2021

Metody Sztucznej Inteligencji. 2. System logicznego wnioskowania

2. SYSTEM LOGICZNEGO WNIOSKOWANIA

WŁODZIMIERZ KASPRZAK

SKŁADNIA I SEMANTYKA JĘZYKA PREDYKATÓW, REGUŁY WNIOSKOWANIA, WNIOSKOWANIE WPROST, REZOLUCJA, SYSTEMY LOGICZNE W PRAKTYCE

W drugim module bloku MSI przedstawiony zostanie system logicznego wnioskowania bazujący na języku predykatów jako języku reprezentacji wiedzy. Opisane zostaną składnia i semantyka języka predykatów (język logiki pierwszego rzędu). Wprowadzone zostaną, uogólnione względem rachunku zdań a właściwe dla logiki predykatów, reguły wnioskowania: „Modus Ponens” i „Rezolucja”. Wymagają one uzgadniania formuł w poprzedniku każdej reguły poprzez znalezienie odpowiednich podstawień pod zmienne występujące w tych formułach. Przedstawione zostaną algorytmy wnioskowania korzystające z tych reguł – wnioskowanie wprost (i jej odmiany „w przód” lub „wstecz”) korzysta z reguły „Modus Ponens” a wnioskowanie przez zaprzeczenie korzysta z reguły „Rezolucja”. Omówione zostaną systemy logicznego wnioskowania występujące w praktyce: PROLOG, logika deskrypcyjna, sieci semantyczne, systemy regułowe. Wspomniane będą rozszerzenia logiki klasycznej: logiki niemonotoniczne i modalne.

Spis treści

1 Język predykatów

Podczas gdy rachunek zdań zakłada, że świat składa się z faktów, **logika pierwszego rzędu** (**logika predykatów**), podobnie jak język naturalny, pozwala specyfikować te fakty znacznie dokładniej.

1.1 Składnia języka predykatów

1.1.1 Kategorie symboli języka

W logice predykatów zakładamy, że w modelowanym świecie występują:

- Obiekty - Np. osoby, domy, liczby, kolory, gry, wojny, …
- Relacje - Np. jest czerwony, jest okrągły, jest liczbą pierwszą, jest bratem, większy niż, jest częścią, jest pomiędzy, …
- Funkcje - Np. jego ojciec, jego najlepszy przyjaciel, o jeden więcej, suma, …

Pozwala to wyrazić fakty jako relacje zachodzące na obiektach wyznaczanych przez funkcje. Stąd składnia języka predykatów obejmuje następujące zasadnicze elementy:

- Stałe np. *Jan, 2, PW*,...
- Symbole predykatów np. *Brat, >,...*
- Symbole funkcji np. *Sqrt, LewaNoga*,...
- Zmienne *x, y, a, b*,...
- Negacja i spójniki $\neg, \Rightarrow, \wedge, \vee, \Leftrightarrow$
- Predykat równości =
- Kwantyfikatory (dla wyrażenia własności zbioru obiektów) $\forall, \exists$

1.1.2 Wyrażenia – termy i formuły

Z elementów składni języka tworzone są wyrażenia języka: termy i formuły. **Termy** wskazują na obiekty i przyjmują postać funkcji, stałej lub zmiennej:
funkcja (*term*$_1$,...,*term*$_n$) lub *stała* lub *zmienna .*
Formuła atomowa to wyrażenie zbudowane na pojedynczym predykacie, tzn. o ogólnej postaci:
predykat (*term*$_1$,...,*term*$_n$) lub *term*$_1$ = *term*$_2$. Predykat „=" („równość") został wyróżniony dlatego, gdyż we wszystkich zastosowaniach języka powinien posiadać podobne znaczenie.
Przykłady:

Term: *Brat*(*Jan*)*;*
Formuła atomowa*: >(Długość(LewaNoga(Andrzej)), Długość(LewaNoga(Jan)))*

Formuły złożone powstają z połączenia formuł atomowych spójnikami z możliwością wykorzystania kwantyfikatorów i negacji:

$\neg S$, $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, $S_1 \Leftrightarrow S_2$,

Przykłady formuł:

Rodzeństwo(*Jan, Andrzej*) $\Rightarrow$ *Rodzeństwo*(*Andrzej, Jan*)
$>(1,2) \vee \leq (1,2)$
$>(1,2) \wedge \neg >(1,2)$

1.1.3 Kwantyfikatory

Ogólna postać uniwersalnego kwantyfikatora to:

$\forall$<*zmienne*> <formuły>

Np. zdanie „*Każda osoba studiująca na PW jest inteligentna*" zapiszemy z użyciem kwantyfikatora jako: $\forall x\ (Studiuje(x, PW) \Rightarrow Inteligentna(x))$.

Wartościowanie formuły z kwantyfikatorem ogólnym może my wyrazić następująco: formuła $\forall x\ P$ jest prawdziwa w modelu *M* wtw. gdy *P* jest prawdziwe dla *x* wartościowanego dowolnym obiektem w tym modelu. W przybliżeniu jest to równoważne koniunkcji wszystkich możliwych wartościowań *P:* $(Studiuje(Jan, PW) \Rightarrow Inteligentna(Jan)) \wedge (Studiuje(Andrzej, PW) \Rightarrow Inteligentna(Andrzej)) \wedge (Studiuje(PW, PW) \Rightarrow Inteligentna(PW)) \wedge ...$

Egzystencjalny kwantyfikator jest ogólnej postaci

$\exists$<*zmienne*> <formuły>

Np. zdanie „*Ktoś spośród osób studiujących na PW jest inteligentny*" zapiszemy jako:

$\exists x\ (Studiuje(x, PW) \wedge Inteligentna(x))$

Formuła $\exists x\ P$ jest prawdziwa w modelu *M* wtw. gdy *P* jest prawdziwe dla *x* wartościowanego jakimś obiektem modelu. W przybliżeniu jest to równoważne alternatywie różnych wartościowań *P*:

$(Studiuje(Jan, PW) \wedge Inteligentna(Jan)) \vee (Studiuje(Andrzej, PW) \wedge Inteligentna(Andrzej)) \vee (Studiuje(PW, PW) \wedge Inteligentna(PW)) \vee ...$

Własności kwantyfikatorów:

- Wyrażenie $\forall x\ \forall y$ jest równoważne z $\forall y\ \forall x$
- $\exists x\ \exists y$ jest równoważne z $\exists y\ \exists x$
- $\exists x\ \forall y$ **nie** jest równoważne $\forall y\ \exists x$

Np. formuła, $\exists x\ \forall y$ *Kocha*(x, y), oznaczająca "*Istnieje osoba, która kocha wszystkich na świecie*", nie jest tożsama z formułą, $\forall y\ \exists x$ *Kocha*(x,y), oznaczającą "*Każdy na świecie jest kochany przez przynajmniej jedną osobę*".

Dualność kwantyfikatorów oznacza, że wyrażenie z jednym kwantyfikatorem może zostać przekształcone na równoważne wyrażenie z drugim kwantyfikatorem (stosując podwójną negację). Np.:

$\forall x\ Lubi(x, lody) \equiv \neg\exists x\ \neg Lubi(x, lody)$

$\exists x\ Lubi(x, brokuły) \equiv \neg\forall x\ \neg Lubi(x, brokuły)$

1.2 Semantyka języka predykatów

1.2.1 Model i wartościowanie zmiennych

W przypadku języka predykatów znaczenie formuły określamy ze względu na: ustaloną **dziedzinę** *D,* **funkcję interpretacji** *m* i funkcję **wartościowania** *a*. **Model** zbioru formuł nadal jest postaci, *M* = [*D*, *m*], ale aby wyznaczyć wartość formuły należy też wartościować jej zmienne, a to wymaga podania funkcji wartościowania *a*.

Dziedzina *D* zawiera obiekty (elementy dziedziny) i relacje oraz funkcje pomiędzy nimi. Funkcja interpretacji *m* przyporządkowuje:

- symbolom stałych → obiekty, czyli elementy dziedziny;
- - *n*-arg. symbolom predykatów → *n*-argumentowe relacje określone na zbiorze D^n;
- - n-arg. symbolom funkcyjnym → funkcje ze zbioru D^n na zbiór *D.*

Formuła atomowa o postaci *predykat*($term_1,...,term_n$) *jest prawdziwa* (*True*) w interpretacji *m* wtw. gdy obiekty wyznaczone przez ($term_1,...,term_n$) spełniają relację referowaną przez *predykat.*
W jaki sposób wyznaczamy obiekt będący wartością termu? Przy interpretowaniu formuły w danym modelu nadawanie wartości zmiennym tej formuły określa się mianem **wartościowania**. Wartościowanie zmiennych *a* w zadanym modelu *M* to odwzorowanie symboli zmiennych na elementy dziedziny. Oznaczmy przez $V_a^M(t)$ wartość termu *t* w modelu *M* = [*D, m*] względem wartościowania *a*. Wartość stałej lub zmiennej obliczamy jako:

- $V_a^M(x) = a(x)$, gdzie *x* jest zmienną.
- $V_a^M(C) = m(C)$, gdzie *C* jest stałą.

Wartość termu $f(t_1,...,t_n)$ wynosi:

- $V_a^M(f(t_1,...,t_n)) = m(f)(V_a^M(t_1),..., V_a^M(t_n))$.

Oznaczmy przez $V_a^M(\tau)$ wartość formuły τ w modelu *M,* gdzie *M* = [*D, m*], względem wartościowania *a*. Mamy:

- Dla predykatu *P*: $V_a^M(P(t_1,...,t_n)) = m(P)(V_a^M(t_1),..., V_a^M(t_n))$.
- Dla równości: $V_a^M(t_1 = t_2)) = (V_a^M(t_1) = V_a^M(t_2))$.
- Dla formuł złożonych:

$$V_a^M(B \wedge C) = V_a^M(B) \wedge V_a^M(C)$$
$$V_a^M(B \vee C) = V_a^M(B) \vee V_a^M(C)$$
$$V_a^M(B \Rightarrow C) = V_a^M(B) \Rightarrow V_a^M(C)$$
$$V_a^M(B \Leftrightarrow C) = V_a^M(B) \Leftrightarrow V_a^M(C)$$

- Dla kwantyfikatorów:

$$V_a^M(\forall x\ B) = \min_{d \in D}[V_{a(x \leftarrow d)}^M(B)]$$
$$V_a^M(\exists x\ B) = \max_{d \in D}[V_{a(x \leftarrow d)}^M(B)]$$

gdzie *min*(True, False) wynosi False, a *max*(True, False) wynosi True.
Przyjmujemy, że $a(x \leftarrow d)$ oznacza wartościowanie identyczne z *a* dla wszystkich zmiennych poza *x*, w którym zmiennej *x* nadawana jest wartość *d*. Czyli $a(x \leftarrow d)$ różni się od *a* co najwyżej wartościowaniem zmiennej *x*.

1.2.2 Rodzaje formuł

W dalszym ciągu badamy znaczenie formuł względem ustalonej dziedziny *D*. Powiemy, że formuła *A* jest **spełnialna** jeśli istnieje interpretacja *m* i wartościowanie *a* przy których *A* jest spełniona (tzn. ma wartość *True*).
A jeśli jej prawdziwość nie zależy od wartościowania? Formuła *A* jest **prawdziwa** w interpretacji *m* jeśli ma wartość *True* w tej interpretacji przy każdym wartościowaniu *a*. Mówimy wtedy, że A jest **aksjomatem** dziedziny [*D*, *m*].

