Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

ALGORYTMY DESKOLEMIZACJI I ICH ZASTOSOWANIA

praca magisterska
rok akademicki 1996/1997

Promotor:
prof.dr.hab. Andrzej Szałas

Autor:
Nguyen Anh Linh

Podziękowanie

Serdecznie dziękuję promotorowi prof. Andrzejowi Szałasowi za rady i za poprawki błędów językowych. Dziękuję koleżance Magdalenie Bajtlikowi za poprawki ortograficzne. Dziękuję Instytucie Informatyki za wykorzytanie drukarki.

SPIS TREŚCI

1 Wstęp

Formuły drugiego rzędu stanowią silne narzędzie opisania rzeczywistości i są używane w ważnych zastosowaniach jak we wnioskowaniu niemonotonicznym i w teorii korespondencji dla nieklasycznych logik. W wielu przypadkach rozpatrujemy możliwość znalezienia formuły pierwszego rzędu, która jest równoważna danej formule drugiego rzędu. Zostały skonstruowane algorytmy DLS [DLS94] i SCAN [GO92], które próbują wyeliminować kwantyfikatory relacyjne (predykatowe) z danej formuły. W tych algorytmach, skolemizacja jest wykorzystywana w trakcie działania w celu umożliwienia dalszych przekształceń. Skolemizacja tworzy nowe stałe i funkcje Skolema, przy czym formuły zawierające te funkcje nadal są drugiego rzędu. W końcowej fazie algorytmy DLS i SCAN próbują wyeliminować funkcje Skolema. Taką operację nazywamy *deskolemizacją*. Celem tej pracy jest opracowanie algorytmów deskolemizacji. Przed podaniem ścisłej definicji deskolemizacji omawiam podstawowe pojęcia, w tym podaję definicję klasycznej logiki pierwszego rzędu, logiki drugiego rzędu i skolemizacji.

1.1 Klasyczna logika pierwszego rzędu

1.1.1 Syntaktyka

Definicja 1.1 *Alfabet klasycznej logiki pierwszego rzędu* tworzą następujące rodzaje wyrażeń:

- *zmienne indywiduowe:*
  $x_1,\ x_2, \ldots,$
- n-argumentowe *symbole funkcyjne* ($n \geq 0$):
  $f_1^n,\ f_2^n, \ldots$
  (symbole 0-argumentowe nazywamy stałymi),
- n-argumentowe *symbole relacyjne* ($n \geq 1$):
  $P_1^n,\ P_2^n, \ldots,$
- *symbole logiczne*:
  $\neg,\ \wedge,\ \vee,\ \rightarrow,\ \leftrightarrow,\ \forall,\ \exists$
  zwane odpowiednio *negacją, koniunkcją, alternatywą, implikacją, równoważnością, kwantyfikatorem ogólnym, kwantyfikatorem egzystencjalnym,*

- *symbole interpunkcyjne*:
  ( ) ,
  (nawiasy i przecinek)

Definicja 1.2 *Językiem* (lub sygnaturą) klasycznej logiki pierwszego rzędu nazywamy zbiór symboli funkcji, relacji, zmiennych indywiduowych i spójników logicznych.

Definicja 1.3 *Zbiór termów* jest najmniejszym zbiorem spełniającym nastepujące warunki:

- stałe są termami,
- zmienne indywiduowe są termami,
- jeśli f jest symbolem funkcji n-argumentowej, a $t_1, \ldots, t_n$ są termami, to także $f(t_1, \ldots, t_n)$ jest termem.

Definicja 1.4 Jeśli P jest symbolem relacji n-argumentowej, $t_1, \ldots, t_n$ są termami, to $P(t_1, \ldots, t_n)$ jest *formułą atomową (atomem).*

Definicja 1.5 *Wystąpienie zmiennej* nazywamy *związanym* wtedy i tylko wtedy, gdy znaduje się w zakresie działania odpowiedniego kwantyfikatora. *Wystąpienie zmiennej* nazywamy *wolnym* wtedy i tylko wtedy, gdy nie jest związane.

Definicja 1.6 *Zmienna* jest *wolna* w danej formule, jeśli przynajmniej jedno jej wystąpienie w tej formule jest wolne. *Zmienna* jest *związana* w danej formule, jeśli wszystkie jej wystąpienia w tej formule są związane.

Definicja 1.7 Formułę bez zmiennych wolnych nazywamy *formułą zamkniętą*.

Definicja 1.8 *Zbiór formuł* jest najmniejszym zbiorem spełniającym warunki:

- atomy są formułami,
- jeśli α i β są formułami, to $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$ są formułami,
- jeśli α jest formuła, x jest zmienną wolną w α, to $(\forall x\ \alpha)$ i $(\exists x\ \alpha)$ są formułami.

1.1.2 Semantyka

Definicja 1.9 *Interpretacją formuły* α nazywamy parę (D, m), gdzie D (*dziedzina*) jest niepustym zbiorem, zaś m taką funkcją przyporządkującą wartości stałym, symbolom fynkcyjnym i symbolom relacyjnym w α, że:

- każdemu symbolowi stałej jest przyporządkowany element zbioru D,
- każdemu n-argumentowemu symbolowi funkcyjnemu f^n przyporządkowane jest odwzorowanie z D^n w D,
- każdemu n-argumentowemu symbolowi relacyjnemu P^n przyporządkowane jest odwzorowanie z D^n w {0,1}.

Oznaczenie Niech $AS(I)$ będzie zbiorem wszystkich *wartościowań zmiennych* w interpretacji $I = (D, m)$, to znaczy wszystkich funkcji v określonych na zbiorze zmiennych indywiduowych takich, że $v(x_k) \in D$.

Definicja 1.10 *Wartość termu* dla ustalonej interpretacji $I = (D, m)$ oraz wartościowania $v \in AS(I)$ określa się następująco:

$$\begin{array}{lcl} Wart_{I,v}(x_k) & = & v(x_k), \\ Wart_{I,v}(f_k^0) & = & m(f_k^0), \\ Wart_{I,v}(f_k^n(t_1, \ldots, t_n)) & = & m(f_k^n)(Wart_{I,v}(t_1), \ldots, Wart_{I,v}(t_n)). \end{array}$$

Definicja 1.11 *Wartość formuły atomowej* dla danej interpretacji $I = (D, m)$ oraz wartościowania $v \in AS(I)$ określa się następująco:

$$Wart_{I,v}(P_k^n(t_1, \ldots, t_n)) = m(P_k^n)(Wart_{I,v}(t_1), \ldots, Wart_{I,v}(t_n)).$$

Oznaczenie Niech $AS_v^x(I)$, dla wartościowania $v \in AS(I)$ oraz zmiennej indywiduowej x, będzie zbiorem wszystkich wartościowań v' takich, że dla dowolnej zmiennej indywiduowej y różnej od x zachodzi $v'(y) = v(y)$. Innymi słowami, wartościowania z $AS_v^x(I)$ mogą się różnić jedynie wartościami dla zmiennej x.

Definicja 1.12 *Wartość formuły złożonej* zależy wyłącznie od wartości symboli występujących w tej formule:

$$
\begin{array}{lcl}
Wart_{I,v}(\neg\alpha) & = & \begin{cases} 1, & \text{jeśli } Wart_{I,v}(\alpha) = 0, \\ 0, & \text{w przeciwnym przypadku} \end{cases} \\
Wart_{I,v}(\alpha \to \beta) & = & \begin{cases} 0, & \text{jeśli } Wart_{I,v}(\alpha) = 1 \text{ i } Wart_{I,v}(\beta) = 0, \\ 1, & \text{w przeciwnym przypadku} \end{cases} \\
Wart_{I,v}(\forall x\ \alpha) & = & \begin{cases} 1, & \text{jeśli } Wart_{I,v'}(\alpha) = 1 \text{ dla każdego } v' \in AS_v^x(I), \\ 0, & \text{w przeciwnym przypadku} \end{cases} \\
Wart_{I,v}(\alpha \wedge \beta) & = & Wart_{I,v}(\neg(\alpha \to \neg\beta)), \\
Wart_{I,v}(\alpha \vee \beta) & = & Wart_{I,v}(\neg\alpha \to \beta), \\
Wart_{I,v}(\alpha \leftrightarrow \beta) & = & Wart_{I,v}((\alpha \to \beta) \wedge (\beta \to \alpha)), \\
Wart_{I,v}(\exists x\ P(x)) & = & Wart_{I,v}(\neg(\forall x\ \neg P(x))).
\end{array}
$$

Definicja 1.13 Formuła α jest *spełniona przy interpretacji I oraz wartościowaniu $v \in AS(I)$* wtedy i tylko wtedy, gdy $Wart_{I,v}(\alpha) = 1$.

Formuła v jest *prawdziwa przy interpretacji I* wtedy i tylko wtedy, gdy $Wart_{I,v}(\alpha) = 1$ dla każdego wartościowania $v \in AS(I)$ (mówimy wówczas, że I *spełnia α, jest modelem* dla α).

Formuła α jest *spełnialna* wtedy i tylko wtedy, gdy istnieje taka interpretacja I oraz wartościowanie $v \in AS(I)$, przy których wartość formuły α wynosi 1.

Formuła α jest *prawdziwa (tautologią)* wtedy i tylko wtedy, gdy każda interpretacja I spełnia α.

Twierdzenie 1.1 (Church) Logika pierwszego rzędu nie jest rozstrzygalna, ale jest częściowo rozstrzygalna, tzn. nie istnieje algorytm, który dla danej (dowolnej) formuły α rozstrzyga czy jest ona tautologią, czy nie; istnieje jednak algorytm, który dla dowolnej formuły β, która jest tautologią, potrafi to stwierdzić.

1.2 Podstawowe pojęcia

Definicja 1.14 *Literałem* nazywamy atom lub jego negację. *Klauzulą* nazywamy alternatywę literałów.

Definicja 1.15 Formuły α i β są *równoważne* (ozn. $\alpha \Longleftrightarrow \beta$) jeśli formuła $\alpha \leftrightarrow \beta$ jest tautologią. Formuła β jest wnioskiem formuły α (ozn. $\alpha \Longrightarrow \beta$) jeśli formuła $\alpha \to \beta$ jest tautologią. (Znaki $\Longrightarrow, \Longleftrightarrow$ mają niższe preferencje łączenia niż spójniki logiczne)

1.2.1 Postaci normalne

Definicja 1.16 Formuła α jest postaci *NNF* (ang. negation normal form) jeśli α nie zawiera spójników $\rightarrow, \leftrightarrow$ i wszystkie spójniki $\neg$ występują bezpośrednio przed relacjami.

W celu sprowadzenia danej formuły do postaci NNF należy:

- wyeliminować spójniki $\rightarrow, \leftrightarrow$ za pomocą reguł:

$$\begin{aligned} \alpha \rightarrow \beta &\iff \neg\alpha \vee \beta, \\ \alpha \leftrightarrow \beta &\iff (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha), \end{aligned}$$

- wprowadzić znak negacji bezpośrednio przed symbole relacji zgodnie z wzorami:

$$\begin{aligned} \neg(\neg(\alpha)) &\iff \alpha, \\ \neg(\alpha \vee \beta) &\iff \neg\alpha \wedge \neg\beta, \\ \neg(\alpha \wedge \beta) &\iff \neg\alpha \vee \neg\beta, \\ \neg\forall x\ \varphi(x) &\iff \exists x\ \neg\varphi(x), \\ \neg\exists x\ \varphi(x) &\iff \forall x\ \neg\varphi(x) \end{aligned}$$

Definicja 1.17 Formuła α jest w *koniunkcyjnej postaci normalnej* (ang. conjunctive normal form) – CNF wtedy i tylko wtedy, gdy jest ona postaci $\alpha = \alpha_1 \wedge \ldots \wedge \alpha_n$ (dla $n \geq 1$), gdzie $\alpha_1, \ldots, \alpha_n$ są klauzulami.

Definicja 1.18 Formuła α jest w *alternatywej (lub dyzjunkcyjnej) postaci normalnej* (ang. disjunctive normal form) – DNF wtedy i tylko wtedy, gdy jest ona postaci $\alpha = \alpha_1 \vee \ldots \vee \alpha_n$ (dla $n \geq 1$), gdzie $\alpha_1, \ldots, \alpha_n$ są koniunkcjami literałów.

W celu sprowadzenia danej formuły nie zawierającej kwantyfikatorów do postaci CNF lub DNF należy sprowadzić ją do postaci NNF i wyprowadzić znak koniunkcji (CNF) lub alternatywy (DNF) na zewnątrz nawiasów (na najwyższy poziom formuły złożonej) przy użyciu praw:

$$\begin{aligned} \alpha \vee (\beta \wedge \gamma) &\iff (\alpha \vee \beta) \wedge (\alpha \vee \gamma), \\ \alpha \wedge (\beta \vee \gamma) &\iff (\alpha \wedge \beta) \vee (\alpha \wedge \gamma). \end{aligned}$$

Definicja 1.19 Formuła α jest w *postaci normalnej* (ang. prenex normal form – PNF) wtedy i tylko wtedy, gdy ona jest postaci $(Q_1x_1 \ldots Q_nx_n\ \mu)$, gdzie Q_kx_k (dla $1 \leq k \leq n$) to albo $\forall x_k$, albo $\exists x_k$, zaś μ – formuła bez kwantyfikatorów. $Q_1x_1 \ldots Q_nx_n$ nazywamy *przedrostkiem*, a μ – *matrycą* formuły α.

Istnieje metoda sprowadzenia danej formuły do postaci normalnej. W tym celu należy sprowadzić daną formułę do postaci NNF i przesunąć kwantyfikatory do przedrostka na podstawie reguł:

$$
\begin{array}{rcl}
(Qx\ \alpha(x)) \vee \beta & \Longleftrightarrow & Qx\ (\alpha(x) \vee \beta), \\
(Qx\ \alpha(x)) \wedge \beta & \Longleftrightarrow & Qx\ (\alpha(x) \wedge \beta), \\
(\forall x\ \alpha(x)) \wedge (\forall x\ \beta(x)) & \Longleftrightarrow & \forall x\ (\alpha(x) \wedge \beta(x)), \\
(\exists x\ \alpha(x)) \vee (\exists x\ \beta(x)) & \Longleftrightarrow & \exists x\ (\alpha(x) \vee \beta(x)), \\
(Q_1x\ \alpha(x)) \vee (Q_2x\ \beta(x)) & \Longleftrightarrow & Q_1x\ Q_2z\ (\alpha(x) \vee \beta(z)), \\
(Q_1x\ \alpha(x)) \wedge (Q_2x\ \beta(x)) & \Longleftrightarrow & Q_1x\ Q_2z\ (\alpha(x) \wedge \beta(z))
\end{array}
$$

(każdy z symboli Q, Q_1, Q_2 oznacza kwantyfikator $\forall$ lub $\exists$; w ostatnich dwóch regułach zmienna z nie może występować w formule $\alpha(x)$).

1.2.2 Skolemizacja

Definicja 1.20 Formuła jest w *standardowej postaci Skolema* wtedy i tylko wtedy, gdy jest w postaci normalnej, w jej przedrostku nie ma kwantyfikatorów egzystencjalnych ($\exists$), zaś jej matryca jest w postaci CNF.

W celu sprowadzenia formuły α do postaci Skolema należy:

- sprowadzić formułę α do postaci normalnej; $\alpha' = Q_1x_1 \ldots Q_nx_n\ \mu$,
- sprowadzić matrycę μ do postaci CNF; $\alpha'' = Q_1x_1 \ldots Q_nx_n\ \mu'$,
- przekształcić przedrostek $Q_1x_1 \ldots Q_nx_n$, powtarzając następujace czynności aż do usunięcia wszystkich kwantyfikatorów egzystencjalnych:
  - rozważmy dowolny kwantyfikator egzystencjalny Q_r z przedrostka, gdzie $1 \leq r \leq n$;

– jeśli w przedrostku przed Q_r nie występują kwantyfikatory ogólne ($\forall$), to w matrycy wymieniamy x_r na c (c jest nową stałą zwaną *stałą Skolema*, która nie występuje w μ'; czyli dla każdego x_r jest to inna stała) oraz usuwamy $Q_r x_r$ z przedrostka formuły α,

– jeśli w przedrostku przed kwantyfikatorem $Q_r x_r$ występuje jeszcze s kwantyfikatorów ogólnych $Q_{j_1} x_{j_1} \ldots Q_{j_s} x_{j_s}$, gdzie $1 \leq j_1 \leq \ldots \leq j_s \leq r$, to w matrycy μ' wymieniamy wystąpienia x_r na $f(x_{j_1}, \ldots, x_{j_s})$ (f jest nowym s-argumentowym symbolem funkcyjnym *funkcji Skolema*) oraz usuwamy $Q_r x_r$ z przedrostka.

Przykład 1.1 Sprowadzić formułę $\exists z\ \forall x\ ((R(z) \wedge P(x)) \to \exists y\ Q(x, y))$ do postaci normalnej Skolema.

Postać normalna (PNF): $\exists z\ \forall x\ \exists y\ (\neg R(z) \vee \neg P(x) \vee Q(x, y))$.

Postać normalna Skolema: $\forall x\ (\neg R(c) \vee \neg P(x) \vee Q(x, f(x)))$, gdzie c jest stałą Skolema, zaś f – funkcją Skolema.

Po sprowadzeniu danej formuły do postaci Skolema otrzymujemy formułę postaci: $\forall x_1\ \ldots \forall x_k\ (\alpha_1 \wedge \ldots \wedge \alpha_m)$, w której podformuły α_k (dla $1 \leq k \leq m$) są klauzulami. Formułę tego typu można reprezentować w postaci zbioru, którego elementami są tworzące ją klauzule. Zakłada się przy tym, że wszystkie zmienne występujące w α_k są związane (znajdują się w zasięgu działania kwantyfikatora ogólnego $\forall$). Również odwrotnie – zbiór klauzul $S = \{\alpha_1, \ldots, \alpha_m\}$ można traktować jako formułę będącą koniunkcją wszystkich jego elementów.

Twierdzenie 1.2 Niech S będzie zbiorem klauzul reprezentujących formułę α w postaci standardowej Skolema. Wówczas formuła α jest sprzeczna wtedy i tylko wtedy, gdy zbiór S jest sprzeczny. ([CL73])

1.3 Logika drugiego rzędu

Logika drugiego rzędu jest rozszerzeniem klasycznej logiki pierwszego rzędu o kwantyfikatory relacyjne i funkcyjne.

Definicja 1.21 *Zbiór formuł drugiego rzędu* jest najmniejszym zbiorem spełniającym warunki:

- Każda formuła klasycznej logiki pierwszego rzędu jest formułą drugiego rzędu.

- jeśli α i β są formułami drugiego rzędu, to $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$ są formułami drugiego rzędu,

- jeśli α jest formułą drugiego rzędu , x jest zmienną (indywiduową) wolną w α, to $(\forall x\ \alpha)$ i $(\exists x\ \alpha)$ są formułami drugiego rzędu.

- jeśli α jest formułą drugiego rzędu, f jest symbolem funkcji, to $(\exists f\ \alpha)$ jest formułą drugiego rzędu. Kwantyfikator $\exists f$ nazywamy *kwantyfikatorem funkcyjnym*, zaś f – zmienną funkcyjną.

- jeśli α jest formułą drugiego rzędu, P jest symbolem relacji, to $(\exists P\ \alpha)$ jest formułą drugiego rzędu. Kwantyfikator $\exists P$ nazywamy *kwantyfikatorem relacyjnym (lub predykatowym)*, zaś P – zmienną relacyjną.

Będziemy używać skrotu $(\forall f\ \alpha)$ dla $(\neg\exists f\ \neg\alpha)$, i $(\forall P\ \alpha)$ dla $(\neg\exists P\ \neg\alpha)$.

Definicja 1.22 *Wartość formuły drugiego rzędu* dla danej interpretacji $I = (D, m)$ oraz wartościowania $v \in AS(I)$ określa się następująco:

- Jeśli α jest formułą drugiego rzędu, f jest symbolem funkcji, to $Wart_{I,v}(\exists f\ \alpha)$ jest zdefiniowana następująco:
  - $Wart_{I,v}(\exists f\ \alpha) = 1$ jeśli $Wart_{I',v}(\alpha) = 1$ dla pewnej interpretacji $I' = (D, m')$, gdzie m' się różni od m jedynie wartością dla symbolu f,
  - $Wart_{I,v}(\exists f\ \alpha) = 0$ wpp

- Jeśli α jest formułą drugiego rzędu, P jest symbolem relacji, to $Wart_{I,v}(\exists P\ \alpha)$ jest zdefiniowana następująco:
  - $Wart_{I,v}(\exists P\ \alpha) = 1$ jeśli $Wart_{I',v}(\alpha) = 1$ dla pewnej interpretacji $I' = (D, m')$, gdzie m' się różni od m jedynie wartością dla symbolu P,
  - $Wart_{I,v}(\exists P\ \alpha) = 0$ wpp

- Dla formuły α, która nie jest postaci $\exists f\ \beta$ ani $\exists P\ \beta$, $Wart_{I,v}(\alpha)$ jest zdefiniowana w taki sam sposób jak w przypadku klasycznej logiki pierwszego rzędu.

