Софийски университет
„Св. Климент Охридски“

Магистърска дипломна работа

Реализация на алгоритъма на Крахт

Автор:
Димитър Красимиров Димитров

Ръководители:
проф. Димитър Вакарелов
проф. Тинко Тинчев

София, 2012

Съдържание

1 Въведение

Предмет на тази дипломна работа е автоматичният превод на условия, изказани на предикатен език от първи ред, към еквивалентни условия, изказани на модален език. Нашата задача е реализация на алгоритъм даден от Маркус Крахт [9], решаващ проблема за един синктатично определен клас предикатни формули. В модалния език ще се ограничим с най-много изброимо количество, едноместни модални оператори $\Box_i$, $\Diamond_i$, за i пробягващо някое изброимо индексно множество. Когато сигнатурата включва само един чифт такива оператори, индекса няма да записваме. Стандартната семантика на Крипке, интерпретира модалните формули в чисто релационни структури X с двуместни релации, в които всяка една от тях R_i играе роля в интерпретирането на $\Box_i$ и $\Diamond_i$. По същество смисълът на всяка модална формула A се задава чрез едноместно, монадично условие от втори ред $ST_x A$ с единствена свободна индивидна променлива x, споменаващо релациите R_i и третиращо съждителните променливи P като свободни предикатни променливи. Всяка оценка на предикатните променливи в подмножества на носителя на някоя структура X, превръща $ST_x A$ в условие от първи ред. Семантика на една формула, разбира се, трябва да е инвариантна относно всевъзможните такива оценки и следователно представлява универсалното затваряне на $ST_x A$ относно предикатните ѝ променливи. Например, ако вземем формулата $P \to \Diamond_i P$ и оценка b в структурата X, то за конкретен елемент $a \in X$, семантиката на Крипке постулира:

$$X, b \Vdash a \in (P \to \Diamond_i P) \iff X, b \vDash (a \in P \to \exists y(R_i a y \wedge y \in P))$$

Абстрахирайки се от b:

$$X \Vdash a \in (P \to \Diamond_i P) \iff X \vDash \forall P\,(a \in P \to \exists y(R_i a y \wedge y \in P))$$

Забележете, че отдясно изискваме удовлетвореност за всяко подмножество P на X, в частност и за $P = \{a\}$. Това показва, че формулата от втори ред влече рефлексивност $R_i aa$ в конкретната точка a. Обратно, знаейки $R_i aa$, условието от втори ред очевидно е удовлетворимо за произволно P. Получихме, че модалната формула $P \to \Diamond_i P$ и предикатната $R_i xx$ са еквивалентни. Две такива формули ще наричаме модално съответни (една на друга). Може да се покаже, че в общия случай съответни формули не винаги съществуват, при това и за двете посоки.

Ван Бентъм [2] поставя въпросите за съответствието:

1. Има ли алгоритъм, който по дадена модална формула намира съответна от първи ред.

2. И обратно, има ли алгоритъм, който по формула от първи ред да намира съответна модална.

Чагрова [5, 4, 6] обаче дава отрицателен отговор и на двата от тях. Това не значи, че не можем да постигнем такава автоматизация за определени класове от формули. Салквист намира [11] един такъв доста широк клас модални формули, за които имаме доста полезно свойство: ако такива формули прибавим като аксиоми към минималната модална логика, наготово получаваме пълнота относно класа, определен от съответните им предикатни формули. Например, заедно двете аксиоми $P \to \Diamond P$ и $\Diamond\Diamond P \to \Diamond P$, ни дават логика пълна относно класа на рефлексивните и транзитивни релации. За всички Салквистови формули тази пълнота се получава чрез канонична конструкция, а намирането на съответните предикатни формули става алгоритмично (алгоритъм на Ван Бентъм – Салквист [1]). Това драматично улеснява анализирането на много модални логики, възникващи в практиката, поради което се развиха няколко компютърни системи като SCAN, DLS, SQEMA, превеждащи от модални към предикатни формули. Всяка една от тях гарантирано дава резултат за класа на Салквистовите формули, а SQEMA [7, Вакарелов, Горанко, Conradie] значително го разширява, запазвайки ценното свойство канонична пълнота.

Маркус Крахт решава другата страна на проблема: описание на тези предикатни формули, които са съответни на Салквистови и как алгоритмично да преминаваме от едните към другите. Не всяка модална формула, имаща съответна от първи ред, е Салквистова, но дори и без това проблемът е достатъчно сложен и в общия случай неразрешим. Класът, който намира Крахт, също като този на Салквист, е синтактично определен. Крахт [9] доказва основната теорема:

- На всяка Крахтова формула модално съответства Салквистова.
- На всяка Салквистова формула модално съответства Крахтова.
- Намирането на тези съответни формули е ефективна процедура.