A co jeśli jej prawdziwość nie zależy od funkcji interpretacji? Mówimy, że formuła *A* jest **tautologią** jeśli jest *prawdziwa* przy każdej interpretacji *m* i wartościowaniu *a* (tzn. jest zawsze prawdziwa dla każdej dziedziny *D*).

Teoria jest to zbiór formuł tworzony dla określonej dziedziny. Teoria jest **niesprzeczna** jeśli istnieją: interpretacja i funkcja wartościująca, dla których prawdziwe są wszystkie formuły teorii.

Aksjomaty teorii to formuły (uznane za) prawdziwe niezależnie od wartościowania. Interesują nas teorie, których formuły dadzą się wyprowadzić w procesie wnioskowania używając aksjomatów i podzbioru formuł początkowej bazy wiedzy jako „punktów startowych" wnioskowania.

Np. aksjomaty w teorii (w świecie) „Osoby spokrewnione":

„Bracia są rodzeństwem" : $\forall x,y\ Brat(x,y) \Leftrightarrow Rodzeństwo(x,y)$

„Matka jest rodzicem i kobietą" : $\forall m,c\ Matka(c) = m \Leftrightarrow (Kobieta(m) \wedge Rodzic(m,c))$

"Rodzeństwo" jest symetryczną relacją : $\forall x,y\ Rodzeństwo(x,y) \Leftrightarrow Rodzeństwo(y,x)$

Równość jest wyróżnionym predykatem, który w każdej interpretacji ma to samo znaczenie – oznacza on relację identyczności, tzn. ($term_1 = term_2$) jest *prawdziwe* wtw. gdy $term_1$ i $term_2$ referują ten sam obiekt. Z predykatu równości korzystamy często opisując własności funkcji lub definiując predykaty w terminach innych funkcji lub relacji, poprzez wprowadzenie wymogu równości lub różności obiektów. Np. poniższa definicja predykatu *Rodzeństwo* korzysta z równości termów i prawdziwości relacji *Rodzic*:

$\forall x,y\ Rodzeństwo(x,y) \Leftrightarrow [\neg(x = y) \wedge \exists m,f\ \neg(m = f) \wedge Rodzic(m,x) \wedge Rodzic(f,x) \wedge Rodzic(m,y) \wedge Rodzic(f,y)]$

1.3 Przekształcanie formuł

1.3.1 Postać predykatowa

Czy możemy dla wnioskowania formuł stosować te same reguły wnioskowania co dla rachunku zdań? Tak, jeśli sprowadzimy formuły do tzw. postaci *predykatowej* (zdaniowej), tzn. wyeliminujemy zmienne i termy.

Załóżmy, że KB składa się z formuły złożonej skwantyfikowanej uniwersalnie,

$\forall x$ Człowiek(x) $\wedge$ Chciwy(x) $\Rightarrow$ Zły(x),

i z literałów (faktów): Człowiek(Jan) , Chciwy(Jan) , Bracia(Andrzej, Jan).

Wartościując formułę uniwersalną wszystkimi możliwymi (znanymi) obiektami otrzymamy bazę wiedzy KB:

Człowiek(Jan) $\wedge$ Chciwy(Jan) $\Rightarrow$ Zły(Jan)

Człowiek(Andrzej) $\wedge$ Chciwy(Andrzej) $\Rightarrow$ Zły(Andrzej)

Człowiek(Jan) , Chciwy(Jan) , Bracia(Andrzej, Jan)

Tak powstała KB nie zawiera zmiennych. Symbolami *predykatowymi* o charakterze zdań są: Człowiek(Jan), Chciwy(Jan), Zły(Jan), Człowiek(Ryszard), itd. Można je traktować jak symbole zdaniowe i stosować wnioskowanie w rachunku zdań. Prawdziwe jest twierdzenie:

„Każda baza wiedzy KB wyrażona w L1R może zostać przekształcona do postaci *predykatowej*, zachowującej własność wynikania, czyli: bazowa formuła wynika z nowej KB wtw. gdy wynika z oryginalnej KB."

Jednak powstaje zasadniczy problem związany z potencjalnie nieskończoną liczbą symboli predykatowych, które mogą powstać ze skończonej liczby formuł. Z powodu istnienia symboli funkcji istnieje nieskończenie wiele termów. Np. Ojciec(Jan), …, Ojciec(Ojciec(Ojciec(Jan))) …
Dlatego też w logice predykatów stosuje się **uogólnione reguły wnioskowania**, tzn. reguły znane nam z rachunku zdań zostają uogólnione dzięki mechanizmom: **podstawienia** pod zmienne i **unifikacji** formuł (uzgadniania zmiennych).

1.3.2 Podstawienie pod zmienne

Rozszerzymy teraz reguły wnioskowania znane nam z rachunku zdań do odpowiednich postaci wymaganych przez logikę predykatów.
Przez **podstawienie** rozumiemy zbór par postaci $\{x_1/t_1,...,x_n/t_n\}$, gdzie $x_1,...,x_n$ są różnymi zmiennymi, natomiast, $t_1,...,t_n$, są termami. Dopuszczamy podstawienie puste ε.
Niech θ będzie podstawieniem i niech α będzie wyrażeniem (tzn. formułą lub termem). Przez **SUBST**(θ , α) oznaczamy wyrażenie powstałe w wyniku jednoczesnego zastąpienia wszystkich wolnych wystąpień zmiennych, $x_1,...,x_n$, termami, $t_1,...,t_n$.
Np. jeśli $\theta = \{x/\text{Jan}, y/\text{Ewa}\}$, to

SUBST(θ, *Lubi*(x, y)) = { *Lubi*(Jan, Ewa) }.

Przypomnijmy, że literał jest to formuła atomowa lub negacja formuły atomowej, a zdanie - formuła bez zmiennych.
Przykład. Stosując predykatowanie generujemy zwykle wiele niepotrzebnych formuł.
Np. dla zbioru formuł: { $\forall x$ *Człowiek*(x) $\wedge$ *Chciwy*(x) $\Rightarrow$ *Zły*(x) , *Człowiek*(*Jan*) , $\forall y$ *Chciwy*(y) , *Bracia*(*Andrzej, Jan*) } wydaje się być oczywiste, że istnieje model, w którym zachodzi *Zło*(*Jan*), ale predykatowanie produkuje również szereg innych zdań, takich jak *Chciwy*(*Andrzej*), które nie mają modelu – nigdzie nie są spełnione.
W ogólności, gdy dziedzina liczy p k-argumentowych predykatów i n obiektów to istnieje $p \cdot n^k$ możliwych wartościowań. Łatwo unikniemy tego nadmiaru zdań, jeśli w jednym kroku wnioskowania znajdziemy podstawienie θ takie, przy którym formuły *Człowiek*(x) i *Chciwy*(y) są równoważne w predykatowej postaci. Dla *Człowiek*(*Jan*) i *Chciwy*(y) zapewnia nam to podstawienie $\theta = \{x/Jan, y/Jan\}$.

1.3.3 Uzgadnianie zmiennych i unifikacja formuł

Obecność symboli zmiennych w logice predykatów powoduje to, że nie możemy poprzestać na prostym wymaganiu identyczności formuł tworzących poprzednik reguły wnioskowania. Stosujemy łagodniejsze wymaganie takie, że pierwsza i druga formuła wejściowa (tworzące poprzednik reguły wnioskowania) są **unifikowalne**, czyli mogą zostać sprowadzone do postaci identycznej przez zastosowanie odpowiednich podstawień dla zmiennych - przypisanie im obiektów lub pewnych termów. W istocie mamy do czynienia z dwoma przypadkami **uzgadniania zmiennych**:

1. *Ujednolicanie* zmiennych – wtedy, gdy dwie formuły różnią się jedynie tym, że w odpowiednich miejscach występują w nich konsekwentnie inne symbole zmiennych,

2. *Uszczegółowienie* – wtedy, gdy w miejscach, gdzie w jednej z nich występuje pewien symbol zmiennej (związany kwantyfikatorem ogólnym), w drugiej konsekwentnie występuje pewien term nie będący zmienną (stała albo zastosowanie symbolu funkcyjnego).

Mówimy, że podstawienie θ jest **unifikatorem** wyrażeń, E_1,...,E_n , jeśli SUBST(θ, E_1) = ... = SUBST(θ, E_n). Np. podstawienie {x/A} jest unifikatorem wyrażeń P(x), P(A). Z kolei wyrażenia P(x), Q(b) nie są unifikowalne.

Reguły wnioskowania w logice predykatów stosują **algorytm unifikacji** (nazwijmy go UNIFY(p, q)), który zwraca takie podstawienie (tzw. **najogólniejszy unifikator**) dla literałów p i q, które czyni je *identycznymi*, lub zwraca *błąd*, jeśli podstawienie nie istnieje.

Wyjaśnijmy czym jest najogólniejszy unifikator? **Złożeniem** podstawień θ_1, θ_2 jest takie podstawienie (oznaczmy je przez COMPOSE(θ_1, θ_2)), które polega na wykonaniu po kolei obu podstawień, czyli: SUBST(COMPOSE(θ_1 , θ_2) , E) = SUBST(θ_2 , SUBST(θ_1 , E)).

Podstawienie θ jest **najogólniejszym** unifikatorem dla {E_1,..., E_n}, jeśli dla każdego unifikatora γ dla tych wyrażeń istnieje podstawienie λ takie, że: γ = COMPOSE(θ, λ). Innymi słowy najogólniejszy unifikator zawiera jedynie to co niezbędne dla unifikacji dwóch literałów, na których działa. Ważną dodatkową obserwacją jest to, że unifikowalne wyrażenia posiadają dokładnie jeden najogólniejszy unifikator.

Przykład. Wyrażenia *P*(Jan, x), *P*(y, z) posiadają nieskończenie wiele unifikatorów. Niektóre z nich to: {y/Jan, x/z}, {y/Jan, x/z, w/Fred}, {y/Jan, x/Jan, z/Jan}. Najogólniejszym unifikatorem jest: {y/Jan, x/z}

Problem uzgadniania zmiennych w wyrażeniach, E_1,..., E_n , polega najpierw na zbadaniu, czy dane wyrażenia są unifikowalne, a jeśli tak to należy znaleźć ich najogólniejszy unifikator. Problem unifikacji jest rozstrzygalny. Wynika to ze skończonej liczby zmiennych w literałach.

Przykład unifikacji literałów:

p = Zna(Jan, x);	q = Zna(Jan, Ewa);	UNIFY(p,q) =	{x/Ewa}
p = Zna(Jan, x)	q = Zna(y, Ewa)		{x/Ewa, y/Jan}
p = Zna(Jan, x)	q = Zna(y, Matka(y))		{y/Jan, x/Matka(Jan)}
p = Zna(Jan, x)	q = Zna(x, Ewa)		*błąd*

1.3.4 Standaryzacja rozłączna

Możemy otrzymać równoważne warianty zadanej formuły zmieniając nazwy zmiennych tej formuły (*przemianowanie zmiennych*). Powiemy, że formuła A jest **wariantem** formuły B, jeśli A można otrzymać z B zastępując niektóre zmienne w B nowymi zmiennymi nie występującymi w B. Np. *Lubi*(x, Jan), jest wariantem formuły, *Lubi*(y, Jan) .

W zasadzie każda formuła dodawana do bazy wiedzy może zostać zastąpiona swoim wariantem (przemianowana) przed takim dodaniem. Jednak dlaczego należy przemianowywać zmienne w formule dodawanej do bazy wiedzy? Jest to motywowane potrzebą uniknięcia błędów unifikacji, które powodowane są przez powtarzanie się nazw zmiennych w rożnych formułach, w sytuacji, gdy w oczywisty sposób nie muszą się one odnosić do tego samego obiektu.