Pojęcia takie jak prawdziwość formuły w modelu, spełnialność, tautologia dla logiki drugiego rzędu są zdefiniowane w taki sam sposób jak w przypadku klasycznej logiki pierwszego rzędu.

Twierdzenie 1.3 Niech po sprowadzeniu formuły I-rzędu α do postaci standardowej Skolema dostajemy formułę β i funkcje Skolema $f_1 \ldots, f_k$, stałe Skolema $c_1, \ldots, c_h$. Wtedy α jest równoważna formule drugiego rzędu $\exists f_1 \ldots \exists f_k \ \exists c_1 \ldots \exists c_h \ \beta$.

1.4 Tematyka i wkład pracy

Nieformalnie rozumiemy deskolemizację jako odwrotny proces do skolemizacji. Po przekształceniach takich jak eliminacja kwantyfikatorów relacyjnych formuła może się znacznie różnić od pierwotnej postaci, stąd często definiujemy deskolemizację jako proces eliminacji kwantyfikatorów funkcyjnych dla formuł bez kwantyfikatorów relacyjnych, które zawierają kwantyfikatory funkcyjne tylko na początku formuły.

Definicja 1.23 Niech dana będzie formuła $\exists f_1 \ \ldots \exists f_n \ \Phi$, gdzie $f_1, \ldots, f_n$ są zmiennymi funkcyjnymi, zaś Φ – formułą pierwszego rzędu. *Deskolemizacją* nazywamy proces, który znajduje taką formułę pierwszego rzędu Ψ, że $(\exists f_1 \ \ldots \exists f_n \ \Phi) \leftrightarrow \Psi$ jest tautologią logiki drugiego rzędu.

Głównym wkładem tej pracy jest algorytm "deskolemizacji opartej na postaci syntaktycznej" i jego implementacja. W tej pracy został skonstruowany także inny schemat deskolemizacji, który polega na wyszukiwaniu podstawień zdefiniowanych w logice wielorodzajowej dla egzystencjalnych zmiennych funkcyjnych. W tej pracy wykażę związek między deskolemizacją a zagadnieniem syntezy programów. Algorytm syntezy programów został skonstruowany i zaimplementowany. Ten ostatni algorytm wykazuje na przydatność schematu deskolemizacji. Oba programy zostały napisane w języku C.

1.5 Omówienie literatury

Niektóre pomysły występujące w tej pracy pochodzą z innych prac. Na przykład, rozdzielenie zmiennej drugiego rzędu pochodzi z [DLS94], wyszukiwanie podstawień dla zmiennych drugiego rzędu występuje wcześniej w wielu pracach (np. w [Sim94]). Pisząc tę pracę oparłem się głównie na następujących opracowaniach:

- [Sz93], [NOS96] – eliminacja kwantyfikatorów drugiego rzędu,
- [Sz93], [Łuk90] – logiki i wnioskowanie,
- [GN87], [Wój91], [Sch94] – rezolucja,
- [Tar94], [LO94] – specyfikacja i synteza programów.

2 Deskolemizacja oparta na postaci syntaktycznej

Algorytm deskolemizacji podany w tym rozdziale wykorzystuje syntaktyczne przekształcenia do eliminacji kwantyfikatorów funkcyjnych. Zostaną podane pewne postaci formuł, dla których można od razu eliminować kwantyfikatory funkcyjne. Strategią algorytmu jest sprowadzenie danej formuły do jednej z tych postaci.

Zakładam, że formuły omawiane w tym rozdziale są już w postaci NNF (tzn. negation normal form). Dodatkowo zakładam, że w formułach wszystkie zmienne związane są różne. Jeśli tak nie jest (na początku lub po pewnym przekształceniu) to zmieniamy je tak, żeby były różne.

Zmienne indywiduowe będę oznaczać przez x, y, z, t, u, v, a zmienne funkcyjne przez f, g, h, k. Zakładamy, że struktura formuł jest drzewem (a nie jest liniowa). W związku z tym, mówimy, że dany symbol występuje *przed* innym symbolem jeśli pierwszy z nich występuje na ścieżce prowadzającej od korzenia do drugiego symbolu. Zmienne funkcyjne to zmienne związane deklarowane pod kwantyfikatorami funkcyjnymi. *Ciąg kwantyfikatorów* (lub *ścieżka kwantyfikatorów*) to ciąg kwantyfikatorów pierwszego rzędu (np. $\forall x\, \forall y\, \exists z\, \forall t$), *zmienna ogólna* – to zmienna indywiduowa związana wymieniona pod kwantyfikatorem ogólnym, a *zmienna egzystencjalna* – to zmienna związana wymieniona pod kwantyfikatorem egzystencjalnym (np. w formule $\forall x\, \exists y\ \varphi(x, y)$ x jest zmienną ogólną, a y jest zmienną egzystencjalną).

2.1 Podstawy teoretyczne

W tym podrozdziale sformułuję cztery lematy, które wprowadzają ważne przekształcenia i będą używane w dowodzie poprawności algorytmu deskolemizacji podanego w tym rozdziale.

Definicja 2.1 *Mówimy, że w formule Φ funkcja f nie występuje wewnątrz siebie jeśli w Φ nie ma termu postaci $f(\ldots\ f(\ldots)\ \ldots)$.*

Lemat 2.1 (szczególny przypadek lematu 2.3) *Niech dana będzie formuła $\exists f\ CK\ \Phi(f(t_1, \ldots, t_n))$, gdzie CK jest ciągiem kwantyfikatów, Φ jest formułą, f jest zmienną funkcyjną, $t_1, \ldots, t_n$ są termami i wszystkie wystąpienia f w Φ są takie same i mają postać $f(t_1, \ldots, t_n)$. Jeśli w Φ, f nie występuje wewnątrz siebie, wszystkie zmienne występujące* *w $f(t_1,\dots,t_n)$ są deklarowane w CK, wszystkie zmienne ogólne z CK występują w $f(t_1,\dots,t_n)$ jako bezpośrednie parametry, to można wyeliminować f z $\exists f\ CK\ \Phi(f(t_1,\dots,t_n))$ zachowując równoważność i wynikiem będzie $CK\ \exists y\ \Phi(y)$, gdzie y jest nową zmienną nie występującą w $CK\ \Phi(f(t_1,\dots,t_n))$.*

Dowód:
($\Longrightarrow$) Oczywiście mamy:

$$\exists f\ CK\ \Phi(f(t_1,\dots,t_n)) \implies CK\ \exists y\ \Phi(y)$$

($\Longleftarrow$) Musimy udowodnić, że:

$$CK\ \exists y\ \Phi(y) \implies \exists f\ CK\ \Phi(f(t_1,\dots,t_n))$$

Skolemizując $CK\ \Phi(f(t_1,\dots,t_n))$ dla wszystkich zmiennych egzystencjalnych z CK dostajemy nową formułę $\exists g_1 \ \dots \exists g_h\ CK^{'}\ \Phi^{'}(f(t^{'}{}_1,\dots,t^{'}{}_n))$. Teraz $CK^{'}$ zawiera tylko zmienne ogólne, niech $CK^{'}$ będzie $\forall x_1 \dots \forall x_k$. Skolemizując $CK\ \exists y\ \Phi(y)$ dla wszystkich zmiennych egzystencjalnych z CK i dla y dostajemy $\exists f_y\ \exists g_1 \ \dots \exists g_h\ CK^{'}\ \Phi^{'}(f_y(x_1,\dots,x_k))$. Bez zmniejszenia ogólności założymy, że $x_1,\dots,x_k$ są pierwszymi parametrami wystąpień f. Ustalmy dowolny model. Jeśli $CK\ \exists y\ \Phi(y)$ nie jest prawdziwa, to niech f będzie dowolną funkcją. W przeciwnym przypadku niech $f_y, g_1,\dots,g_h$ będą takimi funkcjami, że $CK^{'}\ \Phi^{'}(f_y(x_1,\dots,x_k))$ będzie prawdziwa. Określimy f w następujący sposób: dla dowolnych ustalonych $x_1,\dots,x_k$, niech $f(x_1,\dots,x_k,t^{'}_{k+1},\dots,t^{'}_n)$ będzie $f_y(x_1,\dots,x_k)$. Dla pozostałych zestawień parametrów niech wartość f będzie dowolna. W ten sposób f jest dobrze zdefiniowana. Mamy:

$$\begin{array}{ll} & CK\ \exists y\ \Phi(y) \\ \Longrightarrow & \exists f_y\ \exists g_1\ \dots\ \exists g_h\ CK^{'}\ \Phi^{'}(f_y(x_1,\dots,x_k)) \\ \Longrightarrow & \exists f\ \exists g_1\ \dots\ \exists g_h\ CK^{'}\ \Phi^{'}(f(t^{'}{}_1,\dots,t^{'}{}_n)) \\ \Longrightarrow & \exists f\ CK\ \Phi(f(t_1,\dots,t_n))\ \Box \end{array}$$

Przykład 2.1 $(\exists f\ \forall x\ \Phi(x,f(x))) \leftrightarrow (\forall x\ \exists y\ \Phi(x,y))$

Przykład 2.2

$$\begin{array}{ll} & \exists f\ \forall x\ \exists y\ \forall z\ \exists t\ \forall p\ \forall q\ \Phi(x,y,z,t,p,q,f(g(y,z,a),x,z,q)) \\ \Longleftrightarrow & \exists f\ \forall x\ \exists y\ \forall z\ \exists t\ \forall q\ \forall p\ \Phi(x,y,z,t,p,q,f(g(y,z,a),x,z,q)) \\ \Longleftrightarrow & \forall x\ \exists y\ \forall z\ \exists t\ \forall q\ \exists u\ \forall p\ \Phi(x,y,z,t,p,q,u) \end{array}$$

Lemat 2.2 (rozdzielenie zmiennej funkcyjnej) *Niech dana będzie formuła* $\exists f\ CK\ \Phi(f) \vee \Psi(f)$, *gdzie* f *jest zmienną funkcyjną,* CK *jest ciągiem kwantyfikatorów,* Φ, Ψ *są formułami. Niech* $x_1, \ldots, x_k$ *są wszystkimi zmiennymi ogólnymi w* CK. *Jeśli w formule* $\Phi(f) \vee \Psi(f)$, f *nie występuje wewnątrz siebie i istnieją indeksy* $i_1, \ldots, i_k$ *takie, że dla każdego* j $(1 \leq j \leq k)$ x_j *występuje jako argument o numerze* i_j *we wszystkich wystąpieniach* f *w* $(\Phi(f) \vee \Psi(f))$, *to*

$$(\exists f\ CK\ \Phi(f) \vee \Psi(f)) \iff (\exists f_1\ \exists f_2\ CK\ \Phi(f_1) \vee \Psi(f_2))$$

Dowód:
($\Longrightarrow$) Oczywiście mamy:

$$(\exists f\ CK\ \Phi(f) \vee \Psi(f)) \implies (\exists f_1\ \exists f_2\ CK\ \Phi(f_1) \vee \Psi(f_2))$$

($\Longleftarrow$) Musimy udowodnić, że

$$(\exists f_1\ \exists f_2\ CK\ \Phi(f_1) \vee \Psi(f_2)) \implies (\exists f\ CK\ \Phi(f) \vee \Psi(f))$$

Skolemizując $CK\ \Phi(f) \vee \Psi(f)$ dla wszystkich zmiennych egzystencjalnych z CK dostajemy $\exists g_1\ \ldots \exists g_h\ \forall x_1\ \ldots\ \forall x_k\ \Phi^{'}(f) \vee \Psi^{'}(f)$. Ustalmy dowolny model. Jeśli $\exists f_1\ \exists f_2\ CK\ \Phi(f_1) \vee \Psi(f_2)$ nie jest prawdziwa to niech f będzie dowolną funkcją, wpp konstrujemy f w następujący sposób: Niech $f_1, f_2, g_1, \ldots, g_h$ będą dowolnymi funkcjami takimi, że $\forall x_1\ \ldots \forall x_k\ \Phi^{'}(f_1) \vee \Psi^{'}(f_2)$ będzie prawdziwa. Bez zmniejszenia ogólności założymy, że $i_1, \ldots, i_k$ są to $1, \ldots, k$. Dla dowolnych ustalonych $x_1, \ldots, x_k$ jeśli $\Phi^{'}(f_1)$ jest prawdziwa to niech $f(x_1, \ldots, x_k, \_, \ldots, \_) = f_1(x_1, \ldots, x_k, \_, \ldots, \_)$ wpp niech $f(x_1, \ldots, x_k, \_, \ldots, \_) = f_2(x_1, \ldots, x_k, \_, \ldots, \_)$. W ten sposób f jest dobrze zdefiniowana. Mamy

$$\begin{array}{ll}
 & \exists f_1\ \exists f_2\ CK\ \Phi(f_1) \vee \Psi(f_2) \\
\Longrightarrow & \exists f_1\ \exists f_2\ \exists g_1\ \ldots \exists g_k\ \forall x_1 \ldots \forall x_k\ \Phi^{'}(f_1) \vee \Psi^{'}(f_2) \\
\Longrightarrow & \exists g_1\ \ldots \exists g_k\ \exists f\ \forall x_1 \ldots \forall x_k\ \Phi^{'}(f) \vee \Psi^{'}(f) \\
\Longrightarrow & \exists f\ \exists g_1\ \ldots \exists g_k\ \forall x_1 \ldots \forall x_k\ \Phi^{'}(f) \vee \Psi^{'}(f) \\
\Longrightarrow & \exists f\ CK\ \Phi(f) \vee \Psi(f)\ \Box
\end{array}$$

Przykład 2.3 $(\exists f\ \varphi(f) \vee \psi(f)) \iff (\exists f_1\ \exists f_2\ \varphi(f_1) \vee \psi(f_2))$

Przykład 2.4

$$\begin{array}{ll}
 & \exists f\ \forall x\ (\forall y\ \varphi(x, y, f(x, y))) \vee (\forall z\ \psi(x, z, f(x, z))) \\
\iff & \exists f_1\ \exists f_2\ \forall x\ (\forall y\ \varphi(x, y, f_1(x, y))) \vee (\forall z\ \psi(x, z, f_2(x, z))) \\
\iff & \forall x\ (\forall y\ \exists u\ \varphi(x, y, u)) \vee (\forall z\ \exists v\ \psi(x, z, v))
\end{array}$$

Przykład 2.5

$$
\begin{array}{ll}
& \exists f\ \forall x\ \varphi(x, f(x,x)) \vee (\forall y\ \psi(x,y,f(y,x))) \\
\Longleftrightarrow & \exists f_1\ \exists f_2\ \forall x\ \varphi(x, f_1(x,x)) \vee (\forall y\ \psi(x,y,f_2(y,x))) \\
\Longleftrightarrow & \forall x\ (\exists u\ \varphi(x,u)) \vee (\forall y\ \exists v\ \psi(x,y,v))
\end{array}
$$

Definicja 2.2 *Niech $CK1, CK2$ będą ciągami kwantyfikatorów. Piszemy, że $CK1 \leq CK2$ jeśli $CK1$ jest przedziałem początkowym $CK2$.*

Lemat 2.3 będzie rozszerzeniem lematu 2.1 dla przypadku gdy w Φ jest kilka różnych wystąpień funkcji f. Przekształcenie dokonane w lemacie 2.3 nazywamy "kompromisem".

Lemat 2.3 (kompromis) *Niech dana będzie formuła $\exists f\ CK\ \Phi$, gdzie f jest zmienną funkcyjną n-argumentową, CK jest ciągiem kwantyfikatorów, Φ jest formułą zależną od f i CK. Niech $f(t_{11}, \ldots, t_{1n}), \ldots, f(t_{m1}, \ldots, t_{mn})$ będą wszystkimi różnymi wystąpieniami funkcji f w Φ. Jeśli w Φ, f nie występuje wewnątrz siebie i istnieją ciągi kwantyfikatorów $CK_1, \ldots, CK_m$ takie, że $CK_1 \leq \ldots \leq CK_m \leq CK$ i spełnione są następujące warunki:*

1. *Dla dowolnego i ($1 \leq i \leq m$) wszystkie zmienne występujące w $f(t_{i1}, \ldots, t_{in})$ pochodzą z ciągu CK_i.*
2. *Dla dowolnego i ($1 \leq i \leq m$) wszystkie zmienne ogólne z ciągu CK_i występują w $f(t_{i1}, \ldots, t_{in})$ jako bezpośrednie parametry.*
3. *Dla dowolnych i, j ($1 \leq i < j \leq m$) niech $x_1, \ldots, x_{h_i}$ będą wszystkimi zmiennymi ogólnymi ciągu CK_i. Istnieją indeksy (pozycje) $l_1, \ldots, l_{h_i}$ takie, że $x_1, \ldots, x_{h_i}$ występują jako parametry w pozycjach $l_1, \ldots, l_{h_i}$ w $f(t_{i1}, \ldots, t_{in})$ i $f(t_{j1}, \ldots, t_{jn})$ (tzn. dla kazdego k ($1 \leq k \leq h_i$) $t_{il_k} = t_{jl_k} = x_{l_k}$).*

to da się wyeliminować f z formuły $\exists f\ CK\ \Phi$ zachowując równoważność i wynik będzie określony w następujący sposób:

- *Niech $CK_1', \ldots, CK_m', CK_{m+1}'$ będą ciągami kwantyfikatorów takimi, że*

$$
\begin{array}{rcl}
CK_1 & = & CK_1' \\
CK_i & = & CK_{i-1} CK_i' \quad \text{dla } 1 < i \leq m \\
CK & = & CK_1' \ldots CK_m' CK_{m+1}'
\end{array}
$$

  Niech $CK' = CK_1'\ \exists y_1\ \ldots CK_m'\ \exists y_m\ CK_{m+1}'$, gdzie $y_1, \ldots, y_m$ są nowymi zmiennymi.