Този резултат не само значително обогатява познанието ни в областта, но също така има и голяма практическа стойност. Когато искаме да аксиоматизираме модално някой клас, зададен чрез условия от първи ред, достатъчно е те да са в Крахтов вид:

Крахтова формула е формула от първи ред с най-много една свободна променлива, изградена само с операциите $\varphi \wedge \psi$, $\varphi \vee \psi$, $\forall y.R_i xy \to \varphi$, $\exists y.R_i xy \wedge \varphi$, истина $\top$, лъжа $\bot$ и композиции от вида $R^\alpha xy = (R_{a_1} \circ \cdots \circ R_{a_n})xy$, където поне една двете променливи x и y е *наследствено универсална*: свободната променлива на формулата или променлива свързана от квантор $\forall$, който не е в областта на действие на квантор $\exists$. В частния случай, когато в композицията имаме $n = 0$, тя се редуцира до обикновеното равенство $x = y$.

Алгоритъмът на Крахт, по дадена входна Крахтова формула φ, намира модална формула A, съответна на нея. Ако с $\equiv$ бележим модалната съответ-

ност, то можем да кажем че това е процедура f, за която $\varphi \equiv f(\varphi)$. В основата си алгоритъмът се базира на правила за смятане с модални съответности. Например, знаейки $\varphi \equiv A$ и $\psi \equiv B$, можем да изведем $\varphi \wedge \psi \equiv A \wedge B$. Използвайки такива правила, алгоритъмът придобива рекурсивния вид:

$$\begin{aligned} f(R^{\alpha} x y) &= A \\ f(\varphi \wedge \psi) &= f(\varphi) \wedge f(\psi) \\ f(\varphi \vee \psi) &= f(\varphi) \vee f(\psi) \\ f(\forall y. R_i x y \rightarrow \varphi) &= \Box_i f(\varphi) \\ f(\exists y. R_i x y \wedge \varphi) &= \Diamond_i f(\varphi) \end{aligned}$$

Разбира се, горното твърдение е твърде хубаво, за да е истина. Основният проблем е, че подформулите могат да имат повече от една свободна променлива и модалното съответствие трябва да се разшири $\varphi \equiv ?$ за предикатни формули с повече от една свободни променливи. Тук веднага възниква въпросът какво стои от дясната страна, понеже семантиката на една модална формула винаги е едноместен предикат. За да определя n-местни предикати чрез модални условия, Крахт попълва $?$ със списък от модални формули. Това изисква една твърде деликатна дефиниция на релацията $\equiv$. Но дори и това е лъжа. За удобство вместо $\equiv$ се използва дуално дефинирана релация, означавана с $\rightsquigarrow$, и на практика алгоритъмът работи върху отрицания на Крахтови формули. Когато входната формула е с една променлива, на нея винаги съответства точно една модална формула. В този случай между правата $\equiv$ и дуалната $\rightsquigarrow$ се преминава чрез взимане на отрицания:

$$\varphi \equiv A \iff \neg\varphi \rightsquigarrow \neg A$$

Самата работа със списъци от модални формули налага доста допълнителни ограничения на вида на съответствията, с които се работи. Точно тези ограничения определят специалния вид на формулите на Крахт.

В рамките на тази дипломна работа направихме компютърна реализация на алгоритъма на Крахт на програмният език Haskell [12]. В следващите части ще дадем необходимата подготовка, за да може читателят да се запознае с детайлите и проблемите, възникващи около алгоритъма. Накрая ще опишем и самияалгоритъм.

2 Модална логика

Съждителната модална логика предоставя език, в който можем да изразяваме свойства на структури, снабдени с двуместни релации. Пример за такива структури са графите, крайните и безкрайни автомати, частичните наредби и много други. В тези структури логическите формули се интерпретират като едноместни предикати. За да може формулите все пак да

изказват нещо за релациите R_i, модалната логика разширява съждителния език с едноместните оператори:

$$\Box_i A, \Diamond_i A$$

по един за всяка R_i от сигнатурата на структурата. Тази стандартна интерпретация на синтаксиса се нарича семантика на Крипке. По същество тя съответства на фрагмент на монадичната логика от втори ред. Това дава на модалната логика изразителната сила, която я прави интересна. Обичайно, релациите R_i се наричат релации на достижимост и $R_i xy$ указва, че точката y е достижима от точката x (през R_i). Например, в краен автомат R_i може да задава всички преходи с буква i между различните състояния.

