^3 + 1 = b^3 + 1 := by
  rewrite [h]
  rfl

example (h : a = b) : a^3 + 1 = b^3 + 1 := by
  rw [h]
```

Domyślnie te taktyki przepisują lewą stronę równości na prawą. Aby przepisać w drugą stronę, należy użyć ← przed nazwą równości.

```
example {a b c : Nat} (h₁ : 2 * b = a) (h₂ : b = 2 * c) : a = 4 * c := by
  rewrite [←h₁]
  rewrite [h₂]
  rewrite [←Nat.mul_assoc]
  rfl

example {a b c : Nat} (h₁ : 2 * b = a) (h₂ : b = 2 * c) : a = 4 * c := by
  rw [←h₁, h₂, ←Nat.mul_assoc]
```

Jeśli chcemy przepisać nie część celu, a któregoś z założeń, wystarczy go wskazać za pomocą **`at`**. Możemy też przepisywać kilka formuł naraz, oddzielając je spacjami, gdzie cel oznaczamy przez ⊢.

```
example {a b c : Nat} (h₁ : 2 * b = a) (h₂ : b = 2 * c) : a = 4 * c := by
  rw [h₂, ←Nat.mul_assoc] at h₁
  rw [h₁]

example {a b : Nat} (h_eq : a = b + 1) (h : a + b = 4) : 2 * a = 5 := by
  rw [h_eq] at h ⊢
  rw [Nat.two_mul]
  rw [←Nat.add_assoc (b + 1) b 1]
  rw [h]
```

Bardzo często będziemy chcieli udowodnić najpierw jakiś cel pośredni, żeby następnie wykorzystać go do udowodnienia głównego celu. W takiej sytuacji nie trzeba definiować nowego lematu, a wystarczy użyć taktyki **`have`**.

```
example {a : Nat} : a * 2 + 1 = 2 * a + 1 := by
  -- h₁, h₂, h₃ to identyczne założenia
  have h₁ : a * 2 = 2 * a := by rw [Nat.mul_comm]
  have h₂ : a * 2 = 2 * a := Nat.mul_comm a 2
  have h₃ := Nat.mul_comm a 2
  apply Nat.add_left_inj.mpr
  assumption
```

Takie założenie działa jak każde inne twierdzenie, możemy je więc udowodnić w trybie termowym lub taktycznym. Jeśli udowadniamy je w trybie termowym, to oczywiście nie musimy nawet formułować jego pełnej treści. Możemy po `have` nie dać żadnej nazwy, wtedy Lean automatycznie nazwie to założenie `this`.

2.4. Rachunek kwantyfikatorów

Kwantyfikator uniwersalny Z kwantyfikatora $\forall$ korzystaliśmy już wielokrotnie, chociaż niebezpośrednio. Na przykład twierdzenie $\forall x \in \mathbb{N}(x + 0 = x)$ zapisywaliśmy jako

```
theorem add_zero (x : Nat) : x + 0 = x := sorry
```

Możemy je również zapisać i udowodnić jako

```
theorem add_zero' : ∀ x : Nat, x + 0 = x := by
  intro x
  exact add_zero x
```

korzystając z poznanej już taktyki `intro`. Przeciwieństwem `intro` w tym kontekście jest taktyka `revert`, która przenosi podaną zmienną z założeń do kwantyfikatora $\forall$ w celu.

Kwantyfikator egzystencjalny Formuły z kwantyfikatorem $\exists$ najlepiej dowodzić za pomocą dostępnej w `Mathlib.Tactic.Use` taktyki `use`.

```
example : ∃ n : Nat, n > 1000 := by
  use 1001
  decide
```

Problem 2.5. Dowiedzieć się, jak działa kwantyfikator $\exists!$ w Leanie (zdefiniowany w pliku `Mathlib.Logic.ExistsUnique`).

Problem 2.6. Zrobić drugi zestaw zadań:

```
import Mathlib.Logic.ExistsUnique

-- Zadanie 1
example {x y : Nat} (h₁ : x = y + 2) (h₂ : y = 3) : x < 6 := sorry

-- Zadanie 2
example {k : Nat} (h : 15 = k * 3) : 10 ≤ 3 * k := sorry

-- Zadanie 3
example {x y : Nat} (hx : 0 < x) (hy : 0 < y) : 0 < x * y := sorry
```

```
-- Zadanie 4
example : ∃! n : Nat, n ^ 2 = 9 := sorry

-- Zadanie 5
example {x y : Nat} (h_eq : 3 * x = 9) (h_lt : 2 * y < 8) : y * 2 * x * 3 ≤ 100 := sorry

-- Zadanie 6
example (m n : Nat) (h₁ : m ≥ n) (h₂ : n ≥ m) : n = m := sorry

-- Zadanie 7
axiom one_eq_two : 1 = 2

example : 2 = 3 := sorry
```

2.5. Taktyki automatyzujące dowodzenie

Lean posiada wiele taktyk automatyzujących dowodzenie, z których najważniejsze omówimy w tej sekcji.

ring Udowodnijmy, że dla każdych $a, b \in \mathbb{N}$ zachodzi $(a+b)^2 = a^2 + 2ab + b^2$.

```
example {a b : Nat} : (a + b)^2 = a^2 + 2 * a * b + b^2 := by
  rw [Nat.pow_two]
  rw [Nat.mul_add]
  rw [Nat.add_mul]
  rw [Nat.add_mul]
  rw [←Nat.pow_two]
  rw [←Nat.pow_two]
  rw [←Nat.add_assoc]
  rw [Nat.mul_comm]
  rw [Nat.add_assoc (a^2)]
  rw [←Nat.two_mul]
  rw [Nat.mul_assoc]
```

Aby udowadniać podobne tożsamości w pierścieniach przemiennych oraz półpierścieniach przemiennych (na przykład w $\mathbb{N}$), możemy użyć taktyki `ring`.

