Andrzej Szałas

Instytut Informatyki
Uniwersytet Warszawski

Logika na co dzień

Codzienna działalność człowieka wymaga ciągłej analizy sytuacji i wyciągania wniosków. Podobnie dzieje się w systemach informatycznych, w szczególności w systemach inteligentnych, samodzielnie wnioskujących i podejmujących decyzje. Podstawową nauką zajmującą się metodami wnioskowania jest logika i jej właśnie poświęcony jest niniejszy rozdział. Omówimy najpierw podstawowe mechanizmy wnioskowania, w tym rezolucję – najpopularniejszą metodę automatycznego wnioskowania. Następnie przejdziemy do regułowych języków programowania, dla których impulsem była metoda rezolucji. Od uzasadniania poprawności rozumowań przejdziemy do wnioskowania na podstawie baz wiedzy. Obok zastosowań w naukach ścisłych, ekonomicznych czy humanistycznych, metody te mają ogromne znaczenie w informatyce, w tym w robotyce, systemach eksperckich i decyzyjnych.

1. Wprowadzenie

Jeśli jestem głodny, staram się znaleźć coś do zjedzenia. Jeśli chcę przejść na drugą stronę ulicy, rozglądam się i – jeśli sytuacja jest bezpieczna – przechodzę. Jeśli chcę wyszukać dane dotyczące logiki w informatyce, wpisuję w okienko wyszukiwarki Google frazę *logika informatyka*. Jeśli chcę policzyć pierwiastek równania **ax+b=c**, wykonuję instrukcję **jeśli** a=/=0 **to** x=(c–b)/a, być może wpisując ją do arkusza kalkulacyjnego jako =JEŻELI(A1<>0;(C1–B1)/a) lub w języku programowania, takim jak C, C++ czy Java, np. jako **if** a!=0 {x=(c–b)/a}. Jeśli mam podwyższoną temperaturę i jestem przeziębiony – biorę odpowiedni lek. Jeśli nie umiem zdiagnozować choroby, albo jest ona zbyt poważna na samodzielną terapię – idę do lekarza. Jeśli prowadzę pojazd, dojeżdżam do skrzyżowania równorzędnego i z prawej strony nadjeżdża inny pojazd, ustępuję mu pierwszeństwa.

Co łączy te przykłady?

Zauważmy, że wszystkie powyższe zdania zaczynają się od słowa „jeśli". Nasza codzienna działalność bardzo często polega na używaniu zdań tego typu. W logice nazywamy je **implikacjami**, a – przy pewnych ograniczeniach – **regułami**. Do reguł przejdziemy nieco później. Teraz przyjrzyjmy się implikacji. Jest ona powszechnie stosowana – wyraża się przy jej użyciu reguły postępowania, przepisy prawne, sposoby dochodzenia do decyzji, diagnozy lekarskie, strategie gier itd. Wnioskowanie jest często oparte na implikacjach – na podstawie przesłanek wyciągamy wnioski, które stają się przesłankami w kolejnych fazach wnioskowania. Dla zilustrowania tej metody przyjrzyjmy się następującemu rozumowaniu:

(a) pojadę dziś do pracy autobusem lub tramwajem,
(b) jeśli pada deszcz, nie wybieram autobusu (przystanek autobusowy jest dalej niż tramwajowy i bardziej zmoknę),
(c) pada deszcz,
(d) wniosek: pojadę do pracy tramwajem.

Skoro pada deszcz, nie wybieram autobusu. Tego wniosku używam w dalszym rozumowaniu: ponieważ pojadę autobusem lub tramwajem, a nie wybieram autobusu, pozostaje jedynie tramwaj. Uzyskany wniosek uznamy więc za poprawny. Ale na jakiej podstawie? Czy to rozumowanie można uzasadnić metodami takimi, jakie przyjmuje się w naukach ścisłych?

Nauka zajmująca się modelowaniem wnioskowania i jego badaniem nazywa się **logiką** (od greckiego słowa *logos*, oznaczającego rozum, słowo, myśl). Z jednej strony analizuje się w niej poprawność wnioskowań, a z drugiej strony – dostarcza metod i algorytmów wnioskowania.

Korzenie logiki sięgają starożytnej Grecji, ale też Chin czy Indii. Odgrywała ona istotną rolę w średniowieczu, burzliwy jej rozwój datuje się od końca XIX wieku. George Boole, Charles Peirce, John Venn, potem Bertrand Russell czy Gottlob Frege są wybitnymi logikami z tego okresu. Informatyczne pojęcie obliczalności jest też ściśle związane z badaniami logicznymi dotyczącymi rozstrzygalności teorii matematycznych, czyli szukania metod algorytmicznych dla automatycznego znajdowania dowodów twierdzeń tych teorii. Wybitnymi przedstawicielami tego kierunku są Richard Dedekind, Giuseppe Peano, David Hilbert, Arend Heyting, Ernst Zermelo, John von Neumann, Gerhard Gentzen, Kurt Gödel czy Alfred Tarski. Pierwsze matematyczne modele maszyn matematycznych zawdzięczamy kontynuacji prac tych logików, prowadzonych przez Emila Posta, Alonzo Churcha, Stephena Kleenego (prekursorzy języków funkcyjnych), jak również Alana Turinga czy Claude'a Shannona (prekursorzy języków imperatywnych).

Jak wspomnieliśmy – implikacja jest podstawą wnioskowań, jest więc ważnym elementem logik. Rozumowania przeprowadzane w matematyce, informatyce, fizyce czy w życiu codziennym opierają się na zdaniach wyrażających interesujące nas prawdy. Zdania te dzielimy na **proste** (**atomowe**) i złożone. Zdania proste wyrażają pewne fakty, jak „jestem głodny", „chcę przejść na drugą stronę ulicy". Natomiast zdania złożone są tworzone z innych zdań właśnie przy pomocy spójników. Implikacja jest jednym z takich spójników: na przykład zdanie „jeśli jestem głodny, to staram się znaleźć coś do zjedzenia" powstaje z dwóch zdań prostych poprzez ich połączenie spójnikiem „jeśli... to...". Implikacja występuje także w arkuszach kalkulacyjnych i w językach programowania jako podstawa instrukcji warunkowych „**if** {}".

Innymi typowymi spójnikami są **negacja** (nie), **koniunkcja** (i), **alternatywa** (lub) oraz **równoważność** (wtedy i tylko wtedy, gdy). Również te spójniki są wszechobecne w języku naturalnym, a więc i w codziennym wnioskowaniu. Występują również w naukach ścisłych. Informatyka, leżąca na skrzyżowaniu nauk ścisłych i codziennych aktywności człowieka, także nie może się bez nich obyć.

2. Modelowanie

W nauce **modelem** nazywamy uproszczony opis rzeczywistości, pozwalający skutecznie wnioskować, a jednocześnie pomijający szczegóły mniej istotne z punktu widzenia prowadzonych rozumowań. Prawa dynamiki Isaaca Newtona opisują podstawowe zasady rządzące ruchem i siłami. Tworzą prosty model, skuteczny dopóki nie mamy do czynienia z prędkościami bliskimi prędkości

światła. Niels Bohr stworzył słynny model atomu, prosty pojęciowo i oddający podstawowe zasady zachowania atomu.

Działalność człowieka – począwszy od tej codziennej po najbardziej zaawansowane badania i konstrukcje – zaczyna się od rozpoznawania odpowiedniego zestawu zjawisk, a następnie ich modelowania po to, by uzyskany i sprawdzony model później wykorzystywać. Gdy poruszamy się po własnym mieszkaniu, posługujemy się jego modelem stworzonym w czasie rozpoznawania tegoż mieszkania. Model ten mamy zwykle w głowie, niemniej jednak w porównaniu z rzeczywistością jest on bardzo uproszczony. Nie bierze pod uwagę przepływu cząstek powietrza, zachowania elektronów w poszczególnych cząstkach występujących w stropie i ścianach, nie dbamy o złożone procesy przepływu wody w rurach wodociągowych itd. Prostota modelu pozwala jednak na skuteczne wnioskowanie, poruszanie się po pomieszczeniach i działanie. W informatyce też dba się o to, by dla danego zjawiska czy problemu obliczeniowego wybrać możliwie jak najprostszy, ale zarazem skuteczny model.

Wnioskowanie zawsze bazuje na jakimś mniej lub bardziej abstrakcyjnym modelu rzeczywistości.

Wyobraźmy sobie zadanie polegające na opracowaniu bardzo prostego robota przemysłowego, który „obserwuje" taśmę produkcyjną i którego zadaniem jest przestawianie z niej przedmiotów zielonych na taśmę znajdującą się po lewej stronie, a czerwonych – na taśmę znajdującą się po prawej stronie. Przedmioty o innych kolorach robot powinien przepuszczać dalej. Tworzymy więc model – trzy taśmy produkcyjne. Nad środkową taśmą czuwa robot wyposażony w:

- czujniki rozpoznające kolor zielony i czerwony,
- chwytaki służące do chwytania przedmiotów i przestawiania ich na lewą lub prawą taśmę.

Mając ten model możemy teraz opisać działanie robota dwoma prostymi regułami:

(a) jeśli obserwowany obiekt jest zielony, to przenieś go na taśmę z lewej strony,
(b) jeśli obserwowany obiekt jest czerwony, to przenieś go na taśmę z prawej strony.