В този контекст съжденията $\Box_i A$ и $\Diamond_i A$ интуитивно изразяват в кои точки свойството A е „необходимо вярно“ и в кои точки A то е „възможно вярно“. Един начин да формализираме това е чрез транслация на модалните съждения към логика от втори ред. Съпоставената формула $ST_x A$ има точно една свободна индивидна променлива x, а съждителните променливи в A остават свободни и пробягват всевъзможните едномести предикати от структурата. Във формулата $ST_x A$ модалните оператори $\Box_i$ и $\Diamond_i$ се превъплъщават в *ограничените квантори*:

$$\Box_i A \rightsquigarrow \forall y.R_i xy \to \varphi$$
$$\Diamond_i A \rightsquigarrow \exists y.R_i xy \wedge \varphi$$

което показва, че $\Box_i$ и $\Diamond_i$ смислово са дуални един на друг. Прочит на първата дефиниция би бил: φ е „R_i-необходимо вярно в точката x“, ако е вярно във всяка R_i-достижима от нея точка.

По-надолу ще фиксираме синтаксиса на модалните и предикатни формули, а след това ще опишем формално стандартната транслация към монадична логика от втори ред. Най-сетне ще дадем и семантиката на Крипке.

За по-дълбоко запознаване с модалната логика, читателят може да се консултира например с [3].

2.1 Синтаксис

Както вече споменахме в увода, при формулите индексната променлива i пробягва някое изброимо индексно множество. Модалният език ще бележим с FOR_M, а множеството от съждителните променливи с VAR_M:

$$VAR_M \rightleftharpoons \{P, Q, \dots\}$$
$$FOR_M \rightleftharpoons 0 \mid 1 \mid P \mid A \wedge B \mid A \vee B \mid \neg A \mid \Box_i A \mid \Diamond_i A$$

Предикатният език FOR_P, който ще използваме, се различава от стандартния по това, че съдържа само квантори, ограничаващи променливата

до образа $R_i[x]$:

$$\forall y \triangleright_i x.\varphi \rightleftharpoons \forall y.R_i xy \to \varphi$$
$$\exists y \triangleright_i x.\varphi \rightleftharpoons \exists y.R_i xy \wedge \varphi$$

при задължителното условие $x \neq y$. От това следва, че формулите, с които работим, винаги имат поне по една свободна променлива:

$$VAR_P \rightleftharpoons \{x, y, z, \dots\}$$
$$FOR_P \rightleftharpoons \bot \mid \top \mid R^\alpha xy \mid x = y \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \forall y \triangleright_i x.\varphi \mid \exists y \triangleright_i x.\varphi$$

където α е евентуално празен списък от индекси на релации, а степенуването на релация носи смисъла на композиция:

$$R^{a_1 \dots a_n} \rightleftharpoons R_{a_1} \circ \dots \circ R_{a_n}$$

В частния случай когато α е празен, това е просто равенство. За модалните оператори също ще позволяваме степенуване:

$$\Box^{a_1 \dots a_n} \rightleftharpoons \Box_{a_1} \dots \Box_{a_n}$$
$$\Diamond^{a_1 \dots a_n} \rightleftharpoons \Diamond_{a_1} \dots \Diamond_{a_n}$$

отново позволявайки празен списък.

Ще използваме и формули от втори ред. В тях съждителните променливи P ще играят ролята на едноместни предикатни символи, принадлежността към които се ще записваме с $x \in P$. Формулите от втори ред естествено разширяват синтаксиса на предикатните формули, поради което няма да им даваме индуктивна дефиниция.

По конвенция модални формули ще означаваме с буквите A, B, C, а предикатни и такива от втори ред с φ, ψ, χ. Ще използваме $var(A)$ за множеството от съждителни променливи на A, а $free(\varphi)$ за свободните индивидни променливи на φ.

2.2 Семантика и едноместно съответствие

Ще работим със структури от вида $X = \langle X, \{R_i\}_{i \in I} \rangle$. Под *n–местен предикат* в дадена структура X ще разбираме подмножество на X^n.

Стандартната транслация ST превежда модалната формула A до формулата от втори ред $ST_x A$:

$$ST_x P \rightleftharpoons x \in P$$
$$ST_x(A \wedge B) \rightleftharpoons ST_x A \wedge ST_x B$$
$$ST_x(A \vee B) \rightleftharpoons ST_x A \vee ST_x B$$
$$ST_x(\neg A) \rightleftharpoons \neg ST_x A$$
$$ST_x(\Box_i A) \rightleftharpoons \forall y \triangleright_i x.ST_y A$$
$$ST_x(\Diamond_i A) \rightleftharpoons \exists y \triangleright_i x.ST_y A$$