Np. załóżmy, że w istnieją dwie formuły, *Zna*(*x, Ewa*) , *Zna*(*Jan, x*) $\Rightarrow$ *Nienawidzi*(*Jan, x*) . Powinniśmy móc wywnioskować na podstawie tej pary, stosując ogólną regułę *Modus Ponens*, że zachodzi *Nienawidzi*(*Jan, Ewa*). W praktyce nie jest to jednak możliwe, gdyż formuły, *Zna*(*Jan, x*) i *Zna*(*x, Ewa*),

nie są unifikowalne. Dopiero gdy przemianujemy jedną ze zmiennych *x* (np. formułę *Zna*(*x, Ewa*), zamienimy na *Zna*(*y, Ewa*)) to będziemy mogli wyprowadzić *Nienawidzi*(*Jan, Ewa*).
Proces przemianowania zmiennych występujących w formule tak, aby posiadała ona unikalne zmienne względem występujących w bazie wiedzy, nazywamy **standaryzacją rozłączną** formuły.

1.4 Eliminacja kwantyfikatorów

W bazie wiedzy występują formuły pozbawione kwantyfikatorów. Formuła definiowana przez eksperta, w której zmienne związane są kwantyfikatorami, może zostać przekształcona do równoważnej postaci formuły pozbawionej kwantyfikatorów. Zasada eliminacji kwantyfikatorów zakłada wykonanie następujących kolejnych kroków:

1. Standaryzacja rozłączna zmiennych dla kwantyfikatorów – jeśli w formule złożonej występuje powtórne wiązanie tej samej zmiennej przez inny kwantyfikator to należy przemianować każdą kolejną zmienną i odpowiednio jej wystąpienia w formule tak, aby każdy kwantyfikator wiązał unikalną zmienną.
2. Skolemizacja formuły – jest to operacja eliminacji kwantyfikatorów szczegółowych (egzystencjalnych) zachowująca równoważność formuły.
3. Eliminacja (a w zasadzie opuszczenie) kwantyfikatorów uniwersalnych.

1.4.1 Eliminacja kwantyfikatora szczegółowego

Eliminacja kwantyfikatora szczegółowego nosi nazwę **skolemizacji formuły**. Rozróżnimy tu dwa przypadki skolemizacji, zależnie od tego czy w formule występuje przynajmniej jeden kwantyfikator uniwersalny poprzedzający kwantyfikator egzystencjalny, czy też nie.
Eliminacja kwantyfikatora egzystencjalnego nie poprzedzonego żadnym kwantyfikatorem uniwersalnym polega na zastosowaniu następującej reguły wnioskowania:
Dla każdej formuły α, zmiennej *v*, i symbolu stałej *K*, który nie występuje nigdzie indziej w bazie wiedzy, zachodzi reguła wnioskowania o postaci:

$$\frac{\exists v\ \alpha}{SUBST(\{v/K\}, \alpha)}$$

Np. z formuły, ∃x Kapelusz(x) ∧ NaGłowie(x, Jan), wynika, Kapelusz(C_1) ∧ NaGłowie(C_1,Jan), pod warunkiem, że C_1 jest nowym symbolem stałej, zwanej „stałą Skolema".
Jeśli kwantyfikator szczegółowy poprzedzony jest kwantyfikatorem uniwersalnym zmiennej *x* to za *v* podstawiamy unikalny symbol funkcji (np. *F*(*x*)), zwanej „funkcją Skolema" o parametrze *x*:

$$\frac{\forall x\ \exists v\ \alpha}{SUBST(\{v/F(x)\}, \alpha)}$$

1.4.2 Eliminacja uniwersalnego kwantyfikatora

Każde wartościowanie formuły związanej uniwersalnym kwantyfikatorem wynika z tej formuły. Możemy to zapisać w postaci reguły wnioskowania:

$$\frac{\forall_v\ \alpha}{SUBST(\{v/g\}, \alpha)}$$

dla każdej zmiennej *v* i bazowego termu *g* (bazowy term oznacza występowanie co najwyżej jednej funkcji w termie). Tym samym kwantyfikator uniwersalny nie nakłada ograniczeń na wartościowanie związanej nim zmiennej. Po uprzednim unikalnym przemianowaniu zmiennych przy

kwantyfikatorach i po eliminacji ewentualnych kwantyfikatorów szczegółowych w formule, opuszczamy kwantyfikatory uniwersalne.

Np. z formuły , $\forall$x Osoba(x) $\wedge$ Chciwy(x) $\Rightarrow$ Zły(x), wynikają między innymi zdania:

Osoba(Jan) $\wedge$ Chciwy(Jan) $\Rightarrow$ Zły(Jan)

Osoba(Andrzej) $\wedge$ Chciwy(Andrzej) $\Rightarrow$ Zły(Andrzej)

1.5 Rachunek sytuacji w logice predykatów

Wprowadzimy teraz podzbiór języka predykatów, określany jako rachunek sytuacji („*situation calculus*") dla modelowania zjawisk i własności zmiennych w czasie.

1.5.1 Przykład agenta

Prosty agent reaktywny będzie posiadał jedynie reguły wiążące bezpośrednio aktualne obserwacje i akcje. Np. dla agenta w świecie Wumpusa reguła wyrażona w logice predykatów, wiążąca aktualną obserwację błysku złota w chwili *t* z wyborem właściwej akcji „Podnieś", może przyjąć postać dwóch formuł języka:

- formuła dla obserwacji (percepcji): $\forall_{s,\,w,\,u,\,k,\,t}$ *Percepcja*([*s, w,* Błysk*, u, k*], *t*) $\Rightarrow$ *PrzyZłocie*(*t*)
- formuła dla wyboru akcji (refleks): $\forall_{t}$ *PrzyZłocie*(*t*) $\Rightarrow$ *NajlepszaAkcja*(Podnieś, *t*)

Dzięki wprowadzeniu zmiennych reprezentacja w logice predykatów jest znacznie efektywniejsza niż w rachunku zdań. Jednak prosty agent reaktywny nie poradzi sobie ze *światem Wumpusa*, gdyż: (1) nigdy nie będzie wiedział na pewno, czy lepiej jest wrócić do kwadratu startowego czy dalej szukać złota, (2) często się zapętli, gdyż nie rozróżnia on czy niesie już złoto czy też jeszcze nie. Przyczyny obu trudności leżą w braku informacji o stanie środowiska. Agent reaktywny ze stanem reaguje na bieżącą obserwację środowiska w oparciu o jego aktualnie stworzony opis (stan problemu). Baza wiedzy takiego agenta ma charakter dynamiczny – akumuluje ona wszystkie obserwacje w postaci modyfikowalnego stanu problemu.

1.5.2 Wymagane elementy

Rachunek sytuacji („*situation calculus*" - Hayes, McCarthy 1969) jest to pewien sposób opisywania zmian (w czasie) za pomocą języka logiki pierwszego rzędu, czyli do modelowania dynamicznie zmieniającego się świata. Główne jego założenia to:

1. Świat to ciąg sytuacji, z których każda opisuje stan świata w pewnym momencie czasu.
2. Nowa sytuacja powstaje z bieżącej sytuacji w wyniku wykonania akcji przez agenta.

Sytuacje

Zgodnie z tym w rachunku sytuacyjnym każdy symbol predykatu, reprezentujący relację (lub własność) zmieniającą się w czasie, będzie posiadał dodatkowy argument, określający **sytuację**. Sytuacje są to obiekty należące do specyficznej kategorii ***czasu-stanu***.

Przykład

Dla agenta w „świecie Wumpusa" wprowadzimy predykat o 3 argumentach, *Jest*(Agent*, pozycja, sytuacja*), dla wyrażenia pozycji agenta (kratki położenia i kierunku zorientowania) w określonej sytuacji w 2-wymiarowym świecie. Np. możemy zapisać: *Jest*(Agent, [(1,1), 90], S_0) $\wedge$ *Jest*(Agent, [(1,2), 90], S_1)

Funkcja następstwa sytuacji

Kolejne sytuacje (w czasie) nie są jawnie reprezentowane przez predefiniowane obiekty „czasu-stanu" ale są wynikiem poprzedniej sytuacji i wykonanej akcji. W tym celu wprowadzimy funkcję, *Rezultat*(*akcja, sytuacja*), która wyznaczy sytuację będącą wynikiem wykonania akcji w zadanej sytuacji poprzedniej. Np. wyznaczamy nową sytuację bezpośrednio występującą po sytuacji początkowej S_0 jako, S_1= *Rezultat*(RuchWPrzód, S_0) .

1.5.3 Przykład aksjomatów dziedziny

Aksjomaty efektów akcji

W nowej sytuacji będącej wynikiem wykonanej akcji będą zachodzić nowe relacje (własności) reprezentowane predykatami. Wnioskujemy je dzięki **aksjomatom efektów akcji**, zwykle zadanym w postaci formuł zbudowanych na spójniku implikacji.

Np. efektem akcji „*Podnieś złoto*" w pozycji „*x*" i sytuacji „*s*" będzie:

$$\forall_{x,s}\ PrzyZłocie(s) \wedge Jest(Agent, x, s) \Rightarrow TrzymaZłoto(Miejsce(a, x)\ ,\ Rezultat(\text{Podnieś}, s))$$

Np. efektem akcji „*Puść*" będzie:

$$\forall_{x,s}\ Jest(Agent, x, s) \Rightarrow \neg TrzymaZłoto(Miejsce(a, x), Rezultat(\text{Puść}, s))$$

Aksjomaty tła

Aksjomaty **tła** reprezentują przeniesienie relacji (własności) bez ich zmiany do następnej sytuacji.

Np. agent trzymający złoto, którego nie upuścił, w następnej sytuacji nadal będzie to złoto trzymał:

$$\forall a,x,s\ TrzymaZłoto(x, s) \wedge (a \neq \text{Puść}) \Rightarrow TrzymaZłoto(Miejsce(a, x)\ ,\ Rezultat(a, s))$$

Np. jeśli agent nie posiadał złota i go nie podniósł, to nadal go nie posiada:

$$\forall a,x,s\ \neg TrzymaZłoto(x, s) \wedge (a \neq \text{Podnieś}) \Rightarrow \neg TrzymaZłoto(Miejsce(a, x)\ ,\ Rezultat(a, s))$$

Aksjomaty następstwa stanów

Łączymy aksjomaty efektów akcji i aksjomaty tła dla tego samego predykatu w jeden aksjomat **następstwa stanów.** Zebrane są w nim wszystkie warunki dla określenia wartości danego predykatu w następnym stanie. Np. dla predykatu *TrzymaZłoto*:

$$\forall a,x,s\ TrzymaZłoto(Miejsce(a, x),\ Rezultat(a,\ s)) \Leftrightarrow (\ PrzyZłocie(s) \wedge (a=\text{Podnieś})\ ) \vee (TrzymaZłoto(x, s) \wedge (a \neq \text{Puść}))$$

2 Wnioskowanie w logice predykatów

2.1 Twierdzenia o dedukcji

W logice pierwszego rzędu również zachodzą oba znane nam już **twierdzenia o dedukcji**:

- $KB \models \alpha$ wtw. gdy $(KB \Rightarrow \alpha)$ jest tautologią.
- $KB \models \alpha$ wtw. gdy formuła $(KB \wedge \neg\alpha)$ jest niespełnialna.