```
import Mathlib.Tactic.Ring

example {a b : Nat} : (a + b)^2 = a^2 + 2 * a * b + b^2 := by
  ring
```

Jeśli nie chcemy od razu udowadniać równości, a jedynie uprościć cel (np. nierówność), zamiast `ring` możemy użyć taktyki `ring_nf` (od ang. *normal form*).

simp Lean posiada taktykę `simp`, która upraszcza cel (lub założenie) za pomocą zbioru predefiniowanych reguł. Na przykład

```
example {n : Nat} : n ∣ 1 ↔ n = 1 := by
  exact Nat.dvd_one
```

możemy udowodnić też jako

```
example {n : Nat} : n ∣ 1 ↔ n = 1 := by
  simp
```

ponieważ twierdzenie `Nat.dvd_one` jest poprzedzone atrybutem `@[simp]` przy swojej definicji, a więc tym samym dodanym do zbioru reguł używanych przez `simp`.

```
@[simp] theorem Nat.dvd_one {n : Nat} : n ∣ 1 ↔ n = 1
```

Twierdzenia otagowane przez `@[simp]` powinny być w postaci $A = B$ lub $A \Leftrightarrow B$, wtedy Lean zastąpi każde wystąpienie A przez B (nie odwrotnie). Zarówno sam Lean, jak i Mathlib posiadają wiele takich twierdzeń. Możemy również sami je dodawać; należy jednak przy tym uważać – takie twierdzenie powinno *zawsze* upraszczać formułę, prowadząc ją do pewnego rodzaju postaci normalnej.

Zamiast prostego `simp` możemy użyć dowolnego z jego wariantów:

- `simp [a, b]`, który używa lematów oznaczonych przez `@[simp]` oraz dodatkowo podanych `a` i `b`,
- `simp [←a]`, który używa lematów oznaczonych przez `@[simp]` oraz `a` w odwrotną stronę,
- `simp [-a]`, który używa lematów oznaczonych przez `@[simp]` z wyjątkiem `a`,
- `simp [*]`, który używa lematów oznaczonych przez `@[simp]` oraz wszystkich dostępnych w lokalnym kontekście,
- `simp at h`, który upraszcza założenie `h`,
- `simp only [a, b]`, który używa tylko podanych lematów `a` i `b`.

> **Uwaga 2.7**
>
> Ponieważ z każdą aktualizacją Leana i Mathliba zbiór reguł używanych przez `simp` może się zmieniać, nie powinno się używać `simp` w środku dowodu. Można używać go na końcu, ponieważ zakładamy, że `simp` nie zacznie działać *gorzej*.
>
> W środku dowodu można użyć `simp only [...]`. Jeśli nie wiemy, jakich dokładnie twierdzeń chcemy użyć, możemy użyć `simp?`, która podpowie nam, jakie reguły zostały użyte do uproszczenia formuły.

omega Taktyka `omega` używa *testu omega* W. Pugha. Jest użyteczna do udowadniania nierówności liniowych z liczbami całkowitymi i naturalnymi.

```
example {x y : Nat} : x ≤ y ∨ y ≤ x := by
  omega

example {x : Int} (h1 : 0 < x) (h2 : x < 1) : False := by
  omega
```

linarith Taktyka `linarith` również służy do udowadniania nierówności liniowych, ale raczej na liczbach rzeczywistych i wymiernych. Na liczbach naturalnych i całkowitych również działa, ale jest mniej skuteczna niż `omega`.

```
import Mathlib.Data.Real.Basic
import Mathlib.Tactic.Linarith

example {x y : ℝ} (h : x < y) : x - 1 < y := by
  linarith

example {x y : ℝ} (h₁ : x < 4) (h₂ : y < 4) : x + y ≤ 10 := by
  linarith
```

nlinarith Taktyka `nlinarith` działa tak jak `linarith`, ale potrafi też rozwiązać niektóre nieliniowe problemy arytmetyczne.

```
import Mathlib.Data.Real.Basic
import Mathlib.Tactic.Linarith

example {x : ℝ} (h₁ : x < 4) (h₂ : x > 0) : x ^ 2 < 16 := by
  nlinarith
```

3. Studia przypadków

W ten sekcji zajmiemy wykazaniem kilku ciekawych faktów matematycznych. Każdy z tych przykładów będzie okazją do zaznajomienia się z pewnym fragmentem Mathliba oraz poznania nowych konstrukcji składniowych i taktyk Leana.

3.1. Zbiór z relacją równoważności

Cele (czego się nauczymy?)
Struktury, klasy typów, instancje.

Definicja 3.1 (relacja równoważności)**.** Relację $\sim$ nazywamy relacją równoważności, jeśli spełnia następujące warunki dla dowolnych $a, b, c \in A$:

- $a \sim a$ (zwrotność),
- jeśli $a \sim b$, to $b \sim a$ (symetryczność),
- jeśli $a \sim b$ oraz $b \sim c$, to $a \sim c$ (przechodniość).

W języku angielskim zbiór z relacją równoważności nazywa się *setoid* i tej właśnie nazwy będziemy używać. Sformalizujemy w Leanie poniższe twierdzenie.

Twierdzenie 3.2
Setoidem jest para $(\mathbb{Z}^2, \approx)$, gdzie $(x_1, y_1) \approx (x_2, y_2)$ wtedy i tylko wtedy, gdy

$$x_1 + y_1 = x_2 + y_2.$$

Zdefiniujemy teraz swoją pierwszą strukturę w Leanie.