Във $ST_x A$, единствената свободна индивидна променлива е x. Всяка друга е под областа на действие на някакъв ограничен квантор. Предикатните променливи обаче са винаги свободни, т.е. $ST_x A$ е формула от вида $\varphi(x, \vec{P})$. За да определим едноместния предикат, съответстващ на модалната формула A, затваряме универсално $ST_x A$ относно съждителните ѝ променливи $\vec{P}$:

$$\forall \vec{P}.ST_x A$$

Алгебричен вариант на горната дефиниция е семантиката на Крипке. В нея работим с *оценки* $b \in \mathbb{B}$, задаващи стойности на съждителните променливи из подмножествата на универсума на структурата X:

$$\mathbb{B} \rightleftharpoons VAR_M \to \mathscr{P}(X)$$
$$b \in \mathbb{B}$$

Такива оценки разширяваме индуктивно към всички модални формули, съпоставяйки им отново едноместни предикати $A_b \in \mathscr{P}(X)$:

$$\begin{aligned}
P_b &\rightleftharpoons b(P) \\
(A \wedge B)_b &\rightleftharpoons A_b \wedge B_b \\
(A \vee B)_b &\rightleftharpoons A_b \vee B_B \\
(\neg A)_b &\rightleftharpoons X \setminus A_b \\
(\Box_i A)_b &\rightleftharpoons \{x \mid R_i[x] \subseteq A_b\} \\
(\Diamond_i A)_b &\rightleftharpoons R_i^{-1}[A_b]
\end{aligned}$$

По дефиниция елементът $a \in X$ удовлетворява A тогава и само тогава, когато го удовлетворява за всяка оценка. Така формулата задава предиката:

$$[\![A]\!] \rightleftharpoons \cap\{A_b\}_{b \in \mathbb{B}}$$

По-често обаче ще използваме дуално определения предикат:

$$(\!|A|\!) \rightleftharpoons \cup\{A_b\}_{b \in \mathbb{B}}$$

Очевидно те съответстват на затварянията на стандартната транслация:

$$\begin{aligned}
a \in [\![A]\!] &\iff \forall \vec{P}.ST_a A \\
a \in (\!|A|\!) &\iff \exists \vec{P}.ST_a A
\end{aligned}$$

Нека въведем подобно означение за предикатни формули $\varphi(x)$, притежаващи най-много една свободна променлива:

$$[\![\varphi(x)]\!] \rightleftharpoons \{a \in X \mid X \vDash \varphi(a)\}$$

Разбира се, навсякъде стойностите на $[\![]\!]$ и $(\!|\,|\!)$ зависят неявно от структурата X.

Формула от първи ред $\varphi(x)$ е *съответна* на модална формула A тогава и само тогава, когато за всяка структура X и елемент $a \in X$:

$$X \vDash \varphi(a) \iff X \vDash \forall \vec{P}.ST_a A$$

В алгебричен вид това е равенството:

$$[\![\varphi(x)]\!] = [\![A]\!]$$

Ще въведем и дуалната релация $\rightsquigarrow$, която изисква съществуването на поне една оценка b, за която $a \in A_b$. Дефинираме $\varphi(x) \rightsquigarrow A$ (при същите предпоставки), тогава и само тогава, когато:

$$X \vDash \varphi(a) \iff X \vDash \exists \vec{P}.ST_a A$$

Или в алгебричен вид:

$$[\![\varphi(x)]\!] = \langle\!\langle A \rangle\!\rangle$$

От дефинициите следва, че $\varphi(x)$ е съответна на A точно тогава, когато $\neg\varphi(x) \rightsquigarrow \neg A$. По-нататък за смятането с модални съответсвия ще ни се наложи да разширим тази дефиниция до предикатни формули с повече от една свободна променлива.

2.3 Модални оператори

В семантиката на Крипке, неявно използвахме операторите:

$$\begin{aligned} \Box_i, \Diamond_i : \mathscr{P}(X) &\to \mathscr{P}(X) \\ \Box_i U &\rightleftharpoons \{x \mid R_i[x] \subseteq U\} \\ \Diamond_i U &\rightleftharpoons R_i^{-1}[U] \end{aligned}$$

които заедно с $U \cap V$, $U \cup V$ и $X \setminus U$ са аналог на логическите операции, но върху подмножесва на структурата X. Самите $\Box_i$ и $\Diamond_i$ са дуални един на друг:

$$\begin{aligned} \Box_i U &= X \setminus \Diamond_i (X \setminus U) \\ \Diamond_i U &= X \setminus \Box_i (X \setminus U) \end{aligned}$$

което върху формули изглежда като еквивалентностите:

$$\begin{aligned} \Box_i A &\equiv \neg\Diamond_i \neg A \\ \Diamond_i A &\equiv \neg\Box_i \neg A \end{aligned}$$

и ни позволява да придвижваме отрицанието навътре или навън според необходимостта.