Powyższe reguły nie gwarantują przeniesienia wszystkich przedmiotów zielonych na lewą stronę, a czerwonych na prawą, bo zależy to od prędkości taśm, akcji rozpoznawania koloru i akcji przenoszenia przedmiotów. Przytoczone dwie reguły odzwierciedlają bardzo proste wnioskowanie. W opisanym przypadku niekoniecznie skuteczne, ale za to skuteczne i wystarczające w innym modelu. Mianowicie, jeśli założymy, że robot nie myli się w rozpoznawaniu kolorów i niezawodnie przenosi obiekty oraz że środkowa taśma zatrzymuje się na czas rozpoznawania koloru i przenoszenia przedmiotu przez robota, a na dodatek zatrzymuje każdy przedmiot w zasięgu czujników i chwytaków robota – to takie

proste wnioskowanie będzie skuteczne. Daje jednocześnie wiedzę o warunkach, jakie powinno spełniać otoczenie robota, by ten mógł działać efektywnie wykorzystując swoje możliwości.

Możemy więc dostrzec, że skuteczność wnioskowań zależy od przyjętego modelu rzeczywistości. I znów zauważmy, że omawiane modele biorą pod uwagę jedynie to, co niezbędne dla skutecznego rozwiązania problemu, zaniedbując to, co z punktu widzenia tej skuteczności nie jest wymagane.

Aby modelować rzeczywistość, zwykle zaczynamy od identyfikacji przedmiotów (**obiektów**), rodzajów obiektów (**pojęć**), ich cech (**atrybutów**) i związków między nimi. Tak postępuje się np. w projektowaniu relacyjnych baz danych (jak Access, Oracle, MySQL...). Każda baza danych jest modelem pewnej rzeczywistości, a wyniki zapytań kierowanych do baz danych – uzyskanymi informacjami, prawdziwymi w tej rzeczywistości. Podobnie postępuje się w wielu innych obszarach informatyki, w tym choćby w projektowaniu obiektowym niesłychanie ważnym we współczesnych systemach.

W przykładzie z taśmami produkcyjnymi interesują nas przedmioty, których atrybutem jest kolor. Gdybyśmy nieco skomplikowali zadanie, zakładając, że na lewo przenosimy zielone owoce, na prawo – czerwone pomidory, zaś inne obiekty przepuszczamy dalej, zaczynają się pojawiać pojęcia takie jak „owoc", „pomidor", których atrybutem jest kolor.

3. Od modelu do bazy wiedzy

Zwykle model opisujemy początkowo w nieformalny sposób, najczęściej w języku naturalnym. Aby można go było wykorzystać we wnioskowaniu logicznym – musimy ten nieformalny opis przetworzyć na opis wykorzystujący notację logiczną. Powstaje w ten sposób pewna **baza wiedzy**. Następnie jesteśmy zainteresowani wyciąganiem wniosków wynikających z tej bazy wiedzy. Bazy wiedzy mogą opisywać stosunkowo prostą rzeczywistość, jak meble w danym pokoju, poprzez sytuacje dużo bardziej skomplikowane, jak opis chorób, czy zasad ruchu drogowego, po naprawdę trudne do obsługi i wnioskowania, jak obejmujące wybrane teorie matematyczne, fizyczne, chemiczne, biologiczne itd.

Co to znaczy, że wyciągamy wnioski z bazy wiedzy? Otóż interesuje nas, jakie wnioski wynikają z wiedzy zgromadzonej w danej bazie. Jeśli Δ jest bazą wiedzy, zaś W – interesującym nas wnioskiem, badamy, czy z Δ można wywnioskować W, czyli czy Δ implikuje W. Odnosząc się do prostej bazy wiedzy, dotyczącej wyboru pomiędzy autobusem i tramwajem, samą bazę stanowią zdania (a), (b), (c), zaś wnioskiem jest zdanie (d). Podkreślmy, że jeśli baza wiedzy zawiera wiele zdań, przyjmujemy, że zdania te są połączone spójnikiem „i". Baza składająca się ze zdań (a), (b), (c) jest więc rozumiana jako zdanie „(a) i (b) i (c)".

Z praktycznego punktu widzenia potrzebujemy więc języka, w którym będziemy opisywali wiedzę i wnioski, oraz mechanizmów wnioskowania, najlepiej dających się skutecznie implementować. Wprowadzimy więc logikę tradycyjnie nazywaną **klasycznym rachunkiem zdań**.

4. Klasyczny rachunek zdań

Klasyczny rachunek zdań zajmuje się badaniem prawdziwości zdań złożonych na podstawie zdań składowych i w konsekwencji – badaniem poprawności wnioskowania.

4.1. Język klasycznego rachunku zdań

Aby wprowadzić rachunek zdań, zaczyna się od **zmiennych zdaniowych** reprezentujących wartości logiczne **prawda**, **fałsz**, a zarazem zbiory obiektów mających cechy opisywane tymi zmiennymi (w tym ujęciu zmienne zdaniowe odpowiadają cechom, czyli atrybutom obiektów). Możemy na przykład użyć zmiennych „czerwony", „zielony". Mogą one przyjąć wartości prawda, fałsz w zależności od tego, czy dany obiekt jest czerwony (zielony). Możemy też patrzeć na te zmienne, jako na reprezentujące odpowiednio czerwone i zielone obiekty.

Bardziej złożone wyrażenia, zwane **formułami**, uzyskujemy stosując **spójniki logiczne** negacji, koniunkcji, alternatywy, implikacji i równoważności. Czasem wprowadza się też inne spójniki. Tak naprawdę wszystkie możliwe spójniki można zdefiniować przy pomocy np. negacji i koniunkcji, jednak przyjęty przez nas zestaw spójników, choć z tego punktu widzenia nadmiarowy, jest z jednej strony prosty i naturalny, a z drugiej wystarczający w wielu typowych zastosowaniach.

4.2. Tablice logiczne

Znaczenie (czyli **semantykę**) spójników logicznych podaje się często przy pomocy **tablic logicznych**, w których w kolumnach podaje się wartości poszczególnych wyrażeń. Przyjmujemy, że wartościami tymi mogą być jedynie 0, 1; 0 – to fałsz, a 1 – to prawda. Przyjmujemy też notację dla tych spójników: $\neg$ (negacja), $\wedge$ (koniunkcja), $\vee$ (alternatywa), $\Rightarrow$ (implikacja) oraz $\Leftrightarrow$ (równoważność).

Mamy następującą tablicę dla negacji:

p	$\neg p$
0	1
1	0

oraz kolejną dla pozostałych rozważanych spójników:

		koniunkcja	**alternatywa**	**implikacja**	**równoważność**
p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Te tablice dają podstawę bardzo skutecznemu mechanizmowi sprawdzania poprawności wnioskowania dla formuł ze stosunkowo niewielką liczbą zmiennych zdaniowych. Mianowicie konstruujemy tablice logiczne, w których:

- w pierwszych kolumnach umieszczamy zmienne zdaniowe,
- w kolejnych kolumnach umieszczamy wyrażenia występujące w badanej formule ułożone w ten sposób, by wartość danego wyrażenia można było policzyć na podstawie wartości wyrażeń występujących we wcześniejszych kolumnach.

Wiersze w tabeli wypełnia się zaczynając od kolumn odpowiadających zmiennym zdaniowym. Wypełniamy te kolumny wszystkimi możliwymi układami wartości logicznych, jakie można przypisać zmiennym zdaniowym. W następnych krokach wyliczamy wartości wyrażeń w kolejnych kolumnach.

Formuła nazywa się **tautologią,** jeśli przyjmuje wartość 1 (prawda) niezależnie od wartości wchodzących w jej skład zmiennych zdaniowych. Jest ona **spełnialna**, gdy przyjmuje wartość 1 co najmniej dla jednej kombinacji wartości zmiennych zdaniowych. Jest nazywana **kontrtautologią**, jeśli zawsze przyjmuje wartość 0 (fałsz).

Dla przykładu sprawdźmy jakie wartości przyjmuje formuła $\neg(p \vee q) \Rightarrow \neg p$. Tworzymy tablicę logiczną:

p	q	$p \vee q$	$\neg(p \vee q)$	$\neg p$	$\neg(p \vee q) \Rightarrow \neg p$
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	0	0	1
1	1	1	0	0	1

Badana formuła jest zawsze prawdziwa, jest więc tautologią (każda tautologia jest również spełnialna). Formuły występujące we wcześniejszych kolumnach są spełnialne, ale nie są tautologiami.

Sprawdźmy jeszcze **zasadę dowodzenia przez doprowadzanie do sprzeczności**.

Zasada dowodzenia przez doprowadzanie do sprzeczności, nazywana też dowodem nie wprost, została odkryta już przed niemal 2,5 tysiącami lat. Przypisuje się ją Zenonowi z Elei (495 r. p.n.e.) choć – może mniej jawnie – była używana przez jego poprzedników. Zenon z Elei znany był jak polemista, doskonalący sztukę prowadzenia sporów. Słynne są również jego paradoksy.

Zasada ta jest również bardzo ważna współcześnie. Stosuje się ją w matematyce, ale też odgrywa zasadniczą rolę we wnioskowaniu z baz wiedzy. Będziemy z niej korzystać przy omawianiu metody rezolucji, najpopularniejszej współczesnej metody automatycznego wnioskowania. Zasada sprowadzania do sprzeczności mówi, że w celu wykazania implikacji $p \Rightarrow q$, zaprzeczamy q i wykazujemy, że takie zaprzeczenie prowadzi do fałszu. Innymi słowy, chcemy wykazać, że formuła:

$$(p \Rightarrow q) \Leftrightarrow ((p \wedge \neg q) \Rightarrow 0)$$

jest tautologią. Konstruujemy odpowiednią tablicę logiczną:

p	q	$p \Rightarrow q$	$\neg q$	$p \wedge \neg q$	$(p \wedge \neg q) \Rightarrow 0$	$(p \Rightarrow q) \Leftrightarrow ((p \wedge \neg q) \Rightarrow 0)$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	1	0	0	1	1

W kolumnie odpowiadającej badanej formule występuje wyłącznie wartość 1, badana formuła jest więc rzeczywiście tautologią.