- *Dla każdych i, j ($1 \leq i < j \leq m$) niech φ_{ij} będzie formułą $(\bigwedge_{1 \leq h \leq n} t_{ih} = t_{jh}) \rightarrow y_i = y_j$ (niektóre równości $t_{ih} = t_{jh}$ są zbędne bo dwie strony są identyczne).*

- *Wynikiem będzie*
$$CK' \bigwedge_{1 \leq i < j \leq m} \varphi_{ij} \wedge \Phi'$$
gdzie Φ' pochodzi z Φ przez zastąpienie $f(t_{i1}, \ldots, t_{in})$ przez y_i dla wszystkich $1 \leq i \leq m$

Dowód:
($\Longrightarrow$) Trywialnie mamy

$$(\exists f \ CK \ \Phi) \rightarrow (CK' \bigwedge_{1 \leq i < j \leq m} \varphi_{ij} \wedge \Phi')$$

($\Longleftarrow$) Musimy udowodnić, że

$$(CK' \bigwedge_{1 \leq i < j \leq m} \varphi_{ij} \wedge \Phi') \rightarrow (\exists f \ CK \ \Phi)$$

Niech po skolemizacji $CK \quad \Phi$ dla zmiennych egzystencjalnych z CK dostajemy $\exists g_1 \ldots \exists g_p \quad \forall x_1 \ldots \forall x_q \quad \Phi^*$, a po skolemizacji $CK' \quad \bigwedge_{1 \leq i < j \leq m} \varphi_{ij} \quad \wedge \Phi'$ dla zmiennych egzystencjalnych z CK' dostajemy $\exists f_1 \ldots \exists f_m \ \exists g_1 \ldots \exists g_p \ \forall x_1 \ldots \forall x_q \ \Phi'^*$, gdzie $f_1, \ldots, f_m$ są funkcjami Skolema odpowiednio dla $y_1, \ldots, y_m$. Niech po skolemizacji, t_{ij} stanie się t^*_{ij}, a y_i stanie się t_i ($1 \leq i \leq m$, $1 \leq j \leq n$). Ustalmy dowolny model. Załóżmy, że prawdziwa jest formuła $CK' \quad \bigwedge_{1 \leq i < j \leq m} \varphi_{ij} \quad \wedge \Phi'$. Wtedy istnieje $f_1, \ldots, f_m, g_1, \ldots, g_p$ takie, że formuła $\forall x_1 \quad \ldots \forall x_q \quad \Phi'^*$ będzie prawdziwa. Konstruujemy f w następujący sposób: Dla dowolnych $x_1, \ldots, x_q$, dla dowolnego i, niech $f(t^*_{i1}, \ldots, t^*_{in})(x_1, \ldots, x_q) = t_i(x_1, \ldots, x_q)$. Dla pozostałych zestawień parametrów niech wartość funkcji f będzie dowolna. Wystarczy udowodnić, że f jest dobrze zdefiniowana. Czyli trzeba udowodnić, że dla dowolnych ustalonych $x^0_1, \ldots, x^0_q$, $x^1_1, \ldots, x^1_q$, dla dowolnych i, j ($1 \leq i \leq m, 1 \leq j \leq m$) jeśli $\bigwedge_{1 \leq l \leq n} t^*_{il}(x^0_1, \ldots, x^0_q) = t^*_{jl}(x^1_1, \ldots, x^1_q)$ (oznaczamy to przez (*)), to $t_i(x^0_1, \ldots, x^0_q) = t_j(x^1_1, \ldots, x^1_q)$. Zakładamy, że zachodzi (*). Jeśli $i = j$, to warunki 1, 2 zapewniają prawdziwość powyższej formuły (zobacz dowód lematu 2.1). Więc możemy zakładać, że $i < j$. Z (*) i z warunku 3 wynika, że dla wszystkich k ($1 \leq k \leq h_i$)

mamy $x_k^0 = x_k^1$. Dalej warunki 1,2 gwarantuja, że dla wszystkich l $(1 \le l \le n)$ $t_{il}^*(x_1^0, \ldots, x_q^0) = t_{il}^*(x_1^1, \ldots, x_q^1)$. Stąd z (*) wynika, że dla każdego l $(1 \le l \le n)$ $t_{il}^*(x_1^1, \ldots, x_q^1) = t_{jl}^*(x_1^1, \ldots, x_q^1)$. Z założeń i z definicji φ_{ij} wynika, że $t_i(x_1^1, \ldots, x_q^1) = t_j(x_1^1, \ldots, x_q^1)$. Z drugiej strony, mamy $t_i(x_1^0, \ldots, x_q^0) = t_i(x_1^1, \ldots, x_q^1)$ (bo $x_k^0 = x_k^1$ dla $1 \le k \le h_i$). Więc $t_i(x_1^0, \ldots, x_q^0) = t_j(x_1^1, \ldots, x_q^1)$ $\Box$

Przykład 2.6

$$\begin{aligned} & \exists f\ \forall x\ \exists y\ \exists z\ \Phi(x, y, z, f(x, g(x)), f(x, h(y, z))) \\ \Longleftrightarrow\ & \forall x\ \exists u\ \exists y\ \exists z\ \exists v\ (g(x) = h(y, z) \to u = v) \wedge \Phi(x, y, z, u, v) \end{aligned}$$

Przykład 2.7

$$\begin{aligned} & \exists f\ \forall y\ \forall x\ \Phi(x, y, f(x, x), f(x, y)) \\ \Longleftrightarrow\ & \exists f\ \forall x\ \forall y\ \Phi(x, y, f(x, x), f(x, y)) \\ \Longleftrightarrow\ & \forall x\ \exists u\ \forall y\ \exists v\ (x = y \to u = v) \wedge \Phi(x, y, u, v) \end{aligned}$$

Definicja 2.3 *Mówimy, że podformuła Ψ formuły Φ jest pozytywna w Φ jeśli w Φ formuła Ψ nie jest poprzedzona przez negację $(\neg)$.*

Lemat 2.4 (pominięcie szumu) *Niech dana będzie formuła $\exists f\ CK\ \Omega$, gdzie f jest zmienną funkcyjną, CK jest ciągiem kwantyfikatorów, Ω jest formułą zależną od f i CK. Niech Φ będzie pozytywną podformułą formuły Ω taką, że w Ω tylko Φ zawiera f i w Ω formuła Φ nie jest poprzedzona przez kwantyfikatory.*
(1) Jeśli przez przekształcenie omówione w lemacie 2.2 dla formuły $\exists f\ CK\ \Phi$ dostajemy $\exists f_1\ \exists f_2\ CK\ \Psi$ to $(\exists f\ CK\ \Omega) \Longleftrightarrow (\exists f_1\ \exists f_2\ CK\ \Omega[\Phi \leftarrow \Psi])$.
(2) Jeśli przez przekształcenie omówione w lemacie 2.3 dla formuły $\exists f\ CK\ \Phi$ dostajemy $CK'\ \Psi$ to $(\exists f\ CK\ \Omega) \Longleftrightarrow (CK'\ \Omega[\Phi \leftarrow \Psi])$.
Gdzie $\Omega[\Phi \leftarrow \Psi]$ jest dostany z Ω przez podstawienie Φ przez Ψ.

Dowód:
((1) $\Longrightarrow$) Trywialne.
((1) $\Longleftarrow$) (Używanie symboli w tym lemacie i w lemacie 2.2 jest dwuznaczne, ale nie ma wpływu na dowód) Sposób konstruowania f się nie zmieni. Zasadniczym dowodem tego przeniesienia jest to, że dla ustalonych wartości dla zmiennych z CK jeśli $\Psi \to \Phi$ to $\Omega[\Phi \leftarrow \Psi] \to \Omega$ bo Φ jest pozytywna w Ω.
((2) $\Longrightarrow$) Trywialne.

((2) $\Longleftarrow$) Sposób konstruowania funkcji f będzie nieco zmodyfikowany. Różnica polega na tym, że teraz nie możemy zakładać, że formuła $CK' \quad \bigwedge_{1\leq i<j\leq m} \varphi_{ij} \quad \wedge \Phi'$ jest prawdziwa. Definiuję f następująco: Dla dowolnych $x_1,\ldots,x_q$, dla dowolnego i, *jeśli* Φ'^* *jest prawdziwa to* niech $f(t^*_{i1},\ldots,t^*_{in})(x_1,\ldots,x_q) = t_i(x_1,\ldots,x_q)$. Dla pozostałych zewstawień parametrów niech wartość funkcji f będzie dowolna. Żeby udowodnić, że f jest dobrze zdefiniowana trzeba udowodnić, że dla dowolnych $1 \leq i \leq m, 1 \leq j \leq m$, dla dowolnych $x^0_1,\ldots,x^0_q$, $x^1_1,\ldots,x^1_q$, jeśli prawdziwe są trzy formuły $\Phi'^*(x^0_1,\ldots,x^0_q)$, $\Phi'^*(x^1_1,\ldots,x^1_q)$, $\bigwedge_{1\leq l\leq n} t^*_{il}(x^0_1,\ldots,x^0_q) = t^*_{jl}(x^1_1,\ldots,x^1_q)$ to $t_i(x^0_1,\ldots,x^0_q) = t_j(x^1_1,\ldots,x^1_q)$. Dowód tej tezy jest analogiczny jak w lemacie 2.3. Więc dla dowolnych ustalonych $x_1,\ldots,x_q$ mamy $(\bigwedge_{1\leq i<j\leq m} \varphi_{ij} \quad \wedge \Phi'^*) \rightarrow \Phi^*$. Niech po skolemizacji $CK' \quad \Omega$ dla zmiennych egzystencjalnych z CK' dostajemy $\exists f_1 \ldots \exists f_m \quad \exists g_1 \ldots \exists g_p \quad \forall x_1 \ldots \forall x_q \quad \Omega^*$. Przy dowolnych ustalonych $x_1,\ldots,x_q$ mamy $\Omega^*[\Phi^* \leftarrow (\bigwedge_{1\leq i<j\leq m} \varphi_{ij} \quad \wedge \Phi'^*)] \rightarrow \Omega^*$ bo $(\bigwedge_{1\leq i<j\leq m} \varphi_{ij} \quad \wedge \Phi'^*) \rightarrow \Phi^*$ i Φ^* jest pozytywna w Ω^*. Stąd wynika, że $(CK' \ \Omega[\Phi \leftarrow (\bigwedge_{1\leq i<j\leq m} \varphi_{ij} \quad \wedge \Phi')]) \Longrightarrow (\exists f \ CK \ \Omega)$ $\square$

2.2 Omówienie podstawowych przekształceń

W tym podrozdziale omawiam o podstawowe przekształcenia używane w algorytmie deskolemizacji podanym w tym rozdziale. Używam następujących symboli: f to zmienna funkcyjna, Φ to formuła zawierająca f.

Podstawowe operacje (przekształcenia):

- "Redukowanie zakresu"(f,Φ): to funkcja, która wybiera z Φ najmniejszą podformułę Φ' taką, że w Φ tylko Φ' zawiera f i Φ' jest poprzedzona przez tylko spójniki $\wedge, \vee$. (zobacz lemat 2.4 "pominięcie szumu")
- "Rozdzielenie zmiennej funkcyjnej"(f,Φ): zobacz lemat 2.2.
- "Kompromis"(f,Φ): zobacz lemat 2.3.
- "Sklejanie zmiennych ogólnych"(f,Φ): Sklejanie zmiennych ogólnych, które występują w tych samych pozycjach w listach parametrów różnych wystąpień funkcji f w Φ (zobacz algorytm 2.2) . Ta operacja daje więcej możliwości "rozdzielenia zmiennej funkcyjnej"(f,Φ) i bierze

udział w procesie sprowadzającym Φ do postaci, dla której można zastosować lematy 2.3, 2.4.

Przykład 2.8

$$
\begin{array}{ll}
 & \exists f\ (\forall x_1\ \varphi(x_1, f(x_1))) \wedge (\forall x_2\ \psi(x_2, f(x_2))) \\
\Longleftrightarrow & \exists f\ \forall x\ \varphi(x, f(x)) \wedge \psi(x, f(x)) \\
\Longleftrightarrow & \forall x\ \exists y\ \varphi(x, y) \wedge \psi(x, y)
\end{array}
$$

Przykład 2.9

$$
\begin{array}{ll}
 & \exists f\ (\forall x\ \varphi(x, f(x)) \wedge (\forall y\ \psi(x, y, f(y)))) \wedge (\forall z\ \chi(z, f(z))) \\
\Longleftrightarrow & \exists f\ \exists x\ \forall y\ \varphi(x, f(x)) \wedge \psi(x, y, f(y)) \wedge \chi(y, f(y)) \\
\Longleftrightarrow & \exists x\ \exists u\ \forall y\ \exists v\ (x = y \to u = v) \wedge \varphi(x, u) \wedge \psi(x, y, v) \wedge \chi(y, v)
\end{array}
$$

Przykład 2.10

$$
\begin{array}{ll}
 & \exists f\ (\forall x_1\ \forall y_1\ \varphi(x_1, y_1, f(x_1, y_1)))\ \wedge \\
 & \quad (\forall x_2\ \ (\forall y_2\ \psi(x_2, y_2, f(x_2, y_2)))\ \vee \\
 & \qquad\quad (\forall y_3\ \chi(x_2, y_3, f(x_2, y_3)))) \\
\Longleftrightarrow & \exists f\ \forall x\ (\forall y_1\ \varphi(x, y_1, f(x, y_1)))\ \wedge \\
 & \qquad ((\forall y_2\ \psi(x, y_2, f(x, y_2))) \vee (\forall y_3\ \chi(x, y_3, f(x, y_3)))) \\
\Longleftrightarrow & \exists f\ \forall x\ ((\forall y_1\ \varphi(x, y_1, f(x, y_1))) \wedge (\forall y_2\ \psi(x, y_2, f(x, y_2))))\ \vee \\
 & \qquad ((\forall y_1\ \varphi(x, y_1, f(x, y_1))) \wedge (\forall y_3\ \chi(x, y_3, f(x, y_3)))) \\
\Longleftrightarrow & \exists f_1\ \exists f_2\ \forall x \\
 & \quad ((\forall y_1\ \varphi(x, y_1, f_1(x, y_1))) \wedge (\forall y_2\ \psi(x, y_2, f_1(x, y_2))))\ \vee \\
 & \quad ((\forall y_1\ \varphi(x, y_1, f_2(x, y_1))) \wedge (\forall y_3\ \chi(x, y_3, f_2(x, y_3)))) \\
\Longleftrightarrow & \exists f_1\ \exists f_2\ \forall x\ (\forall y_{12}\ \varphi(x, y_{12}, f_1(x, y_{12})) \wedge \psi(x, y_{12}, f_1(x, y_{12})))\ \vee \\
 & \qquad\qquad\quad (\forall y_{13}\ \varphi(x, y_{13}, f_2(x, y_{13})) \wedge \chi(x, y_{13}, f_2(x, y_{13}))) \\
\Longleftrightarrow & \forall x\ (\forall y_{12}\ \exists u\ \varphi(x, y_{12}, u) \wedge \psi(x, y_{12}, u))\ \vee \\
 & \quad (\forall y_{13}\ \exists v\ \varphi(x, y_{13}, v) \wedge \chi(x, y_{13}, v))
\end{array}
$$

- "Przesunięcie kwantyfikatorów wgłąb"(Φ): Ta operacja zmniejszy zakresy, w których się obozują kwantyfikatory. Można zrozumieć tę operację jako przesunięcie spójników $\wedge, \vee$ na zewnątrz kwantyfikatorów. To daje więcej możliwości manipulacji formułą. (zobacz algorytm 2.1)

- "and-in"(Φ): Ta operacja przesuwa spójniki $\wedge$ wgłąb względem spójnika $\vee$. Inaczej mówiąc, ta operacja przekształca Φ w dyzjunkcję (możliwie najwięcej) formuł.

$$(\varphi \vee \psi) \wedge \chi \Longleftrightarrow (\varphi \wedge \chi) \vee (\psi \wedge \chi)$$
$$\chi \wedge (\varphi \vee \psi) \Longleftrightarrow (\chi \wedge \varphi) \vee (\chi \wedge \psi)$$

- "or-in"(Φ): dualnie do "and-in".

- "Skracanie formuły"(Φ): Ta operacja jest używana w procedurach "and-in", "or-in", "przesunięcie kwantyfikatorów wgłąb". W praktyce ta operacja okazuje się bardzo ważna bo rozmiar formuły dynamicznie rośnie po przekształceniach. To jest jedyna operacja mająca "charakter semantyczny" używana w algorytmie deskolemizacji podanym w tym rozdziale.

- "Zmiana kolejności kwantyfikatorów":

$$(\forall x\ \forall y\ \varphi(x,y)) \Longleftrightarrow (\forall y\ \forall x\ \varphi(x,y))$$
$$(\exists x\ \exists y\ \varphi(x,y)) \Longleftrightarrow (\exists y\ \exists x\ \varphi(x,y))$$

- "Wyciąnięcie kwantyfikatorów na zewnątrz":

$$((\forall x\ \varphi(x)) \wedge \psi) \Longleftrightarrow (\forall x\ (\varphi(x) \wedge \psi))$$
$$(\psi \wedge (\forall x\ \varphi(x))) \Longleftrightarrow (\forall x\ (\psi \wedge \varphi(x)))$$
$$((\exists x\ \varphi(x)) \wedge \psi) \Longleftrightarrow (\exists x\ (\varphi(x) \wedge \psi))$$
$$(\psi \wedge (\exists x\ \varphi(x))) \Longleftrightarrow (\exists x\ (\psi \wedge \varphi(x)))$$

W dalszej części opiszę algorytmy "przesunięcie kwantyfikatorów wgłąb" i "sklejanie zmiennych ogólnych"; algorytm "skracanie formuł" będzie opisany w rozdziale 4. Zakładam, że inne procedury (np. *"and-in"*,*"or-in"*") są znane czytelnikom.

Algorytm 2.1 (przesunięcie kwantyfikatorów wgłąb)

Parametr wejściowy: Φ (formuła)

1. Wykonuj (rekurencyjnie) *"przesunięcie kwantyfikatorów wgłąb"* dla bezpośrednich podformuł formuły Φ.

2. Stosując przekształcenia

$$\begin{aligned}
\forall x\ (\varphi(x) \wedge \psi(x)) &\iff (\forall x\ \varphi(x)) \wedge (\forall x\ \psi(x)) \\
\exists x\ (\varphi(x) \vee \psi(x)) &\iff (\exists x\ \varphi(x)) \vee (\exists x\ \psi(x)) \\
\forall x\ (\varphi(x) \vee \psi) &\iff (\forall x\ \varphi(x)) \vee \psi \\
& \qquad \textit{gdzie } \psi \textit{ nie zawiera } x. \\
\exists x\ (\varphi(x) \wedge \psi) &\iff (\exists x\ \varphi(x)) \wedge \psi \\
& \qquad \textit{gdzie } \psi \textit{ nie zawiera } x.
\end{aligned}$$

rozkładaj Φ na mniejsze części. Jeśli Φ została rozłożona to wykonuj *"przesunięcie kwantyfikatorów wgłąb"* dla części i stop .

3. Jeśli Φ jest postaci $\forall x \forall y\ \varphi(x, y)$ (lub $\exists x \exists y\ \varphi(x, y)$) to zmień Φ na $\forall y \forall x\ \varphi(x, y)$ (lub odp. $\exists y \exists x\ \varphi(x, y)$), wykonuj *"przesunięcie kwantyfikatorów wgłąb"* dla $\forall x\ \varphi(x, y)$ (lub odp. $\exists x\ \varphi(x, y)$) i stop.

4. Jeśli Φ jest postaci $\forall x\ \varphi(x) \vee \psi(x)$ (lub $\exists x\ \varphi(x) \wedge \psi(x)$) to wykonuj *"or-in"*$(\varphi(x) \vee \psi(x))$ (lub odp. *"and-in"*$(\varphi(x) \wedge \psi(x))$). Jeśli przez to formuła Φ się zmieniła na $\forall x\ \varphi_2(x) \wedge \psi_2(x)$ (lub odp. $\exists x\ \varphi_2(x) \vee \psi_2(x)$), to skocz do kroku 2.

Algorytm 2.2 (sklejanie zmiennych ogólnych)

Parametry wejściowe: f (zmienna funkcyjna do eliminacji), Φ (formuła zawierająca f), *opcja* (1 lub 2).

- Dla każdej bezpośredniej podformuły φ formuły Φ wywołaj "sklejanie zmiennych funkcyjnych"$(f, \varphi, opcja)$ (rekurencyjnie).
- Jeśli Φ jest postaci $\varphi \wedge \psi$ to
  - Niech $CK = pusty$ (ciąg kwantyfikatorów)
  - Wykonuj następującą pętlę
    - Wyciągnij wszystkie kwantyfikatory $\exists$, które nie są poprzedzone przez $\forall$ w zakresie $\varphi \wedge \psi$, z $\varphi \wedge \psi$ na zewnątrz i dopisz je na końcu CK.
    - Znajdź kwantyfikatory $\forall x_1$ z φ i $\forall x_2$ z ψ takie, że:
      - nie są poprzedzone przez $\exists$ w zakresach odpowiednio φ i ψ.

· istnieje indeks i taki, że: jeśli $opcja = 1$ to x_1 występuje jako parametr numer i *we wszystkich* wystąpieniach funkcji f w φ i x_2 występuje jako parametr numer i *we wszystkich* wystąpieniach funkcji f w ψ; jeśli $opcja = 2$ to zamiast *"we wszystkich"* używaj *"w niektórych"* dla powyższego zdania.

  * Wyciągnij te kwantyfikatory na zewnątrz, sklej (utożsamiaj) je i dopisz wspólny kwantyfikator na końcu CK
  * Wykonuj tę pętlę dopóki nie można dalej nic wyciągać.

– Dopisz CK na początku Φ

2.3 Algorytm deskolemizacji

Zadanie: Dana jest formuła drugiego rzędu $\exists f_1 \ldots \exists f_k\ \Phi$, gdzie $f_1, \ldots, f_k$ są zmiennymi funkcyjnymi, a Φ to formuła zawierająca $f_1, \ldots, f_k$ (Φ jest formułą pierwszego rzędu na sygnaturze rozszerzonej o nowe funkcje $f_1, \ldots, f_k$). Znajdź formułę pierwszego rzędu Ψ taką, że $(\exists f_1 \ldots \exists f_k\ \Phi) \iff \Psi$.

Algorytm wygląda następująco: Najpierw włożymy $f_1, \ldots, f_k$ do pewnej kolejki, oznaczamy tę kolejkę przez *zmf*. Po kolei bierzemy jedną zmienną (niech to będzie f) z *zmf*, i próbujemy wyeliminować ją z Φ. W trakcie próby eliminacji f z Φ mogą zostać utworzone nowe zmienne funkcyjne (przez "rozdzielenie zmiennej funkcyjnej"), w takim przypadku te nowe zmienne zostaną wstawione do kolejki *zmf*. Eliminacja jednej zmiennej funkcyjnej może dać możliwość eliminacji innych, więc jeśli nie da się od razu eliminować f z Φ to wstawiamy ją na końcu kolejki *zmf*. Pętlę (przechodzenie po kolejce *zmf*) wykonujemy dopóki *zmf* stanie się pusta lub nie da się rozdzielać lub eliminować żadnej zmiennej funkcyjnej z *zmf* lub chcemy ją zatrzymać (przekroczenie limitu czasu). Strategią eliminacji f z Φ jest doprowadzenie Φ do postaci , dla której można od razu eliminować f przez operacje omówione w lematach 2.3,2.4. Można podzielić ten proces na dwie fazy. W pierwszej fazie próbujemy rozdzielić f na instancje w celu zmniejszenia ilości różnych wystąpień jednej instancji w Φ. Wybieramy jedną instancję i oznaczamy ją nadal przez f, inne instancje włożymy do kolejki *zmf*. "Sklejanie zmiennych ogólnych"(z opcją 1) jest włączone do tej fazy, bo to daje więcej możliwości rozdzielenia f. Druga faza przygotowuje Φ do zastosowania lematów 2.3,2.4. Wykonuje "sklejanie zmiennych ogólnych"(z opcją 2), wyciąga potrzebne kwantyfikatory na zewnątrz i w razie potrzeby dokonuje zmiany kolejności kwantyfikatorów.