Относно $[\![]\!], (\!)$ имаме следните тъждества:

$$\begin{aligned}
\Box_i [\![A]\!] &= [\![\Box_i A]\!] \\
\Diamond_i (\!(A)\!) &= (\!(\Diamond_i A)\!) \\
[\![A \wedge B]\!] &= [\![A]\!] \cap [\![B]\!] \\
(\!(A \vee B)\!) &= (\!(A)\!) \cup (\!(B)\!)
\end{aligned}$$

Тези за $\Box_i, \Diamond_i$ следват от:

$$\begin{aligned}
\Box_i(\cap\{U_\lambda\}_{\lambda\in\Lambda}) &= \cap\{\Box_i U_\lambda\}_{\lambda\in\Lambda} \\
\Diamond_i(\cup\{U_\lambda\}_{\lambda\in\Lambda}) &= \cup\{\Diamond_i U_\lambda\}_{\lambda\in\Lambda}
\end{aligned}$$

2.4 Специални видове формули

Ще ни потрябват някои специални видове модални формули. От техните свойства зависи коректността на правилата, чрез които ще комбинираме съответни формули.

Множеството от всички оценки на съждителните променливи $\mathbb{B}$ образува пълна булева алгебра, използвайки поточковата наредба:

$$b_1 \leq b_2 \rightleftharpoons \forall P \in VAR_M . b_1(P) \subseteq b_2(P)$$

В този случай инфинимумът $b_{inf} = \wedge\{b_\lambda\}_{\lambda\in\Lambda}$ на произволна фамилия от оценки е:

$$b_{inf}(P) = \cap\{b_\lambda(P)\}_{\lambda\in\Lambda}$$

Според индуктивно дефинираното разширение A_b на оценка $b \in \mathbb{B}$, всяка модална формула можем разглеждаме като изображение между алгебрите $\mathbb{B}$ и $\mathscr{P}(X)$:

$$\begin{aligned}
A_ : \mathbb{B} &\to \mathscr{P}(X) \\
b &\mapsto A_b
\end{aligned}$$

Полезността на различните видове формули идва именно от свойствата на това изображение. Въвеждаме следните класове модални формули:

Позитивни. A е позитивна $\rightleftharpoons$ $A\_$ е монотонно изображение:

$$b_1 \leq b_2 \text{ влече } A_{b_1} \subseteq A_{b_2}$$

Негативни. A е негативна $\rightleftharpoons$ $A\_$ е антимонотонно изображение:

$$b_1 \leq b_2 \text{ влече } A_{b_1} \supseteq A_{b_2}$$

Силно позитивни. За силно позитивните формули ще дадем индуктивна дефиниция: A е с-п $\rightleftharpoons$ A е изградена само от съждителни променливи, 1, $\wedge$ и $\Box_i$.

Една формула е позитивна (негативна) тогава и само тогава, когато всяка променлива е в областта на действие на четен (респ. нечетен) брой отрицания.

Негативните формули ще бележим с N, а силно позитивните с S. Всяка силно позитивна формула е позитивна. Двете свойства, заради които въвеждаме силно позитивни формули е, че те $\rightsquigarrow$ съответсват на предикатната истина $\top$, както и че запазват инфинимумите:

$$\top \rightsquigarrow S$$
$$S_{\wedge\{b_\lambda\}_{\lambda\in\Lambda}} = \cap\{S_{b_\lambda}\}_{\lambda\in\Lambda}$$

на всяка фамилия от оценки $\{b_\lambda\}_{\lambda\in\Lambda}$. Тези свойства лесно се показват с индукция по построението на S.

2.5 Списъци от модални формули

Смятането с модални съответствия изисква да определяме n-местни предикати чрез модални формули. Естествен начин затова е да разширим дефинициите на ST_x, $()_b$, $[\![]\!]$ и $(\!()\!)$ към списъци от модални формули $\vec{A}$:

$$ST_{x_1\ldots x_n}(A_1 \cdots A_n) \rightleftharpoons ST_{x_1} A_1 \wedge \cdots \wedge ST_{x_n} A_n$$
$$(A_1 \cdots A_n)_b \rightleftharpoons (A_1)_b \times \cdots \times (A_n)_b$$

Дефиницията на $[\![]\!]$ и $(\!()\!)$ е на практика същата:

$$[\![\vec{A}]\!] \rightleftharpoons \cap\{(\vec{A})_b\}_{b\in\mathbb{B}}$$
$$(\!(\vec{A})\!) \rightleftharpoons \cup\{(\vec{A})_b\}_{b\in\mathbb{B}}$$

Имаме свойствата:

$$[\![A_1 \cdots A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$
$$(\!(A_1 \cdots A_n)\!) \subseteq (\!(A_1)\!) \times \cdots \times (\!(A_n)\!)$$