4.3. Dlaczego metoda tablic logicznych nie jest dobra dla większych zadań?

W praktycznych zastosowaniach często trzeba sprawdzać spełnialność formuł zawierających dużą liczbę zmiennych. Potrafi ona dochodzić do tysięcy. Wykonajmy teraz prosty rachunek. Załóżmy, że mamy formułę mającą 100 zmiennych. Tabela logiczna będzie więc miała 2^{100} wierszy, bo tyle jest różnych przypisań dwóch wartości logicznych stu zmiennym. Ile czasu spędziłby na obliczeniach bardzo szybki komputer, wykonujący, powiedzmy 2^{54} operacji na sekundę (to więcej niż 10^{16} operacji na sekundę)?

Aby wygenerować 2^{100} wierszy potrzebujemy:

- więcej niż $2^{100}/2^{54}$ $(=2^{46})$ sekund, czyli więcej niż $2^{46}/60$ minut,
- to więcej niż $2^{46}/2^{6}$ $(=2^{40})$ minut
- i więcej niż $2^{40}/2^{6}$ $(=2^{34})$ godzin,
- i więcej niż $2^{34}/2^{5}$ $(=2^{29})$ dób,
- to z kolei więcej niż 2^{20} lat (czyli więcej niż milion lat).

5. Automatyczne wnioskowanie

Dotychczas omawialiśmy metody wnioskowania skuteczne w niewielkich przykładach. Rzeczywiste systemy obsługujące klasyczny rachunek zdań, nazywane SAT Solvers (SAT pochodzi od angielskiego *satisfiability*, czyli **spełnialność**), potrafią sobie radzić z bardzo dużymi formułami występującymi w niemałym obszarze zastosowań. Ich siła polega na stosowaniu dużej liczby algorytmów skutecznych dla wybranych rodzajów formuł.

Poniżej przedstawimy jeden z takich algorytmów, bardzo skuteczny i powszechnie stosowany w informatyce oraz sztucznej inteligencji, zwany **metodą rezolucji**. Metoda rezolucji działa na koniunkcjach klauzul, przy czym **klauzula** jest alternatywą zmiennych zdaniowych lub ich negacji. Na przykład klauzulą jest:

$$p \vee \neg q \vee \neg r \vee s$$

zaś nie jest: $p \vee \neg\neg q$ (bo $\neg\neg q$ nie jest negacją zmiennej, a negacją negacji zmiennej). Nie jest też klauzulą $p \wedge \neg r$ (ponieważ jest to koniunkcja, a nie alternatywa).

Przyjmuje się, że pusta klauzula (niemająca żadnych wyrażeń) jest równoważna fałszowi (czyli 0).

5.1. Dlaczego klauzule są ważne?

Wiedza w systemach sztucznej inteligencji, w tym w bazach wiedzy, systemach eksperckich itd., zwykle ma postać klauzulową. Mianowicie implikacja postaci:

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_k) \Rightarrow (r_1 \vee r_2 \vee \ldots \vee r_m)$$

jest równoważna klauzuli:

$$\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee r_1 \vee r_2 \vee \ldots \vee r_m.$$

Implikacje wspomnianej postaci nazywamy **regułami**. Podobnymi regułami posługujemy się codziennie, np.:

- gorączka ∧ kaszel ⇒ przeziębienie ∨ grypa,
- deszcz ∧ bezwietrznie ⇒ parasol ∨ kurtka_z_kapturem,
- deszcz ∧ wiatr ⇒ samochód.

Klauzule są maszynową reprezentacją reguł, wygodną z punktu widzenia stosowania metody rezolucji.

5.2. Przekształcanie formuł do postaci klauzulowej

Okazuje się, że każdą formułę rachunku zdań można przekształcić do równoważnej jej formuły w postaci klauzulowej. Aby dokonać takiego przekształcenia, zastępujemy podformuły występujące w formule wejściowej zgodnie z podanymi poniżej zasadami aż do momentu uzyskania koniunkcji klauzul.

Łatwo się przekonać, stosując metodę tablic logicznych, że formuła zastępowana przyjmuje zawsze tę samą wartość logiczną co formuła ją zastępująca:

Formuła zastępowana	Formuła zastępująca
$A \Leftrightarrow B$	$(\neg A \vee B) \wedge (A \vee \neg B)$
$A \Rightarrow B$	$\neg A \vee B$
$\neg\neg A$	A
$\neg(A \wedge B)$	$\neg A \vee \neg B$
$\neg(A \vee B)$	$\neg A \wedge \neg B$
$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$
$(B \wedge C) \vee A$	$(A \vee B) \wedge (A \vee C)$

Zakładamy, że usuwane są:

- zbędne nawiasy (np. $((A))$ można zastąpić przez (A), zaś $(A \vee B) \vee C$ – przez $A \vee B \vee C$,

- powtórzenia wyrażeń występujących w alternatywach (np. $(A \vee A \vee B)$ można zastąpić przez równoważną formułę $(A \vee B)$),
- powtórzenia wyrażeń występujących w koniunkcjach (np. $(A \wedge A \wedge B)$ można zastąpić przez równoważną formułę $(A \wedge B)$).

Dla przykładu rozważmy formułę: $(\neg(p \wedge \neg q) \vee (p \Rightarrow r)) \vee (r \Leftrightarrow \neg s)$. Jej sprowadzenie do postaci klauzulowej może składać się z kroków:

- $((\neg p \vee \neg\neg q) \vee (p \Rightarrow r)) \vee (r \Leftrightarrow \neg s)$,
- $((\neg p \vee \neg\neg q) \vee (\neg p \vee r)) \vee (r \Leftrightarrow \neg s)$,
- $(\neg p \vee \neg\neg q \vee \neg p \vee r) \vee (r \Leftrightarrow \neg s)$,
- $(\neg p \vee q \vee \neg p \vee r) \vee (r \Leftrightarrow \neg s)$,
- $(\neg p \vee q \vee \neg p \vee r) \vee ((\neg r \vee \neg s) \wedge (r \vee \neg\neg s))$,
- $(\neg p \vee q \vee \neg p \vee r) \vee ((\neg r \vee \neg s) \wedge (r \vee s))$,
- $(\neg p \vee q \vee r) \vee ((\neg r \vee \neg s) \wedge (r \vee s))$,
- $(\neg p \vee q \vee r \vee \neg r \vee \neg s) \wedge (\neg p \vee q \vee r \vee r \vee s)$,
- $(\neg p \vee q \vee r \vee \neg r \vee \neg s) \wedge (\neg p \vee q \vee r \vee s)$.

Wynikową formułę można z łatwością uprościć zauważając, że pierwsza klauzula ma zawsze wartość prawda (ponieważ zawiera alternatywę $r \vee \neg r$):

- $1 \wedge (\neg p \vee q \vee r \vee s)$,

czyli:

- $(\neg p \vee q \vee r \vee s)$.

5.3. Metoda rezolucji

Metoda rezolucji wykorzystuje zasadę dowodzenia przez doprowadzanie do sprzeczności.

Metodę rezolucji wprowadził John Alan Robinson w 1965 roku. Była ona jednak stosowana już w XIX wieku, pod inną nazwą i w zakresie ograniczonym do formuł wybranej postaci. Wykorzystywał ją Charles Lutwidge Dodgson, znany także jako Lewis Carroll, autor m.in. *Alicji w krainie czarów*. W jego podręczniku *Symbolic Logic* (1896 r.) metoda ta nosi nazwę *method of underscoring*.

Metoda rezolucji jest w swojej istocie oparta na przechodniości implikacji, czyli na regule mówiącej że:

z przesłanek $(p \Rightarrow q)$ oraz $(q \Rightarrow r)$ mamy prawo wnioskować $(p \Rightarrow r)$.

Na przykład:

z przesłanek (deszcz ⇒ mokro) oraz (mokro ⇒ ślisko)
możemy wywnioskować, że (deszcz ⇒ ślisko).

Sformułowanie tej reguły w postaci klauzulowej jest następujące:

z przesłanek $(\neg p \vee q)$ oraz $(\neg q \vee r)$ wywnioskuj $(\neg p \vee r)$.

Uogólniając na dowolne klauzule uzyskamy regułę rezolucji mówiącej, że z klauzul:

$$\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k \vee r_1 \vee r_2 \vee \ldots \vee r_m$$
$$p_1 \vee \neg q_1 \vee \ldots \vee \neg q_n \vee s_1 \vee s_2 \vee \ldots \vee s_m$$

mamy prawo wywnioskować klauzulę:

$$\neg p_2 \vee \ldots \vee \neg p_k \vee r_1 \vee r_2 \vee \ldots \vee r_m \vee \neg q_1 \vee \ldots \vee \neg q_n \vee s_1 \vee s_2 \vee \ldots \vee s_m,$$

czyli usuwamy jedną ze zmiennych występującą w pierwszej klauzuli wraz z jej negacją występującą w drugiej klauzuli. Dla wygody zapisaliśmy je jako pierwsze, ale miejsce wystąpienia w klauzulach nie ma znaczenia – ważne jest tylko, by wystąpiły one w dwóch klauzulach.