Bardziej formalny opis algorytmu podaję w czterech procedurach: *"deskolemizuj"*, *"rozdziel"*, *"sklej i rozdziel"*, *"kompromis"*. Pierwsza z nich to główną procedurą, następne dwie tworzą fazę 1, a ostania procedura jest fazą 2. W tych procedurach symbole Φ, Φ', Φ^* są zmiennymi strukturalnymi programu ; jeśli ich podformuły się zmienią to one też się zmienią. W wywołaniach tych procedur parametr Φ jest typu *in-out*. Traktujemy *zmf* jako zmienną globalną.

Procedura 2.3 (deskolemizuj)

Parametry wejściowe: Φ.
Wynik: $(powodzenie, \Phi)$ lub (*porażka,pusta*)

1. *"przesunięcie kwantyfikatorów wgłąb"* (Φ);
2. Wykonuj pętlę
   (a) Jeśli *zmf* jest pusta to zwróć wynik $(powodzenie, \Phi)$;
   (b) Wyciągnij jedną zmienną funkcyjną z *zmf*, niech to będzie f;
   (c) Jeśli f nie występuje w Φ to skocz do kroku 2a;
   (d) Próbuj wyeliminować f z Φ wywołaniem *"rozdziel"*(f, Φ);
   (e) *"przesunięcie kwantyfikatorów wgłąb"* (Φ);
   (f) Jeśli nie można eliminować ani rozdzielić żadnej zmiennej funkcyjnej z *zmf* lub przekroczono limit czasu to przerwij działanie i zwróć wynik (*porażka,pusta*);

Procedura 2.4 (rozdziel)

Parametry wejściowe: f, Φ.

1. Wyciągnij kwantyfikatory egzystencjalne, które nie są poprzedzone (bezpośrednio lub pośrednio) przez kwantyfikatory ogólne, z Φ na zewnątrz i dopisz je do ciągu kwantyfikatorów CK (na początku CK jest pusty).
2. Niech Φ' będzie wynikiem operacji "redukowania zakresu"(f, Φ).
3. Wykonuj *"and-in"*(Φ').

4. Niech Φ' będzie postaci $\Phi_1'(f) \vee \ldots \vee \Phi_k'(f)$. Zmień Φ' na $\Phi_1'(f_1) \vee \ldots \vee \Phi_k'(f_k)$, gdzie $f_1, \ldots, f_k$ są nowymmi zmiennymi funkcyjnymi. Po kolei dla $1 \leq i \leq k$ próbuj wyeliminować f_i z Φ_i' wywołaniem *"sklej i rozdziel"*(f_i, Φ_i'), jeśli to się nie uda to dopisz f_i do kolejki *zmf*.

5. Dopisz kwantyfikatory z CK na początku Φ.

Procedura 2.5 (sklej i rozdziel)

Parametry wejściowe: f, Φ.

1. Niech Φ' będzie wynikem "redukowanie zakresu"(f, Φ).

2. Wykonuj *"sklejanie zmiennych ogólnych"* $(f, \Phi', 1)$;

3. Jeśli Φ' jest postaci $\exists x_1 \ldots \exists x_k \; \forall y_1 \ldots \forall y_h \; \Phi^*$, gdzie $0 \leq k$, $1 \leq h$ i dla każdego j $(1 \leq j \leq h)$ istnieje indeks l_j taki, że y_j występuje jako argument numer l_j we wszystkich wystąpieniach funkcji f w Φ^*, to wywołaj *"rozdziel"*(f, Φ^*) żeby dalej rozdzielić f;

4. W przeciwnym przypadku, wywołaj *"kompromis"*(f, Φ');

Procedura 2.6 (kompromis)

Parametry wejściowe: f, Φ.

1. Wykonuj *"sklejanie zmiennych ogólnych"* $(f, \Phi, 2)$;

2. Wyciągnij wszystkie kwantyfikatory, które poprzedzą przynajmniej jedno wystąpienie f, z Φ na zewnątrz i dopisz je do ciągu kwantyfikatorów CK.

3. Wykonuj zmiany kolejności kwantyfikatorów w CK (jeśli jest to potrzebne i możliwe) w celu sprowadzenia $CK \; \Phi$ do postaci, dla której można zastosować lematy 2.3.

4. Jeśli formuła $\exists f \; CK \; \Phi$ spełnia przesłanki lematu 2.3 to zastosuj ten lemat do niej, przypisz wynik (razem z ciągiem kwantyfikatorów) tego zastosowania do Φ.

5. W przeciwnym przypadku dopisz CK na początku Φ.

Twierdzenie 2.5 (Poprawność algorytmu) *Jeśli na początku* zmf $=$ $(f_1, \ldots, f_k)$ *i wywołanie* "deskolemizuj"*(Ω) daje wynik (powodzenie,Ω') to mamy* $(\exists f_1 \ \ldots \exists f_k \ \Omega) \Longleftrightarrow \Omega'$.

Dowód: Udowodnimy, że *dla dowolnego ustalonego modelu prawdziwość formuły $\exists$ zmf Ω się nie zmieni* w trakcie realizacji algorytmu (tu przez $\exists$ *zmf* rozumiemy jako $\exists g_1 \ \ldots \exists g_h$, gdzie $g_1, \ldots, g_h$ są wszystkimi zmiennymi funkcyjnymi kolejki *zmf*). Tę tezę nazwamy *niezmiennikiem* algorytmu. Wystarczy udowodnić, że krok 4 procedury *"rozdziel"* i krok 4 procedury *"kompromis"* zachowują prawdziwość niezmiennika.

Poprawność kroku 4 procedury "rozdziel" *względem niezmiennika:* Ustalmy moment, w którym realizacja algorytmu dochodzi do kroku 4 procedury *"rozdziel"*. Niech Φ' będzie formułą wymienioną w kroku 4 procedury *"rozdziel"*. Φ' jest podformułą formuły Ω i w formule Ω tylko Φ' zawiera f. Niech SK będzie ścieżką kwantyfikatorów prowadzącą do Φ' w kontekście Ω. SK składa się z kwantyfikatorów wymienionych w krokach 1 procedury *"rozdziel"* i 3 procedury *"sklej i rozdziel"*, więc SK razem z f i Φ' spełnia przesłanki lematu 2.2. Stąd z lematu 2.2 i lematu 2.4 mamy poprawność kroku 4 procedury *"rozdziel"* względem niezmiennika.

Poprawność kroku 4 procedury "kompromis" *względem niezmiennika:* Ustalmy moment, w którym realizacja algorytmu dochodzi do kroku 1 procedury *"kompromis"* . W tym momencie Φ jest podformułą formuły Ω i w formule Ω tylko Φ zawiera f. Niech SK będzie ścieżką kwantyfikatorów prowadzającą do Φ w kontekscie Ω. Łatwo widać, że dla każdej zmiennej ogólnej x_j z SK istnieje indeks l_j taki, że x_j występuje jako argument numer l_j we wszystkich wystąpieniach funkcji f w Φ. Teraz niech realizacja algorytmu dochodzi do kroku 4 (w samym wywołaniu tej procedury). Jeśli $\exists f \ CK \ \Phi$ spełnia przesłanki lematu 2.3 i wynikiem zastosowania lematu 2.3 do tej formuły jest Ψ to $\exists f \ SK, CK \ \Phi$ też spełnia przesłanki lematu 2.3 i wynikiem zastosowania lematu 2.3 do tej formuły będzie $SK \ \Psi$. Stąd z lematu 2.3 i lematu 2.4 wynika poprawność kroku 4 względem niezmiennika. $\square$

Przykład 2.11 W poniższym przykładzie najpierw trzeba eliminować g, potem przesunąć kwantyfkatory wgłąb i eliminować f.

$$
\begin{array}{ll}
 & \exists g \ \exists f \ \forall x \ \varphi(x) \vee (\exists y \ \forall z \ \psi(y, z, f(z), g(x, y))) \\
\Longleftrightarrow & \exists f \ \exists g \ \forall x \ \varphi(x) \vee (\exists y \ \forall z \ \psi(y, z, f(z), g(x, y))) \\
\Longleftrightarrow & \exists f \ \forall x \ \exists y \ \exists t \ \varphi(x) \vee (\forall z \ \psi(y, z, f(z), t))
\end{array}
$$

$$
\begin{aligned}
&\iff (\forall x\ \varphi(x)) \vee (\exists f\ \exists y\ \exists t\ \forall z\ \psi(y,z,f(z),t))\\
&\iff (\forall x\ \varphi(x)) \vee (\exists y\ \exists t\ \forall z\ \exists u\ \psi(y,z,u,t))
\end{aligned}
$$

3 Redukcja do logiki wielorodzajowej

W tym rozdziale zostanie podany inny schemat deskolemizacji. Jest on oparty na wyszukiwaniu podstawień zdefiniowanych w logice wielorodzajowej dla zmiennych funkcyjnych. Ograniczymy przestrzeń wartości zmiennych funkcyjnych do klasy funkcji częściowo definiowalnych wykorzystujących funkcje Skolema. Ta klasa będzie zdefiniowana w logice wielorodzajowej. Podajemy aksjomaty dla tej klasy i wywołujemy system automatycznego dowodzenia twierdzeń do syntezy podstawień dla zmiennych funkcyjnych.

3.1 Logika wielorodzajowa

Logika wielorodzajowa jest rozszerzeniem klasycznej logiki pierwszego rzędu o rodzaje (tzn. typy). Każdy symbol funkcyjny lub relacyjny musi teraz mieć konkretny typ (w tym, typy dla argumentów). W formalizacji podanej w tym paragrafie polimorfizm nie jest dozwolony. Logika wielorodzajowa jest wygodniejsza niż logika klasyczna w opisaniu rzeczywistości, co wynika z występowania typów. Można wykazać, że jeśli chodzi o własciwości teoretyczne – te logiki są równoważne[1].

3.1.1 Syntaktyka

Język logiki wielorodzajowej tworzą następujace rzeczy:

- *symbole stałych, symbole funkcyjne, symbole relacyjne, spójniki logiczne, znaki interpunkcyjne:* jak w klasycznej logice pierwszego rzędu,
- *rodzaje:* $\tau_1, \tau_2, \ldots$,
- *funkcja określająca typy:* TYP
  - Dla symbolu stałej c_i, $TYP(c_i)$ jest pewnym τ_i.
  - Dla symbolu funkcji n-argumentowej f_i^n $(n > 0)$, $TYP(f_i^n)$ ma postać $\tau_{i_1} * \ldots * \tau_{i_n} \to \tau_i$
  - Dla symbolu relacji n-argumentowej P_i^n, $TYP(P_i^n)$ ma postać $\tau_{i_1} * \ldots * \tau_{i_n}$

Mówimy, że symbol s ma typ $TYP(s)$.

[1] w pewnym sensie

Definicja 3.1 *Zbiór termów* jest najmniejszym zbiorem spełniającym nastepujące warunki:

- stałe są termami odpowiednich typów,
- zmienne indywiduowe z typami są termami tych typów,
- jeśli f_i^n jest symbolem funkcji n-argumentowej typu $\tau_{i_1} * \ldots * \tau_{i_n} \to \tau_i$, a $t_1, \ldots, t_n$ są odpowiednio termami typu $\tau_{i_1}, \ldots, \tau_{i_n}$, to $f_i^n(t_1, \ldots, t_n)$ jest termem typu τ_i.

Definicja 3.2 Jeśli P jest symbolem relacji n-argumentowej typu $\tau_{i_1} * \ldots * \tau_{i_n}$, a $t_1, \ldots, t_n$ są termami typu $\tau_{i_1}, \ldots, \tau_{i_n}$ odpowiednio, to $P(t_1, \ldots, t_n)$ jest *formułą atomową (atomem).*

Definicja 3.3 *Zbiór formuł* jest najmniejszym zbiorem spełniającym warunki:

- atomy są formułami,
- jeśli α i β są formułami, to $\neg\alpha$, $\alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \to \beta$, $\alpha \leftrightarrow \beta$ są formułami,
- jeśli α jest formułą, x jest zmienną wolną w α, τ jest rodzajem, to $(\forall x : \tau)\ \alpha$ i $(\exists x : \tau)\ \alpha$ są formułami. W zakresie formuły α zmienna x ma typ τ.

Uwagi: W formułach otwartych, typy zmiennych wolnych nie są jawnie wymienione. Można określić typy zmiennych wolnych na podstawie typów argumentów funkcji i relacji pod warunkiem, że w formule nie występuje konflikt typów.

3.1.2 Semantyka

Definicja 3.4 *Interpretacją formuły* α nazywamy parę $((D_1, D_2, \ldots), m)$, gdzie D_i jest niepustym zbiorem, zaś m taką funkcją przyporządkującą wartości stałym, symbolom funkcyjnym i symbolom relacyjnym w α, że:

- każdemu symbolowi stałej typu τ_i jest przyporządkowany element zbioru D_i,

- każdemu n-argumentowemu symbolowi funkcyjnemu f_i^n typu $\tau_{i_1} * \ldots * \tau_{i_n} \to \tau_i$ przyporządkowane jest odwzorowanie z $D_{i_1} * \ldots * D_{i_n}$ w D_i,
- każdemu n-argumentowemu symbolowi relacyjnemu P_i^n typu $\tau_{i_1} * \ldots * \tau_{i_n}$ przyporządkowane jest odwzorowanie z $D_{i_1} * \ldots * D_{i_n}$ w $\{0,1\}$.

Definicja 3.5 *Wartościowaniem zmiennych* przy interpretacji $I = ((D_1, D_2, \ldots), m)$ nazwamy parę (v, typ), która każdej zmiennej indywiduowej x przyporządkuje typ $\tau_{typ(x)}$ i wartość $v(x)$ z dziedziny $D_{typ(x)}$.

Definicja 3.6 *Wartość formuły α przy interpretacji I i wartościowaniu (v, typ) (przy interpretacji I)* (oznaczamy tę wartość przez $Wart_{I,(v,typ)}(\alpha)$) jest zdefiniowana jak w przypadku dla klasycznej logiki pierwszego rzędu z wyjątkiem:

- $Wart_{I,(v,typ)}((\forall x : \tau)\ \beta) = 1$ wtedy i tylko wtedy, gdy $Wart_{I,(v',typ')}(\beta) = 1$ dla każdego wartościowania (v', typ') przy interpretacji I, która się różni od (v, typ) jedynie wartościami dla zmiennej x i spełnia $\tau_{typ'(x)} = \tau$.
  Dla pozostałych przypadków, $Wart_{I,(v,typ)}((\forall x : \tau)\ \beta) = 0$.
- $Wart_{I,(v,typ)}((\exists x : \tau)\ \beta) = Wart_{I,(v,typ)}(\neg(\forall x : \tau)\ \neg\beta)$

Pojęcia takie jak *"prawdziwość formuły przy interpretacji i wartościowaniu"*, *"prawdziwość formuły przy interpretacji"*, *"spełnialność formuły"*, *"tautologia"* są zdefiniowane w zwykły sposób (zobacz rozdział 1).

3.2 Rezolucja

Rezolucja jest procedurą automatycznego dowodzenia twierdzeń dla logiki pierwszego rzędu. Rezolucja została zdefiniowana przez Robinsona i rozszerzona przez wielu autorów. Pierwotnie rezolucja była zdefiniowana dla klasycznej logiki pierwszego rzędu, okazuje się, że można rozszerzyć ją dla logiki wielorodzajowej. Jeśli chodzi o podstawowe operacje takie jak *rezolucja binarna* lub *faktoryzacja*, to nic nie trzeba zmieniać. Dopiero kiedy adoptujemy inne operacje (np. *paramodulacja*), to trzeba sprawdzić zgodność typów. *Unifikacja* termów dla logiki wielorodzajowej jest zdefiniowana w taki sam sposób jak dla klasycznej logiki pierwszego rzędu (bo w formalizacji podanej w tym rozdziale nie dopuszczamy polimorfizmu). Dla prostoty, w tym podrozdziale podaję definicję rezolucji tylko dla klasycznej logiki pierwszego rzędu. Ten podrozdział jest napisany na podstawie pracy [Wój91].

3.2.1 Unifikacja

Unifikacja (ang. unification) jest to proces doprowadzenia danych wyrażeń do tej samej postaci (ujednolicenia) przez zastosowanie odpowiednich podstawień do występujących w tych wyrażeniach zmiennych.

Definicja 3.7 *Podstawieniem* (ang.substitution) nazywamy skończony zbiór *przypisań* (ang. association) wyrażeń zmiennym, przy czym:

- każdej zmiennej przypisane jest co najwyżej jedno wyrażenie,
- żadna ze zmiennych, posiadających przypisane jakieś wyrażenie, nie wystepuje w żadnym z przypisanych zmiennym wyrażeń.

Podstawienie, w którym wszystkie wyrażenia są termami podstawowymi, nazywamy podstawieniem podstawowym (ang. ground substitution).

Podstawienie z pustym zbiorem przypisań nazywamy *podstawieniem pustym.*

Przykład 3.1 Instancją wyrażenia $W = P(x, x, y, v)$ przy podstawieniu $\sigma = \{x \leftarrow a, y \leftarrow f(b), z \leftarrow w\}$ jest wyrażenie

$$W\sigma = P(x, x, y, v)\{x \leftarrow a, y \leftarrow f(b), z \leftarrow w\} = P(a, a, f(b), v)$$

Definicja 3.8 *Złożeniem* podstawienia σ i podstawienia τ (oznaczanym przez $\sigma\tau$) nazywamy podstawienie uzyskane przez zastosowanie τ do wyrażeń z σ, a następnie dodanie do σ tych wiązań z τ, które nie dotyczą zmiennych posiadających wiązania w σ.

Przykład 3.2 Wynikiem złożenia podstawienia $\sigma = \{z \leftarrow g(x, y)\}$ oraz $\tau = \{x \leftarrow a, y \leftarrow b, w \leftarrow c, z \leftarrow d\}$ jest podstawienie

$$\begin{aligned} \sigma\tau &= \{z \leftarrow g(x, y)\}\{x \leftarrow a, y \leftarrow b, w \leftarrow c, z \leftarrow d\} \\ &= \{z \leftarrow g(a, b), x \leftarrow a, y \leftarrow b, w \leftarrow c\} \end{aligned}$$

Definicja 3.9 Zbiór wyrażeń $\{W_1, \ldots, W_n\}$ jest *unifikowalny* (ang. unifiable), jeśli istnieje takie podstawienie σ, że wyrażenia $W_1\sigma, \ldots, W_n\sigma$ są identyczne, czyli $W_1\sigma = \ldots = W_n\sigma$. Podstawienie σ nazywane jest *unifikatorem* (ang. unifier) tego zbioru.

Definicja 3.10 Podstawienie σ jest *tak samo lub bardziej ogólne* niż podstawienie τ, jeśli istnieje takie podstawienie δ, że $\sigma\delta = \tau$.

Definicja 3.11 Podstawienie τ nazywamy *najbardziej ogólnym unifikatorem* (ang. most general unifier, w skrócie MGU) wyrażeń α i β, jeśli τ jest unifikatorem zbioru $\{\alpha, \beta\}$ ($\alpha\tau = \beta\tau$) oraz dla każdego podstawienia σ, które jest unifikatorem zbioru $\{\alpha, \beta\}$, istnieje takie podstawienie δ, że $\tau\delta = \sigma$, czyli $\alpha\tau\delta = \alpha\sigma = \beta\sigma = \beta\tau\delta$.

Istnieje algorytm, który znajduje najbardziej ogólny unifikator skończonego, unifikowalnego zbioru niepustych wyrażeń. Jeśli ten zbiór nie jest unifikowalny, algorytm również stwierdza ten fakt. Opis tego algorytmu można znaleźć w pracy [Wój91].

3.2.2 Zasada rezolucji

Definicja 3.12 Jeśli dla danych dwóch klauzul C_1 i C_2 istnieje podstawienie σ i literały L_1 w C_1 oraz L_2 w C_2 takie, że literały $L_1\sigma$ i $L_2\sigma$ tworzą parę komplementarną, to klauzulę $(C_1 - L_1)\sigma \bigcup (C_2 - L_2)\sigma$ nazywamy *rezolwentą binarną* (ang. binary resolvent) klauzul C_1 i C_2.