С това преминаване към списъци обаче въведохме едно съществено различие между $ST_{\vec{x}}\vec{A}$ и $[\![]\!]/(\!()\!)$. В $\vec{x}$ променливите могат да участват по няколко пъти, поради което затвариянията на $ST_{\vec{x}}\vec{A}$ не са директно еквивалентни на предикатите $[\![\vec{A}]\!]/(\!(\vec{A})\!)$. За синтактични преобразувания над формули варианта с повторения е доста по-удобен, затова по-нататък дефиницията ни за модална съответност ще зависи директно от реда и повторенията на променливите в списъка $\vec{x}$. За да запазим алгебричния вид от едноместния случай, ще използваме формули с асоцииран към тях контекст от индивидни променливи, а в самото съответствие ще имаме предикатни формули заедно с асоциирана към тях субституция.

3 Формули в контекст

В тази секция ще се занимаем с проблема как еднозначно да интерпретираме формулите от първи ред като многоместни предикати, т.е. да дефинираме семантиката $[\![\varphi]\!]$ на формули с повече от една свободна променлива. За да си осигурим това условие, ще ни се наложи към тях да предполагаме и контекст от индивидни променливи. Проблемът с обикновените формули идва от това, че няма канонична връзка между техните променливи и реда на аргументите на предикатите.

3.1 Дефиниция

Контекст ще наричаме списък от индивидни променливи $\vec{x}$ без повторения. Понеже няма повторения, субституцията $\vec{x} \mapsto \vec{y}$ е добре дефинирана и ще я бележим с:

$$|^{\vec{x}}_{\vec{y}}$$

Субституция също ще наричаме и резултата $\varphi|^{\vec{x}}_{\vec{y}}$ от прилагането на субституция към формула.

Фромулите, с променливи измежду контекста $\vec{x}$, обозначаваме с:

$$FOR_{\vec{x}} \rightleftharpoons \{\varphi \mid free(\varphi) \subseteq \vec{x}\}$$

Всяка $\varphi \in FOR_{\vec{x}}$ определя предиката

$$\{\vec{a} \in X \mid X \vDash \varphi|^{\vec{x}}_{\vec{a}}\}$$

който обаче зависи от избора на $\vec{x}$. Естествен начин да премахнем тази зависимост е да използваме формули заедно с еднозначно асоцииран към тях контекст, т.е. двойки $\varphi = (\varphi, \vec{x})$. Тях ще наричаме *формули в контекст* или по-често просто, формули и също ще ги бележим с малки гръцки букви:

$$CON_{\vec{x}} \rightleftharpoons \{\text{тези } \varphi \text{ с контекст } \vec{x}\}$$
$$CON \rightleftharpoons \{\text{формули в контекст}\}$$

т.е. от покритието $\{FOR_{\vec{x}}\}$ на FOR_P се сдобихме с разбиването $\{CON_{\vec{x}}\}$ на CON.

Асоцииранят контекст ни позволява да си мислим за променливите като на аргументи на предикат. За формула $\varphi \in CON_{\vec{x}}$ дефинираме операцията

$$\varphi(\vec{y}) \rightleftharpoons \varphi|^{\vec{x}}_{\vec{y}}$$

Сега можем еднозначно да укажем предиката, който формулата в контекст φ определя в дадена структура X:

$$[\![\varphi]\!] \rightleftharpoons \{\vec{a} \in X \mid X \vDash \varphi(\vec{a})\}$$

Когато използваме $[\![\varphi]\!]$ обикновено няма да упоменаваме структурата.

По-нататък ще използваме означението $!\,\vec{y}$, за списъка, който се получава от $\vec{y}$ като премахнем повторенията на променливи, запазвайки реда на първите им срещания:

$$!\,\vec{y} \rightleftharpoons \text{променливите от } \vec{y}\text{, но без повторения}$$

Множествата $CON_{\vec{x}}$ са затворени относно операциите $\wedge$, $\vee$ и $\neg$. Кванторите пък сменят контекста на $\varphi \in CON_{\vec{x}\cdot y}$:

$$\exists y.\varphi \in CON_{\vec{x}}$$
$$\exists y \triangleright_i z.\varphi \in CON_{!(\vec{x}\cdot z)}$$

За да затворим CON относно операцията $\varphi(\vec{y}) \rightleftharpoons \varphi|^{\vec{x}}_{\vec{y}}$, постулираме:

$$\varphi(\vec{y}) \in CON_{!\vec{y}}$$

Този нов контекст не размества реда на аргументите на $\varphi(\vec{y})$.