```
structure Point where
  x : Int
  y : Int
```

Elementy struktury tworzymy za pomocą jednego z trzech sposobów:

```
def p := (⟨0, 1⟩ : Point)   -- lub def p : Point := ...
def q := Point.mk 1 0       -- lub def q : Point := ...
def r : Point := { x := 1, y := 1 }
```

W Leanie istnieją *klasy typów* (ang. *type classes*), które są trochę jak interfejsy w Javie czy C#, ale z pewnymi różnicami. Między innymi: można dodawać instancję klasy typów do istniejącego typu bez jego modyfikacji, istnieją parametryzowane klasy typów, klasy

typów mogą dziedziczyć po innych klasach typów, klasy typów mogą być instancjonowane automatycznie lub w wyniku dowodzenia.

Udowodnienie, że para (`Point`, $\approx$) jest setoidem będzie polegało właśnie na zdefiniowaniu *instancji* klasy typów `Setoid` dla typu `Point`.

```
instance : Setoid Point where
  r a b := a.x + a.y = b.x + b.y
  iseqv := by
    constructor
    · -- refl (zwrotność, reflexivity)
      intro p
      rfl
    · -- symm (symetryczność, symmetry)
      intro p q
      intro h
      rw [h]
    · -- trans (przechodniość, transitivity)
      intro p q r
      intro h₁ h₂
      rw [h₁, h₂]
```

Pola `r` oraz `iseqv` odpowiadają odpowiednio relacji równoważności oraz dowodowi, że jest to relacja równoważności. Alternatywny, krótszy zapis dowodu pokazano poniżej.

```
instance : Setoid Point where
  r a b := a.x + a.y = b.x + b.y
  iseqv := {
    refl := by intro p; rfl
    symm := by intro p q h; rw [h]
    trans := by intro p q r h₁ h₂; rw [h₁, h₂]
  }
```

Teraz możemy używać relacji $\approx$ na elementach typu `Point` i korzystać z jej własności.

```
example : p ≈ q := by
  rfl
```

3.2. Grupa abelowa

> **Cele** (czego się nauczymy?)
> Typy zależne, koercja typów, inferencja klasy typów, taktyki `change`, `unfold`, `field_simp`, `infer_instance`.

Definicja 3.3 (magma)**.** Niech R będzie zbiorem niepustym oraz $\cdot : R \times R \to R$ działaniem wewnętrznym na R. Parę $(R, \cdot)$ nazywamy wtedy magmą lub grupoidem.

Definicja 3.4 (półgrupa)**.** Magma $(R, \cdot)$ jest półgrupą, jeśli działanie $\cdot$ jest łączne, tzn. dla dowolnych $a, b, c \in R$ zachodzi

$$(a \cdot b) \cdot c = a \cdot (b \cdot c).$$

Definicja 3.5 (monoid)**.** Półgrupa $(R, \cdot)$ jest monoidem, jeśli istnieje element neutralny $e \in R$, tzn. dla dowolnego $a \in R$ zachodzi

$$e \cdot a = a \cdot e = a.$$

Definicja 3.6 (grupa)**.** Monoid $(R, \cdot)$ jest grupą, jeśli dla dowolnego $a \in R$ istnieje element odwrotny $a^{-1} \in R$, tzn. zachodzi

$$a \cdot a^{-1} = a^{-1} \cdot a = e.$$

Definicja 3.7 (grupa abelowa)**.** Grupa $(R, \cdot)$ jest grupą abelową, jeśli działanie $\cdot$ jest przemienne, tzn. dla dowolnych $a, b \in R$ zachodzi

$$a \cdot b = b \cdot a.$$

Warunek przemienności możemy wprowadzić na dowolnym poziomie, tworząc magmy, półgrupy czy monoidy przemienne (abelowe).

Będziemy chcieli udowodnić, że pewna przykładowa struktura jest grupą abelową.

Twierdzenie 3.8

Grupą abelową jest para $(A, \star)$, gdzie $A = \mathbb{Q} \setminus \{1\}$, a działanie $\star$ jest zdefiniowane jako

$$a \star b = ab - a - b + 2.$$

Czytelnik powinien w tym momencie udowodnić to twierdzenie na kartce.

Zacznijmy od zdefiniowania działania $\star$. Najłatwiej będzie to zrobić w ten sposób, że zdefiniujemy specjalny typ A, a działanie $\star$ jako mnożenie na tym typie.

```
import Mathlib.Data.Rat.Init

def A : Type := {x : ℚ // x ≠ 1}
```

Po raz pierwszy spotykamy się z *podtypami* (ang. *subtypes*), które pozwalają nam definiować typy jako inne typy z dodatkowym warunkiem. Aby skonstruować element podtypu, musimy podać element typu bazowego oraz dowód spełnienia warunku. Na przykład, aby skonstruować element $0 : A$, napiszemy:

```
#check (⟨0, by decide⟩ : A)
```

Element podtypu jest automatycznie wyposażony w wiele przydatnych pól, w tym wartość i twierdzenie o spełnianiu warunku. Warto zauważyć, że $a : \mathbb{Q} = b : \mathbb{Q} \iff a : A = b : A$. Takie twierdzenie też jest oczywiście dostępne.