Regułę tę można zapisać następująco:

$$\frac{L_1 \vee D_1 \ \ , \ \ L_2 \vee D_2}{(D_1 \vee D_2)\sigma} \quad gdzie\ \sigma = MGU(\{L_1, L_2\}).$$

Przykład 3.3 Wyprowadzenie rezolwenty binarnej:

$$\frac{P(x) \vee Q(x, y) \ \ , \ \ \neg P(a) \vee R(b, z)}{Q(a, y) \vee R(b, z)} \quad \sigma = \{x \leftarrow a\}$$

Definicja 3.13 Jeśli σ jest najbardziej ogólnym unifikatorem pewnego podzbioru literałów klazuli C, to klauzulę C', uzyskaną z C przez zastosowanie do niej σ i usunięcie (poza jednym) powtarzających się literałów, nazywamy *faktorem* (ang. factor) klauzuli C.

Przykład 3.4 Faktorem klauzuli $C = P(x) \vee P(f(y)) \vee R(x, y)$ przy podstawieniu $\sigma = MGU(\{P(x), P(f(y))\}) = x \leftarrow f(y)$ jest klauzula $C' = P(f(y)) \vee R(f(y), y)$.

Definicja 3.14 *Wywodem rezolucyjnym* (ang. resolution deduction) klauzuli C ze zbioru klauzul S nazywamy ciąg klauzul $C_1, \ldots, C_n$, którego elemetami są wyłącznie elementy zbioru S oraz rezolwenty i faktory klauzul wystepujących wcześniej w tym ciagu, zaś $C_n = C$.

Definicja 3.15 Klauzula C jest $wyprowadzalna$ ze zbioru klauzul S, jeśli istnieje wywód C z S.

Definicja 3.16 Traktując klauzulę jako zbiór literałów, *klauzulą pustą* nazywamy pusty zbiór literałów.

Twierdzenie 3.1 (poprawność zasady rezolucji) Jeśli istnieje wywód rezolucyjny klauzuli C ze zbioru klauzul S, to S implikuje logicznie C.

Twierdzenie 3.2 (pełność zasady rezolucji) Jeśli zbiór klauzul S jest niespełnialny, to istnieje rezolucyjny wywód klauzuli pustej z S.

Dowody tych twierdzeń można znaleźć w pracach [GN87],[Wój91].

3.3 Redukcja do logiki wielorodzajowej

Definicja 3.17 Niech dany będzie język klasycznej logiki pierwszego rzędu L. Konstruujemy język wielorodzajowy $L^{'}$ przez dodanie do języka L:

- *nowych rodzajów: Indeks, Stan, Term, Formuła, Instrukcja.* (rodzaj języka L nazywamy $Pierwotny$),
- *stałych:* $0, 1, 2, \ldots : Stan$
- (dla każdego symbolu funkcji n-argumetowej f języka L), *nowej funkcji* n-argumentowej: $\_f \;:\; Term * \ldots * Term \rightarrow Term$ (te nowe symbole nazywamy *metafunkcjami* (lub *metastałymi* jeśli $n = 0$)),
- (dla każdego symbolu relacji n-argumetowej r języka L), *nowej relacji* n-argumentowej: $\_r \;:\; Term * \ldots * Term \rightarrow bool$ (te nowe symbole nazywamy *metarelacjami*),
- *nowych symboli:* (te symbole nazywamy *symbolami sematyki*).

X	:	$Indeks \rightarrow Term$
		(X konstruuje metazmienne,
		$X(i)$ jest zmienną pamięciową o numerze i),
$\_or$	:	*Formuła* $*$ *Formuła* $\rightarrow$ *Formuła*,
$\_not$	:	*Formuła* $\rightarrow$ *Formuła*,
$\_forall$	:	$Indeks$ $*$ *Formuła* $\rightarrow$ *Formuła*,
$null$	:	$Stan$
		(pusta pamięć lub pusta lista parametrów),
zastąp	:	$Stan * Indeks * Pierwotny \rightarrow Stan$
		(*zastąp(s,i,a)* – pamięć otrzymana z s przez
		zastąpienie zmiennej i wartością a),
$przypisz$	:	$Indeks * Term \rightarrow Instrukcja$
		(instrukcja przypisania),
złóż	:	$Instrukcja * Instrukcja \rightarrow Instrukcja$
		(instrukcja złożenia),
$\_if$	:	*Formuła* $*$ *Instrukcja* $\rightarrow$ *Instrukcja*,
$\_while$	:	*Formuła* $*$ *Instrukcja* $\rightarrow$ *Instrukcja*,
$elem$	:	$Stan * Indeks \rightarrow Pierwotny$
		($elem(s,i)$ – wartość zmiennej o numerze i w s),
W	:	$Term * Stan \rightarrow Pierwotny$
		(wartościowanie termów),
B	:	*Formuła* $*Stan \rightarrow$ *bool*
		(wartościowanie formuł),
P	:	$Instrukcja * Stan \rightarrow Stan$,
		($P(f,s)$ – stan pamięciowy po wykonywaniu
		instrukcji f dla stanu pamięciowego s)

Język L' nazywamy *rozszerzeniem o funkcje częściowo definiowalne* języka L i oznaczamy L' przez $RFCD(L)$.

Definicja 3.18 Niech dany będzie język klasycznej logiki pierwszego rzędu L. Definiujemy $AFCD(L)$ jako zbiór następujących aksjomatów (nad językiem $RFCD(L)$) :

1. $\forall\ s : Stan, i : Indeks, a : Pierwotny$ *elem(zastąp(s,i,a),i)*$= a$

2. $\forall\ s : Stan, i : Indeks, j : Indeks, a : Pierwotny$
   $i \neq j \rightarrow$ *elem(zastąp(s,i,a),j)*$= elem(s,j)$

3. $W(\_c) = c$
   (dla każdego symbolu stałej c języka L)

4. $\forall\ s : Stan, i : Indeks\ W(X(i), s) = elem(s, i)$

5. $\forall\ s : Stan, t_1 : Term, \ldots, t_n : Term$
   $W(\_f(t_1, \ldots, t_n), s) = f(W(t_1, s), \ldots, W(t_n, s))$
   (dla każdego symbolu funkcji n-argumentowej f języka L)

6. $\forall\ s : Stan, t_1 : Term, \ldots, t_n : Term$
   $B(\_r(t_1, \ldots, t_n), s) \leftrightarrow r(W(t_1, s), \ldots, W(t_n, s))$
   (dla każdego symbolu relacji n-argumentowej r języka L)

7. $\forall\ s : Stan, b_1$:*Formuła*,b_2 :*Formuła*
   $B(\_or(b_1, b_2), s) \leftrightarrow B(b_1, s) \vee B(b_2, s)$

8. $\forall\ s : Stan, b$:*Formuła*
   $B(\_not(b), s) \leftrightarrow \neg B(b, s)$

9. $\forall\ s : Stan, i : Indeks, b$:*Formuła*
   $B(\_forall(i, b), s) \leftrightarrow (\forall\ x : Pierwotny\ \ B(b,zastąp(s,i,x)))$

10. $\forall\ i : Indeks, t : Term, s : Stan$
    $P(przypisz(i, t), s) = zastąp(s,i,W(t,s))$

11. $\forall\ f : Instrukcja, g : Instrukcja, s : Stan$
    $P(złóż(f,g),s)=P(g,P(f,s))$

12. $\forall\ b$:*Formuła*,f : $Instrukcja, s : Stan$
    $(B(b, s) \rightarrow P(\_if(b, f, g), s) = P(f, s)) \wedge$
    $(\neg B(b, s) \rightarrow P(\_if(b, f, g), s) = P(g, s))$

13. $\forall\ b$:*Formuła*,f : $Instrukcja, s : Stan$
    $(\neg B(b, s) \rightarrow P(\_while(b, f), s) = s) \wedge$
    $(B(b, s) \rightarrow P(\_while(b, f), s) = P(złóż(f,\_\ while(b,f)),s))$

Definicja 3.19 Niech dana będzie formuła $\Phi = \exists f_1\ \ldots \exists f_k\ \ \Psi$, gdzie $f_1, \ldots, f_k$ są zmiennymi funkcyjnymi, Ψ jest zwykłą formułą. *Redukcją do logiki wielorodzajowej* formuły Φ nazywamy formułę:

$\exists\ \_f_1 : Instrukcja, \ldots, \_f_k : Instrukcja\ \Psi^*$

gdzie Ψ^* jest konstruowany z Ψ przez zastąpienie wszystkich wystąpień

$f_i(t_1, \ldots, t_{n_i})$ $(1 \leq i \leq k)$ w Ψ przez

$elem(P(\_f_i, zastąp(\ldots zastąp(null, 1, t_1) \ldots, n_i, t_{n_i})), 0)$.

Taką formułę oznaczamy przez Φ^r. (Jeśli Φ jest formułą nad językiem L, to Φ^r jest formułą nad językiem wielorodzajowym $RFCD(L)$.)

Twierdzenie 3.3 (poprawność redukcji) Niech dane będą formuły Φ_1, Φ_2 nad językiem L, gdzie $\Phi_2 = \exists f_1 \ldots \exists f_k\ \Psi$; $f_1, \ldots, f_k$ będą zmiennymi funkcyjnymi, Φ_1, Ψ zwykłymi formułami. Niech Φ_1^s oraz nowe funkcje Skolema $g_1, \ldots, g_h$ będą uzyskane w wyniku skolemizacji formuły Φ_1. Niech $L_2 = L \bigcup \{g_1, \ldots, g_h\}$. Jeśli formuła $AFCD(L_2) \wedge \Phi_1^s \wedge \neg\Phi_2^r$ (nad językiem L_2) jest niespełnialna, to formuła $\Phi_1 \wedge \neg\Phi_2$ też jest niespełnialna (czyli $\Phi_1 \to \Phi_2$ jest tautologią).

Dowód:
Niech M będzie dowolnym ustalonym modelem nad językiem L, niech $g_1, \ldots, g_h$ będą dowolnymi funkcjami w modelu M. Rozszerzymy M do modelu M_2 nad językiem L_2 (aby M_2 spełniał $AFCD(L_2)$) w następujący sposób:

- Dziedziną $Indeks$ będzie zbiór liczb naturalnych.

- Dziedzinę $Stan$ określamy jako zbiór skończonych zbiorów elementów $(i : Indeks, a : Pierwotny)$ taki, że jeśli $s \in Stan$, $(i, a_1) \in s$, $(i, a_2) \in s$ to $a_1 = a_2$. $null$ będzie pustym zbiorem. Dla każdych $s : Stan, i : Indeks, a : Pierwotny$ jeśli istnieje $a_0 : Pierwotny$ taki, że $(i, a_0) \in s$ to niech $zastąp(s,i,a) = (s - \{(i, a_0)\}) \bigcup \{(i, a)\}$, wpp niech $zastąp(s,i,a) = null$.

- Dziedziny *Term, Formuła, Instrukcja* zostaną zbudowane jako odpowiednie algebry (wielorodzajowe) termów. W związku z tym wszystkie metastałe, metafunkcje, metarelacje i symbole sematyki $X, \_or, \_not, \_forall, przypisz, \_złóż, \_if, \_while$ są określone.

- Funkcja $elem$ będzie określona według reguły: dla każdego $i : Indeks$ niech $elem(null, i)$ będzie pewnym wyróżnionym elementem dziedziny $Pierwotny$; dla każdego $s : Stan$ $(s \neq null)$, $i : Indeks$, jeśli istnieje element $a : Pierwotny$ taki, że $(i, a) \in s$, to niech $elem(s, i) = a$, wpp niech $elem(s, i) = elem(null, i)$. W ten sposób aksjomaty 1,2 są spełnione w M_2.

- Fukcje W, B będą określone według aksjomatów $3 \to 9$.

- Dla każdego elementu f : $Instrukcja$, jeśli nie istnieje b :*Formuła*, g : $Instrukcja$ takie, że f = $\_while(b, g)$ to niech $P(f, \_)$ będzie określony według aksjomatów 10,11,12. Dla każdych b :*Formuła*, g : $Instrukcja$, s : $Stan$, jeśli program "$while(b, g)''$ zatrzymuje się dla danych s to niech $P(\_while(b, g), s)$ będzie określony według aksjomatu 13, wpp niech $P(\_while(b, g), s) = null$. W ten sposób aksjomat 13 jest spełniony w M_2.

Z taką konstrukcją M_2 spełnia $AFCD(L_2)$. Z założenia , $AFCD(L_2) \wedge \Phi_1^s \wedge \neg\Phi_2^r$ jest fałszywa w M_2, wynika, że $\Phi_1^s \wedge \neg\Phi_2^r$ jest fałszywa w M_2. Stąd istnieją $\_f_1$: $Instrukcja, \ldots, \_f_k$: $Instrukcja$ takie, że $\Phi_1^s \wedge \neg\Psi^*$ jest fałszywa w M_2. Niech $f_1, \ldots, f_k$ będą określone następująco:
$\forall x_1, \ldots, x_{n_i}\ f_i(x_1, \ldots, x_{n_i}) =$
$elem(P(\_f_i, zastąp(\ldots zastąp(null, 1, x_1) \ldots, n_i, x_{n_i})), 0)$.
W ten sposób $\Phi_1^s \wedge \neg\Psi$ jest równoważna z $\Phi_1^s \wedge \neg\Psi^*$ i jest fałszywa w M_2. Stąd wynika, że $\Phi_1 \wedge \neg\Phi_2$ jest fałszywa w M □

3.4 Podejście do deskolemizacji

Zadanie: Niech dana będzie formuła drugiego rzędu $\exists f_1 \ldots \exists f_k\ \Phi(f_1, \ldots, f_k)$, gdzie $f_1, \ldots, f_k$ są zmiennymi funkcyjnymi, a Φ to formuła I-rzędu zawierająca $f_1, \ldots, f_k$ Trzeba znaleźć formułę pierwszego rzędu Ψ taką, że $(\exists f_1 \ldots \exists f_k\ \Phi(f_1, \ldots, f_k)) \leftrightarrow \Psi$.

Schemat rozwiązywania: Jeśli znajdziemy formułę pierwszego rzędu $Baza$ taką, że

$$\begin{aligned}
(\exists f_1 \ldots \exists f_k\ \Phi(f_1, \ldots, f_k)) &\iff (\exists g_1 \ldots \exists g_h\ \Phi_2(g_1, \ldots, g_h)),\\
(\exists g_1 \ldots \exists g_h\ \Phi_2(g_1, \ldots, g_h)) &\implies Baza,\\
Baza &\implies (\exists g_1 \ldots \exists g_h\ \Phi_2(g_1, \ldots, g_h))
\end{aligned}$$

to $Baza$ jest rozwiązaniem naszego zadania.

Kroki wykonywania:

1. *Uproszczenie:* Przekształcimy formułę $\exists f_1 \ldots \exists f_k\ \Phi(f_1, \ldots, f_k)$ do prostszej (w pewnym sensie) postaci $\exists g_1 \ldots \exists g_h\ \Phi_2(g_1, \ldots, g_h)$. Można w tym celu zastosować algorytm *"deskolemizacja oparta na postaci syntaktycznej"*. Można zmodyfikować algorytm *"deskolemizacja oparta na*

postaci syntaktycznej", aby w przypadku porażki dawał prostszą formułę równoważną formule podanej na wejściu. W przypadku, gdy wynik uproszczenia jest formułą pierwszego rzędu to zatrzymujemy się tutaj.

2. Na początku niech $Baza$ będzie pustym zbiorem formuł (lub zbiorem aksjomatów teorii, na której pracujemy). Wyróżnimy *część pierwszego rzędu* formuły $\exists g_1 \ \ldots \exists g_h \ \Phi_2(g_1, \ldots, g_h)$ i włożymy tę część do $Baza$.

3. Uruchamiamy dwa procesy $Proces1, Proces2$, które działają współbieżnie i wykonują następujące akcje:

   - $Proces1$ generuje wnioski pierwszego rzędu formuły $\exists g_1 \ \ldots \exists g_h \ \Phi_2(g_1, \ldots, g_h)$ i wkłada te wnioski do $Baza$. Tu można zastosować różne rodzaje osłabienia. Na przykład można rozdzielić zmienną funkcyjną nie zachowując równoważności (np. $(\exists f \ \varphi(f) \wedge \psi(f)) \rightarrow (\exists f_1 \ \varphi(f_1)) \wedge (\exists f_2 \ \psi(f_2)))$ i wywołać *"deskolemizacja oparta na postaci syntaktycznej"* dla nowej formuły. W ten sposób $Proces1$ wzbogaci $Baza$.
   - $Proces2$ próbuje udowodnić $Baza \rightarrow (\exists g_1 \ \ldots \exists g_h \ \Phi_2(g_1, \ldots, g_h))$. W tym celu można redukować formułę do logiki wielorodzajowej i używając rezolucję próbować udowodnić prawdziwość nowej formuły. Można w tym celu zastosować systemy dowodzenia twierdzeń dla logiki wielorodzajowej (np. Otter).

Przykład 3.5 Niech dana będzie formuła do deskolemizacji:

$$\begin{aligned}
&\exists f \ \forall x \ [\ (\varphi(x) \vee \forall y \ \exists z \ \psi(x, y, z, f(x, y, z))) \ \wedge \\
&\qquad\qquad (\neg\varphi(x) \vee \exists u \ \forall v \ \chi(x, u, v, f(x, u, v)))]
\end{aligned}$$

Przez osłabienie i zastosowanie algorytmu "deskolemizacji opartej na postaci syntaktycznej" dla osłabionej formuły mamy:

$$\begin{aligned}
&\exists f \ \forall x \ [\ (\varphi(x) \vee \forall y \ \exists z \ \psi(x, y, z, f(x, y, z))) \ \wedge \\
&\qquad\qquad (\neg\varphi(x) \vee \exists u \ \forall v \ \chi(x, u, v, f(x, u, v)))] \\
\Longrightarrow \ &\exists f_1 \ \exists f_2 \ \forall x \ [\ (\varphi(x) \vee \forall y \ \exists z \ \psi(x, y, z, f_1(x, y, z))) \ \wedge \\
&\qquad\qquad\qquad (\neg\varphi(x) \vee \exists u \ \forall v \ \chi(x, u, v, f_2(x, u, v)))] \\
\Longrightarrow \ &\forall x \ [\ (\varphi(x) \vee \forall y \ \exists z \ \exists p \ \psi(x, y, z, p)) \ \wedge \\
&\qquad (\neg\varphi(x) \vee \exists u \ \forall v \ \exists q \ \chi(x, u, v, q))]
\end{aligned}$$

Okazuje się, że ostatnia formuła jest równoważna pierwszej formule. W istotcie, jeśli skolemizujemy ostatnią formułę dla kwantyfikatorów $\exists p$ i $\exists q$, to dostajemy

$$\begin{aligned}\exists f_p\ \exists f_q\ \forall x\ [\ &(\varphi(x) \lor \forall y\ \exists z\ \psi(x,y,z,f_p(x,y,z))) \land \\ &(\neg\varphi(x) \lor \exists u\ \forall v\ \chi(x,u,v,f_q(x,u,v)))]\end{aligned}$$

Dla dowolnego ustalonego modelu, dla dowolnych ustalonych f_p, f_q niech f będzie funkcją określoną następująco:

$$f(x,y,z) = \begin{cases} f_p(x,y,x) & jeli\ \varphi(x)\ jest\ faszywa \\ f_q(x,y,z) & wpp \end{cases}$$

Widać, że z taką definicją mamy:

$$\begin{aligned} & \forall x\ [\ (\varphi(x) \lor \forall y\ \exists z\ \psi(x,y,z,f_p(x,y,z))) \land \\ & \qquad (\neg\varphi(x) \lor \exists u\ \forall v\ \chi(x,u,v,f_q(x,u,v)))] \\ \Longrightarrow\ & \forall x\ [\ (\varphi(x) \lor \forall y\ \exists z\ \psi(x,y,z,f(x,y,z))) \land \\ & \qquad (\neg\varphi(x) \lor \exists u\ \forall v\ \chi(x,u,v,f(x,u,v)))]\end{aligned}$$

Stąd mamy:

$$\begin{aligned} & \exists f_p\ \exists f_q\ \forall x\ [\ (\varphi(x) \lor \forall y\ \exists z\ \psi(x,y,z,f_p(x,y,z))) \land \\ & \qquad (\neg\varphi(x) \lor \exists u\ \forall v\ \chi(x,u,v,f_q(x,u,v)))] \\ \Longrightarrow\ & \exists f\ \forall x\ [\ (\varphi(x) \lor \forall y\ \exists z\ \psi(x,y,z,f(x,y,z))) \land \\ & \qquad (\neg\varphi(x) \lor \exists u\ \forall v\ \chi(x,u,v,f(x,u,v)))]\end{aligned}$$

Właściwość schematu: Sam schemat tworzy algorytm deskolemizacji, ale jeśli realizacja algorytmu nie zatrzymuje się w kroku "uproszczenie", to proces szukania rozwiązania jest bardzo złożony. Główną propozycją algorytmu jest redukcja zagadnienia do logiki wielorodzajowej i zastosowanie istniejących systemów dowodzenia twierdzeń. Klasa formuł wejściowych, dla których ten algorytm może się powieść, jest dość szeroka, bo w szukanych konstrukcjach zmiennych funkcyjnych mogą wystąpić funkcje Skolema (to znaczy konstrukcje zmiennych funkcyjnych są zbudowane w sposób wykorzystujący (w pewnym sensie) te same zmienne funkcyjne) i konstrukcje zmiennych funkcyjnych są zależne od modelu (to znaczy, że wystarczy udowodnić, że te konstrukcje istnieją, a nie trzeba wyznaczyć konkretnych). Klasa funkcji, do której ograniczymy kwantyfikatory funkcyjne, jest większa niż klasa funkcji definiowalnych w normalnym sensie. Uważam, że ten schemat jest ramą, na której można skonstruować inny, lepszy algorytm deskolemizacji. Trzeba dołożyć więcej strategii i heurystyk do tego schematu, aby dostać algorytm, który byłby w rzeczywistości lepszy od algorytmu *"deskolemizacji opartej na postaci syntaktycznej"*.