Zauważmy, że po wywnioskowaniu nowej klauzuli pewne wyrażenia mogą się w niej powtarzać. Jak już wiemy $A \vee A$ jest równoważne A, więc powtórzenia wyrażeń po prostu usuwamy. Na przykład, mając klauzule $p \vee q \vee r$ oraz $p \vee \neg q$ możemy zastosować regułę rezolucji i uzyskać wniosek $q \vee q \vee r$. Ponieważ q się powtarza, możemy usunąć powtórzenie i uzyskujemy $q \vee r$.

Jak wspomnieliśmy, reguła rezolucji to zasada wnioskowania przez doprowadzanie do sprzeczności. Oznacza to, że najpierw negujemy sprawdzaną formułę, a następnie staramy się z tej negacji uzyskać fałsz, a więc klauzulę pustą. Jeśli się to uda, początkowa formuła jest tautologią. Jeśli pustej klauzuli nie da się w żaden sposób uzyskać, formuła tautologią nie jest.

Ważnym zastosowaniem metody rezolucji jest wykazywanie, że pewna formuła wynika z bazy danych wiedzy, w której wiedza jest reprezentowana jako zbiór klauzul. Chcemy sprawdzić czy $\Delta \Rightarrow q$, gdzie Δ jest bazą wiedzy, a q jest formułą wykazywaną przy założeniu, że wiedza zawarta w Δ odzwierciedla interesującą nas rzeczywistość. W metodzie rezolucji zaprzeczamy implikacji $(\Delta \Rightarrow q)$. Zaprzeczenie to jest równoważne formule $(\Delta \wedge \neg q)$. Czyli $\neg q$ dokładamy do bazy wiedzy Δ (przekształcając $\neg q$ do postaci klauzulowej) i staramy się wyprowadzić klauzulę pustą.

Zauważmy, że baza Δ nie zmienia się. Z wydajnościowego punktu widzenia jest to bardzo ważne, bo zwykle taka baza danych jest duża, podczas gdy badane

wnioski zwykle stosunkowo małe, gdyż reprezentują one typowe zapytania użytkowników. Zwykle zapytania mają bardzo niewielkie rozmiary w porównaniu z rozmiarem bazy danych.

Dla przykładu wykażmy, że z koniunkcji klauzul $(p \Rightarrow q) \wedge (\neg p \Rightarrow q)$ można wywnioskować q. Jest to formalizacja wnioskowania przez przypadki, bo spełniony jest warunek p albo warunek $\neg p$. Bez względu na to, który z nich jest spełniony, konsekwencją jest q. W życiu często stosujemy takie wnioskowanie. Na przykład, gdy chcemy zabezpieczyć się przed zmoknięciem, bierzemy parasol. W tej sytuacji stosujemy wnioskowanie:

$$(\neg\text{deszcz} \wedge \text{parasol} \Rightarrow \neg\text{zmoknę}) \wedge (\text{deszcz} \wedge \text{parasol} \Rightarrow \neg\text{zmoknę}),$$

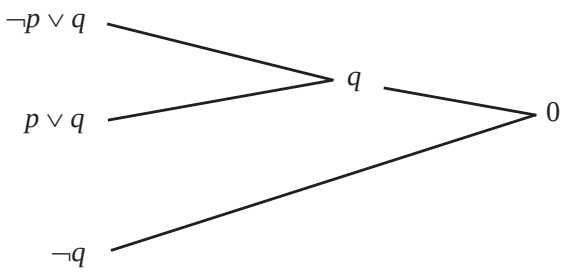
a więc wnioskuję, że skoro zawsze biorę parasol, nie zmoknę bez względu na to, czy będzie deszcz, czy nie. Może to niezbyt praktyczny wniosek, zwłaszcza w czasie upałów, ale rzeczywiście gwarantuje ochronę przed zmoknięciem.

Zasadę wnioskowania przez przypadki można zapisać w postaci klauzulowej jako:

$(\neg p \vee q)$ – pierwsze założenie w postaci klauzulowej
$(p \vee q)$ – drugie założenie w postaci klauzulowej
$\neg q$ – zaprzeczona konkluzja.

Stosując regułę rezolucji do dwóch pierwszych klauzul uzyskujemy klauzulę $(q \vee q)$, usuwamy zbędne powtórzenie q, uzyskujemy więc klauzulę zawierającą jedynie q. Teraz z tej klauzuli oraz z $\neg q$ uzyskujemy klauzulę pustą.

Graficznie to wnioskowanie można przedstawić jak na rysunku 1.

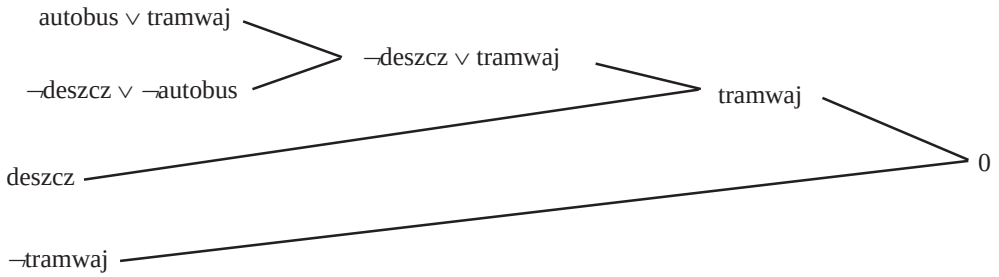


Rysunek 1. Graficzna reprezentacja wnioskowania rezolucyjnego

Zbadajmy jeszcze słuszność wcześniej rozważanego rozumowania dotyczącego wyboru pomiędzy autobusem lub tramwajem. Reprezentacja zdań (a)–(d) w logice może być następująca:

autobus ∨ tramwaj,
deszcz ⇒ ¬autobus (ta implikacja jest równoważna klauzuli ¬deszcz ∨ ¬autobus),
deszcz,
wniosek: tramwaj.

Aby zastosować metodę rezolucji, negujemy wniosek, dołączamy go do bazy wiedzy i staramy się uzyskać klauzulę pustą reprezentującą fałsz (0). Odpowiednie wnioskowanie rezolucyjne ilustruje rysunek 2.



Rysunek 2. Przykładowe wnioskowanie rezolucyjne

Rozważmy jeszcze inny przykład wnioskowania. Mamy do czynienia z sytuacją, w której robot ma wybrać kierunek ruchu: w lewo, na wprost lub w prawo. Jeśli nie pada deszcz, powinien iść w prawo. Jeśli pada deszcz, nie powinien iść w lewo, ani na wprost. Jaki kierunek robot może wybrać?

Stwórzmy najpierw odpowiednią bazę wiedzy:

- lewo ∨ wprost ∨ prawo,
- ¬deszcz ⇒ prawo,
- deszcz ⇒ ¬lewo ∧ ¬wprost.

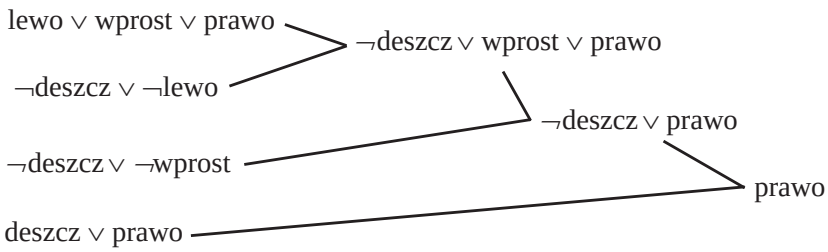
Po przekształceniu tej bazy do postaci klauzulowej uzyskamy:

- lewo ∨ wprost ∨ prawo,
- deszcz ∨ prawo,
- ¬deszcz ∨ ¬lewo,
- ¬deszcz ∨ ¬wprost.

Dwie ostatnie klauzule są uzyskane z implikacji deszcz ⇒ ¬lewo ∧ ¬wprost. Mianowicie, zgodnie z podanymi wcześniej zasadami, jest ona równoważna: ¬deszcz ∨ (¬lewo ∧ ¬wprost), czyli:

$$(\neg\text{deszcz} \vee \neg\text{lewo}) \wedge (\neg\text{deszcz} \vee \neg\text{wprost}).$$

Rysunek 3 przedstawia odpowiednie wnioskowanie rezolucyjne (dla czytelności kolejność klauzul została nieco zmieniona, co oczywiście nie wpływa na wynik).



Rysunek 3. Przykładowe wnioskowanie rezolucyjne

Właściwym wnioskiem jest więc wybór kierunku w prawo. Możemy zauważyć bowiem, że gdybyśmy rozważali wnioskowanie „baza wiedzy implikuje wniosek", to stosując metodę rezolucji dodajemy zaprzeczony wniosek do bazy wiedzy i staramy się uzyskać fałsz. Gdybyśmy dołączyli do bazy zaprzeczony wniosek, czyli „¬prawo", uzyskanie fałszu byłoby już natychmiastowe, ponieważ mamy wywnioskowany fakt „prawo".

6. Logika wyszukiwania

Możemy już stwierdzić, że logika jest we wnioskowaniu wszechobecna, gdyż fałsz, prawda, wnioskowanie, model, pojęcie, związek (relacja), reguła, czy spójniki (∧, ∨, ¬, ⇒) są podstawowymi konceptami rozważanymi w logice.

Przejdźmy teraz do wyszukiwarek internetowych. Internet jest ogromną bazą wiedzy (bez względu na to, jak oceniamy jakość tych czy innych obszarów tej wiedzy). Zasadniczą różnicą w porównaniu z tradycyjnymi bazami danych, zorganizowanymi w bardzo uporządkowany sposób, jest to, że w Internecie występuje ogromna różnorodność zasobów i brak ich jednolitej struktury.