```
def a0 := (⟨0, by decide⟩ : A)
#check a0.val     -- ↑a0 : ℚ
#check a0.prop    -- ↑a0 ≠ 1
#check a0.coe_inj -- ↑a = ↑b ↔ a = b
```

Strzałka w górę oznacza tutaj koercję (rzutowanie) typu z podtypu na typ bazowy.

Zdefiniujmy działanie $\star$, póki co na liczbach wymiernych.

```
def op (a b : ℚ) : ℚ := a * b - a - b + 2
```

Teraz możemy pokazać, że $(A, \star)$ jest magmą, tzn. że $a \star b \in A$ dla dowolnych $a, b \in A$.

```
lemma op_ne_one {a b : ℚ} (ha : a ≠ 1) (hb : b ≠ 1) : op a b ≠ 1 := by
  intro h
  unfold op at h
  have : (a - 1) * (b - 1) = 0 := by linarith
  rcases mul_eq_zero.mp this with ha' | hb'
  · have : a = 1 := by linarith
    contradiction
  · have : b = 1 := by linarith
    contradiction
```

Użyliśmy tutaj taktyki `unfold`, która *rozwija* definicję `op` w danym miejscu.

Mając powyższy lemat, możemy już stwierdzić, że $(A, \star)$ rzeczywiście jest magmą. W Leanie magma multiplikatywna jest nazwana po prostu `Mul`, więc wystarczy zdeklarować instancję tej klasy typów dla naszego podtypu.

```
instance : Mul A where
  mul a b := ⟨op a.val b.val, op_ne_one a.prop b.prop⟩
```

Teraz chcemy udowodnić, że $(A, \star)$ jest nie tylko magmą, ale i podgrupą, czyli że $\star$ jest łączne. Jest to nawet prostsze niż na karce.

```
import Mathlib.Tactic.Ring

instance : Semigroup A where
  mul_assoc a b c := by
    apply Subtype.ext
    change op (op a.val b.val) c.val = op a.val (op b.val c.val)
    unfold op
    ring
```

Musieliśmy użyć tutaj twierdzenia

```
Subtype.ext {p : α → Prop} {a1 a2 : { x // p x }} : ↑a1 = ↑a2 → a1 = a2
```

oraz nowej taktyki, `change`, która pozwala nam zmienić cel na równoważny *z definicji*.

Teraz pokażemy, że $(A, \star)$ jest monoidem, czyli ma element neutralny. W tym celu należy powiedzieć, że typ A ma jedynkę (równą 2), a następnie, że ta jedynka istotnie jest elementem neutralnym, tzn. $2 \star a = a \star 2 = a$ dla każdego $a \in A$.

```
instance : One A where
  one := ⟨2, by decide⟩

instance : Monoid A where
  one_mul a := by
    apply Subtype.ext
    change op 2 a.val = a.val
    unfold op
    ring
  mul_one a := by
    apply Subtype.ext
    change op a.val 2 = a.val
    unfold op
    ring
```

Aby pokazać, że $(A, \star)$ jest grupą, trzeba najpierw udowodnić, że $a^{-1} = \frac{a}{a-1} \in A$, wskazać wzór na odwrotność, a na końcu pokazać, że istotnie spełnia on $a^{-1}a = 2$.

```
import Mathlib.Tactic.FieldSimp
import Mathlib.Tactic.Linarith

lemma op_inv_ne_one {a : ℚ} (ha : a ≠ 1) : a / (a - 1) ≠ 1 := by
  intro h
  have a_sub_one_ne_zero : a - 1 ≠ 0 := sub_ne_zero.mpr ha
  have := (div_eq_one_iff_eq a_sub_one_ne_zero).mp h
  linarith

instance : Inv A where
  inv a := ⟨a.val / (a.val - 1), op_inv_ne_one a.prop⟩

instance : Group A where
  inv_mul_cancel a := by
    apply Subtype.ext
    change op (a.val / (a.val - 1)) a.val = 2
    unfold op
    field_simp
    have : (a.val - 1) / (a.val - 1) = 1 := by
      have a_sub_one_ne_zero : a.val - 1 ≠ 0 := sub_ne_zero.mpr a.prop
      exact div_self a_sub_one_ne_zero
    rw [this]
    ring
```

Użyliśmy tutaj taktyki `field_simp`, która próbuje sprowadzić wyrażenia do wspólnego mianownika.

Na koniec zostało pokazać przemienność $\star$.

```
instance : CommGroup A where
  mul_comm a b := by
    apply Subtype.ext
    change op a.val b.val = op b.val a.val
    unfold op
    ring
```

Z tych wszystkich własności możemy wyprowadzić inne fakty, np. że $(A, \star)$ jest przemiennym monoidem z własnością skracania ($ac = bc \implies a = b$). Jest to oczywiste, ale o tyle ciekawe, że nie każdy monoid przemienny ma własność skracania (np. $(\mathbb{N}, \cdot)$) i nie każdy monoid przemienny z własnością skracania jest grupą (np. $(\mathbb{N}, +)$). W Leanie takie instancje można *inferować* przy pomocy `infer_instance`.