3.5 Synteza programów

W proceduralnym podejściu do programowania, często mówimy o zmiennych pamięciowych, o stanie realizacji programu . Programowanie funkcyjne odbiega od tej reguły i koncentruje się na sematyce funkcji. W programowaniu czysto funkcyjnym, nie ma operacji *przypisania* i pętli *while*. Programy są traktowane jako funkcje. Pętle *while* są zastąpione przez bezpośrednie wywołania rekurencyjne. Funkcje są zdefiniowane głównie przez polecenie *if* i wywołania rekurencyjne. Takie definicje można uważać za formuły logiczne mimo, że są programami.

To podejście do programowania pozwala łatwiej sprawdzić poprawność programów względem specyfikacji, ponieważ ścisła $specyfikacja$ często jest napisana jako formuła logiki i jeśli $program$ też jest formułą to sprawdzenie poprawności programu jest redukowane do sprawdzenia prawdziwość formuły $program \to specyfikacja$.

Można nieformalnie formułować zagadnienie *syntezy programów* następująco: Niech dana będzie teoria T i formuła $S(f)$ jako specyfikacja funkcji f. Trzeba znaleźć formułę $D(f)$ taką, że $D(f)$ należy do *klasy formuł definiujących programy* i formuła $T \to (D(f) \to S(f))$ jest prawdziwa. Zagadnienie syntezy programów jest związane z eliminacją kwantyfikatorów funkcyjnych następująco: trzeba udowodnić prawdziwość formuły $T \to \exists f\ S(f)$. Różnica jest tylko w tym, że musimy wyznaczyć konkretną funkcję f przez podanie poprawnej definicji $D(f)$.

Definicja 3.20 Niech L będzie językiem klasycznej logiki pierwszego rzędu. Niech f będzie symbolem funkcji n-argumentowej nie należącym do L. Niech $i_1, \ldots, i_n$ będą symbolami stałych nie należącymi do L (te symbole są zwane parametrami) i niech $L_2 = L \bigcup \{f, i_1, \ldots, i_n\}$. Zbiorem *programów* definujących f nad L nazywamy najmniejszy zbiór spełniający następujące warunki:

- Każdy term nie zawierający zmiennych nad L_2 jest programem.

- Jeśli b jest zamkniętą formułą bez kwantyfikatorów nad L_2 i t_1, t_2 są programami to $\_if(b, t_1, t_2)$ jest programem.

Definicja 3.21 Niech P będzie programem definiującym funkcję n-argumentową f nad językiem L. Niech $less$ będzie symbolem relacji (2n)-argumentowej należącym do L i niech T będzie teorią nad L. Mówimy, że P ma *własność stopu* względem relacji $less$ w teorii T jeśli dla każdego wystąpienia $f(t_1, \ldots, t_n)$ w P prawdziwa jest formuła

$$T \to \forall i_1 \ \ldots \forall i_n \ less(t_1, \ldots, t_n, i_1, \ldots, i_n)$$

Definicja 3.22 Niech P będzie programem definiującym funkcję n-argumentową f. *Postacią formułową* programu P nazywamy formułę $\forall x_1, \ldots, \forall x_n$ *do_formuły(P)*, gdzie

- *do_formuły*$(\_if(b, t_1, t_2)) \equiv$
  $(b[i_1 \leftarrow x_1, \ldots, i_n \leftarrow x_n] \to$ *do_formuły*$(t_1)) \wedge$
  $(\neg b[i_1 \leftarrow x_1, \ldots, i_n \leftarrow x_n] \to$*do_formuły*$(t_2))$

- jeśli t nie zawiera $\_if$, to
  do_formuły(t) $\equiv \ f(x_1, \ldots, x_n) = t[i_1 \leftarrow x_1, \ldots, i_n \leftarrow x_n]$

Oznaczmy postać formułową programu P przez $F(P)$.

Definicja 3.23 Niech dany będzie język (logiki klasycznej pierwszego rzędu) L, teoria T nad L, symbol funkcji n-argumentowej f nie należącej do L, formuła $S(f)$ (traktowana jako specyfikacja funkcji f) nad językiem $L \bigcup \{f\}$. Mówimy, że program P definiujący f *spełnia specyfikację* $S(f)$ jeśli prawdziwa jest formuła $T \to (F(P) \to S(f))$.

W dalszych częściach tego podrozdziału zakładam, że mamy dany język klasycznej logiki pierwszego rzędu L, teorię T nad L, symbol funkcji n-argumentowej f nie należący do L, formułę $S(f)$ (jako specyfikację funkcji f), symbol relacji (2n)-argumentowej $less$ (która definiuje własność stopu) i zadanie syntezy funkcji f. Program definiujący funkcję f musi spełniać specyfikację $S(f)$ i mieć własność stopu względem $less$ w teorii T.

3.5.1 Sugerowanie postaci syntezowanego programu

Pierwszym krokiem syntezy funkcji f jest próba przeprowadzenia syntaktycznymi przekształceniami $S(f)$ do postaci formułowej pewnego programu.

Wynik tego kroku jest tylko sugesją (to znaczy, przekształcenia nie muszą zachowywać równoważności formuły; sugerowany program będzie weryfikowany względem specyfikacji i własności stopu w następnym kroku). Ten krok może zatrzymywać się z sukcesem lub z porażką. W przypadku sukcesu, mówimy, że specyfikacja $S(f)$ jest *pseudoprogramowa*. W przypadku porażki ten krok powienien sugerować, jak wygląda program syntezowany (np. $\_if(i2 = 0, i1, x)$). Symbole w sugesji dzielą się na dwie grupy: niezmienne i zmienne (lub *wiadomome* i *niewiadome*). Im więcej wiadomomych, tym lepiej (pod względem, że te wiadomome mają uzasadnienie).

Algorytm 3.1 (sugerowanie postaci syntezowanego programu)

1. Sprowadź $S(f)$ do postaci NNF.

2. Używając "przesunięcie kwantyfikatorów wgłąb" i "or_in" przekształcaj $S(f)$ w koniunkcję pewnych formuł; niech tymi formułami będą $czynnik_1, \ldots, czynnik_k$.

3. Usuń *czynniki*, które nie zawierają f. Dalej można założyć, że wszystkie $czynnik_1, \ldots, czynnik_k$ zawierają f.

4. for $i = 1$ to k do

   (a) Usuń kwantyfikatory ogólne na początku $czynnik_i$;

   (b) Przekształć $czynnik_i$ w postaci $\neg warunek_i \vee wariant_i$, gdzie $warunek_i$ nie zawiera f ($warunek_i$ może być pusty);

   (c) Sprowadź $warunek_i$, $wariant_i$ do postaci NNF.

   (d) Jeśli $wariant_i$ nie jest równością, to usuń $czynnik_i$, ustaw flagę $need\_extra\_variant$ (na początku ta flaga jest wyłączona), zwiększ i o 1 i skocz do początku pętli (do kroku 4a);

   (e) Jeśli jest możliwe, ustaw kierunek równości w $wariant_i$ w taki sposób, aby $wariant_i$ miał własność stopu względem $less$ w teorii T. (używaj Ottera w tym celu)

   (f) Jeśli lewa strona równości $wariant_i$ nie jest postaci $f(\ldots)$ lub jest postaci $f(\ldots f \ldots)$, to usuń $czynnik_i$, ustaw flagę $need\_extra\_variant$, zwiększ i o 1 i skocz do początku pętli (do kroku 4a)

(g) Niech $f(t_1, \ldots, t_n)$ będzie lewą stroną równości $wariant_i$. Po kolei dla $1 \leq j \leq n$, jeśli t_j jest zmienną, to zastąp wszystkie wystąpienia tej zmiennej w $czynnik_i$ przez i_j (i jest literą, a j – numerem), gdzie i_j jest symbolem stałej (zakładamy, że L nie zawiera i_j).

(h) Zmodyfikuj $warunek_i$ w następujący sposób: zastąp wszystkie podformuły formuły $warunek_i$, które się zaczynają kwantyfikatorem ogólnym, przez nowe zmienne boolowskie (typu *Formuła* w logice wielorodzajowej); usuń wszystkie kwantyfikatory egzystencjalne występujące w $warunek_i$.

(i) Niech $f(t_1, \ldots, t_n)$ będzie lewą stroną równości $wariant_i$. Po kolei dla $1 \leq j \leq n$, jeśli t_j nie zawiera zmiennych i t_j nie jest i_j, to dopisz $i_j = t_j$ do $warunek_i$ (tzn. $warunek_i := warunek_i \wedge (i_j = t_j)$), w przeciwnym przypadku dopisz nową zmienną boolowską do $warunek_i$ (jeśli tej operacji nie wykonano wcześniej).

5. Uporządkuj (przenumeruj) nieusunięte $czynnik_i$ $(1 \leq i \leq k)$ według pewnego kryterium. Niech wynikiem tego kroku będzie $czynnik_1, \ldots, czynnik_h$.

6. Jeśli istnieje i $(1 \leq i \leq h)$ taki, że $warunek_i$ jest pusty to zwróć prawą stronę równości $wariant_i$ jako wynik.

7. Jeśli flaga $need\_extra\_variant$ jest ustawiona, to niech $rs = xt$, gdzie xt jest nową zmienną typu $Term$; w przeciwnym razie przypisz do rs prawą stronę równości $wariant_h$ i zmniejsz h o 1.

8. for $i = h$ downto 1 do $temp :=$ prawa strona równości $wariant_i$
   $rs := \_if(warunek_i, temp, rs)$

9. Zwróć rs jako wynik.

Przykład 3.6 Dla specyfikacji

$$(\forall x\ sum(x, null) = x) \wedge$$
$$(\forall x\ \forall y\ sum(x, succ(y)) = succ(sum(x, y)))$$

wynikiem algorytmu sugerowania postaci syntezowanego programu jest

$$\_if(\_eq(i2, \_null), i1, \_succ(\_sum(i1, y)))$$

3.5.2 Formułowanie zadania w logice wielorodzajowej

W tym paragrafie roszerzymy język L do języka wielorodzajowego L_2 , teorię T do T_2 (nad L_2) i zmienimy $S(f)$ na S_2.

Następujące rzeczy będą dołączone do L, aby uzyskać L_2:

- *nowe rodzaje: Term, Formuła, Stan.* (rodzaj języka L nazywamy $Pierwotny$),
- *stałe:* $i_1, \ldots, i_n : Term$
- (dla każdego symbolu funkcji k-argumetowej g języka L), *nowa funkcja* k-argumentowa: $\_g \ : \ Term * \ldots * Term \rightarrow Term$
- (dla każdego symbolu relacji k-argumetowej r języka L), *nowa relacja* k-argumentowa: $\_r \ : \ Term * \ldots * Term \rightarrow bool$
- *nowe symbole:*

$\_f$	:	$\underbrace{Term * \ldots * Term}_{n\ razy} \rightarrow Term$ (dla rekurencyjnych wywołań)
$\_not$	:	*Formuła* $\rightarrow$ *Formuła*,
$\_or$	:	*Formuła* $*$ *Formuła* $\rightarrow$ *Formuła*,
$\_and$	:	*Formuła* $*$ *Formuła* $\rightarrow$ *Formuła*,
$\_if$	:	*Formuła* $*$ *Term* $\rightarrow$ *Term*,
mem	:	$\underbrace{Pierwotny * \ldots * Pierwotny}_{n\ razy} \rightarrow Stan$ (lista parametrów),
W	:	$Term * Term * Stan \rightarrow Pierwotny$, (wartościowanie termów; $W(p, f, s)$ to wartość termu f programu p obliczona na podstawie listy parametrów s)
B	:	*Term* $*$ *Formuła* $*$ *Stan* $\rightarrow$ *bool*, (wartościowanie formuł; $B(p, b, s)$ to wartość formuły b programu f obliczona na podstawie listy parametrów s)

Pierwszy parametr funkcji W i B i śledzi program syntezowany. To jest użyteczne przy podaniu semantyki funkcji rekurencyjnych.

Następujące *aksjomaty syntezy* będą dołączone do T, aby uzyskać T_2:

1. $\forall\ p: Term, a_1 : Pierwotny, \ldots, a_n : Pierwotny$
   $W(p, i_j, mem(a_1, \ldots, a_n)) = a_j$
   (dla każdego $1 \leq j \leq n$)

2. $\forall\ p: Term, s: Stan\ W(p, \_c, s) = c$
   (dla każdego symbolu stałej c języka L)

3. $\forall\ p: Term, s: Stan, t_1 : Term, \ldots, t_k : Term$
   $W(p, \_g(t_1, \ldots, t_n), s) = g(W(p, t_1, s), \ldots, W(p, t_k, s))$
   (dla każdego symbolu funkcji k-argumentowej g języka L)

4. $\forall\ p: Term, s: Stan, t_1 : Term, \ldots, t_k : Term$
   $B(p, \_r(t_1, \ldots, t_n), s) \leftrightarrow r(W(p, t_1, s), \ldots, W(p, t_k, s))$
   (dla każdego symbolu relacji k-argumentowej r języka L)

5. $\forall\ p: Term, b$:*Formuła*,$s: Stan$
   $B(p, \_not(b), s) \leftrightarrow \neg B(p, b, s)$

6. $\forall\ p: Term, b_1$:*Formuła*,b_2 :*Formuła*,$s: Stan$
   $B(p, \_or(b_1, b_2), s) \leftrightarrow B(p, b_1, s) \vee B(p, b_2, s)$

7. $\forall\ p: Term, b_1$:*Formuła*,b_2 :*Formuła*,$s: Stan$
   $B(p, \_and(b_1, b_2), s) \leftrightarrow B(p, b_1, s) \wedge B(p, b_2, s)$

8. $\forall\ p: Term, b$:*Formuła*,$t_1 : Term, t_2 : Term, s: Stan$
   $(B(p, b, s) \rightarrow W(p, \_if(b, t_1, t_2), s) = W(p, t_1, s)) \wedge$
   $(\neg B(p, b, s) \rightarrow W(p, \_if(b, t_1, t_2), s) = W(p, t_2, s))$

9. $\forall\ p: Term, s: Stan, t_1 : Term, \ldots, t_n : Term$
   $(\exists\ a_1 : Pierwotny, \ldots, a_n : Pierwotny\ s = mem(a_1, \ldots, a_n) \wedge$
   $less(W(p, t_1, s), \ldots, W(p, t_n, s), a_1, \ldots, a_n)) \rightarrow$
   $W(p, \_f(t_1, \ldots, t_n), s) =$
   $W(p, p, mem(W(p, t_1, s), \ldots, W(p, t_n, s)))$

Nowa specyfikacja S_2 będzie formułą $(\exists\ xf\ :\ Term\ S')$, gdzie xf jest nową zmienną i S' pochodzi z $S(f)$ przez zastąpienie wszystkich wystąpień $f(t_1, \ldots, t_n)$ w $S(f)$ przez $W(xf, xf, mem(t_1, \ldots, t_n))$.

3.5.3 Algorytm syntezy programów

Algorytm omówiony w tym paragrafie został zaimplementowany w języku C; jego implementacja używa zmodyfikowanej wersji Ottera do syntezy programu, do weryfikacji poprawności i własności stopu programu. Otter jest systemem dowodzenia twierdzeń klasycznej logiki pierwszego rzędu opartym na rezolucji i napisanym przez Williama McCune'a z Argonne National Laboratory. Opis oprogramowania znajduje się w następnym rodziale.

Algorytm 3.2

1. Sugeruj postać syntezowanego programu;
2. Jeśli wszystko zostało zasugerowane to
   (a) Używając systemu dowodzenia twierdzeń weryfikuj własność stopu programu względem $less$ w teorii T; (trzeba ustawić limit czasu)
   (b) Używając systemu dowodzenie twierdzeń weryfikuj poprawność programu względem specyfikacji; (trzeba ustawić limit czasu)
   (c) Jeśli dwa poprzednie kroki kończą się sukcesem zwróć sugerowany program jako wynik, wpp zatrzymuj się z porażką;
3. Formułuj L_2, T_2, S_2 jak wymienione w poprzednim paragrafie;
4. Używaj rezolucji wielorodzajowej do rozwiązywania problemu w następujący sposób:
   (a) Przekształć T_2 w zbiór klauzul i wstaw te klauzule do bazy;
   (b) Neguj S_2 na $\forall\ xf : Term\ S_3$; Niech $S_4 = S_3 \vee \$ANS(xf)$;
   (c) Zastąp wszystkie wystąpienia xf w S_4 przez sugerowaną postać programu;
   (d) Przekształć S_4 w zbiór klauzul i wstaw te klauzuli do bazy;
   (e) Używaj rezolucji do wyprowadzenia klauzuli postaci $\{\$ANS(P)\}$ z bazy.
   (f) Jeśli poprzedni krok zatrzymuje się z powodzeniem, to
      i. Jeśli P zawiera zmienne, to zastąp je przez dowolne stałe odpowiednich typów;

ii. Zwróć P jako wynik;

Przykłady będą podane w rozdziale 4

4 Oprogramowanie

Algorytm deskolemizacji opartej na postaci syntetycznej i algorytm syntezy programów zostały zaimplementowane w języku C w systemie operacyjnym Linux. Można zmodyfikować jednak te programy tak, aby działały w systemie DOS z kompilatorem Turbo C. Pierwszy program się nazywa *desk* (deskolemizacja), a drugi *prs* (synteza programów). W tym rozdziale opiszę strukturę programów, sposób instalacji i uruchamiania, postać danych wejściowych, testy i wyniki testów tych programów. Poza tym omówię rozwiązywanie problemów technicznych: Jak skracać formuły, jak zmodyfikować system Otter, aby wykorzystać go w programie *prs*. Otter jest dostępny w Internecie poprzez FTP pod adresem $info.mcs.anl.gov$ w pliku $pub/Otter/otter - 3.0.4.tar.Z$. Aktualna (w momencie pisania tej pracy) wersja Ottera posiada numer 3.0.4, zmodyfikowanej przeze mnie wersji nadałem numer 3.0.4p.

4.1 Instalacja i uruchamianie

Oprogramowanie wraz z dokumentacją użytkownika zawarte jest w dyskietce dołączonej do tej pracy. Katalog *deskolem* zawiera dokumentację i kody programów *desk* i *prs*. Katalog *otter*-3.0.4*p* zawiera zmodyfikowaną wersją programu Otter wersji 3.0.4. Katalog *przyklad* zawiera dane wejściowe i wyniki testów obóch programów *desk* i *prs*.