Dla ustalenia uwagi załóżmy, że interesującymi nas obiektami z Internetu są strony WWW. Wpisując w okienko wyszukiwarki Google zestaw słów kluczowych, pytamy o te strony, na których występują wszystkie wypisane słowa kluczowe. Jest to tzw. **wyszukiwanie AND**, odpowiadające koniunkcji (w języku polskim „and" znaczy „i").

Jak już wiemy, aby koniunkcja *p* AND *q* była prawdziwa, prawdziwe muszą być oba zdania składowe: zdanie *p* i zdanie *q*. Jeśli wpiszemy dwa słowa: *logika informatyka* to z punktu widzenia wyszukiwarki Google oznacza to wpisanie wyrażenia *logika* AND *informatyka*, czyli wyszukanie stron (naszych obiektów), na których występuje słowo *logika* i słowo *informatyka*. Tak naprawdę nie zawsze pojawią się oba słowa, co odbiega od logicznego rozumienia koniunkcji. Aby mieć „prawdziwą" koniunkcję powinniśmy wpisać wyrażenie *+logika +infor-*

matyka (operator + umieszczony przed danym słowem oznacza, że musi ono wystąpić na wyszukanej stronie)[1].

Co jeszcze pojawia się w Google? Twórcy tej wyszukiwarki oferują też **wyszukiwanie OR**. W języku polskim „or" to „lub", czyli logiczny spójnik alternatywy. Przypomnijmy sobie, że aby alternatywa *p* OR *q* była prawdziwa, prawdziwe musi być co najmniej jedno ze zdań składowych: zdanie *p* lub zdanie *q* (lub oba te zdania). Na przykład wpisanie w Google wyrażenia

logika OR *informatyka*

spowoduje wyszukanie stron, na których występuje słowo *logika* lub słowo *informatyka* lub oba te słowa. Ponieważ nie używamy nawiasów, potrzebna jest zasada wyjaśniająca, jak rozumieć wpisywane wyrażenia. Przy dodawaniu i mnożeniu wyrażenie x+y*z rozumiemy jako x+(y*z). Znak mnożenia ma większą siłę niż znak dodawania. Podobnie przyjmujemy w wyrażeniach zawierających spójnik OR oraz AND, przy czym rolę mnożenia odgrywa OR, zaś dodawania – AND[2]. Oznacza to, że wyrażenie

lekcja informatyka OR *logika*

wyszukiwarka Google rozumie jako

$$lekcja \wedge (informatyka \vee logika),$$

a nie jako (*lekcja* ∧ *informatyka*) ∨ *logika*. Wyszukane zostaną więc strony, na których pojawia się słowo *lekcja* oraz co najmniej jedno ze słów: *informatyka* lub *logika*.

I mamy jeszcze operator '-', który umieszczony przed słowem oznacza, że *nie* może ono wystąpić na wyszukanej stronie. Spójnik ten odpowiada więc negacji. Wpisanie do wyszukiwarki wyrażenia *-logika* spowoduje więc wyszukanie tych stron WWW, na których nie występuje słowo *logika*.

Google posługuje się logiką, interpretując wyrażenia logiczne i wyszukując zgodnie z nimi interesujące nas zasoby.

7. Programowanie w logice

Tradycyjne języki programowania, jak C, C++ czy Java, odzwierciedlają metodologię wyrażoną tytułem klasycznej już książki Niklausa Wirtha:

Algorytmy + struktury danych = programy.

[1] Tak naprawdę Google czasem łamie tę zasadę, bowiem algorytmy porządkownia stron w połączeniu z aukcjami związanymi ze sprzedażą reklam nie zawsze dobrze odpowiadają potrzebom wyrażonym przez wyszukującego.

[2] Ta konwencja jest charakterystyczna dla Google. Tradycyjnie koniunkcja jest „silniejsza" niż alternatywa.

Główny nacisk jest tu położony na łatwość zapisywania algorytmów i mniejszą (np. w C) lub większą (w programowaniu obiektowym, jak C++, czy Java) łatwość opisu struktur danych. Programowanie polega tu na opracowaniu właściwych struktur danych oraz na wprowadzeniu algorytmu rozumianego jako dokładny opis procesu przetwarzania danych, precyzujący kolejne kroki do wykonania w danej sytuacji. Opis ten jest następnie kompilowany lub interpretowany i wykonywany przez komputer. W wielu obszarach zastosowań okazuje się, że to tradycyjne podejście może być zastąpione takim, w którym podaje się fakty i reguły, a same obliczenia pozostawia komputerowi. To podejście, zaproponowane przez Roberta Kowalskiego w roku 1972, zakłada zmianę powyższej zasady na zasadę:

algorytm = logika + sterowanie,

w której logika mówi o tym, jaki cel należy osiągnąć, zaś sterowanie o tym, jak osiągnąć oczekiwany cel. Następuje tu oddzielenie logiki od sterowania: logikę określa użytkownik, zaś sterowanie jest wyznaczane przez komputer. Pojawia się naturalne pytanie: czy taki cel można osiągnąć? Czyli – czy możliwe jest programowanie w logice?

Aby wyjaśnić zasadę wyznaczania przez komputer sterowania przyjrzyjmy się typowemu sposobowi rozumowania, w którym z posiadanej wiedzy staramy się uzyskać interesujące nas wnioski. Mamy więc pewną, być może złożoną, bazę wiedzy i reguły wnioskowania. Na przykład:

- z faktu, że Jan jest ojcem Marii i Jacka wnioskujemy, że Maria i Jacek są – być może przyrodnim – rodzeństwem,
- z faktu, że mamy zielone światło wnioskujemy, że możemy przejść przez ulicę (nadal zachowując pewną ostrożność).

Dane są więc fakty: „Jan jest ojcem Marii", „światło jest zielone" oraz reguły, które na podstawie tych faktów pozwalają wyciągnąć wnioski „Maria i Jacek są rodzeństwem", „możemy przejść przez ulicę".

Pojawia się naturalne pytanie, w jakich dziedzinach zastosowań takie podejście ma sens, a w jakich zastosowanie podejść tradycyjnych jest się lepsze. Okazuje się, że podejście tradycyjne, oparte na algorytmach i strukturach danych, jest skuteczniejsze w zastosowaniach nastawionych na obliczenia numeryczne, w których podstawowe są operacje na liczbach, lub nastawionych na złożone struktury danych. W obliczeniach związanych z wnioskowaniem i przetwarzaniem danych symbolicznych (tj. nieliczbowych) wygodniejsze okazuje się podejście regułowe, wchodzące w skład **deklaratywnej** metody programowania, w której wyznacza się cele do osiągnięcia, pozostawiając komputerowi znalezienie metody osiągnięcia tych celów. Do języków deklaratywnych zaliczamy języki funkcyjne np. Lisp, Schema, ML, języki programowania w logice, jak

Prolog, oraz wybrane języki zapytań, jak choćby SQL czy Datalog. Spośród wymienionych języków podejście regułowe realizuje Prolog i Datalog. Języki regułowe są też stosowane w dużych systemach komercyjnych, jak Oracle Business Rules, IBM ILOG czy Business Rules Framework wchodzącym w skład Microsoft BizTalk Server.

Przykład

Rozpocznijmy od prostego przykładu – chcemy wnioskować o relacjach rodzinnych. Zacznijmy najpierw od pytania, jakie informacje wystarczają do określenia związków rodzinnych. Można wybrać różne informacje bazowe, na przykład możemy zauważyć, że wystarczy informacja o tym, kto jest czyim rodzicem i o płci. Załóżymy więc, że mamy dane relacje bycia rodzicem, mężczyzną i kobietą. W języku naturalnym relacja bycia rodzicem dotyczy dwóch osób – rodzica i potomka. Relacja płci dotyczy pojedynczych osób. Możemy na przykład powiedzieć, że Jan jest rodzicem Marii, Jan jest mężczyzną i Maria jest kobietą. W logice używamy zwykle nieco innej notacji: nazwę relacji umieszczamy na początku, a następnie piszemy **argumenty** tej relacji, czyli wyrażenia określające konkretne obiekty lub zmienne. Podobnie zapisuje się relacje w językach programowania.

Relacje występujące w naszym przykładzie zapiszemy w notacji logicznej jako:

- rodzic(X, Y) – oznaczające, że osoba X jest rodzicem osoby Y,
- kobieta(X) – oznaczające, że X jest kobietą,
- mężczyzna(X) – oznaczające, że X jest mężczyzną.

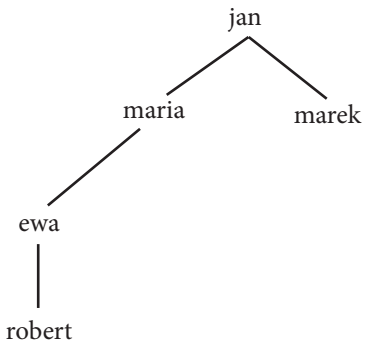
Możemy teraz opisywać fakty:

- rodzic(jan, maria), rodzic(jan, marek), rodzic(maria, ewa), rodzic(ewa, robert),
- kobieta(maria), kobieta(ewa),
- mężczyzna(jan), mężczyzna(marek), mężczyzna(robert).

Zauważmy, że imiona napisaliśmy małymi literami. To celowy zabieg – w omawianych językach regułowych stałe zapisuje się rozpoczynając nazwę małą literą. Gdy nazwę rozpoczyna wielka litera – mamy do czynienia ze **zmienną**. Jeśli napiszemy rodzic(jan, X), X oznacza zmienną, mogącą w trakcie obliczeń przyjąć konkretne wartości. Dotychczas mieliśmy do czynienia ze zmiennymi zdaniowymi, przyjmującymi wartości prawda, fałsz. Zmienne występujące powyżej mają inną rolę – reprezentują obiekty z danej dziedziny.