```
instance : CancelCommMonoid A := by
  infer_instance
```

Wbrew pozorom, nie będziemy z tego często korzystać, bo Lean automatycznie próbuje inferować instancje w momencie, kiedy jest to konieczne (np. przy aplikacji do twierdzenia).

Warto wiedzieć, że zamiast budować dowód krok po kroku, można od razu stworzyć dużą deklarację `instance : CommGroup A`, w której należy udowodnić każdą konieczną własność.

Problem 3.9. Zrobić trzeci zestaw zadań:

```
import Mathlib.Data.Rat.Init
import Mathlib.Data.Real.Basic
```

```
-- Zadanie 1
example {x : ℝ} (hx : 0 < x) : 2 ≤ x + 1 / x := sorry

-- Zadanie 2
-- Udowodnić, że (A, op) jest grupą przemienną.
def A := {x : ℚ // -1 < x ∧ x < 1}
def op (a b : ℚ) := (a + b) / (1 + a * b)

instance : CommGroup A := sorry
```

3.3. Równanie funkcyjne

Cele (czego się nauczymy?)
Słowo kluczowe **obtain**, taktyki `specialize`, `funext`, `refine`, `simpa`.

Zajmiemy się teraz pewnym zadaniem, które pojawiło się na shortliście Międzynarodowej Olimpiady Matematycznej w 2013 roku (zadanie N1). Oryginalnie należało znaleźć wszystkie rozwiązania równania funkcyjnego, ale my ograniczymy się do pokazania, że podane rozwiązanie jest jedynym możliwym.

Twierdzenie 3.10
Funkcja $f(x) = x$ jest jedyną funkcją $f : \mathbb{N}_+ \to \mathbb{N}_+$ spełniającą

$$m^2 + f(n) \mid mf(m) + n$$

dla wszystkich $m, n \in \mathbb{N}_+$.

Jako że dowód może być trudniejszy, niż w poprzednich przykładach, najpierw przedstawimy go w języku naturalnym.

Dowód. Łatwo sprawdzić, że $f(x) = x$ spełnia warunek zadania. Pokażemy teraz, że jest to jedyne rozwiązanie.

Podstawmy $m = n = 2$. Otrzymujemy

$$4 + f(2) \mid 2f(2) + 2.$$

Z tego wynika, że istnieje $k \in \mathbb{N}_+$ takie, że

$$\begin{aligned} k(4 + f(2)) &= 2f(2) + 2 \\ 4k + kf(2) &= 2f(2) + 2 \\ (k-2)f(2) &= 2 - 4k \\ f(2) &= \frac{2-4k}{k-2}, \end{aligned}$$

z czego wynika $k = 1$, więc dostajemy $f(2) = 2$.

Teraz podstawmy $m = 2$. Otrzymujemy

$$\begin{aligned} 4 + f(n) &\mid 2f(2) + n \\ 4 + f(n) &\mid 4 + n, \end{aligned}$$

więc $f(n) \leqslant n$.

Na koniec podstawmy $m = n$. Otrzymujemy

$$\begin{aligned} n^2 + f(n) \mid nf(n) + n \\ n^2 + f(n) \leqslant nf(n) + n \\ n(n-1) \leqslant f(n)(n-1). \end{aligned}$$

Jeśli $n = 1$, to oczywiście $f(1) = 1$. Jeśli $n \neq 1$, to możemy podzielić przez $n - 1$ i otrzymujemy $n \leqslant f(n)$. Ostatecznie mamy $f(n) = n$ dla każdego $n \in \mathbb{N}_+$. □

Sformalizowane twierdzenie wygląda następująco.

```
import Mathlib.Data.PNat.Basic

example {f : ℕ+ → ℕ+} : (∀ m n : ℕ+, m * m + f n ∣ m * f m + n) ↔ f = id := sorry
```

Funkcja `id` to funkcja identycznościowa, to znaczy $\mathrm{id}(x) = x$ dla każdego x.

Chcemy pokazać równoważność, więc musimy udowodnić oba kierunki. Możemy więc użyć taktyki `constructor`, która tworzy dwa cele do udowodnienia (z czego jeden będzie bardzo prosty).

```
example {f : ℕ+ → ℕ+} : (∀ m n : ℕ+, m * m + f n ∣ m * f m + n) ↔ f = id := by
  constructor
  · intro h
    sorry -- dowód trudniejszego kierunku
  · intro h
    simp [h]
```

Zacznijmy od pokazania, że $f(2) = 2$. Aby to zrobić, należy podstawić $m = n = 2$, które można oczywiście zrealizować za pomocą `have` oraz aplikacji, ale my użyjemy taktyki `specialize`.

```
have f2_eq_2 : f 2 = 2 := by
  -- h : ∀ (m n : ℕ+), m * m + f n ∣ m * f m + n
  specialize h 2 2
  -- h : 2 * 2 + f 2 ∣ 2 * f 2 + 2
  sorry
```

Teraz musimy skorzystać z faktu, że $a \mid b$ wtedy i tylko wtedy, gdy istnieje c takie, że $b = ca$. W Leanie takie twierdzenie jest dostępne jako `exists_eq_mul_left_of_dvd`. Czytelnik powinien zwrócić uwagę na jego dokładne sformułowanie – korzystamy z faktu, że $(\mathbb{N}_+, \cdot)$ jest półgrupą przemienną.

```
have f2_eq_2 : f 2 = 2 := by
  specialize h 2 2
  have := exists_eq_mul_left_of_dvd h
  -- this : ∃ c, 2 * f 2 + 2 = c * (2 * 2 + f 2)
  sorry
```

Z założeniem, które zawiera kwantyfikator egzystencjalny, dosyć niewygodnie jest pracować. Wolelibyśmy mieć bezpośrednio zmienną i założenie o niej. Oby to osiągnąć, wystarczy zamienić `have` na `obtain`.