Instalacja oprogramowania polega na skopiowaniu plików z dyskietki na dysk twardy i wykonaniu polecenia:

```
do.sh
```

Dla programów *desk* i *prs* należy przygotować przed uruchomieniem plik danych wejściowych . Format pliku danych wejściowych jest opisany dalej w tym rozdziale. Sposób uruchamiania:

```
desk < plik_wejściowy
prs  < plik_wejściowy
```

4.2 Moduły programów

Pliki deklaracji:

consts.h :	stałe
teksts.h :	stałe tekstowe (komunikaty w języku polskim)
limits.h :	stałe ograniczeń
types.h :	definicje struktur danych
all.h :	deklaracje ważnych funkcji używanych w programie *desk*
synlib.h	deklaracje dodatkowych funkcji używanych w programie *prs*

Moduły:

rl.c :	definicje globalnych danych, konstruktory, destruktory
leks.c :	moduł leksykalny
misc.c :	funkcje pomocnicze
skom.c :	funkcje używane w skolemizacji
res.c :	rezolucja – podstawowe funkcje, rezolucja pojedyncza
skroc.c :	skracanie formuł
pdesk.c :	deskolemizacja – funkcje
desk.c :	deskolemizacja – program
leks2.c :	kopia pliku *leks.c*
sug.c :	sugerowanie postaci programu na podstawie specyfikacji
wer.c :	weryfikacja własności stopu programu
synlib.c :	funkcje używane w *prs*
prs.c :	synteza programów – program

Oryginalna wersja Ottera została rozszerzona o dwa pliki: *prs_lib.h* i *prs_lib.c*. Te pliki zawierają definicje i deklaracje dodatkowych funkcji dla syntezy programów. Pliki *cos.h*, *av.c*, *io.c*, *misc.c*, *options.c*, *process.c*, *resolve.c* oryginalnej wersji Ottera zostały zmodyfikowane. Te modyfikacje są zawarte między deklaracjami "ifdef PROG_SYNT" i "endif PROG_SYNT".

4.3 Wejście i wyjście

Przykład pliku danych wejściowych dla programu "desk":

```
# deklaracja symboli
    a.                       % stałe
    .                        % zbiór funkcji - pusty
```

```
    P:3, H:3, G:3, K:3.      % relacje

# Aksjomaty
    % nie ma

# Funkcje do deskolemizacji

   f:2, g:1.    % zmienne funkcyjne do eliminacji

# Formuła do deskolemizacji

     (Vx Vy P(x,y,f(x,y)) | G(x,a,g(x)))
   & (Vx Vy P(x,y,f(x,y)) | K(x,a,g(x)))
   & (Vx Vy H(x,y,f(x,y)) | G(x,a,g(x)))
   & (Vx Vy H(x,y,f(x,y)) | K(x,a,g(x))).

Komentarz ...
```

Dane wyjściowe dla pliku danych wejściowych podanego powyżej:

```
Formuła:
           (Vx Vy P(x,y,f(x,y)) | G(x,a,g(x)))
         & (Vx Vy P(x,y,f(x,y)) | K(x,a,g(x)))
         & (Vx Vy H(x,y,f(x,y)) | G(x,a,g(x)))
         & (Vx Vy H(x,y,f(x,y)) | K(x,a,g(x)))

Zmienne funkcyjne:  f, g,

Wynik : Vx_22 (Vy_28 Ec_5_4 H(x_22,y_28,c_5_4) &
                            P(x_22,y_28,c_5_4))
            | (Ec_8_2 G(x_22,a,c_8_2) & K(x_22,a,c_8_2))
```

Cztery znaki # są obowiązkowe i oznaczają początki czterech sekcji: *"deklaracja symboli"*, *"aksjomaty"*, *"zmienne funkcyjne do eliminacji"*, *"formuła do deskolemizacji"*. Napisy po # do końca linii nie mają znaczenia dla programu, są tylko komentarzami. Inne komentarze zaczynają się znakiem % i kończą się końcem linii.

Sekcja *"deklaracja symboli"* zawiera trzy części: *"deklaracja stałych"*, *"deklaracja funkcji"* i *"deklaracja relacji"*. Te części kończą się kropką. Symbole oddzielane są przecinkiem. Dla funkcji i relacji trzeba deklarować liczbę argumentów (ta liczba jest poprzedzona znakiem dwukropka). Nazwy symboli muszą zaczynać się literą, i mogą zawierać litery, cyfry i znak podkreślenie "_".

Sekcja *"aksjomaty"* zawiera formuły traktowane jako aksjomaty. Te aksjomaty biorą udział w procesie *skracanie formuł*. Często pomijamy tę sekcję. Każda formuła kończy się kropką. Spójniki logiczne są następujące:

```
V  - kwantyfikator ogólny,       ->   - implikacja
E  - kwantyfikator szczególny,   <->  - równoważność
&  - koniunkcja,                 !    - negacja
|  - alternatywa,                =    - równość
```

Syntaktyka formuł jest następująca:

```
<term> ::=    <zmienna>
              <funkcja>
              <funkcja> ( <lista_argumentów> )

<lista_argumentów> ::=
              <term>
              <term> , <term>

<formuła> ::= <relacja> ( <lista_argumentów> )
              <term> = <term>
              ! <formuła>
              <formuła> & <formuła>
              <formuła> | <formuła>
              <formuła> -> <formuła>
              <formuła> <-> <formuła>
              E <zmienna> <formuła>
              V <zmienna> <formuła>
              ( <formuła> )
```

Uszeregowanie spójników logicznych w kolejności malejących priorytetów wygląda następująco: $!\ \ \&\ \ |\ \ \leftrightarrow\ \ \rightarrow\ \ V\ \ E$, przy czym spójniki $\leftrightarrow$ i $\rightarrow$ mają równe priorytety. Wszystkie spójniki posiadają lewą łączność.

Sekcja *"zmienne funkcyjne do eliminacji"* musi zawierać jedną zmienną funkcyjną do eliminacji. Ta sekcja ma postać taką samą jak część *"deklaracja funkcji"* sekcji *"deklaracja symboli"*.

Sekcja *"formuła do deskolemizacji"* zawiera jedną formułę zakończoną kropką. Pozostała część pliku danych wejściowych jest ignorowana.

Przykład pliku danych wejściowych dla programu "prs":

```
%
% sum(a,b) - na podstawie succ, pred i rekurencyjnych wywoływań
%

# Deklaracja symboli
        null.                   % zero
        succ:1, pred:1.         % funkcje
        .                       % pusty zbiór relacji

# Aksjomaty
        Vx pred(succ(x)) = x.
        Vx -(null = succ(x)).

# Funkcja do syntezy
                           sum:2.

# Specyfikacja
                 (Vx sum(x,null) = x) &
                  (Vx Vy  sum(x,succ(y)) = succ(sum(x,y))).

# Ograniczenia
                 use(if,rec).

# Część dla Ottera
assign(max_literals,5).
assign(max_weight,300).

% POPRAWNOŚĆ REKURENCJI
list(usable).
  less2(x,succ(x)).
```

```
  less(mem(x,y1),mem(x,y2)) | -less2(y1,y2).
  less(mem(x1,y),mem(x2,y)) | -less2(x1,x2).
end_of_list.
```

Dane wyjściowe dla powyższego pliku danych wejściowych:

```
Uruchomiono Otter.

-------- PROOF -------- 76 [binary,75.1,11.1]
$ANS(if(_eq(_null,i2),i1,_succ(_sum(i1,_pred(i2))))).

Search stopped by max_proofs option.
UDAŁO SIĘ !!!
```

Dane wyjściowe programu *prs* są pisane w sposób inny niż standardowy. Syntezowany program jest zawarty w literale *$ANS*. $i1, i2, \ldots$ są kolejnymi parametrami. Symbole pierwotne są poprzedzone znakiem podkreślenia.

Format pliku danych wejściowych programu "prs":
Plik danych wejściowych dla programu *prs* zawiera sześć sekcji: *"deklaracja symboli"*, *"aksjomaty"*, *"program (funkcja) do syntezy"*, *"specyfikacja"*, *"ograniczenia"*, *"część dla Ottera"*. Pierwsze cztery sekcje są takie samie jak dla programu *desk*. Sekcja *"część dla Ottera"* musi być w postaci zrozumiałej dla programu *otter*. W tej części często podajemy pewne ograniczenia (np. *max_seconds, max_memory*) i definicję relacji *less*, która ma gwarantować własność stopu programu (relacja *less* jest typu $Stan * Stan \rightarrow bool$).

Sekcja *"ograniczenia"* jest ciągiem instrukcji. Każda instrukcja kończy się kropką. Zbiór dostępnych instrukcji to:

- *set*(flag) - ustawia flagę.
- *clear*(flag) - wyłączy flagę.
- *assign*(parametr, wartość) - przypisuje parametrowi wartość.
- *use*($\{if|and|or|not|rec\}\ldots$) - Jeśli chcemy używać *if,and,or,not,rec* w programie syntezowanym, to musimy je jawnie zadeklarować tym poleceniem. Np. przez *use(if,rec)* informujemy, że chcemy używać *if, rekurencji* i blokować *and,or,not*.

- *blocked_syms*(lista_symboli) - blokuje używanie pewnych symboli w programie syntezowanym. Np. jeśli deklarowane są stałe *a,b,c*, funkcje *f:1, g:2*, relacje *P:2, R:2* i polecenie $blocked\_syms(b, c, g, P, =)$ to *prs* będzie używać a, f, R i ignorować $b, c, g, P, =$ w trakcie syntezy programu.

Parametry ograniczenia:

- *max_if* - domyślnie 5, zakres $0 \rightarrow 10$, maksymalna ilość *if* używanych w programie syntezowanym.
- *max_dept_if* - domyślnie 5, zakres $0 \rightarrow 10$, maksymalna głębokość zagnieżdzania *if* w sobie.
- *max_andor* - domyślnie 5, zakres $0 \rightarrow 10$, maksymalna ilość *and,or* używanych w programie syntezowanym.
- *max_rec* - domyślnie 5, zakres $0 \rightarrow 10$, maksymalna ilość wywołań rekurencyjnych używanych w programie syntezowanym.
- *max_dept_term* - domyślnie 4, zakres $1 \rightarrow 20$, maksymalna głębokość termów występujących w programie syntezowanym.
- *max_length_term* - domyślnie MAX_INT, zakres $1 \rightarrow$ MAX_INT, maksymalna długość termów występujących w programie syntezowanym.
- *max_length_formula* - domyślnie MAX_INT, zakres $1 \rightarrow$ MAX_INT, maksymalna długość formuł występujących w programie syntezowanym.

Flagi:

- *top_down_synt* - domyślnie ustawiona. Jeśli ta flaga jest ustawiona to wszystkie literały inne niż pierwszy wszystkich *aksjomatów syntezy* będą blokowane (tzn. nie mogą być aktywnymi literałami) w procesie wnioskowania (binarna rezolucja, paramodulacja). Umożliwia to rozkładanie złożonych termów na mniejsze.
- *rec_in_rec* - domyślnie wyłączona. Jeśli ta flaga jest wyłączona, to niedozwolone jest zagnieżdzanie wywołań rekurencyjnych.

- *rec_ in_formula* - domyślnie wyłączona. Jeśli ta flaga jest wyłączona to niedozwolone jest występowanie wywołań rekurencyjnych w warunkach logicznych.
- *check_ types* - domyślnie ustawiona. Jeśli ta flaga jest ustawiona to *prs* będzie sprawdzać zgodność typów (*prs* używa logiki wielorodzajowej).
- *adjust_ otter_ input* - domyślnie wyłączona. Jeśli ta flaga jest ustawiona to przed wywołaniem Ottera program *prs* zawiesza działanie , aby pozwolić użytkownikowi zmodyfikować plik pośredni.

Przykład pliku pośredniego (dla podanej wcześniej specyfikacji funkcji *"sum"*) :

```
set(binary_res).
set(para_into).
set(para_into_vars).
clear(para_into_right).
clear(para_from_right).

clear(print_kept).
clear(print_new_demod).
clear(print_back_demod).
clear(print_back_sub).

list(usable). % SYNTEZA
  W(xp,if(xb,xt1,xt2),xs)=W(xp,xt1,xs) | -B(xp,xb,xs).
  W(xp,if(xb,xt1,xt2),xs)=W(xp,xt2,xs) | B(xp,xb,xs).
  W(xp,_sum(xt1,xt2),xs) =
              W(xp,xp,mem(W(xp,xt1,xs),W(xp,xt2,xs)))
    | -less(mem(W(xp,xt1,xs),W(xp,xt2,xs)),xs).

  W(xp,i1,mem(xt1,xt2)) = xt1.
  W(xp,i2,mem(xt1,xt2)) = xt2.
  W(xp,_null,xs) = null.
  W(xp,_pred(xt1),xs) = pred(W(xp,xt1,xs)).
  W(xp,_succ(xt1),xs) = succ(W(xp,xt1,xs)).

  B(xp,_eq(xt1,xt2),xs) | -(W(xp,xt1,xs) = W(xp,xt2,xs)).
```

```
  -B(xp,_eq(xt1,xt2),xs) | (W(xp,xt1,xs) = W(xp,xt2,xs)).

  x = x.
end_of_list.

list(demodulators).
  W(xp,i1,mem(xt1,xt2)) = xt1.
  W(xp,i2,mem(xt1,xt2)) = xt2.
  W(xp,_null,xs) = null.
  W(xp,_pred(xt1),xs) = pred(W(xp,xt1,xs)).
  W(xp,_succ(xt1),xs) = succ(W(xp,xt1,xs)).
  B(xp,_eq(xt1,xt2),xs) = (W(xp,xt1,xs) = W(xp,xt2,xs)).

  and(y,x) = and(x,y).
  or(y,x) = or(x,y).
  _eq(y,x) = _eq(x,y).
end_of_list.

list(usable). % AXIOMS
  pred(succ(xnx_3)) = xnx_3.
  null != succ(xnx_7).
end_of_list.

list(sos). % QUERY
W(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))),
  if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))),
  mem(f_x_16(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3)))),null))
 != f_x_16(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))))
| W(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))),
    if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))),
    mem(f_x_17(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3)))),
      succ(f_y_4(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3)))))))
   != succ(W(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))),
             if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3))),
       mem(f_x_17(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3)))),
           f_y_4(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3)))))))
| $ANS(if(_eq(i2,_null),i1,_succ(_sum(i1,xny_3)))).
end_of_list.
```

```
set(prog_synt).
assign(num_axioms_wolfa,10).

fun_to_synt([_sum(_,_)]).
term([i1,i2,_pred(_),_succ(_),_null]).
formula([_eq(_,_)]).
skolem([f_x_16(_),f_x_17(_),f_y_4(_)]).
skolem2([f_x_16(_),f_x_17(_),f_y_4(_)]).

weight_list(pick_given).
  weight(B($(0),$(0),$(0)),3).
  weight(W($(0),$(0),$(0)),3).
end_of_list.

weight_list(purge_gen).
  weight($ANS($dots(not($dots(and($(0),$(0)))))),10000).
  weight($ANS($dots(not($dots(or($(0),$(0)))))),10000).
  weight($ANS($dots(or($dots(and($(0),$(0))),$(0)))),10000).
  weight($ANS($dots(or($(0),$dots(and($(0),$(0)))))),10000).
end_of_list.

assign(max_literals,5).
assign(max_weight,300).

% POPRAWNOŚĆ REKURENCJI
list(usable).
  less2(x,succ(x)).
  less(mem(x,y1),mem(x,y2)) | -less2(y1,y2).
  less(mem(x1,y),mem(x2,y)) | -less2(x1,x2).
end_of_list.
```

Kody wyjściowe programów "desk", "prs":

201	błąd danych wejściowych
202	zatrzymanie programu opcją *adjust_otter_input*
203	powodzenie
204	zła specyfikacja
205	nie udało się zweryfikować programu

206	porażka
207	błąd danych wejściowych w części dla Ottera
208	brak pamięci
209	przerwanie z terminala
210	błąd wewnątrz (ang. bug)

4.4 Rozwiązanie problemów technicznych

4.4.1 Jak skracać formuły zachowując równoważność

Skracanie formuł jest używane w programie *desk* w procedurach "and-in", "or-in", "przesunięcie kwantyfikatorów wgłąb". W praktyce ta operacja okazuje się bardzo ważna bo rozmiar formuły dynamicznie rośnie po przekształceniach.

Jedną z metod jest uporządkowanie podformuł i zastosowanie aksjomatów idempotencji ($a \wedge a = a, \;\; a \vee a = a$). Wiemy, że spójniki logiczne takie jak $\&, |$ są przemienne, siąsiednie kwantyfikatory tego samego typu mogą być zamienione. Uporządkujemy podformuły według pewnego porządku leksykalnego i potem zastosujemy aksjomaty idempotencji do usuwania zbędnych podformuł. Wywołanie jest rekurencyjne, najpierw skracamy podformuły, a potem główną formułę. Taką metodą zastosowano w Otterze, autorzy nazywają ją *leksykalną demodulacją*. W praktyce ta metoda nie skraca formuły do końca i w przypadku *desk* zajmuje ona dużo czasu.

Drugą metodą skracania formuł jest zastosowanie systemu dowodzenia. W programie *desk* zastosowałem *rezolucję pojedynczą* i okazało się, że ta metoda działa efektywnie.

Definicja 4.1 Rezolucją pojedynczą (ang. unit resolution) *nazywamy rezolucję, w której rezolwentę uzyskuje się przy użyciu co najmniej jednej przesłanki pojedynczej, bądź faktora pojedynczego przesłanki.*

Rezolucja pojedyncza nie jest pełna, ale zawsze się zatrzymuje i jest zbieżna bo zawsze produkuje klauzule mniejsze niż jedna z przesłanek. Zastosujemy rezolucję pojedynczą do skracania formuł w następujący sposób:

Algorytm 4.1 (skracanie formuły)

Dane wejściowe: formuła Φ (może być otwarta)

1. Jeśli Φ jest otwarta to czasowo zmień Φ na zamkniętą przez zastąpienie wszystkich zmiennych wolnych przez różne nowe stałe.

2. Wykonuj *"skracanie formuły"* dla wszystkich bezpośrednich podformuł formuły Φ.

3. Jeśli Φ jest postaci $\varphi_1 \wedge \ldots \wedge \varphi_n$ to dla każdego j $(1 \leq j \leq n)$

   (a) sprawdź czy φ_j jest wnioskiem pozostałych czynników i aksjomatów podanych w pliku danych wejściowych (używaj rezolucji pojedynczej w tym celu)

   (b) jeśli tak jest, to usuń φ_j z Φ.

4. W przeciwnym przypadku, jeśli Φ jest postać $\varphi_1 \vee \ldots \vee \varphi_n$ to

   (a) neguj Φ (prowadzić Φ do koniunkcji pewnych formuł)

   (b) wykonuj akcje takie jak w kroku 3

   (c) neguj Φ

5. Jeśli krok 1 został wykonany to wykonuj przeciwny proces.

4.4.2 Modyfikacje Ottera w celu wykorzystania w programie "prs"

Otter jest systemem automatycznego dowodzenia twierdzeń klasycznej logiki opartym na rezolucji. Reguły wnioskowania używane w systemie Otter są to między innymi: binarna rezolucja (w tym faktoryzacja), hyperrezolucja, paramodulacja, demodulacja. Strategia kontroli procesu wnioskowania w Otter'ze jest następująca:

- Na początku wszystkie klauzule są podzielone na dwie listy: lista aksjomatów (USABLE) i lista uzasadnień (ang. set of support – SOS).

- W każdej iteracji głównej pętli, jedna klauzula (najlepszy kandydat według pewnego kryterium) będzie wybrana z listy SOS, wszystkie wnioskowania używające tej klauzuli, listy USABLE i wcześniej wybranych klauzul będą generowane. Nowe klauzule będą wkładane do listy SOS.

Program *prs* formalizuje zadanie w sposób rozumialny dla Ottera, zapisuje tę formalizację do pliku *otter.in* i wywołuje *otter* do rozwiązywania problemu zapisanego w pliku *otter.in*. Założenia dwóch programów są różne, stąd konieczne są pewne modyfikacje dla oryginalnego Ottera. W Otter'ze dokonane zostały cztery modyfikacje:

- *Ograniczenia na postać wyniku:* W celu zmniejszenia przestrzeni wyszukiwania wyniku pewne nowe *parametry* i *flagi* są włączone do Ottera. Te parametry i flagi są wymienione w sekcji 4.3 "Wejście i wyjście". Kiedy nowa klauzula jest dołączona do bazy, sprawdzamy czy wykracza poza ograniczenia, jeśli tak jest to rezygnujemy z operacji. Klauzula wykracza poza ograniczenia wtedy i tylko wtedy, gdy zawiera literał odpowiedzi ($\$ANS(\ldots)$) i ta odpowiedź wykroczy poza ograniczenia.

- *Blokowanie pewnych wnioskowań:* Do tego celu służy flaga *top_down_synt*. Jeśli ta flaga jest ustawiona to będą blokowane wszystkie literały inne niż pierwszy wszystkich *aksjomatów syntezy* (tzn. te literały nie mogą być aktywne) w procesie wnioskowania (binarna rezolucja, paramodulacja). To umożliwia rozkładanie złożonych termów na mniejsze. Dla każdej nowej klauzuli sprawdzamy skąd ona pochodzi, jeśli jest wynikiem operacji, która używa blokowanych literałów, to nie będziemy dołączać jej do bazy.