Ze względu na potencjalnie nieprzeliczalną liczbę obiektów dla badanej teorii, która jest możliwa w języku logiki pierwszego rzędu, pojawia się problem z generalną rozstrzygalnością procesu wnioskowania. W logice predykatów zachodzi następujące twierdzenie:

„Problem stwierdzenia, czy dana formuła jest tautologią czy nie, jest problemem nierozstrzygalnym".

Można jedynie pokazać, że:

„istnieje algorytm, który dla zadanej formuły *A* stwierdza, że *A* jest tautologią, pod warunkiem, że tak istotnie jest."

Będziemy w stanie uogólnić reguły wnioskowania, stosowane w rachunku zdań, w taki sposób, a by stanowiły tautologie w logice predykatów i w związku z tym były poprawnymi regułami wnioskowania w tym języku logiki.

2.2 Postacie normalne formuł i reguły wnioskowania

Procedury wnioskowania zakładają, że formuły występują we właściwej postaci **normalnej**. Pozwala to zdefiniować obliczeniowo efektywną procedurę wnioskowania.

2.2.1 Klauzula Horna i uogólniona reguła odrywania

Dla procedur stosujących regułę *Modus Ponens* (zwaną także *regułą odrywania*) zdania (lub formuły) przyjmują postać tzw. **klauzul Horna**. Klauzula Horna to pojedynczy prosty literał lub implikacja o postaci: (*koniunkcja prostych literałów*) $\Rightarrow$ *prosty literał* .

„Prosty" oznacza „pozytywny", nie zanegowany.

Uogólniona reguła odrywania (*General Modus Ponens*) stosowana jest w procedurach wnioskowania dla logiki predykatów. Jest to następująca reguła:

$$\frac{p_1', \cdots, p_n', \qquad p_1 \wedge \cdots \wedge p_n \Rightarrow q}{SUBST(\theta, q)}$$

gdzie (p_1', … ,p_n' ,p_1 , ... ,p_n ,q) są literałami, a θ jest podstawieniem takim, że dla każdego i= 1,2,...,n:

SUBST(θ, p') = SUBST(θ, p).

Np. p_1'=*Król*(*Jan*), p_1= *Król*(*x*) , p_2' = *Chciwy*(*y*) , p_2 = *Chciwy*(*x*) , q = *Zło*(*x*)

θ = {*x*/*Jan*, *y*/*Jan*}, SUBST(θ, q) = *Zło*(*Jan*)

Zastosowanie uogólnionej reguły odrywania poprzedzone jest zamianą każdej formuły dodawanej do bazy wiedzy do postaci klauzul Horna, wykonywaną w następujących krokach:

1. Ewentualne przemianowanie zmiennych w formule - każda zmienna w formule związana osobnym kwantyfikatorem musi być unikalnie nazwana.
2. Eliminacja kwantyfikatorów egzystencjalnych
3. Opuszczenie kwantyfikatorów uniwersalnych
4. Przekształcenie do postaci normalnej (postać klauzulowa Horna) zgodnie z zasadami rachunku zdań.
5. Dodawanie do bazy wiedzy:
   - Standaryzacja rozłączna zmiennych formuły względem bazy wiedzy
   - Formuła o postaci koniunkcji klauzul jest przekształcana do zbioru tych klauzul (stosując „regułę eliminacji koniunkcji").

2.2.2 Postać normalna CNF i reguła rezolucji

Drugą postacią normalną dla zdań (lub formuł) jest **koniunkcyjna postać normalna** (ang. *Conjunctive Normal Form*, **CNF**) będąca koniunkcją klauzul, z których każda ma postaci alternatywy literałów.

Przykład konwersji formuły

Konwersja formuły do postaci CNF. Zdanie „*każdy kto kocha wszystkie zwierzęta jest kochany przez kogoś*" wyrazimy w postaci formuły logiki predykatów jako:

$$\forall x\,[\forall y\ Zwierz(y) \Rightarrow Kocha(x,y)] \Rightarrow [\exists y\ Kocha(y,x)].$$

1. Eliminacja obu implikacji:

$$\forall x\,[\neg\forall y\ \neg Zwierz(y) \vee Kocha(x,y)] \vee [\exists y\ Kocha(y,x)]$$

2. Przesuwamy $\neg$ w prawo: $\neg\forall x\ p \equiv \exists x\ \neg p$, $\neg\ \exists x\ p \equiv \forall x\ \neg p$

$$\forall x\,[\exists y\ \neg(\neg Zwierz(y) \vee Kocha(x,y))] \vee [\exists y\ Kocha(y,x)]$$

$$\forall x\,[\exists y\ \neg\neg Zwierz(y) \wedge \neg Kocha(x,y)] \vee [\exists y\ Kocha(y,x)]$$

$$\forall x\,[\exists y\ Zwierz(y) \wedge \neg Kocha(x,y)] \vee [\exists y\ Kocha(y,x)]$$

3. Standaryzacja rozłączna zmiennych: każdy kwantyfikator korzysta z innej zmiennej

$$\forall x\,[\exists y\ Zwierz(y) \wedge \neg Kocha(x,y)] \vee [\exists z\ Kocha(z,x)]$$

4. Skolemizacja: każda egzystencjalna zmienna (i jej kwantyfikator) jest zastępowana przez funkcję Skolema dla zmiennej należącej do poprzedzającego kwantyfikatora uniwersalnego:

$$\forall x\,[Zwierz(F(x)) \wedge \neg Kocha(x,F(x))] \vee Kocha(G(x),x)$$

5. Pomijamy uniwersalny kwantyfikator:

$$[Zwierz(F(x)) \wedge \neg Kocha(x,F(x))] \vee Kocha(G(x),x)$$

6. Rozdzielamy $\vee$ względem $\wedge$:

$$[Zwierz(F(x)) \vee Kocha(G(x), x)] \wedge [\neg Kocha(x,F(x)) \vee Kocha(G(x),x)]$$

Uzyskaliśmy postać CNF, w tym przypadku jest to iloczyn dwóch klauzul.

7. Ewentualnie z klauzul usuwamy każdy literał o postaci $\neg$True i False. Usuwamy też klauzule zawierające literał $\neg$False lub True.

Zakładając, że dodawana formuła o postaci CNF jest prawdziwa w aktualnym świecie agenta, prawdziwe są wtedy wszystkie klauzule z których składa się taka formuła. Po uprzednim sprawdzeniu **unikalności nazw zmiennych** względem zmiennych już występujących w bazie wiedzy możemy dodać aktualną formułę CNF do bazy wiedzy w postaci zbioru jej klauzul :

$$KB' = KB \cup \{ (Zwierz(F(x)) \vee Kocha(G(x), x)) , (\neg Kocha(x,F(x)) \vee Kocha(G(x),x)) \}$$

Uogólniona reguła rezolucja, właściwa dla logiki predykatów, jest postaci:

$$\frac{l_1 \vee \ldots \vee l_k\ ,\ m_1 \vee \ldots \vee m_n}{SUBST(\theta, (l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n))}$$

gdzie UNIFY(l_i, $\neg m_j$) = θ.

Zakłada się, że obie formuły w poprzedniku reguły mają postaci klauzul, tzn. wszystkie składowe l_i, m_i są literałami. Np. z pary klauzul, $\neg$*Bogaty*(*x*) $\vee$ *Nieszczęśliwy*(*x*), i , *Bogaty*(*Jan*), o unifikowalnych komplementarnie literałach $\neg$*Bogaty*(*x*) i *Bogaty*(*Jan*), wnioskujemy zachodzenie klauzuli, *Nieszczęśliwy*(*Jan*), przy zastosowaniu podstawienia: θ= {x/Jan}.

2.3 Procedury wnioskowania

W rachunku zdań procedura wnioskowania sprawdza jedynie, czy podany symbol zdaniowy wynika z bazy wiedzy czy też nie. W logice predykatów możemy spytać bazę wiedzy o prawdziwość formuły dla wielu obiektów na raz. Mając formułę *S* pytamy się, czy istnieje podstawienie θ dla którego *S* wynika z bazy wiedzy. Wywołanie funkcji ASK(KB, S) zwraca wszystkie podstawienia θ takie, że: KB $\models$ SUBST(θ, S) . Pytania kierowane do bazy wiedzy są typu: kto, gdzie, kiedy?
Np. „*Kto jest bratem Piotra*?": ASK(KB, $\exists$x Brat(x, Piotr)). Oczekiwana odpowiedź to zbiór alternatywnych podstawień, np.: {{x/Jan}, {x/Stefan}} .

2.3.1 Wnioskowanie w przód

Stosujemy **wnioskowanie w przód** zasadniczo wtedy, gdy nowa formuła *p* zostaje dodana do bazy wiedzy *KB* (implementujemy to jako podfunkcję funkcji **TELL**). Przykładową implementacją jest procedura FC_TELL() (Tabela 1). Korzystają z reguły „uogólnione modus ponens" wyprowadza ona i dodaje do bazy wiedzy wszystkie możliwe implikacje wynikające z dodania nowej formuły (a w zasadzie zdania *p* reprezentującego obserwowany fakt).

Tabela 1. Wnioskowanie wprzód jako element funkcji TELL

```
procedure FC_TELL(KB, p)
begin
  if (w KB istnieje już wariant formuły p) then return;
  Dodaj p do KB;
  for ((p1 ∧ p2 ∧ … ∧ pn ⇒q) ∈ KB takie, że zachodzi unifikacja ∃i UNIFY(pi, p) = θ ) do
      FC_WNIOSKUJ(KB, [p1,..., pi-1,pi+1,…,pn ], q, θ );
  end
end
```

Rekursywna procedura FC_WNIOSKUJ (Tabela 2) uzgadnia, jeśli to możliwe, dalsze klauzule w poprzedniku formuły. Jeśli wszystkie one zostaną uzgodnione, to wyprowadzone zostają nowe zdania (jedno lub więcej, w zależności od liczby możliwych podstawień) a dla każdego nowego zdania wywoływana jest ponownie procedura początkowa FC_TELL.

Tabela 2. Procedura "FC_Wnioskuj"

```
procedure FC_WNIOSKUJ(KB, warunki, konkluzja, θ)
begin
  if (warunki == []) then FC_TELL(KB, SUBST(θ, konkluzja));
  else for (każde p' ∈ KB takie, że θ2 = UNIFY(p', SUBST(θ , Pierwszy(warunki)))) do
                        θ = COMPOSE(θ, θ2);
                        FC_WNIOSKUJ(KB, Reszta(warunki), konkluzja, θ);
       end
```

```
  end
end
```

Przy powyższym rozwiązaniu, gdy sterowanie agenta wysyła zapytanie (w ramach funkcji komunikacji z bazą wiedzy **ASK**) o wynikanie z KB pewnej formuły $q(\boldsymbol{Z})$ (Tabela 3), gdzie Z jest listą zmiennych w formule, to implementacja wnioskowania w przód polega jedynie na sprawdzeniu możliwości uzgodnienia (unifikacji) formuły $q(\boldsymbol{Z})$ ze zdaniami już istniejącymi w KB.

Tabela 3. Wnioskowanie wprzód jako element funkcji ASK

```
function FC_ASK(KB, q(Z)) returns [ wynik, θ ]
begin
   wynik = False,
   θ = ∅
   for (każda formuła q' w KB unifikowalna z q ) do
      wynik = True;
      ρ = UNIFY(q', q);
      θ = θ ∪ ρ|Z;
   end
   return [ wynik, θ ];
end
```

2.3.2 Wnioskowanie wstecz

Wnioskowanie wstecz stosujemy wtedy, gdy chcemy otrzymać odpowiedź na pytanie zadane do bazy wiedzy (implementacja w postaci funkcji BC_ASK) ale w przeciwieństwie do wnioskowania wprzód nie przechowujemy w bazie wiedzy wszystkich implikacji faktów dodanych uprzednio do bazy wiedzy (Tabela 4). Tworzone jest jedno drzewo wyprowadzenia I/LUB idąc od „korzenia", którym jest formuła zapytania wstecz, do „liści" drzewa, które odpowiadają faktom zapisanym w bazie wiedzy.