3.2 Повторения на променливите

Полезно е да видим в по-директен вид зависимостта на $[\![\varphi(\vec{y})]\!]$ от $\vec{y}$ и $[\![\varphi]\!]$. В следващата дефиниция ще означаваме множеството естествени числа строго по-малки от n със самото число n. Нека $\vec{y} = (y_0 \dots y_{m-1})$ и $n = |!\,\vec{y}|$. От списъка $\vec{y}$ получаваме изображението d:

$$d : m \to n$$
$$d(i) \rightleftharpoons \text{позицията на } y_i \text{ в } !\,\vec{y}$$

Чрез него за произволно множество X се сдобиваме с друго изображение :

$$[\vec{y}] : X^n \to X^m$$
$$(a_0, \dots, a_{n-1}) \mapsto (a_{d(0)}, \dots, a_{d(m-1)})$$

Дефиницията е коректна, понеже d е сюрективно. В теория на типовете такива функции, валидни за всякакви множества, X се наричат полиморфни. В теорята на категориите те съответстват на естествени трансформации.

За пример нека вземем $\vec{y} = \langle xyyxxz\rangle$. Тогава $!\,\vec{y} = \langle xyz\rangle$, изображението $d : 6 \to 3$ е:

$$\langle 0,1,2,3,4,5\rangle \mapsto \langle 0,1,1,0,0,2\rangle$$

а пък $[\vec{y}]$ действа над произволно множество X^3 с:

$$(a_0, a_1, a_2) \mapsto (a_0, a_1, a_1, a_0, a_0, a_2)$$

Понеже $[\vec{y}]$ е инективно, взимането на образи и прообрази преминава през всякакви сечения, обединения и допълнения. Получаваме и търсената връзка:

$$[\![\varphi(\vec{y})]\!] = [\vec{y}]^{-1}\,[\![\varphi]\!]$$

Нека читателят да обърне внимание и на един друг обичаен клас полиморфни функции – тези пермутиращи елементите на списъци с фиксирана дължина. Тях наричаме просто пермутации.

3.3 Маркирани участия

В субституцията $\varphi(\vec{x})$ една променлива може да се среща няколко пъти в списъка $\vec{x}$. По-нататък ще ни трябва да различаваме от коя позиция идва конкретно участие на променливата. Например, ако $\varphi \in CON_{\vec{u}\cdot v\cdot w}$ то във формулата

$$\varphi(\vec{x}, y, y) = \varphi|^{\vec{u}\cdot v\cdot w}_{\vec{x}\cdot y\cdot y}$$

ще искаме да различаваме участията на y според това дали произлизат от v или w. Такива участия ще наричаме *маркирани*. Цялата информация за тях е налична в контекста на φ плюс изображението $[\vec{x}]$. Въвеждаме нов вид формули по подобие на CON, в които всички участията на свободните променливи са маркирани. Тях ще записваме с:

$$\varphi[\vec{x}]$$

По същество една такава формула представя $\varphi(\vec{x})$ и можем да я идентифицираме с двойката $(\varphi, [\vec{x}])$.

При формулите с маркирани участия отново ще искаме да използваме логическите операции $\wedge$, $\vee$, $\neg$, $\exists\triangleright$, $\forall\triangleright$. Дефинирането на кванторните операции е малко по-проблемно, но то е много важно при формализацията на алгоритъма на Крахт. Ще разгледаме само случая $\exists y \triangleright_i z$. Интуитивно искаме да разширим операцията $\exists\triangleright$, така че от $\varphi[\vec{x}\cdot y]$ да получим формула $\varphi'[\vec{x}\cdot \not{y}\cdot z]$ еквивалентна на:

$$\exists y(R_i z y \wedge \varphi[\vec{x}\cdot y])$$

Нека фиксираме контекста на $\varphi \in CON_{\vec{u}\cdot v}$. Позицията на z в добавения конюнктивен член $R_i z y$ ще маркираме с нова променлива w. За да не маркираме инцидентно други позиции, трябва $w \notin \vec{u}\cdot v$, а за да не свържем други променливи освен y, също и $y \notin \vec{u}\cdot v$. При условие че $y \notin \vec{x}$, дефинираме търсената формула φ' и самата операция:

$$\varphi' \rightleftharpoons \exists y(R_i w y \wedge \varphi|^{v}_{y}) \in CON_{\vec{u}\cdot w}$$
$$(\exists y \triangleright_i z)(\varphi[\vec{x}\cdot y]) \rightleftharpoons \varphi'[\vec{x}\cdot z]$$

Ако имахме $y \in \vec{x}$, то във $\varphi'[\vec{x}\cdot z]$ щяхме да маркираме участия на свързаната променлива y, което би било пречка.

4 Смятане с модални съответствия

Ще представим система от правила, които са леко олекотен вид на дадените в [10, глава 5]. Тези преобразувания са основата на нашата алгоритмична имплементация. Ще обсъдим коректност само на логически правила. За по-подробна дискусия нека читателят се обърне към посочената глава, но с предупреждението, че нашето изложение е в по-алгебричен вид, различавайки се леко. Навсякъде по-долу силно позитивни формули ще означаваме с S, а негативни с N.