```
have f2_eq_2 : f 2 = 2 := by
  specialize h 2 2
  obtain ⟨k, hk⟩ := exists_eq_mul_left_of_dvd h
  -- k : ℕ+
  -- hk : 2 * f 2 + 2 = k * (2 * 2 + f 2)
  sorry
```

Teraz możemy już pokazać, że $k = 1$. Z takim celem powinna poradzić sobie już taktyka `nlinarith`, ale, jak Czytelnik raczy sprawdzić, sobie nie radzi[2]. Problemem jest to, że pracujemy na podtypie typu $\mathbb{N}$, a nie na samym $\mathbb{N}$. Zmieńmy to za pomocą `PNat.dvd_iff` oraz `PNat.eq`, kończąc tę część dowodu.

```
import Mathlib.Data.PNat.Basic
import Mathlib.Tactic.Linarith

example {f : ℕ+ → ℕ+} : (∀ m n : ℕ+, m * m + f n ∣ m * f m + n) ↔ f = id := by
  constructor
  · intro h
    have f2_eq_2 : f 2 = 2 := by
      specialize h 2 2
      obtain ⟨k, hk⟩ := exists_eq_mul_left_of_dvd (PNat.dvd_iff.mp h)
      have k_eq_1 : k = 1 := by
        nlinarith
      apply PNat.eq
      nlinarith
    sorry -- ⊢ f = id
  · intro h
    simp [h]
```

Aktualny cel to $f = \mathrm{id}$. Wolelibyśmy pracować z celem postaci $f(x) = x$ dla każdego x. Aby wykonać takie przejście, możemy użyć taktyki `funext`, która zamienia równość funkcji na równość wartości funkcji dla dowolnego argumentu oraz twierdzenia `id_eq`, które upraszcza $\mathrm{id}(x)$ do x.

```
have f2_eq_2 : f 2 = 2 := sorry -- ...
funext x
-- x : ℕ+
-- ⊢ f x = id x
rw [id_eq]
-- x : ℕ+
-- ⊢ f x = x
sorry
```

Teraz moglibyśmy kolejno udowodnić, że $f(x) \leqslant x$ oraz $f(x) \geqslant x$, następnie użyć `le_antisymm` ($f(x) \leqslant x \wedge f(x) \geqslant x \implies f(x) = x$), ale możemy od razu zasygnalizować do czego dążymy za pomocą `refine`, a Lean sam wygeneruje cele do udowodnienia.

```
intro h
have f2_eq_2 : f 2 = 2 := sorry -- ...
funext x
rw [id_eq]
refine le_antisymm ?_ ?_
· -- ⊢ f x ≤ x
  sorry
```

[2]Przynajmniej w wersji Lean v26.0.0, w przyszłości może się to zmienić.

```
· -- ⊢ x ≤ f x
  sorry
```

Reszta dowodu jest już raczej prosta – Czytelnik powinien spróbować go ukończyć samodzielnie (korzystając z twierdzenia `PNat.le_of_dvd` i taktyki `change` kiedy będzie to potrzebne). Cały dowód znajduje się poniżej.

```
import Mathlib.Data.PNat.Basic
import Mathlib.Tactic.Linarith

example {f : ℕ+ → ℕ+} : (∀ m n : ℕ+, m * m + f n ∣ m * f m + n) ↔ f = id := by
  constructor
  · intro h
    have f2_eq_2 : f 2 = 2 := by
      specialize h 2 2
      obtain ⟨k, hk⟩ := exists_eq_mul_left_of_dvd (PNat.dvd_iff.mp h)
      have k_eq_1 : k = 1 := by
        nlinarith
      apply PNat.eq
      nlinarith
    funext x
    rw [id_eq]
    refine le_antisymm ?_ ?_
    · specialize h 2 x
      rw [f2_eq_2] at h
      have := PNat.le_of_dvd h
      simpa
    · specialize h x x
      have := PNat.le_of_dvd h
      change (x * x + f x : ℕ) ≤ x * f x + x at this
      change (x : ℕ) ≤ f x
      nlinarith [PNat.pos x, PNat.pos (f x)]
  · intro h
    simp [h]
```

Taktyka `simpa` to połączenie `simp` oraz `assumption`, które domyślnie działa z hipotezą `this`. Zamiast `simpa` moglibyśmy więc napisać `simpa using this` albo `simp at this` i `assumption`.

3.4. Kongruencja modulo

> **Cele** (czego się nauczymy?)
> Typy induktywne, taktyki `induction` i `gcongr`, środowisko `calc`.

W tej sekcji będziemy chcieli udowodnić proste twierdzenie dotyczące kongruencji modulo 9. Zanim jednak do tego przejdziemy, musimy się dowiedzieć, jak w Leanie są zdefiniowane liczby naturalne.

Typ induktywny to typ definiowany przez podanie jego konstruktorów. Dotychczas stworzyliśmy jeden typ, `Point`, który miał dokładnie jeden konstruktor, domyślnie nazwany `Point.mk`. Korzystaliśmy już jednak z typów induktywnych z wieloma konstruktorami, np. `Or` z dwoma konstruktorami: `Or.inl` oraz `Or.inr`. Również `Nat` jest typem induktywnym, który, zgodnie z aksjomatyką Peano, ma dwa konstruktory: `Nat.zero` (liczba 0) oraz `Nat.succ` (następnik liczby naturalnej). Każdą liczbę naturalną można więc zbudować, zaczynając od zera i wielokrotnie stosując następnika. Poniżej znajduje się definicja liczb naturalnych w Leanie.

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Uwaga

Składnia **`structure`** (której użyliśmy do zdefiniowania `Point`) jest w rzeczywistości skróconą formą składni **`inductive`**. Oprócz uproszczenia zapisu, struktury automatycznie generują pola dostępu (tzn. możemy użyć `point.x`). Zamiast