Tabela 4. Funkcja wnioskowania wstecz

```
function BC_ASK(KB, q(Z)) returns [wynik, θ|Z ]
// wynik = True/False, θ|Z = zbór podstawień pod zmienne z listy Z;
begin
   θ = {};
   wynik = BC_LISTA(KB, [q(Z)], θ);
   return [wynik, θ|Z ];
end
```

Parametry podfunkcji BC_LISTA() to: baza wiedzy *KB*, lista celów [*q*], szukany zbiór podstawień θ. Jest to funkcja rekursywna, wywoływana najpierw dla celu (formuły zapytania) a następnie dla kolejnych literałów warunkujących zadanych cel (Tabela 5). Przy powrocie z sukcesem z wywołania funkcja zwraca rozszerzony zbiór podstawień pod zmienne w dotychczas analizowanych formułach-celach.

Tabela 5. Wywołania rekursywnej funkcji BC_LISTA tworzą drzewo wyprowadzenia dla zapytania

function BC_LISTA(*KB*, *cele,* θ) : **return** *wynik*
//zwraca ewentualnie rozszerzony zbiór podstawień θ;
begin
  wynik ← $\varnothing$;
  if (*cele* są puste) **then return** {θ};
  q←SUBST(θ, Perwszy(cele));
  for (każdy $r \in$ KB taki, że STAND-ROZŁĄCZNA(r) = ($p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q'$) $\in$ *KB* i zachodzi θ' ←UNIFY(*q*, *q'*)) **do**
    wynik ← BC_LISTA(*KB*, SUBST(θ', [p_1,...,p_n|Reszta(*cele*)], COMPOSE(θ, θ')) $\cup$ *wynik*;
  return *wynik*;
end

Podane powyżej funkcje implementujące wnioskowanie wstecz są potencjalnie niepełne z powodu możliwości istnienia pętli zależności w grafie I-LUB. Dla zabezpieczenia przed taką sytuacją należy sprawdzać aktualny cel z każdym celem pamiętanym na stosie i unikać wielokrotnego umieszczania celu na stosie. Procedura jest też potencjalnie nieefektywna na skutek możliwego powtarzania celów już opracowanych (zarówno z pozytywnym jak i negatywnym wynikiem). Zabezpieczeniem przed tym jest pamiętanie w procedurze poprzednio wywoływanych celów (co wiąże się z potrzebą dodatkowej pamięci).

2.3.3 Wnioskowanie poprzez rezolucję

Wnioskowanie z regułą rezolucji prowadzone jest nie wprost i polega na sprawdzeniu warunków niespełnialności formuły będącej zaprzeczeniem formuły zapytania. Jeśli α jest formułą zapytania to jej zaprzeczenie dodawane jest do bazy wiedzy (jest ona wtedy postaci CNF(KB $\wedge \neg\alpha$)) a następnie stosuje się iteracyjnie krok rezolucji. Jeśli zostanie wygenerowana formuła pusta to będzie oznaczać, że formuła zapytania zachodzi, przy podstawieniu pod zmienne warunkujące prawdziwość poprzedników rezolwenty pustej. Procedura wnioskowania zwraca wszystkie możliwe podstawienia pod zmienne w zapytaniu, warunkujące uzyskanie formuły pustych. Proces wnioskowania kończy się w momencie braku nowych wyników wnioskowania w kolejnej iteracji. Jeśli nie wygenerowano żadnej formuły pustej to zwracany jest wynik „False” (Tabela 6).

Tabela 6. Funkcja wnioskowania przez rezolucję

function RESOLUTION_ASK(*KB*, *q*(***Z***)) **return [***wynik,* $\theta_{\backslash Z}$ **]**

```
// wynik = True/False, θ\Z = zbór podstawień pod zmienne z listy Z;
begin
  wynik = False; θ\Z = ∅; θ = ∅ ;
  klauzule = KB ∪ {klauzule od zanegowanej formuły q };
  nowe = klauzule;
  REPEAT
     aktualne = ∅ ;
     FOR każda klauzula p w nowe DO
        FOR każda klauzula p' w { klauzule - p} DO
           [rezolwenty, θ] = KROK_REZOLUCJI(p, p');
           IF (θ <> „fail") & (rezolwenty == ∅) THEN
              wynik = True;
              θ\Z= COMPOSE(θ\Z ,θ) ;
           end;
           aktualne = aktualne ∪ rezolwenty ;
        end;
     end;
     klauzule= klauzule ∪ aktualne;   nowe = aktualne;
  UNTIL aktualne < > ∅
  return [wynik, θ\Z ];
end
```

Baza wiedzy powinna być niesprzeczna i zawierać formuły o postaci CNF. Lista szukanych podstawień $\theta_{|Z}$ jest początkowo pusta. W pierwszej iteracji zbiór *nowe*(1) tworzą wszystkie klauzule w KB i klauzule pochodzące z negacji zapytania. Zbiór *aktualne*(1) jest początkowo pusty.
W *i*-tej iteracji wykonywane są kroki rezolucji dla każdej klauzuli *p* ze zbioru *nowe*(i) z każdą klazulą, pochodzącą ze zbioru {*klazule*(i) - *p*}. Nowe rezolwenty (nie wygenerowane wcześniej) są dodawane do zbioru *aktualne*(i).
Jeśli w danym kroku została wyprowadzona klauzula pusta (jedno- lub wielokrotnie) to nastąpiło to przy podstawieniu (-ach) θ pod zmienne tych formuł. Funkcja wnioskowania zapamiętuje wszystkie podstawienia pod zmienne zapytania w zbiorze $\theta_{|Z}$ które dały wynik pozytywny wnioskowania.

3 Inżynieria wiedzy

Inżynieria wiedzy zajmuje się metodologią konstruowania i korzystania z systemów z bazą wiedzy. Jednym z jej przejawów jest „programowanie w logice", uznawane też za jeden z 4 głównych paradygmatów programowania.

3.1 Inżynieria wiedzy a inżynieria oprogramowania

Poniższa Tabela 7Tabela 7. Porównanie inżynierii oprogramowania i inżynierii wiedzy. pokazuje zasadnicze różnice pomiędzy inżynierią oprogramowania a inżynierią wiedzy.

Tabela 7. Porównanie inżynierii oprogramowania i inżynierii wiedzy.

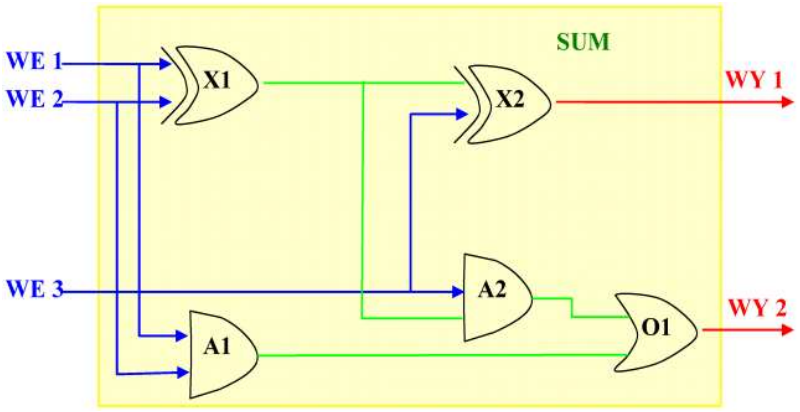
Inżynieria oprogramowania	Inżynieria wiedzy
Języki programowania	Języki reprezentacji wiedzy
Dane	Bazy wiedzy
Operacje	Procedury wnioskowania (dowodzenia zdań)
Wykonanie programu	Wnioskowanie (dowodzenie zdania zapytania)

Zgodnie z zasadami inżynierii w procesie tworzenia bazy wiedzy, wyrażonej w języku logiki, wyróżnimy następujące etapy:

1. Identyfikacja zadania.
2. Zebranie odpowiedniej wiedzy o dziedzinie (jakie obiekty i fakty modelowanej dziedziny należy reprezentować w systemie a jakie są zbędne).
3. Wybór słownika dla relacji, funkcji i stałych (ontologia).
4. Zakodowanie ogólnej wiedzy o dziedzinie (aksjomaty).
5. Zakodowanie opisu specyficznego zadania (formuły atomowe wyrażające fakty).
6. Testowanie - wysyłanie zapytania do procedury wnioskowania i odbieranie jej odpowiedzi.
7. Przeanalizowanie wyników testowania i ewentualnie poprawianie zawartości bazy wiedzy.

Przykład

Definiowanie bazy wiedzy dla układu elektronicznego - sumatora jednobitowego (Rysunek 1):



Rysunek 1. Schemat sumatora jednobitowego

1)Identyfikacja zadania: sprawdzenie, czy układ wykonuje prawidłową operację dodawania?

2) Zebranie odpowiedniej wiedzy o zasadzie działania:

- zauważenie tego, że układ składa się z połączeń i bramek logicznych;

- określenie typów stosowanych bramek (AND, OR, XOR, NOT)
- bez znaczenia są takie cechy, jak: rozmiar, kształt, kolor, koszt bramek

3) Wybór słownika:

- stałe oznaczają bramki; np. X1, X2.
- typ bramki podany będzie dzięki funkcji; np. Typ(X1) = XOR;

inne alternatywne (ale gorsze) rozwiązanie – poprzez predykaty: Typ(X1, XOR), XOR(X1).

- wejścia / wyjścia bramki podają kolejne funkcje; np. We(1,X1), Wy(1,X1).
- istnienie połączenia bramek wyznacza relacja: Połączenie(Wy(..), We(..)).
- wartości sygnału to obiekty 1, 0 – podaje je funkcja: Sygnał(Wy(..)).

4) Zakodowanie ogólnej wiedzy o dziedzinie (zasady rządzące dziedziną zadane są w postaci formuł korzystających z symboli przyjętego słownika):

$\forall$t1,t2 Połączenie(t1, t2) $\Rightarrow$ Sygnał(t1) = Sygnał(t2)

$\forall$t Sygnał(t) = 1 $\vee$ Sygnał(t) = 0

1 ≠ 0

$\forall$t1,t2 Połączenie (t1, t2) $\Rightarrow$ Połączenie (t2, t1)

$\forall$g Typ(g) = OR $\Rightarrow$ (Sygnał(Wy(1,g)) = 1 $\Leftrightarrow$ $\exists$n Sygnał(We(n,g)) = 1)

$\forall$g Typ(g) = AND $\Rightarrow$ (Sygnał(Wy(1,g)) = 0 $\Leftrightarrow$ $\exists$n Sygnał(We(n,g)) = 0)

$\forall$g Typ(g) = XOR $\Rightarrow$ (Sygnał(Wy(1,g)) = 1 $\Leftrightarrow$ Sygnał(We(1,g)) ≠ Sygnał(We(2,g)))

$\forall$g Typ(g) = NOT $\Rightarrow$ (Sygnał(Wy(1,g)) ≠ Sygnał(We(1,g)))

5) Zakodowanie specyficznego problemu, czyli określenie własności konkretnie badanego układu elektronicznego (logiczne fakty):

Typ(X_1) = XOR *Typ*(X_2) = XOR

Typ(A_1) = AND *Typ*(A_2) = AND

Typ(O_1) = OR

Połączenie(*Wy*(1,X_1),We(1,X_2)) , *Połączenie* (We(1,C_1),We(1,X_1))

Połączenie(*Wy*(1,X_1),We(2,A_2)), *Połączenie* (We(1,C_1),We(1,A_1))

Połączenie(*Wy*(1,A_2),We(1,O_1)) , *Połączenie* (We(2,C_1),We(2,X_1))

Połączenie(*Wy*(1,A_1),We(2,O_1)) , *Połączenie* (We(2,C_1),We(2,A_1))

Połączenie(*Wy*(1,X_2), *Wy*(1,C_1)), *Połączenie* (We(3,C_1),We(2,X_2))

Połączenie(*Wy*(1,O_1), *Wy*(2,C_1)), *Połączenie* (We(3,C_1),We(1,A_2))

6) Testowanie – generowanie zapytań kierowanych do procedury wnioskowania.