- *Jednoznaczność wyniku:* Jeśli na końcu dostajemy $\$ANS(wynik1) \vee \$ANS(wynik2)$ to nie wiadomo który ($wynik1$ czy $wynik2$) jest właściwym wynikiem. Przyjmujemy następujące rozwiązanie : dla każdej nowej klauzuli, po generowaniu wszystkich faktorów sprawdzimy, czy zawiera ona przynajmniej dwa literały odpowiedzi. Jeśli tak jest, to nie będziemy dołączać jej do bazy.

- *Zgodność typów:* Ponieważ *prs* formalizuje zadanie w logice wielorodzajowej, a Otter jest oparty o klasyczną logikę pierwszego rzędu, więc sprawdzenie zgodność typów musi być dołączone do Ottera. Nowymi rodzajami są *Term, Formuła, Pamięć.* Można w prosty sposób określić typ i typy parametrów wszystkich symboli z wyjątkiem zmiennych. Zmienne mogą mięć dowolny typ, ale w zakresie jednej klauzuli jedna zmienna nie może mieć jednocześnie dwóch różnych typów. Dla każdej nowej klauzuli sprawdzamy czy zawiera ona konflikt typów, jeśli tak jest to nie będziemy dołączać jej do bazy.

4.5 Testy

4.5.1 Testy dla programu "desk"

W katalogu .../*przyklady* jest plik *testdesk*, ten plik zawiera testy dla programu *desk*. Wszystkie przykłady (2.1 → 2.11) zostały testowane.

Program *test0* został napisany do testowania prgramu *desk*. Program *test0* bierze formułę, skolemizuje ją, przesuwa kwantyfikatory w głąb, wykonuje "or-in", a potem deskolemizuje zmodyfikowaną formułę. W większości przypadków program *test0* zatrzymuje się z powodzeniem. Format pliku danych wejściowych dla programu *test0* jest taki sam jak dla programu *desk* z wyjątkiem: teraz druga i trzecia sekcja nie mają znaczenia, a czwarta seksja zawiera formułę do testowania. A oto wyniki kilku testów:

Wynik testu 1 (programu *test0*):

```
Pierwotna formuła :

          Vx (Vy Ez G(x,y,z) & H(x,y,z)) |
             (Vu Ev P(x,u,v) & K(x,u,v)).

Po ‘skolemizacji’,
   ‘przesuwaniu kwantyfikatorów wgłąb’, ‘or-in’ :

    (Vx (Vy G(x,y,fz(x,y))) | (Vu P(x,u,fv(x,u))))
  & (Vx (Vy G(x,y,fz(x,y))) | (Vu K(x,u,fv(x,u))))
  & (Vx (Vy H(x,y,fz(x,y))) | (Vu P(x,u,fv(x,u))))
  & (Vx (Vy H(x,y,fz(x,y))) | (Vu K(x,u,fv(x,u)))).

Zmienne funkcyjne do eliminacji:  fv, fz

Wynik :
          Vx (Vu Ec_5 P(x,u,c_5) & K(x,u,c_5)) |
             (Vy Ec_9 H(x,y,c_9) & G(x,y,c_9))
```

Wynik testu 2 (programu *test0*):

```
Pierwotna formuła:
```

```
        Vx (Vy Ez P(x,y,z) | H(x,y,z)) |
           (Et Vu Ev G(x,t,u,v) & K(x,t,v)).
```

```
Po ‘skolemizacji’,
   ‘przesuwaniu kwantyfikatorów wgłąb’, ‘or-in’ :

    (Vx (Vy P(x,y,fz(x,y)) | H(x,y,fz(x,y))) |
        (Vu G(x,ft(x),u,fv(x,u))))
  & (Vx (Vy P(x,y,fz(x,y)) | H(x,y,fz(x,y))) |
        (Vu K(x,ft(x),fv(x,u)))).

Zmienne funkcyjne do eliminacji:  fv, ft, fz

Wynik :
   Vx Ec_13 (Vu Ec_10 G(x,c_13,u,c_10) & K(x,c_13,c_10)) |
            (Vy (Ec_17 P(x,y,c_17)) | (Ec_19 H(x,y,c_19)))
```

4.5.2 Testy dla programu “prs”

Program *prs* działa efektywnie (zatrzymuje się z powodzeniem po kilku sekundach) tylko dla małych zadań lub dla specyfikacji pseudoprogramowych. Dla zagadnienia syntezy programów jest potrzebnych więcej heurystyk i metod (np. indukcja). Program *prs* syntezuje programy *max, sum, mult, nwd* w kilkunastu sekundach. Wsród tych programów *sum, mult* są ciekawsze (program *max* jest za mały, a podana tu specyfikacja dla *nwd* jest trywialna). A oto pliki danych wejściowych dla *max, mult, nwd*:

```
% SYNTEZA FUNKCJI "max"

# Deklaracja symboli
  .                      % pusty zbiór stałych
  .                      % pusty zbiór funkcji
  egt:2.                 % >=

# Aksjomaty
                Vx egt(x,x).
                Vx Vy egt(x,y) | egt(y,x).
```

```
# Funkcja do syntezy
                          max:2.

# Specyfikacja
                Vx Vy (max(x,y)=x | max(x,y)=y) &
                      egt(max(x,y),x) & egt(max(x,y),y).

# Ograniczenia i Sterowanie
                use(if).
                blocked_syms(=).

# Część dla Ottera: pusta
```

Wynikiem tego testu jest *if(_egt(i2,i1),i2,i1).*

```
%  SYNTEZA FUNKCJI ‘mult’
%  na podstawie sum, succ, pred, null

# Deklaracja symboli
        null.                    % zero
        succ:1, pred:1, sum:2 .  % funkcje
        .                        % pusty zbiór relacji

# Aksjomaty
        Vx pred(succ(x)) = x.
        Vx -(null = succ(x)).

# Funkcja do syntezy
                           mult:2.

# Specyfikacja
               (Vx mult(x,null) = null) &
               (Vx Vy  mult(x,succ(y)) = sum(mult(x,y),x)).

# Ograniczenia
               use(if,rec).
```

```
# Część dla Ottera
assign(max_literals,5).
assign(max_weight,300).

% poprawność rekurencji
list(usable).
  less2(x,succ(x)).
  less(mem(x,y1),mem(x,y2)) | -less2(y1,y2).
  less(mem(x1,y),mem(x2,y)) | -less2(x1,x2).
end_of_list.
```

Wynikiem tego testu jest
if(_eq(_null,i2),_null,_sum(_mult(i1,_pred(i2)),i1)).

```
%  SYNTEZA FUNKCJI nwd : największy wspólny dzielnik

# deklaracja symboli
   .            % pusty zbiór stałych
   minus:2.     % funkcje
   gt:2.        % relacja >

# aksjomaty
            Vx x=x.
            Vx Vy x=y | gt(x,y) | gt(y,x).
            Vx Vy x=y -> -gt(x,y).
            Vx Vy gt(x,y) -> -gt(y,x).

# funkcja do syntezy
                          nwd:2.

# specyfikacja

   (Vx nwd(x,x)=x) &
   (Vx Vy gt(x,y) -> nwd(x,y)=nwd(minus(x,y),y)) &
   (Vx Vy gt(y,x) -> nwd(x,y)=nwd(x,minus(y,x))).

# Ograniczenie i Sterowanie
```

```
            use(if,rec).

# Część dla Ottera
list(usable). % poprawność rekurencji
  less2(minus(x,y),x).
  less(mem(x,y1),mem(x,y2)) | -less2(y1,y2).
  less(mem(x1,y),mem(x2,y)) | -less2(x1,x2).
end_of_list.
```

Wynikiem tego testu jest
if(_eq(i2,i1),i1,if(_gt(i1,i2),_nwd(_minus(i1,i2),i2), _nwd(i1,_minus(i2,i1)))).

5 Zastosowania

Deskolemizacja jest używana w algorytmach eliminacji kwantyfikatorów relacyjnych. Algorytmy eliminacji kwantyfikatorów relacyjnych DLS i SCAN używają skolemizacji w trakcie działania i na końcu używają deskolemizacji do eliminacji kwantyfikatorów funkcyjnych. Jeśli chodzi o zastosowania algorytmów eliminacji kwantyfikatorów relacyjnych, to można wymienić *circumscription* i *teorię korespondencji* (opis tych zastosowań można znaleźć w [NOS96]).

W rozdziale 3 opisałem jeden z algorytmów syntezy programów i wykazałem, że zagadnienie syntezy programów można sformułować w sposób podobny do zagadnienia deskolemizacji.

W tym rozdziale krótko opiszę algorytmy DLS i SCAN i podam przykłady dla nich. Bardziej szczególne opisy tych algorytmów można znaleźć w [NOS96].

Mówimy, że formuła Φ jest pozytywna względem relacji P jeżeli w $NNF(\Phi)$ (ang. negation normal form of Φ) nie istnieje żadne negatywne wystąpienie relacji P. Dualnie mówimy, że Φ jest negatywna względem relacji P jeżeli w $NNF(\Phi)$ nie ma żadnego pozytywnego wystąpienia relacji P.

Następujący lemat został udowodniony przez Ackermanna, można znaleźć dowód tego lematu w pracy [Sz92].

Lemat 5.1 (Ackermann) Niech P będzie zmienną relacyjną i niech Φ i $\Psi(P)$ będą formułami pierwszego rzędu. Jeśli $\Psi(P)$ jest pozytywna względem P i Φ nie zawiera żadnego wystąpienia relacji P, to

$$(\exists P\ \forall \overline{x}\ (\neg P(\overline{x}) \vee \Phi) \wedge \Psi(P)) \Longleftrightarrow \Psi\Big[P(\overline{\alpha}) \leftarrow [\Phi]^{\overline{x}}_{\overline{\alpha}}\ \Big],$$

gdzie $[\Phi]^{\overline{x}}_{\overline{\alpha}}$ jest instancją formuły Φ przez podstawienie $[\overline{x}/\overline{\alpha}]$, a prawa strona powyższej równości pochodzi z Ψ przez zastąpienie wszystkich wystąpień $P(\overline{\alpha})$ przez $[\Phi]^{\overline{x}}_{\overline{\alpha}}$. (Zmieniamy nazwy zmiennych związanych jeśli jest to potrzebne.) Podobnie dla przypadku kiedy znak $\neg$ przed P jest pominięty i Ψ jest negatywna względem P.

Algorytm DLS jest zdefiniowany w pracy [DLS94], stanowi on rozszerzenie algorytmu eliminacji kwantyfikatorów relacyjnych podanego w [Sz92].

Algorytm 5.1 (DLS)
Dane wejściowe: formuła $\exists P\ \Phi$, gdzie P jest zmienną relacyjną a Φ formułą pierwszego rzędu.
Cel: eliminować kwantyfikator $\exists P$ z danej formuły.

1. *Przygotowanie i rozdzielenie:* Formuła, którą chcemy uzyskać, powinna być postaci

$$\exists P\ \exists \overline{x}\ [(\Phi_1(P) \wedge \Psi_1(P)) \vee \ldots \vee (\Phi_n(P) \wedge \Psi_n(P))],$$

gdzie, dla każdego $1 \leq i \leq n$, $\Phi_i(P)$ jest pozytywna względem P, a $\Psi_i(P)$ jest negatywna względem P. Kroki tej fazy są następujące:

   (a) Sprowadź Φ do postaci NNF (ang. negation normal form).

   (b) Przesuń kwantyfikatory ogólne wgłąb i przesuń kwantyfikatory egzystencjalne na zewnątrz.

   (c) Wykonuj przekształcenie "and-in".

   (d) Jeśli wynik poprzednich kroków nie spełnia oczekiwań, to zatrzymaj się z porażką; wpp zastąp formułę wejściową przez

$$\exists \overline{x}\ [(\exists P\ \Phi_1(P) \wedge \Psi_1(P)) \vee \ldots \vee (\exists P\ \Phi_n(P) \wedge \Psi_n(P))].$$

   (e) Dla każdego $1 \leq i \leq n$, próbuj znaleźć formułę pierwszego rzędu równoważną formule $\exists P\ \Phi_i(P) \wedge \Psi_i(P)$

2. *Przygotowanie do zastosowania lematu Ackermanna:* Ta faza próbuje przekształcić formułę postaci $\exists P\ \Phi(P) \wedge \Psi(P)$, gdzie $\Phi(P)$ jest pozytywna względem P, a $\Psi(P)$ jest negatywna względem P, do (dwóch) postaci, dla których można zastosować lemat Ackermanna. Trzeba próbować obie możliwości, bo deskolemizacja w następnej fazie może się udać tylko dla jednej. W tym celu można używać skolemizacji i nastepujących przekształceń:

$$\begin{array}{rcl} P(\overline{x}) & \Longleftrightarrow & \forall \overline{y}\ (P(\overline{y}) \vee \overline{x} \neq \overline{y}) \\ P(\overline{x_1}) \vee \ldots \vee P(\overline{x_n}) & \Longleftrightarrow & \exists \overline{z}\ [(\overline{z} = \overline{x_1} \vee \ldots \vee \overline{z} = \overline{x_n}) \wedge P(\overline{z})] \end{array}$$

3. *Zastosowanie lematu Ackermanna:* Zastosuj lemat Ackermanna do eliminacji kwantyfikatora $\exists P$. W razie potrzeby wykonuj deskolemizację.

Przykład 5.1

$$\exists P\ (\forall x\ \exists y\ \forall z\ \forall t\ (P(z,t,x) \rightarrow (\phi(z,t,x) \wedge \psi(z,x,y)))) \wedge$$
$$(\forall x\ \exists y\ \forall z\ P(x,y,z))$$

Algorytm DLS po zastosowaniu lematu Ackermanna przeprowadza powyższą formułę na:

$$\exists f\ \forall x\ \exists y\ \forall z\ \phi(x,y,z) \wedge \psi(x,z,f(z))$$

a po zastosowaniu programu *desk* sprowadzi do:

$$\forall z_2\ \exists t\ \forall x\ (\exists y\ \forall z_1\ \phi(x,y,z_1)) \wedge \psi(x,z_2,t).$$

Przykład 5.2

$$\exists P\ (\forall x_0\ \forall x_1\ \exists y\ \forall x_2\ (P(x_0,x_1,x_2) \rightarrow \Phi(x_1,x_2,y))) \wedge$$
$$\exists x_1\ \forall x_2\ P(x_2,g(x_2),x_1) \wedge P(x_1,x_1,x_2)$$

Po zastosowaniu lematu Ackermanna, algorytm DLS przeprowadza powyższą formułę na:

$$\exists f\ \exists x_1\ \forall x_2\ \Phi(g(x_2),x_1,f(x_2,g(x_2))) \wedge \Phi(x_1,x_2,f(x_1,x_1))$$

a po zastosowaniu programu *desk* sprowadzi do:

$$\exists x_1\ \exists y\ \forall x_2\ \exists z\ ((x_2 = x_1 \wedge g(x_2) = x_1) \rightarrow z = y) \wedge$$
$$\Phi(g(x_2),x_1,z) \wedge \Phi(x_1,x_2,y)$$

Algorytm SCAN jest zdefiniowany w pracy [GO92]. SCAN na wejściu dostaje formułę α postaci $\exists P_1 \ldots \exists P_k\ \Phi$, gdzie P_i są zmiennymi relacyjnymi, a Φ jest formułą pierwszego rzędu i próbuje równocześnie wyeliminować wszystkie kwantyfikatory relacyjne.

Algorytm 5.2 (SCAN)

1. Sprowadź Φ do zbioru klazul.

2. Wszystkie C-rezolwenty i C-faktory z aktywnym literałem zawierającym jedną z zmiennych relacyjnych $P_1, \ldots, P_n$ będą generowane. C-rezolucja jest zdefiniowana następująco:

$$\frac{P(s_1,\ldots,s_n) \vee C\ \ ,\ \ P(t_1,\ldots,t_n) \vee D}{C \vee D \vee s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n}$$

natomiast C-faktoryzacja:

$$\frac{P(s_1, \ldots, s_n) \vee P(t_1, \ldots, t_n) \vee C}{P(s_1, \ldots, s_n) \vee C \vee s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n}$$

(zakładamy, że klauzule nie mają wspólnych zmiennych). Kiedy zostaną wygenerowane wszystkie rezolwenty i faktory między wybranym literałem i resztą bazy klauzul, to klauzula zawierająca ten literał będzie usuwana z bazy. Jeśli wszystkie klauzule zostaną usunięte w ten sposób to α jest tautologią. Jeśli została wygenerowana pusta klauzula to α jest niespełnialna.

3. Jeśli krok 2 zatrzymuje się z niepustą bazą, to wykonuj deskolemizację do eliminacji kwantyfikatorów funkcyjnych (utworzonych w kroku 1 w procesie skolemizacji) dla bazy.

Przykład 5.3 (Ten przykład jest wzięty z pracy [NOS96])

$$\exists P\ \forall x\ \forall y\ \exists z\ (\neg P(a) \vee Q(x)) \wedge (P(y) \vee Q(a)) \wedge P(x)$$

Po wykonywaniu kroku 1 i kroku 2 algorytmu SCAN dostajemy bazę

$$\{\ \{Q(a)\},\ \{a \neq f(x,y), Q(x)\}\ \},$$

czyli formułę

$$\exists f\ \forall x\ \forall y\ Q(a) \wedge (a \neq f(x,y) \vee Q(x)).$$

Po deskolemizacji (wykonaniu programu *desk*) dla powyższej formuły dostajemy

$$\forall x\ \exists z\ Q(a) \wedge (a \neq z \vee Q(x))$$

6 Podsumowanie

W tej pracy został skonstruowany i zaimplementowany algorytm deskolemizacji opartej na postaci syntaktycznej. Jedną z zalet tego algorytmu jest krótki czas działania (wynik może być powodzeniem lub porażką). Opracowano także inny schemat deskolemizacji, który oparty na wyszukiwaniu podstawień dla zmiennych funkcyjnych. Ten schemat nie został zaimplementowany. Powienien zostać ulepszony i uszczegółowiony przed zastosowaniem. W tej pracy wykazałem także związek między deskolemizacją a zagadnieniem syntezy programów. Algorytm syntezy programów został skonstruowany i zaimplementowany. Ten ostatni algorytm wskazuje na przydatność schematu deskolemizacji. Program deskolemizacji opartej na postaci syntaktycznej został napisany w języku C. Można wykorzystać ten program jako uzupełnienie algorytmów eliminacji kwantyfikatorów predykatowych DLS i SCAN.

References

[CL73] C.L.Chang, R.C.T.Lee. *Symbolic logic and mechanical theorem proving*, Academic Press, 1973.

[DLS94]
P.Doherty, W.Łukaszewicz, A.Szałas. *Computing Circumscription Revisited: A Reduction Algorithm.* Technical Report LiTH-IDA-R-94-42, Institutionen för Datavetenskap, University of Linköping, 1994. A preliminary report published in *Proceedings of IJCAI 95*, 95. To appear in *Journal of Automated Reasoning*

[GN87]
M.R.Genesereth and N.J.Nilsson. *Logical foundations of artificial intelligence.* Morgan Kaufmann, 1987.

[GO92]
D.M.Gabbay and H.J.Ohlbach. *Quantifier Elimination in Second-Order Predicate Logic.* In Bernhard Nebel. Charles Rich, and William Swartout, editors, *Principles of Knowledge Representation and Reasoning (KR92)*, 425-435. Morgan Kaufmann, 1992. Also published in the South African Computer Journal, 7:35-43, 1992.

[LO94]
K.Lau and M.Ornaghi. *On Specification Frameworks and Deductive Synthesis of Logic Programs.* Lecture Notes In Computer Science – 883, Verlag Springer, 1994.

[Łuk90]
W. Łukaszewicz. *Non-motononic Reasoning, Formalization of commonsense reasoning.* E.Horwood, 1990.

[McC90]
W.McCune. *OTTER* 2.0. In Mark Stickel, editor. *Proc. of* 10^{th} *International Conference on Automated Deduction, LNAI 449*, 663–664. Verlag Springer, 1990.

[NOS96]
A.Nonnengart, H.J.Ohlbach, and A.Szałas. *Elimination of Predicate Quantifiers*, 1996.

[Sch94]
R.Scherl. *Equality and Constrained Resolution.* Automated Deduction – CADE-12. Verlag Springer, 1994.

[Sim94] H.Simmons. *Monotonous elimination of predicate quantifiers.* Journal of Logic and Computation, 1994.

[Sz92]
A. Szałas. *On the Correspondence between Modal and Classical Logic: an Automated Approach.* Technical Report MPI-I-92-209, Max-Planck-Institut für Informatik, Saarbücken, 1992. Also published in *Journal of Logic and Computation,* 3:605-620, 1993.

[Sz93]
A. Szałas. *Zarys dedukcyjnych metod automatycznego wnioskowania*, 1993.

[Tar94]
A.Tarlecki. *Programowanie funkcyjne*, 1994.

[Wój91]
M.Wójcik. *Zasada rezolucji - Metoda automatycznego wnioskowania*, PWN, 1991.