```
structure Point where
  x : Int
  y : Int
```

można więc napisać

```
inductive Point where
  | mk (x : Int) (y : Int)

def Point.x (p : Point) : Int :=
  match p with
  | Point.mk x _y => x

def Point.y (p : Point) : Int :=
  match p with
  | Point.mk _x y => y
```

Wracając do konguencji, będziemy chcieli udowodnić poniższe twierdzenie przez indukcję względem n.

Twierdzenie 3.11

Dla każdej liczby naturalnej n zachodzi

$$4^n \equiv 1 + 3n \pmod 9.$$

Dowód. Dla $n = 0$ mamy $4^0 = 1$ oraz $1 + 3 \cdot 0 = 1$, więc teza zachodzi. Załóżmy więc, że teza zachodzi dla pewnego n i udowodnijmy ją dla $n + 1$ (to znaczy dla następnika n). Mamy

$$4^{n+1} \equiv 1 + 3(n + 1) \pmod 9.$$

Wykonajmy ciąg przekształceń:

$$\begin{aligned} 4^{n+1} &= 4 \cdot 4^n \\ &\equiv 4(1 + 3n) \pmod 9 \\ &= 4 + 12n \\ &= 4 + 3n + 9n \\ &\equiv 4 + 3n + 0n \pmod 9 \\ &= 1 + 3(n + 1). \end{aligned}$$

□

Teraz sformalizujemy ten dowód w Leanie. Ciąg przekształceń został tak przygotowany, żeby każdy krok był prosty do uzasadnienia.

```
import Mathlib.Data.Nat.ModEq

example {n : ℕ} : 4^n ≡ 1 + 3*n [MOD 9] := sorry
```

Aby przeprowadzić dowód przez indukcję, użyjemy taktyki `induction`.

```
example {n : ℕ} : 4^n ≡ 1 + 3*n [MOD 9] := by
  induction n with
  | zero => rfl
  | succ n ih => sorry
```

W tym przypadku `ih` to nasza hipoteza indukcyjna (ang. *induction hypothesis*). Teraz możemy przeprowadzić ciąg przekształceń za pomocą środowiska **`calc`**, które pozwala na pisanie łańcuchów relacji równoważności (takich jak równość czy kongruencja modulo), gdzie każdy krok jest uzasadniony dowodem.

```
example {n : ℕ} : 4^n ≡ 1 + 3*n [MOD 9] := by
  induction n with
  | zero => rfl
  | succ n ih =>
    calc
      -- ⊢ 4 ^ (n + 1) ≡ 1 + 3 * (n + 1) [MOD 9]
      _ = 4 * 4^n := sorry
      -- ...
      _ ≡ 1 + 3*(n + 1) [MOD 9] := sorry
```

Czytelnik powinien już być w stanie wypełnić luki samodzielnie, korzystając z twierdzeń dostępnych w `Mathlib.Data.Nat.ModEq` oraz taktyki `ring`. Cały dowód wygląda następująco.

```
import Mathlib.Data.Nat.ModEq
import Mathlib.Tactic.Ring

example {n : ℕ} : 4^n ≡ 1 + 3*n [MOD 9] := by
  induction n with
  | zero => rfl
  | succ n ih =>
    calc
      _ = 4 * 4^n := by ring
      _ ≡ 4 * (1 + 3*n) [MOD 9] := by
        apply Nat.ModEq.mul_left
        exact ih
      _ = 4 + 12*n := by ring
      _ = 4 + 3*n + 9*n := by ring
      _ ≡ 4 + 3*n + 0*n [MOD 9] := by
        apply Nat.ModEq.add_left
        apply Nat.ModEq.mul_right
        rfl
      _ = 1 + 3*(n + 1) := by ring
```

Sformułowanie ostatniego kroku można uprościć, pisząc `_ = _ := by ring`, ponieważ ostateczny wynik jest naszym celem, więc Lean go zna. Poza tym, w niektórych krokach możemy użyć taktyki `gcongr`, która automatycznie stosuje pewne twierdzenia dotyczące kongruencji. Ostatecznie otrzymujemy:

```
import Mathlib.Data.Nat.ModEq
import Mathlib.Tactic.Ring

example {n : ℕ} : 4^n ≡ 1 + 3*n [MOD 9] := by
  induction n with
  | zero => rfl
  | succ n ih =>
    calc
      _ = 4 * 4^n := by ring
      _ ≡ 4 * (1 + 3*n) [MOD 9] := by gcongr
      _ = 4 + 12*n := by ring
      _ ≡ 4 + 3*n [MOD 9] := by gcongr; rfl
      _ = _ := by ring
```

Problem 3.12. Zrobić czwarty zestaw zadań:

```
import Mathlib.Algebra.Ring.Defs
import Mathlib.Algebra.Field.IsField
import Mathlib.Data.Finite.Defs

-- Zadanie 1
-- Udowodnić, że każdy skończony pierścień całkowity jest ciałem.
theorem finite_integral_domain_is_field (R : Type*) [CommRing R] [IsDomain R] [Finite R] :
    IsField R := sorry