Np. pytanie jest postaci: „jakie sygnały wejściowe spowodują, że wyjście pierwsze bramki C1 będzie „0” a jej wyjście drugie „1” ?”:

$\exists$i1,i2,i3 Sygnał(We(1,C1))=i1 $\wedge$ Sygnał(We(2,C1))=i2 $\wedge$ Sygnał(We(3,C1))=i3 $\wedge$ Sygnał(Wy(1,C1))=0 $\wedge$ Sygnał(Wy(2,C1))=1

Poprawna odpowiedź to zbiór podstawień:

θ = { {i1/1, i2/1, i3/0}, {i1/1, i2/0, i3/1}, {i1/0, i2/1, i3/1} }

7) Ewentualne poprawianie bazy wiedzy. Można też opuścić oczywiste związki, jak np. 1 ≠ 0.

3.2 Systemy ekspertowe

System ekspertowy jest praktycznym wykorzystaniem metodyk języków reprezentacji wiedzy i skojarzonych z nimi mechanizmów wnioskowania. W zależności od sposobu reprezentacji wiedzy wyróżnimy następujące kategorie systemów ekspertowych:

1. Systemy logiki predykatów – przykładami są systemy dowodzenia twierdzeń i systemy stosujące język PROLOG.
2. Systemy regułowe (np. OPS5, CLIPS, SOAR) – stosują formuły implikacji dla reprezentacji warunków wykonania i opisu akcji; wśród akcji są operacje wprowadzania i usuwania do/z bazy wiedzy i operacje we/wy; stosują wnioskowanie w przód.
3. „Ramy" (frames) i sieci semantyczne – wiedza dana jest w postaci strukturalnej i zorientowanej obiektowo.
4. Logika deskrypcyjna – popularny podzbiór języka predykatów i jego wnioskowania.

Systemy **dowodzenia twierdzeń** (ang. *theorem provers*) (np. AURA, OTTER) stosują wnioskowanie metodą rezolucji w logice predykatów i ich celem jest dowodzenia twierdzeń w zadaniach matematycznych oraz realizacja zapytań do bazy wiedzy.

Języki **programowania w logice**, (np. język **Prolog**), stosują zwykle wnioskowanie wstecz, nakładają ograniczenia na język predykatów i dysponują proceduralnymi elementami wejścia/wyjścia. Program w Prologu to uporządkowany ciąg formuł a dane to formuła celu (zapytanie). Wykonanie programu to realizacja wnioskowania wstecz z przeszukiwaniem według zasady „*przeszukiwanie w głąb*", i „*od lewej-do prawej*". Programowanie w Prologu ma zasadniczo charakter deklaratywny (kod programu jest logiczną specyfikacją problemu a jego wykonanie – wnioskowaniem logicznym). Jednak w celu zwiększenia efektywności występuje też wiele mechanizmów pozalogicznych: jest duży zbiór wbudowanych predykatów związanych z arytmetyką i wejściem/wyjściem – sprawdzenie poprawności tych predykatów polega na wykonaniu odpowiedniego kodu procedur a nie na wnioskowaniu logicznym; występuje też szereg funkcji systemowych i obsługujących bazę wiedzy.

Systemy **regułowe** (np. OPS5, CLIPS, SOAR) stosują *reguły produkcji* (implikacje wiążące warunki wykonania z opisami akcji) i są podstawą wielu systemów ekspertowych. Systemy produkcji realizują wnioskowanie w przód, używając bardzo ograniczonego języka. Typowy system produkcji składa się z: pamięci roboczej – zawiera pozytywne literały bez zmiennych (zdania atomowe); zbioru reguł – wyrażeń postaci:

$$P_1 \wedge ... \wedge P_n \Rightarrow akcja_1 \wedge ... \wedge akcja_m ,$$

gdzie (P_1,...,P_n) są literałami (formułami atomowymi), a ($akcja_1$,..., $akcja_m$) są działaniami, które należy wykonać wtedy, jeśli wszystkie zdania P_i są prawdziwe (tzn. znajdują się w pamięci roboczej). Wśród akcji są operacje wprowadzania i usuwania do/z bazy wiedzy i operacje we/wy.

„**Ramy**" (ang. *frames*) i **sieci semantyczne** (np. SNePS, Netl, KL-ONE) są to etykietowane grafy o ściśle zdefiniowanych etykietach łuków. Obiekty i ich kategorie są węzłami grafu a ich relacje binarne reprezentują zależności typu: „*jest częścią*", „*jest specjalizacją*", „*jest instancją*". Każdy węzeł sieci,

posiada nazwane atrybuty, gdzie każdy atrybut wiąże obiekt z jakimś termem (w szczególności ze stałą). Atrybuty kategorii ogólnej są dziedziczone przez jej specjalizacje. W zasadzie każda sieć semantyczna może być wyrażona w postaci zbioru formuł logiki predykatów. Jednak sieć semantyczna realizująca dziedziczenie z możliwością wystąpienia **wyjątków** jest potencjalnie **niemonotoniczna** – dodanie nowej przesłanki może unieważnić dotychczas wyprowadzone formuły.

3.2.1 PROLOG

Język programowania logicznego umożliwia implementację logicznego systemu wnioskowania dzięki temu, że:

- posiada reprezentację logicznych formuł,
- umożliwia dołączanie informacji sterującej procesem wnioskowania,
- posiada mechanizm wnioskowania.

Typowym przedstawicielem takich języków jest PROLOG. Program w Prologu jest zbiorem klauzul Horna - każda formuła jest postaci formuły atomowej lub implikacji, gdzie w poprzedniku występują dodatnie literały a następnik jest pojedynczym literałem. Dużymi literami oznacza się zmienne, małymi – nazwy predykatów, funkcji i stałych. Prawdziwość wbudowanych predykatów sprawdzana jest za pomocą dołączonego kodu programu.

Operatory

Prolog wprowadza własna notację operatorów (Tabela 8).

Tabela 8. Operatory logiczne w Prologu

Logika klasyczna	Operacja	PROLOG
$\neg$	Negacja	**not**
$\vee$	Alternatywa	**;**
$\wedge$	Koniunkcja	**,**
$\Rightarrow$	Implikacja	**:-**

Przyjęte jest założenie o zupełności świata - symbol "negacji" nie jest używany do tworzenia zanegowanych formuł, lecz odpowiada operacji **not** („nieprawda, że w aktualnej interpretacji zachodzi dany fakt"). Np. mając klauzulę: *alive*(X) :- not *dead*(X), konkretny fakt *alive*(*joe*) zajdzie wtedy, jeśli wykażemy, że *dead*(*joe*) jest nieprawdziwe.

Klauzule

W Prologu stosuje się zapis klauzul Horna w formacie implikacji pisanej w odwrotnej kolejności (następnik poprzedza poprzednika implikacji) a literały poprzednika są niejawnie połączone koniunkcją, czyli:

nagłówek **:-** *literał*$_1$, …, *literał*$_n$.

Np.: *przestępca*(X) **:-** *Amerykanin*(X), *broń*(Y), *sprzedaje*(X,Y,Z), *wrogi*(Z).

Każda klauzula Horna w Prologu jest jednej z postaci:

- $\neg P_1 \vee \ldots \vee \neg P_n \vee Q$ (*równoważne:* $(P_1 \wedge \ldots \wedge P_n) \Rightarrow Q$)
- Q (*formuła atomowa – literał*)

- $P_1 \wedge ... \wedge P_n$ (*klauzula celu - zapytanie*)
- $\perp$ (*klauzula pusta*)

W Prologu podane powyżej klauzule Horna zapisujemy w następującej postaci:

- Q :- P_1, ... , P_n . Klauzulę interpretujemy jako procedurę o nagłówku Q i treści P_1,... , P_n. Wywołanie procedury Q sprowadza się do kolejnego wywołania procedur P_1, ... , P_n .
- Q :- . (*formuła atomowa – literał*) Klauzulę interpretujemy jako procedurę o pustej treści.
- ?- P_1, ... , P_n . (*klauzula celu - zapytanie*)
- Q . (fakt) Interpretujemy jako dane programu).
- . (*klauzula pusta*) Interpretujemy jako instrukcje *stop*.

Predykaty

Specyficzne typy danych to *listy* i *struktury*. Lista ma postać sekwencji termów i list, rozdzielonych przecinkami i zawartych w nawiasach [ ]. Struktury mają postać predykatów, których argumentami są predykaty.

Typową operacją jest badanie przynależności termu X do listy L za pomocą predykatu *member*(X,L).

Przypisanie wartości zmiennym reprezentuje operacja **is**. Np. : *zmienna* **is** *wyrażenie_arytmetyczne.*

Dostępne są podstawowe operacje arytmetyczne: +, -, *, /, **, mod.

Istnieją wbudowane predykaty:

- do badania typu argumentu: var(V), atom(A), number(N), itp.;
- dla arytmetycznych relacji binarnych: A < B, A =< B, itp.;
- sprawdzające prawdziwość relacji: A ?= B (czy predykaty A i B są unifikowalne?), A==B (czy wartości A i B są równe?);
- operacje specjalne: A = B (zunifikuj), read(X), write(X), print(X) (operacje wejścia/wyjścia)

Program

Program w Prologu to uporządkowany ciąg procedur:

- zapytanie to klauzula celu;
- wykonanie programu to realizacja wnioskowania metodą rezolucji (albo wnioskowania wstecz).

Programowanie w Prologu ma zasadniczo charakter deklaratywny (kod programu jest logiczną specyfikacją problemu a jego wykonanie – wnioskowaniem logicznym). Jednak w celu zwiększenia efektywności występuje też wiele mechanizmów pozalogicznych :

- duży zbiór wbudowanych predykatów związanych z arytmetyką i wejściem/wyjściem – sprawdzenie poprawności tych predykatów polega na wykonaniu odpowiedniego kodu procedur a nie na wnioskowaniu logicznym;
- szereg funkcji systemowych i funkcji obsługujących bazę wiedzy.

Odpowiedzią na zapytanie są podstawienia pod zmienne, przy których formuła celu jest spełnione. Jeśli zapytanie nie zawiera zmiennych to odpowiedzią są napisy *yes* lub *no*.

Przykłady zapytań i wyników wnioskowanie

?- X **is** 2+2.

Wynik: X=4

Niech *append* oznacza operację połączenia dwóch list wraz z podaniem wyniku połączenia. Wśród danych programu mogą wystąpić klauzule:

append([], Y, Y) :- . (połączenie listy pustej i Y daje w wyniku Y)

append([X|L], Y, [X|Z]) :- append(L, Y, Z) (rozszerzenie łączonej listy L zawsze odpowiednio rozszerza wynikowa listę Z).