Pisząc fakt, taki jak rodzic(jan, maria) stwierdzamy, że ten fakt **zachodzi**, czyli jest prawdziwy w opisywanej sytuacji.

Podane wcześniej fakty definiują następujące drzewo genealogiczne:



Rysunek 4. Przykładowe drzewo genealogiczne

Możemy teraz zapisać reguły dotyczące związków rodzinnych. Zacznijmy od reguły definiującej relację bycia matką:

osoba X jest matką osoby Y jeśli X jest rodzicem Y i X jest kobietą.

Regułę tę możemy zapisać jako:

matka(X, Y) jeśli rodzic(X, Y) i kobieta(X).

Fakty wypisane po słowie „jeśli" nazywamy **przesłankami** reguły, zaś fakt występujący przed „jeśli" – jej **wnioskiem**[3]**.**

Zauważmy, że implikację piszemy teraz odwrotnie: zaczynamy od wniosku, a kończymy przesłankami. Ten styl notacji wynika z konwencji przyjętej w językach regułowych. Zauważmy też, że fakty są prostymi regułami, bowiem każdy fakt *R* można zapisać jako „*R* jeśli prawda", bowiem z prawdy możemy wywnioskować tylko fakty prawdziwe.

Używając powyższych faktów możemy wywnioskować np., że prawdziwy jest fakt:

matka(maria, ewa),

ponieważ korzystając z naszej reguły mamy:

matka(maria, ewa) jeśli rodzic(maria, ewa) i kobieta(maria),

co w świetle prawdziwości faktów rodzic(maria, ewa) oraz kobieta(maria), pozwala nam wyciągnąć omawiany wniosek.

Bycie ojcem można zapisać analogicznie:

ojciec(X, Y) jeśli rodzic(X, Y) i mężczyzna(X).

Jeśli siostrę danej osoby zdefiniujemy jako kobietę mającą wspólnego z nią rodzica, to odpowiednią regułę możemy zapisać:

[3] W literaturze wniosek nazywany jest często **nagłówkiem**, zaś zbiór przesłanek – **ciałem** reguły.

siostra(X, Y) jeśli kobieta(X) i rodzic(Z, X) i rodzic(Z, Y) i X =/= Y.

W powyższej regule Z oznacza tego samego rodzica dla X i dla Y. Oczywiście musieliśmy założyć, że X jest osobą różną od Y, ponieważ nikt nie jest swoją siostrą.

Jak zdefiniować dziadka? Możemy użyć poniższej reguły:

dziadek(X, Y) jeśli mężczyzna(X) i rodzic(X, Z) i rodzic(Z, Y).

Spróbujmy teraz zdefiniować przodka w dowolnym pokoleniu. Aby to osiągnąć, możemy użyć definicji:

(i) przodek(X, Y) jeśli rodzic(X, Y),
(ii) przodek(X, Y) jeśli rodzic(X, Z) i przodek(Z, Y).

Pierwsza z tych reguł stwierdza oczywisty fakt, iż rodzic jest przodkiem. Druga z nich stwierdza, że rodzic przodka jest także przodkiem. Użyliśmy więc **rekursji**, czyli w definicji reguły przodek, wśród przesłanek pojawia się odwołanie do przodka.

Popatrzmy jak wygląda przykładowe wnioskowanie z użyciem tych reguł. Chcemy wywnioskować, że Jan jest przodkiem Ewy. Na podstawie reguły (i) wnioskujemy, że Maria jest przodkiem Ewy, gdyż jest jej rodzicem. Stwierdzamy więc, że prawdziwy jest fakt przodek(maria, ewa). Wiemy, że rodzic(jan, maria). Zapisujemy regułę (ii) przyjmując, że X = jan, Z = maria oraz Y = ewa:

przodek(jan, ewa) jeśli rodzic(jan, maria) i przodek(maria, ewa).

Ponieważ przesłanki są prawdziwe, prawdziwy jest także wniosek. Podobnie możemy wywnioskować, że Maria jest przodkiem Roberta i znów stosując regułę (ii) mamy:

przodek(jan, robert) jeśli rodzic(jan, maria) i przodek(maria, robert),

czyli wnioskujemy również, że Jan jest przodkiem Roberta.

8. Podstawy języka Prolog

Prolog jest najbardziej znanym językiem programowania realizującym podejście regułowe. Istnieje wiele jego implementacji, w tym bardzo dobra jest darmowa implementacja SWI Prolog, dostępna pod adresem http://www.swi-prolog.org/. Inną wartą polecenia darmową implementacją jest Eclipse: http://eclipseclp.org/.

Programy zapisane w języku Prolog składają się z reguł i faktów. Fakty zapisuje się jako $R(t_1,..., t_n)$, gdzie R jest relacją n-argumentową, zaś $t_1,..., t_n$ są wyrażeniami bez zmiennych. Przykłady faktów już poznaliśmy wcześniej. Reguły mają postać:

$$R_1(t_1,..., t_{n1}) :- R_2(s_1,..., s_{n2}),..., R_k(u_1,..., u_{nk}).$$

Symbol ':-' czytamy jako „jeśli", zaś przecinki oddzielające przesłanki oznaczają spójnik „i".

Zakłada się, że argumenty relacji R_1,..., R_k są dowolnymi wyrażeniami zawierającymi stałe, zmienne i symbole funkcyjne, przy czym zmienne występujące we wniosku muszą też występować w przesłankach. Na przykład poprawna jest reguła:

$$R(\mathrm{X}, f(\mathrm{Y})) :- S(\mathrm{X}, a),\ T(\mathrm{Y}).$$

Natomiast nie jest poprawna:

$$R(\mathrm{X}, f(\mathrm{Y})) :- S(\mathrm{X}, a),\ T(\mathrm{X}).$$

ponieważ zmienna Y występuje we wniosku, a nie występuje w przesłankach.

Jeśli zbiór przesłanek jest pusty, przyjmujemy, że występuje tam prawda. Jeśli jakaś zmienna występuje w przesłankach, a nie występuje we wniosku oznacza ona istnienie pewnego obiektu. Na przykład zmienna Z w rozważanej wcześniej regule:

dziadek(X, Y):- mężczyzna(X), rodzic(X, Z), rodzic(Z, Y).

nie występuje we wniosku. Regułę tę odczytujemy więc jako:

X jest dziadkiem Y jeśli X jest mężczyzną
i istnieje osoba Z taka, że X jest rodzicem Z i Z jest rodzicem Y.

Gdy zdefiniowaliśmy już wszystkie reguły i opisaliśmy fakty, możemy zadawać pytania wyrażone przez ciąg przesłanek:

$$R(t_1,..., t_n),..., S(u_1,..., u_m).$$

Odpowiedź na tak zadane pytanie jest zdefiniowana następująco:

- jeśli w zapytaniu nie występują zmienne, to:
  - jeśli fakty występujące w zapytaniu są prawdziwe, to odpowiedzią jest „yes" (tak), czyli potwierdzenie prawdziwości zapytania, tzn. potwierdzenie, że da się ono wywnioskować z podanego zestawu faktów za pomocą zdefiniowanych reguł,
  - jeśli przynajmniej jeden fakt jest fałszywy, to odpowiedzią jest „no" (nie), czyli stwierdzenie, że zapytanie nie da się wywnioskować,
- jeśli w zapytaniu występują zmienne, to wynikiem są wszystkie krotki (ciągi) wartości, które podstawione w miejsce zmiennych powodują, że zapytanie da się wywnioskować.

Na przykład:

- zapytanie dziadek(jan, ewa), siostra(maria, marek) da odpowiedź „yes",
- zapytanie dziadek(jan, ewa), siostra(maria, ewa) da odpowiedź „no",
- zapytanie rodzic(X, Y) da w wyniku:

X = jan Y = maria
X = jan Y = marek

X = maria Y = ewa
X = ewa Y = robert

- zapytanie przodek(X, Y), kobieta(X) da w wyniku:

X = maria Y = ewa
X = ewa Y = robert
X = maria Y = robert.

Jeśli nie jesteśmy zainteresowani wartością pewnej zmiennej, np. interesują nas kobiety będące przodkami, ale nie jesteśmy zainteresowani danymi tych przodków, możemy użyć **zmiennej anonimowej** oznaczanej znakiem podkreślenia:

przodek(X, _), kobieta(X).

Wartości zmiennych anonimowych są obliczane, ale nie są przekazywane na zewnątrz.

W implementacjach języka Prolog podstawowym mechanizmem obliczeniowym jest przedstawiona wcześniej zasada rezolucji, dostosowana do radzenia sobie z regułami wychodzącymi poza rachunek zdań. Szczegóły tego dostosowania tu pominiemy. Ważne jest, że oparty o tę zasadę mechanizm obliczeniowy jest poprawny (wyniki są rzeczywiście wnioskami z faktów, uzyskanymi przez stosowanie reguł) i pełny (tzn. jeśli zapytanie jest wnioskiem, to będzie ono odpowiednio obliczone).

Język Prolog zawiera arytmetykę i listowe struktury danych. Istnieją bogate biblioteki implementujące wiele innych struktur. Przyjrzyjmy się jeszcze operacjom na listach. **Lista** to ciąg pewnych elementów. Listy zapisujemy jako $[a_1,..., a_n]$. Na przykład listą jest [1, 2, 3, *a, z*, 4]. Listy też często zapisujemy w postaci [G | Og], gdzie G jest wyróżnionym elementem nazywanym **głową** listy, zaś Og – listą pozostałych elementów, zwaną **ogonem** listy. Listę pustą zapisujemy jako [].