-- Zadanie 2
-- Podpowiedź: użyj Nat.le_induction
example {n : ℕ} (hn : 2 ≤ n) : 3^n > n * 2^n := sorry
```

Wstęp teoretyczny do zadania 1 znajduje się w dodatku A.1.

4. Literatura

[1] J. Avigad, P. Massot. Mathematics in Lean, https://leanprover-community.github.io/mathematics_in_lean/.

[2] J. Avigad, L. de Moura, S. Kong, S. Ullrich. Theorem Proving in Lean 4, https://leanprover.github.io/theorem_proving_in_lean4/.

[3] H. Barendregt, W. Dekkers, R. Statman. Lambda Calculus with Types, *Perspectives in Logic*, Cambridge University Press, 2013.

[4] L. Csirmaz, Z. Gyenis. Mathematical Logic: Exercises and Solutions, *Problem Books in Mathematics*, Springer, 2022.

[5] E. Palmgren. Semantics of intuitionistic propositional logic: lecture notes for Applied Logic, 2009, https://www2.math.uu.se/~palmgren/tillog/heyting3.pdf.

[6] M. H. Sørensen, P. Urzyczyn. Lectures on the Curry-Howard Isomorphism, *Studies in Logic and the Foundations of Mathematics*, vol. 149, Elsevier, 2006.

[7] P. Urzyczyn. Rachunek lambda: materiały do wykładu, 2025, https://www.mimuw.edu.pl/~urzy/Lambda/erlambda.pdf.

[8] R. Zach et. al, The Open Logic Text, https://openlogicproject.org/.

A. Dodatek

A.1. Pierścienie całkowite i ciała

W tej sekcji przedstawimy bardzo krótki wstęp teoretyczny dotyczący pierścieni całkowitych i ciał. Dowód twierdzenia mówiącego o tym, że każdy skończony pierścień całkowity jest ciałem, jest przedstawiony na końcu.

Definicja A.1 (pierścień)**.** Niech R będzie zbiorem niepustym oraz $+$ i $\cdot$ działaniami wewnętrznymi na R. Trójkę $(R, +, \cdot)$ nazwiemy pierścieniem, jeśli spełnione są następujące warunki:

- $(R, +)$ jest grupą abelową,
- $(R, \cdot)$ jest półgrupą,
- dla dowolnych $a, b, c \in R$ zachodzą prawa rozdzielności:

$$a \cdot (b + c) = a \cdot b + a \cdot c,$$
$$(b + c) \cdot a = b \cdot a + c \cdot a.$$

Definicja A.2 (pierścień całkowity)**.** Pierścień $(R, +, \cdot)$ jest pierścieniem całkowitym, jeśli:

- jest nietrywialny (ma co najmniej dwa różne elementy),
- jest przemienny (działanie $\cdot$ jest przemienne),
- ma jedynkę (istnieje element neutralny dla działania $\cdot$),
- nie ma dzielników zera (jeśli $a, b \in R$ oraz $a \cdot b = 0$, to $a = 0$ lub $b = 0$).

Definicja A.3 (ciało)**.** Pierścień $(R, +, \cdot)$ jest ciałem, jeśli:

- jest nietrywialny (ma co najmniej dwa różne elementy),
- jest przemienny (działanie $\cdot$ jest przemienne),
- ma jedynkę (istnieje element neutralny dla działania $\cdot$),
- każdy niezerowy element posiada element odwrotny.

Fakt A.4. Dla każdego elementu a pierścienia zachodzi $a \cdot 0 = 0$.

Dowód.

$$\begin{aligned} a \cdot 0 &= a \cdot (0 + 0) \\ a \cdot 0 &= a \cdot 0 + a \cdot 0 \\ a \cdot 0 - a \cdot 0 &= a \cdot 0 + a \cdot 0 - a \cdot 0 \\ 0 &= a \cdot 0 + 0 \\ 0 &= a \cdot 0 \end{aligned}$$

□

Twierdzenie A.5

Każdy skończony pierścień całkowity jest ciałem.

Dowód. Niech $(R, +, \cdot)$ będzie skończonym pierścieniem całkowitym. Rozważmy funkcję $f : R \ni x \mapsto ax \in R$ dla dowolnie wybranego, niezerowego $a \in R$. Funkcja ta jest injekcją, ponieważ (skoro nie ma dzielników zera) z $ax_1 = ax_2$ wynika $x_1 = x_2$. Injekcja na zbiorze skończonym jest surjekcją, w szczególności istnieje $b \in R$ takie, że $f(b) = ab = 1$. Element b jest wtedy elementem odwrotnym elementu a, więc $(R, +, \cdot)$ jest ciałem. □

Skończony pierścień całkowity można w Leanie opisać jako taki typ `R`, który spełnia `[CommRing R]`, `[IsDomain R]` i `[Finite R]`. Klasa typów `CommRing` opisuje przemienny pierścień z jedynką[3], a `IsDomain` to własność półpierścienia z jedynką (`Semiring`), która oznacza nietrywialność i brak dzielników zera. Pierścień całkowity to więc po prostu `CommRing` z własnością `IsDomain`, a skończony pierścień całkowity to `CommRing` z własnościami `IsDomain` i `Finite`.

A.2. Dalsza lektura

Jest kilka rzeczy, których w ramach tego kursu nie udało się omówić, a które mogą przydać się przy pierwszej okazji podczas samodzielnej pracy z Leanem. Lista (z pewnością niepełna) takich zagadnień znajduje się poniżej:

- taktyka `nth_rw`,
- taktyka `by_cases`,
- taktyka `congr`,
- taktyka `contrapose!`,
- silna indukcja (`induction n using Nat.strongRecOn`),
- (magiczne) taktyki `grind` i `aesop`,
- taktyki z pytajnikiem (oprócz poznanego `simp?` mamy `exact?`, `apply?`, `easop?`, itp.),
- `structure` ... `deriving` ...,
- komenda `print axioms` (raczej w ramach ciekawostki).

[3] Pierścienie i półpierścienie w Leanie domyślnie mają jedynkę. Istnieją natomiast ich odpowiedniki bez jedynki, nazwane `NonUnitalRing` i `NonUnitalSemiring`.