Zapytanie: ?- append(A, B, [1,2])

Wynik:

A=[] B=[1,2]

A=[1] B=[2]

A=[1,2] B=[]

W tym przypadku istnieją trzy możliwe (alternatywne) podstawienia pod zmienne A i B w zapytaniu. Wszystkie one są zwracane jako wynik wnioskowania.

Prolog oferuje oryginalny sposób ograniczenia liczby rozwiązań. Służy do tego znak „!" umieszczony w zapytaniu. W powyższym przypadku, na zapytanie

?- append(A, B, [1,2]) !

zwrócony zostałby tylko pierwszy wynik: A=[] B=[1,2]

Jeśli w zapytaniu występuje koniunkcja dwóch lub więcej literałów to znak „!" umieszczony po wybranym literale sprawie, że zwracana jest tylko pierwsza możliwość uzgodnienia jego zmiennych w literałach przed znakiem „!", a dla pozostałych – wszystkie możliwe podstawienia pod zmienne, przy których literały te są spełnione.

Znak „!" może występować też w aksjomatach. Jeśli umieszczony zostanie na końcu sekwencji literałów w warunku formuły implikacji to prawdziwość każdego literału sprawdzana jest jedynie do pierwszego jego spełniania.

3.2.2 System regułowy

System regułowy (nazywany też systemem produkcji) jest podstawą większości systemów ekspertowych spotykanych w praktyce. Taki system realizuje **wnioskowanie wprzód**. Typowy system regułowy (produkcji) składa się z:

- **pamięci *roboczej***, która zawiera pozytywne literały bez zmiennych (zdania atomowe);
- zbioru **reguł (produkcji),** czyli wyrażeń o postaci:

$P_1 \wedge ... \wedge P_n \Rightarrow akcja_1 \wedge ... \wedge akcja_m$,

gdzie $P_1,...,P_n$ są literałami (formułami atomowymi), a $akcja_1,..., akcja_m$ są działaniami, które należy wykonać wtedy, jeśli wszystkie zdania P_i są prawdziwe (tzn. znajdują się w pamięci roboczej). W takiej sytuacji regułę nazywamy **realizowalną**. Przykłady akcji - dodanie lub usunięcie elementu z pamięci roboczej.

Praca systemu regułowego składa się z ciągu **cykli**. Z kolei każdy cykl składa się z **trzech faz**:

1. **faza dopasowania** – poszukiwanie realizowalnych reguł;
2. **faza wyboru** – wybór reguł do wykonania i ustalenie ich kolejności;

3. **faza wykonania** – wykonanie wybranych reguł.

Faza dopasowania

Zasadniczym elementem **fazy dopasowywania** jest procedura **unifikacji**, ale w praktycznych systemach wymagana jest jej efektywna implementacja. Przykładem efektywnego algorytmu dopasowywania jest tzw. **algorytm sieci czasu rzeczywistego RT** (tzw. RT-sieć) w OPS-5. Ogranicza on złożoność unifikacji dzięki jednokrotnemu wydzieleniu wspólnych zależności wielu reguł oraz pamiętaniu wcześniej wykonanych unifikacji.

Przykład algortymu „sieć RT"

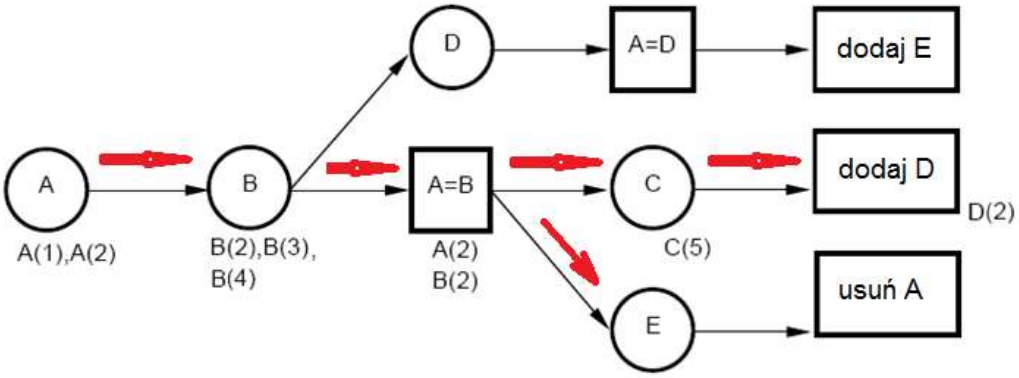
Załóżmy, że dane są następujące fakty (w pamięci *roboczej*) i reguły (w pamięci *długotrwałej*):

- Fakty: { $A(1), A(2), B(2), B(3), B(4), C(5)$ }
- Reguły: $A(x) \wedge B(x) \wedge C(y) \Rightarrow$ *dodaj* $D(x)$.

  $A(x) \wedge B(y) \wedge D(x) \Rightarrow$ *dodaj* $E(x)$.

  $A(x) \wedge B(x) \wedge E(z) \Rightarrow$ *usuń* $A(x)$.

Przed wykonaniem wnioskowania wykonywana jest konwersja reguł do postaci sieci obliczeń: okręgi to odwołania do pamięci roboczej, kwadraty to unifikacje, prostokąty to wyprowadzane akcje (Rysunek 2).



Rysunek 2. Przykład wnioskowania w systemie regułowym "sieci RT"

Faza wyboru akcji

Niektóre systemy wykonują wszystkie realizowalne działania, ale inne systemy wykonują tylko niektóre z nich. W takiej sytuacji są możliwe różne strategie wyboru:

- Strategia (1) - preferujemy reguły, które odnoszą się do ostatnio tworzonych elementów pamięci roboczej;
- Strategia (2) – preferujemy reguły bardziej „specyficzne".

Np. z poniższych reguł należy wybrać drugą z nich:

$Ssak(x) \Rightarrow$ dodaj $Nogi(x,4)$.

$Ssak(x) \wedge Człowiek(x) \Rightarrow$ dodaj $Nogi(x,2)$.

- Strategia (3) – preferujemy działania, którym nadano wyższy priorytet.

Np. z poniższych reguł zapewne preferujemy drugą z nich:

$P(x) \Rightarrow Akcja(Odkurzanie(x))$.

$R(x) \Rightarrow Akcja$(Ewakuacja).

3.2.3 Sieci semantyczne (ramy)

Sieć semantyczna to reprezentacja wiedzy w postaci grafu, przedstawiającego **kategorie obiektów** (lub pojedyncze obiekty) i **relacje** pomiędzy nimi. Węzły grafu reprezentują **pojęcia** (ang. concepts) a łuki grafu – **relacje** pomiędzy pojęciami. Zawsze wyróżniane są podstawowe **relacje** pomiędzy pojęciami:

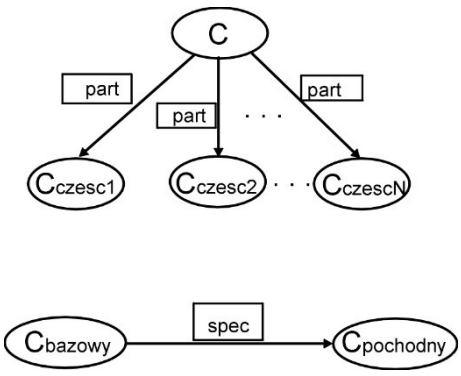
- *Spec* – relacja pomiędzy pojęciem - kategorią ogólną a pojęciem – kategoria specjalizowana (innymi słowy jest to relacja dziedziczenia);
- *Part* – relacja pojęcia będącego częścią do pojęcia reprezentującego całość (innymi słowa - wyznacza hierarchię złożoności pojęć);
- *Member* (lub Instance) – relacja instancji pojęcia (np. obiektu) do jej kategorii (np. typu obiektu).

Każdy węzeł sieci (kategoria obiektów lub obiekt) posiada **nazwane atrybuty** (własności), gdzie każdy atrybut może być przedstawiony jako relacja wiążąca dwa pojęcia (węzły sieci), np. obiekt z pojęciem reprezentującym funkcję lub term (w szczególności z wartością stałą określonego typu). Atrybuty kategorii ogólnej są „dziedziczone" przez kategorie-pojęcia specjalizowane, a także występują w obiektach tej kategorii i jej specjalizacji.
W zasadzie każda sieć semantyczna może być wyrażona w postaci zbioru formuł logiki pierwszego rzędu (Rysunek 3):

- Pojęcia odpowiadają typom obiektów lub obiektom.
- Łuki o etykietach „Inst" wyznaczają relację nazwaną np. „instance" pomiędzy typem obiektu a jego instancją.
- Zależność „zbiór części --part_of--> całość" jest interpretowana jako formuła implikacji:

  $C_{część1} \wedge C_{część2} \wedge ... \wedge C_{częśćN} \Rightarrow C$
- Podobnie, zależność „koncept_pochodny –spec_of--> koncept_bazowy":

  $C_{pochodny} \Rightarrow C_{bazowy}$



Rysunek 3. Dwie hierarchie pojęć w sieci semantycznej

Niech **atrybut** *R* dla węzła *A* wyznacza węzeł *B*. Z uwagi na zasadnicze rozróżnienie w semantyce pojęć pomiędzy kategoriami i instancjami (obiektami) atrybut *R* odpowiada dwu-argumentowej relacji będącej jednej z trzech postaci:

1. relacja zachodzi pomiędzy dwiema kategoriami (np. typami obiektów): $R(A, B)$;
2. relacja zachodzi pomiędzy każdym obiektem kategorii *A* i kategorią *B*: $\forall x \; x \in A \Rightarrow R(x, B)$;
3. relacja zachodzi pomiędzy każdym obiektem kategorii *A* i pewnym obiektem kategorii *B*:

$\forall x\ \exists y\ x \in A \Rightarrow y \in B \wedge R(x, y)$.

Zauważmy, że relacje pierwszej postaci tworzone są podczas definiowania pojęć bazy wiedzy dla określonej dziedziny. Druga postać oznacza możliwość selektywnego stosowania atrybutu dla obiektów typu A podczas wnioskowania. Trzecia postać oznacza przypisanie konkretnego termu (funkcji lub stałej) typu B do atrybutu konkretnego obiektu typu A.

3.2.4 Logika opisowa

Logika opisowa jest pewnym rozstrzygalnym podzbiorem logiki predykatów. Każdy język z rodziny języków logiki opisowej zawiera:

- Pojęcia reprezentujące kategorie, czyli zbiory obiektów;
- Pojęcia atomowe, w tym koncept uniwersalny * (Top), reprezentujący uniwersum, oraz koncept pusty $\perp$ (Bottom), który nie może mieć żadnych wystąpień;
- Role atomowe (atrybuty) - reprezentują relacje między parami obiektów;
- Konstrukcje (operatory) służące do tworzenia złożonych pojęć i ról a także wyrażania aksjomatów dziedziny. Można użyć wielu własności (tzn. innych pojęć lub ograniczeń poprzez atrybuty) równocześnie w definicji pojęcia (Tabela 9).