Rozważmy kilka przykładów przetwarzania list. Na początek sprawdźmy, czy element X występuje w danej liście:

```
jest(X, [X |]).
jest(X, [_, Y]:- jest(X, Y).
```

Pierwsza reguła stwierdza, że X występuje na liście zaczynającej się od X. Druga z nich mówi, że jeśli X jest na liście Y, to jest również na liście, której ogonem jest Y.

Aby zilustrować użycie arytmetyki, obliczmy ile jest elementów na zadanej liście:

```
ile(0, []).
ile(X, [_ | Z]:- ile(Y, Z), X = Y + 1.
```

Pierwsza z reguł mówi, że jest 0 elementów w pustej liście. Druga z nich, że w liście złożonej z jakiegokolwiek elementu początkowego '_' i ogona Z jest o jeden element więcej niż w Z.

Prolog jest stosowany w przetwarzaniu języka naturalnego, w systemach inteligentnych i wszędzie tam, gdzie wnioskowanie jest naturalnym sposobem dochodzenia do wymaganych wyników. Okazuje się, że w takich zastosowaniach kod w języku Prolog jest wielokrotnie (a czasem nawet kilkusetkrotnie) krótszy niż kody programów rozwiązujących takie same zagadnienia, napisanych w językach C, C++ czy Java.

9. Języki zapytań

Na pewno czytelnik zauważył, że opisując fakty mamy do czynienia z bazami danych. Mianowicie relacja odpowiada tabeli w relacyjnej bazie danych, zaś zbiór faktów definiuje kolejne wiersze w tej tabeli. Przykładowe fakty dotyczące bycia rodzicem można interpretować jako tabelę z dwoma kolumnami:

rodzic	**dziecko**
jan	maria
jan	marek
maria	ewa
ewa	robert

Również każdą tabelę można opisać zestawem faktów, po jednym fakcie dla opisu każdego z wierszy. Dlatego też jest naturalne, by faktami opisywać dane o elementach, zaś regułami – tabele, nieistniejące w rzeczywistości, ale obliczane, gdy zachodzi taka potrzeba. Poza zyskiem pamięciowym, można w ten sposób wyrazić zapytania niedające się zapisać w tradycyjnym standardzie SQL.

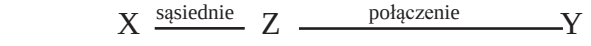
Podstawowym językiem regułowym rozważanym w kontekście baz danych jest **Datalog**. Jest on pewnym ograniczeniem języka Prolog. Mianowicie nie dopuszcza się w nim symboli funkcyjnych. Jedynymi argumentami relacji mogą więc być stałe i zmienne. Baza danych składa się z dwóch części: bazy faktów i bazy reguł. Datalog ma wiele implementacji, w tym darmowe. Bardzo dobrą edukacyjną implementacją Datalogu jest DES, dostępny pod adresem: http://www.fdi.ucm.es/profesor/fernan/des/. Zawiera on także implementację języka SQL, jest więc doskonałym narzędziem do nauczania baz danych.

Aby zilustrować tworzenie baz danych w ujęciu regułowym, rozważmy przykład, w którym interesuje nas znajdowanie połączeń kolejowych, być może z przesiadkami. Projektując taką bazę danych musimy zastanowić się, jakie informacje powinny być przechowywane w bazie danych. Naturalnym pierwszym pomysłem jest trzymanie wszystkich połączeń. Gdybyśmy jednak mieli tylko jedną linię kolejową przechodzącą kolejno przez 100 stacji, to wszystkich

połączeń byłoby 100*99 (każda stacja połączona z 99 innymi stacjami), czyli 9 900. Tymczasem wystarczy przechowywać informacje jedynie o połączeniach pomiędzy sąsiednimi stacjami, a więc informacje o nie więcej niż 99 połączeniach. Zdefiniujmy więc bazę danych zawierającą relację sąsiednie(X, Y), mówiącą o tym, że stacje X i Y są sąsiednie i połączone ze sobą. Teraz połączenia możemy zdefiniować regułami:

(iii) połączenie(X, Y):- sąsiednie(X, Y).
(iv) połączenie(X, Y):- sąsiednie(X, Z), połączenie(Z, Y).

Pierwsza z tych reguł jest oczywista. Druga z nich stwierdza, że X i Y są połączone, jeśli istnieje stacja Z sąsiednia do X (a więc połączona z X), która to stacja Z jest połączona z Y. Sytuację tę ilustruje poniższy diagram:



Zauważmy, że jeśli stacje X i Y są połączone, tzn. połączenie(X, Y) jest prawdą, przejazd między X oraz Y może wymagać zmiany pociągu.

Załóżmy, że mamy bazę faktów:

sąsiednie(warszawa, działdowo), sąsiednie(działdowo, iława),
sąsiednie(iława, tczew), sąsiednie(tczew, gdańsk).

Jak możemy teraz obliczyć relację „połączenie”?

Datalog ma dwie podstawowe metody obliczeniowe – poprzez wspomnianą przy okazji języka Prolog rezolucję oraz tzw. **metodę wstępującą**, w której zaczynając od faktów generujemy przy pomocy reguł wszystkie możliwe wnioski. Okazuje się, że metoda wstępująca zawsze się zatrzymuje i działa w czasie dopuszczalnym z praktycznego punktu widzenia (ograniczonym przez wielomian, gdy rozmiarem danych jest liczba obiektów/stałych występujących we wszystkich relacjach). Zanim sformalizujemy tę metodę przyjrzyjmy się jej działaniu na przykładzie połączeń kolejowych:

- reguła (iii) powoduje wygenerowanie faktów:
  połączenie(warszawa, działdowo), połączenie(działdowo, iława),
  połączenie(iława, tczew), połączenie(tczew, gdańsk),
- reguła (iv) powoduje dodatkowo wygenerowanie najpierw:
  połączenie(warszawa, iława), połączenie(działdowo, tczew),
  połączenie(iława, gdańsk),
  a więc połączeń stacji wymagających jednej stacji pośredniej,
- następnie reguła (iv) generuje dodatkowo:
  połączenie(warszawa, tczew), połączenie(działdowo, gdańsk),
  a więc połączeń stacji wymagających dwóch stacji pośrednich,
- w ostatniej iteracji regułą (iv) generuje fakt:

połączenie(warszawa, gdańsk),
a więc połączenie wymagające trzech stacji pośrednich.

Ogólnie algorytm obliczania relacji zdefiniowanych regułami jest następujący (ten algorytm jest poglądowy, rzeczywiste implementacje wykorzystują szereg zaawansowanych technik):

- niech A będzie zbiorem faktów
- dopóki A się zmienia:
  dodawaj do A nowe fakty, które powstają ze stosowania reguł w ten sposób, że ilekroć wszystkie przesłanki instancji danej reguły (z ustalonymi wartościami wszystkich zmiennych) są w zbiorze A, dołączamy do A również wniosek tej reguły.

Gdybyśmy chcieli rozszerzyć rozważaną bazę danych o połączenia autobusowe i możliwość przesiadek między autobusami i pociągami, wystarczy dodać relację sąsiednieA(X, Y) stwierdzającą, że X oraz Y są połączone autobusem bez przystanków pośrednich. Podane uprzednio reguły (iii) oraz (iv) wystarczy teraz uzupełnić o reguły:

połączenie(X, Y):- sąsiednieA(X, Y).
połączenie(X, Y):- sąsiednieA(X, Z), połączenie(Z, Y).

10. Uwagi o metodologii postępowania

Języki regułowe są doskonałym narzędziem modelowania złożonej rzeczywistości, a jednocześnie dostarczają mechanizmów wnioskowania. Z jednej strony mamy więc model, a z drugiej – natychmiastową możliwość jego testowania poprzez mechanizm zapytań występujący zarówno w języku Prolog, jak i regułowych językach bazodanowych.

Uporządkujmy teraz nieco metodologię stosowaną przy modelowaniu. Jak już podkreślaliśmy – skuteczność wnioskowań zależy od przyjętego modelu rzeczywistości.

Przyjrzyjmy się tej metodologii na jeszcze jednym przykładzie. Załóżmy, że chcemy utworzyć bazę danych w języku Datalog, pozwalającą opisywać ulice w danym mieście oraz interesujące miejsca, do których można tymi ulicami dotrzeć. Interesują nas skrzyżowania ulic oraz informacje, czy dana ulica jest zamknięta dla ruchu, czy przejezdna.

Zacznijmy od identyfikacji obiektów, o jakich będziemy gromadzić informacje. Z pewnością będą to ulice oraz interesujące miejsca. Ich atrybutami będą:

- w przypadku ulic - nazwa ulicy oraz czy informacja o przejezdności ulicy, opisywane jako ulica(U, P),
- w przypadku interesujących miejsc - typ miejsca (pomnik, muzeum, teatr...) oraz nazwa miejsca, opisywane jako miejsce(N, T).

Pozostały do określenia relacje. Z punktu widzenia naszego przykładu – mamy dwie relacje:

- skrzyżowanie(U1, U2), gdzie U1, U2 są nazwami ulic, stwierdzające, że istniejące ulice U1 i U2 tworzą skrzyżowanie,
- blisko(N, U), gdzie N jest nazwą miejsca, zaś U jest nazwą ulicy, stwierdzające, że miejsce N jest w pobliżu ulicy U.

Możemy teraz opisywać fakty takie jak:

ulica(kwiatowa, przejezdna), ulica(maków, zamknięta), ulica(bratków, przejezdna),
skrzyżowanie(kwiatowa, maków), skrzyżowanie(maków, bratków),
miejsce(lalka, teatr), miejsce(sienkiewicz, pomnik),
blisko(lalka, bratków), blisko(sienkiewicz, maków).

Zauważmy, że stwierdziliśmy istnienie skrzyżowania ulicy Kwiatowej z ulicą Maków, ale nie mamy informacji, że istnieje skrzyżowanie ulicy Maków z ulicą Kwiatową. Aby nie wypisywać wszystkich takich informacji możemy dodać regułę[4]:

skrzyżowanie(U1, U2):- skrzyżowanie(U2, U1).

Załóżmy, że jesteśmy zainteresowani uzyskiwaniem informacji, czy z danej ulicy możemy dotrzeć do zadanego interesującego miejsca. Aby to było możliwe, musimy mieć dodatkową relację można(U, N), stwierdzającą, że z ulicy U można dotrzeć w pobliże miejsca N. Aby zdefiniować tę relację, potrzebujemy jeszcze wiedzy, czy z danej ulicy można dojechać na podaną ulicę:

dojazd(U1, U2):- U1 = U2.
dojazd(U1, U2):- skrzyżowanie(U1, U3), ulica(U3, przejezdna),
dojazd(U3, U2).

Pierwsza reguła mówi, że z danej ulicy można dojechać na nią samą. Druga stwierdza, że można dojechać z ulicy U1 na ulicę U2, jeśli istnieje przejezdna ulica U3 mająca skrzyżowanie z U1, z której to ulicy U3 można dojechać na ulicę U2.

Relację „można" opisujemy teraz regułą (dlaczego jest poprawna?):

można(U, N):- dojazd(U, U1), blisko(N, U1).

Teraz jesteśmy gotowi do zadawania wiele interesujących pytań, np.:

- miejsce(N, teatr), blisko(N, U) – wyszukaj pary N = nazwa teatru U = nazwa ulicy, takie że N jest blisko U,
- miejsce(N, teatr), można(U, N) – wyszukaj pary N = nazwa teatru U = nazwa ulicy, takie że z ulicy U można dojechać w pobliże N.

Mamy więc mini system doradczy dla przewodnika po mieście.

[4] Ze względu na przyjęte mechanizmy obliczeniowe taka reguła zawsze działa w języku Datalog, ale nie we wszystkich implementacjach języka Prolog, którego obliczenia mogą się tu zapętlić.

Na zakończenie podajmy jeszcze dwie wskazówki, jak tworzyć reguły zawierające alternatywę wśród przesłanek oraz koniunkcję we wniosku. Otóż często chciałoby się mieć reguły postaci:

W:- *A* lub *B*.
R i *P*:- *Q*.

Ani Prolog, ani Datalog nie pozwalają na wyrażanie takich reguł bezpośrednio. Możemy jednak skorzystać z praw logiki i uzyskać reguły równoważne powyższym. Po pierwsze, następująca formuła jest zawsze prawdziwa:

[(*A* lub *B*) implikuje *W*] jest równoważne [(*A* implikuje *W*) i (*B* implikuje *W*)].

Zapisując powyższą równoważność w języku rachunku zdań uzyskamy:

$$[(A \vee B) \Rightarrow W] \Leftrightarrow [(A \Rightarrow W) \wedge (B \Rightarrow W)].$$

Możemy teraz sprawdzić prawdziwość tej równoważności używając metody tablic logicznych lub rezolucji.

Regułę „*W*:- *A* lub *B*" możemy więc zastąpić dwoma regułami zgodnymi ze składnią języka Prolog/Datalog:

W:- *A*.
W:- *B*.

Dla uzyskania drugiego rodzaju reguł skorzystamy z innego prawa logicznego:

[*Q* implikuje (*R* i *P*)] jest równoważne [(*Q* implikuje *R*) i (*Q* implikuje *P*)],

co znów można sprawdzić w rachunku zdań, ponieważ powyższą formułę można w sposób naturalny zapisać jako:

$$[Q \Rightarrow (R \wedge P)] \Leftrightarrow [(Q \Rightarrow R) \wedge (Q \Rightarrow P)].$$

Reguła „*R* i *P*:- *Q*" jest więc równoważna następującym dwóm regułom:

R:- *Q*.
P:- *Q*.

Na przykład:

rodzic(X, Y):- matka(X, Y) lub ojciec(X, Y)

zapisujemy jako reguły:

rodzic(X, Y):- matka(X, Y).
rodzic(X, Y):- ojciec(X, Y).

Natomiast:

silny(X) i zdrowy(X):- wysportowany(X)

zapisujemy jako:

silny(X):- wysportowany(X).
zdrowy(X):- wysportowany(X).

11. Podsumowanie

Na stosunkowo niewielu stronach tego rozdziału przeszliśmy w rzeczywistości bardzo długą drogę: od modelowania i wnioskowania w klasycznym rachunku zdań (o źródłach jeszcze w starożytności) po języki regułowe, dla których impulsem była metoda rezolucji opublikowana przed niecałymi pięćdziesięcioma laty. Od uzasadniania poprawności rozumowań przeszliśmy do wnioskowania na podstawie baz wiedzy na gruncie rachunku zdań po to, by pokazać mechanizmy programowania w logice i powrócić do baz wiedzy w języku bardziej zaawansowanym niż rachunek zdań.

Rachunek zdań, choć wydaje się prosty, swój obecny kształt uzyskał przed prawie stu pięćdziesięciu laty. Trudno przecenić jego rolę i zakres zastosowań. Wielu naukowców poświęca całą swoją aktywność badawczą np. na poszukiwanie efektywnych systemów wnioskowania dla wybranych rodzajów formuł, pojawiających się w danych zastosowaniach w wyniku modelowania lub tłumaczenia łatwiejszych w użyciu formalizmów. W końcu jeden z kilku problemów milenijnych, za rozwiązanie którego czeka nagroda w wysokości miliona dolarów (http://www.claymath.org/millennium/P_vs_NP/), jest równoważny wykazaniu istnienia lub nieistnienia efektywnego algorytmu badania, czy dana formuła klasycznego rachunku zdań jest spełnialna[5].

Mimo swojej siły rachunek zdań nie jest w stanie dobrze modelować złożonej rzeczywistości systemów informatycznych, języka naturalnego, reprezentacji wiedzy czy wnioskowania o świecie rzeczywistym. W minionych czterdziestu latach w informatyce prowadzono bardzo intensywne badania nad logikami, np. modelującymi wnioskowanie człowieka lepiej niż klasyczny rachunek zdań (znanymi w sztucznej inteligencji jako wnioskowanie zdroworozsądkowe lub niemonotoniczne). Dziedzina ta nadal intensywnie się rozwija, znajdując zastosowania w systemach autonomicznych, jak bezzałogowe helikoptery czy złożone systemy robotyki.

Jednym z ważnych rozszerzeń rachunku zdań są języki regułowe. Reguły odpowiadają implikacji, jednak mamy w nich dużo bogatszy repertuar środków wyrazu dzięki relacjom i wyrażeniom funkcyjnym. Języki regułowe są jednym z najważniejszych podejść realizujących paradygmat programowania deklaratywnego, a więc takiego, w którym opisujemy cel do obliczenia, zamiast podawania algorytmu precyzującego krok po kroku obliczenia prowadzące do tego celu. Są one stosowane w systemach sztucznej inteligencji, w tym w systemach doradczych, bazach wiedzy czy w analizie języka naturalnego. Jako narzędzie edukacyjne umożliwiają one wprowadzenie podstawowych zasad logiki wnioskowania. Są też skutecznym narzędziem nauki myślenia rekurencyjnego.

[5] Więcej o problemach milenijnych można przeczytać w rozdziale *Czy wszystko można obliczyć. Łagodne wprowadzenie do złożoności obliczeniowej.*

Świat logik jest bardzo bogaty. Logika występuje praktycznie w każdej dziedzinie życia i nauki i choć jednym z najważniejszych obszarów jej zastosowań jest informatyka, nie można zapomnieć o jej użyciu w codziennych życiowych aktywnościach, naukach ścisłych, ale i w humanistycznych. Trudno sobie bowiem wyobrazić jakiekolwiek prace w filozofii, historii czy prawie bez rygorystycznych zasad stojących za przeprowadzanymi tu rozumowaniami.

Literatura

1. Kowalski R., *Logika w rozwiązywaniu zadań*, WNT, Warszawa 1989
2. Niederliński A., *Programowanie w logice z ograniczeniami. Łagodne wprowadzenie dla platformy ECLiPSe*, Wydawnictwo PKJS, Gliwice 2012 (istnieje również darmowa wersja pierwszego wydania, udostępniona przez Autora: http://www.pwlzo.pl/download/PWLZO_zab.pdf)
3. Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa 2004

Prof. dr hab. Andrzej Szałas

zajmuje się logicznymi podstawami informatyki i sztucznej inteligencji, specjalizując się w nieklasycznych metodach wnioskowania, w tym w językach regułowych. Jego wcześniejsze badania dotyczyły też systemów operacyjnych, języków obiektowych oraz semantyki współbieżności. W roku 1980 ukończył Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego. Następnie odbył studia doktoranckie w Instytucie Matematycznym Polskiej Akademii Nauk. Stopnie doktora (1984) oraz doktora habilitowanego (1991) uzyskał na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego. W roku 1999 otrzymał tytuł naukowy profesora nauk matematycznych w zakresie informatyki. Obecnie jest zatrudniony na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego oraz w Department of Computer and Information Science na Uniwersytecie w Linköpingu (Szwecja). Pracował też jako profesor lub profesor wizytujący na polskich i zagranicznych uczelniach. Opublikował 6 książek oraz ponad 100 artykułów naukowych.

andrzej.szalas@mimuw.edu.pl