Tabela 9. Przykład logiki opisowej: język ALC

Konstruktor (operator)	Znaczenie
$C \equiv D$	Pojęcie C jest równoważne D
$\neg C$	Negacja pojęcia C
$C \sqsubseteq D$	Pojęcie C jest specjalizacją pojęcia D
$C \sqcap D$	Część wspólna pojęć C i D
$C \sqcup D$	Suma pojęć C i D
$\exists R.C$	Kwantyfikacja egzystencjalna – zbiór takich instancji, które są powiązane (przynajmniej raz) rolą R z instancją pojęcia C
$\forall R.C$	Kwantyfikacja ogólna – zbiór takich instancji, których wszystkie istniejące powiązania rolą R dotyczą instancji pojęcia C (ale obejmuje także takie instancje, które nie są powiązane rolą R z żadną instancją)

Baza wiedzy dla danej dziedziny zastosowania zawiera dwa zbiory wyrażeń: TBox i ABox**.** TBox obejmuje terminologię dziedziny zastosowania (zbiór pojęć, zbiór ról (atrybutów), zbiór aksjomatów). ABox obejmuje aktualny opis świata (fakty).

Przykład opisu dziedziny „tradycyjna rodzina”

Bez twierdzenia o zupełności definicji dziedziny, do zbioru Tbox przyjmiemy *atomowe* pojęcia reprezentujące kategorie (*Osoba, Mężczyzna, Kobieta, Rodzic*) i ich role (atrybuty): *maDziecko, maSyna, maCórkę*.

Do zbioru Tbox dodamy definicje ograniczeń i pojęć złożonych:

$$Mężczyzna \sqsubseteq Osoba$$
$$Kobieta \sqsubseteq Osoba$$
$$Kobieta \sqcap Mężczyzna \equiv \bot$$
$$Rodzic \equiv Osoba \sqcap \exists maDziecko.Osoba$$
$$Ojciec \equiv Mężczyzna \sqcap Rodzic$$
$$Matka \equiv Kobieta \sqcap Rodzic$$

Uwzględnimy następujące aksjomaty dziedziny:

$maSyna \sqsubseteq maDziecko$ (jeśli ma syna to ma dziecko)

$\exists maSyna.\neg Mężczyzna \equiv \bot$ (syn jest zawsze mężczyzną)

$\exists maSyna.* \sqsubseteq Rodzic$ (każdy, to ma syna jest też rodzicem)

Załóżmy, że w zbiorze Abox zawarte są znane nam fakty:

$$Instance(Kobieta, Anna), Instance(Kobieta, Ewa), Instance(Mężczyzna, Adam)$$
$$maDziecko(Anna, Ewa), maDziecko(Anna, Adam), maSyna(Adam, Jan)$$

Wnioskowanie. Możemy spytać się jakiego typu jest np. obiekt „Jan": $types(Jan)?$ Proces wnioskowania odpowiada podając wszystkie pojęcia – kategorie obiektu Jan: $\{Mężczyzna, Osoba\}$. Na pytanie $types(Adam)?$ otrzymamy odpowiedź: $\{Rodzic, Mężczyzna, Osoba\}$.

4 Niektóre rozszerzenia logiki klasycznej

Logika niemonotoniczna

Logika modalna

4.1 Logika niemonotoniczna

Logika klasyczna jest **monotoniczna**: „jeśli formuła *A* wynika ze zbioru formuł ***X***, to *A* wynika także z każdego nadzbioru zbioru ***X*** : if KB $\models \alpha$ then (KB $\wedge \beta$) $\models \alpha$.

Własność monotoniczności jest powiązana z założeniem o **zupełności** bazy wiedzy (**zamknięty świat**):

- brakujące literały są niespełnione (False),
- unikamy reprezentowania w bazie wiedzy wielu negatywnych literałów.

Jednak w praktyce w wielu dziedzinach wiedza ma charakter **niemonotoniczny** – dodanie nowej przesłanki (obserwacji) może unieważnić dotychczas przyjęte założenia. Jest to zgodne z założeniem „**otwartego świata**":

- spełnianie brakujących literałów nie jest z góry znane;
- **niemonotoniczność** zachodzi wtedy, gdy mogą być wyjątki od ogólnej reguły.

Przykład: logika „domyślna"

Domyślna wiedza o „świecie kotów” może polegać na tym, że jeśli obiekt jest kotem to ma cztery nogi. Wprowadzamy tę własność świata do bazy wiedzy w postaci formuły definiującej predykat (np. o nazwie *Rel4*):

$$\forall r,a,b\ Rel4(r, a, b) \Leftrightarrow [r, a, b] \in \{[Nogi, Koty, 4], \ldots\}.$$

Jeśli dopuszczamy, że od tej własności mogą być wyjątki – to logika jest potencjalnie ***niemonotoniczna*** – dodanie nowej przesłanki może unieważnić dotychczas wyprowadzone zdania. Np. dodanie obserwacji *Nogi*(*kotek, 3*) unieważnia wyprowadzony poprzednio fakt, że *kotek* ma 4 nogi.

W logice „domyślnej” wprowadza się dwa typy reguł wnioskowania:

- *zwykłe*: mają one znaną nam postać

$$\frac{Przeslanka}{Konkluzja},$$

  czyli są uporządkowaną parą wyrażeń;
- *domyślne*: mające postać uporządkowanej trójki wyrażeń:

$$\frac{Przeslanka : Uzasadnienie}{Konkluzja}$$

  gdzie zdanie *Uzasadnienie* odgrywa rolę uprawomocnienia (ang. *justification*).

Normalna postać reguły *domyślnej* to <A : B /B>, co oznacza, że: „możemy wyprowadzić B z A o ile B jest niesprzeczne z tym, co już jest wiadome.

4.2 Logika modalna

W logie predykatów możemy modelować przekonania (*sądy*) agenta za pomocą relacji typu:

Wierzy (Agent, *x*), *Wie*(Agent, *x*), *Chce*(Agent, *x*).

Np. chcemy tym wyrazić, że: „*agent* wie, że *p”, „agent* wierzy, że *p”, „agent* chce aby, *p”*. Czyli chcemy wyrazić relację pomiędzy agentem i formułą (lub zdaniem). Ale możemy napisać,

Wierzy (Agent*, x*),

jedynie pod warunkiem, że *x* jest **termem.** Np. *Wierzy*(Agent*, Lata*(*Superman*)) jest prawidłowe pod warunkiem, że *Lata*(*Superman*) jest termem.

Związek pomiędzy „przekonaniem” a „wiedzą” może być rożnie definiowany. Filozofowie definiują wiedzę jako „udowodnioną prawdziwość przekonania”. Wyrazimy to następująco:

$$\forall a, p\ \ Wiedza(a,p) \Leftrightarrow Wierzy(a,p) \wedge Prawda(p) \wedge Prawda(KB(a) \Rightarrow p)$$

Problemem logiki klasycznej jest jej **ekstensjonalność** tzn. iż wartość wyrażenia jest funkcją wartości jego podwyrażeń. Prowadzi to np. do następującego wynikania: z faktu, że *Instance(Złodziej, Kowalski)* i z przekonania agenta *Wie*(*Agent, TymOknemWszedł(Złodziej)*) wynikałoby, że *Wie*(*Agent, TymOknemWszedł(*Kowalski*)*).

Taka cecha logiki może prowadzić do generowania zupełnie niepotrzebnych zdań.

W praktyce występuje czasem **intensjonalność**, tzn. zdarza się, że wartość logiczna zdania złożonego, w którym występuje czyjeś „przekonanie”, nie jest jednoznacznie wyznaczona przez wartości logiczne zdań składowych. Rozwiązaniem powyższego problemu może być zastosowanie

logiki modalnej, która jest intensjonalna lub też ograniczenie zakresu termów reprezentujących obiekty-własności tylko do działania na symbolach stałych.

Logiki modalne stanowią rozszerzenie logiki predykatów o tzw. *operatory modalne*, takie jak operator „przekonania" („jest możliwe") (ang. „believes") i „wiary" („jest konieczne") (ang. „knows"), których argumentami są *zdania logiczne* a nie *termy.* Wnioskowanie w logice modalnej ogranicza możliwość podstawiania pod zmienne w kontekście takich wyrażeń modalnych.

Przykład logiki modalnej

W modalnym rachunku zdań wprowadzimy dwa dodatkowe jednoargumentowe operatory (spójniki):

- „jest możliwe, że" – spójnik możliwości, oznaczony ◊ ;
- „jest konieczne, żeby" – spójnik konieczności, oznaczony □ .

To co jest *konieczne* uznaje się za prawdziwe, natomiast to co jest *możliwe* można opatrzyć jakimś współczynnikiem oceny i poddać analizie.

Oba operatory modalne są wzajemnie ze sobą powiązane:

◊¬A ⇔ ¬ □A : coś co (czasem) może być nieprawdziwe (może nie zachodzić) nie może być (zawsze) koniecznie prawdziwe;

◊A ⇔ ¬□¬A : coś co (czasem) może być prawdziwe (może zachodzić) nie może być (zawsze) koniecznie nieprawdziwe;

□¬A ⇔ ¬◊A : coś co jest koniecznie nieprawdziwe to nie może być czasami prawdziwe;

□A ⇔ ¬◊¬A : coś jest koniecznie prawdziwe to nie może być czasami nieprawdziwe.

Za pomocą nowych operatorów zdefiniujemy spójnik *ścisłej implikacji* →:

(A → B) ⇔ □(A ⇒ B) : jeśli z A wynika B to koniecznie prawdziwa jest klasyczna implikacja;

(A → *B*) ⇔ ¬◊(A ∧ ¬B) : jeśli z A wynika B to nie może jednoczesne zachodzić poprzednik A i nie zachodzić następnik B.

Zachodzą także następujące wyrażenia:

□A → A : jeśli coś jest konieczne to zachodzi (zawsze);

◊A → □ ◊A : jeśli coś jest możliwe to konieczna jest możliwość jej zachodzenia;

□ (*A* → *B*) → (□ *A* → □ *B*) : jeśli konieczna jest implikacja z A do B to konieczność poprzednika A implikuje konieczność następnika B.

Reguły wnioskowania:

1) Jeśli coś zachodzi to jest konieczne:

$$\frac{A}{\Box A}$$

2) Klasyczna reguła *Modus Ponens* rozszerzona jest przez uwzględnienie *ścisłej implikacji* w poprzedniku reguły:

$$\frac{A, A \to B}{B}$$

5 Pytania testowe

5.1 Logika predykatów

1. Omówić elementy składni logiki predykatów.
2. Omówić semantykę logiki predykatów.
3. Przedstawić problem predykatowania formuł.
4. Na czym polegają: podstawienie i uzgadniania zmiennych?
5. Omówić typową kwantyfikację formuł i reguły eliminacji kwantyfikatorów
6. Omówić proces unifikacji formuł.
7. Przedstawić rachunek sytuacji – przeznaczenie, podstawowe elementy.

5.2 Inżynieria wiedzy

1. Czym zajmuje się inżynieria wiedzy?
2. Wymienić podstawowe kategorie ontologiczne.
3. Omówić system PROLOG.
4. Omówić przykładowy system regułowy.
5. Przedstawić sieć semantyczną i jej związek z językiem predykatów

5.3 Logiki rozszerzone

1. Scharakteryzować istotne cechy logiki niemonotonicznej.
2. Czym charakteryzuje się logika modalna?

6 Bibliografia

Podstawowa

1. S. Russel, P. Norvig: *Artificial Intelligence. A modern approach.* Prentice Hall, 2002 (2nd ed.), 2013 (3d ed.). [Rozdziały: 1, 2]
2. M. Flasiński: *Wstęp do sztucznej inteligencji*. Wydawnictwo Naukowe, PWN, 2011. [Rozdziały: 2, 14]
3. W. Traczyk: *Inżynieria wiedzy*. Oficyna Wydawnicza EXIT, Warszawa, 2010. [Rozdziały: 2, 3, 7, 8].