4.1 Модални съответствия на повече променливи

Правилата ще касаят релацията $\leftrightsquigarrow$, тъй като тя ни е по-удобна в доказателствата. Въпросът за съответствие между формулите $\varphi[x]$ и A се свежда до въпроса $\neg\varphi[x] \leftrightsquigarrow \neg A$. В процеса на извеждането обаче ще трябва да преминем през подформули с повече от една свободна променлива. Затова ще разширим $\leftrightsquigarrow$ до релация между предикатни формули с маркирани участия и списъци от модални формули. При условие че $\vec{x}$ и $\vec{A}$ имат една и съща дължина, полагаме $\varphi[\vec{x}] \leftrightsquigarrow \vec{A}$ тогава и само тогава, когато:

$$X, \imath \vDash \varphi[\vec{x}] \iff X, \imath \vDash \exists \vec{P}.ST_{\vec{x}}\vec{A}$$

за всяка структура X и оценка на индивидните променливи $\imath$. Преминавайки към позиционни аргументи, получаваме алгебричния вид, който ще използваме в доказателствата:

$$\varphi[\vec{x}] \leftrightsquigarrow \vec{A} \rightleftharpoons [\vec{x}]^{-1}[\![\varphi(\vec{x})]\!] = [\vec{x}]^{-1}(\!|\vec{A}|\!)$$

Забележете, че според дефиницията на $(\!|\,|\!)$ всяко едно включване от вида $U \subseteq (\!|\vec{A}|\!)$, за U подмножество на структурата X, предполага съществуването на индексирана фамилия от оценки:

$$f : U \to \mathbb{B}$$

за която е в сила:

$$u \in (\vec{A})_{f(u)}$$

4.2 Начални правила

Началните правила се прилагат в случая на най-простите формули.

Логически константи.

$$(K)\ \top[\vec{x}] \leftrightsquigarrow 1 \qquad \bot[\vec{x}] \leftrightsquigarrow 0$$

Верността директно следва от факта че $[\![\top]\!] = X = (\!|1|\!)$ и $[\![\bot]\!] = \varnothing = (\!|0|\!)$ за всяка структура X.

Атомарни формули.

$$(A)\ \neg R^\alpha x y \leftrightsquigarrow \Box^\alpha P \cdot \neg P \qquad \neg R^\alpha x y \leftrightsquigarrow \Box^\alpha \neg P \cdot P$$

В частния случай, когато списъкът α е празен, правилото се редуцира до:

$$x \neq y \leftrightsquigarrow P \cdot \neg P \qquad x \neq y \leftrightsquigarrow \neg P \cdot P$$

което просто изразява факта, че $x \neq y$ тогава и само тогава, когато x и y са отделими чрез някои множества P и $X \setminus P$. Чрез индукция по дължината на α лесно се доказва цялото (A).

4.3 Структурни правила

Структурните правила променят формата на формулите, но не и тяхното съдържание. На тяхната коректност няма да се спираме. Читателят може да се консултира с [10, глава 5] или [9].

Смяна на знака. Правилото за атомарни формули е симетрично относно смяната $P \leftrightarrow \neg P$. Тя е валидна и в общия случай:

$$(N)\ \frac{\varphi[\vec{x}] \leftrightsquigarrow \vec{A}}{\varphi[\vec{x}] \leftrightsquigarrow \vec{A}|^{P\neg P}_{\neg PP}}$$

Това правило основно служи за обръщане на монотонността на $\vec{A}$.

Отслабване. За променлива y можем, да добавим нова позиция с тривиалното ограничение 1:

$$(W)\ \frac{\varphi \in CON_{\vec{u}} \qquad \varphi[\vec{x}] \leftrightsquigarrow \vec{A}}{\varphi \in CON_{\vec{u}\cdot v} \qquad \varphi[\vec{x}\cdot y] \leftrightsquigarrow \vec{A}\cdot 1}\ v \notin \vec{u}$$

Разместване. Както може да се очаква можем да разместваме реда на аргументите:

$$(P)\ \frac{\varphi \in CON_{\vec{u}} \qquad \varphi[\vec{x}] \leftrightsquigarrow \vec{A}}{\varphi \in CON_{\pi(\vec{u})} \qquad \varphi[\pi(\vec{x})] \leftrightsquigarrow \pi(\vec{A})}\ \pi\text{–пермутация}$$

Субституция. Заместваики всички участия на y със z, просто добавяме ограниченията на y към тези на z:

$$(S)\ \frac{\varphi[\vec{x}] \leftrightsquigarrow \vec{A}}{\varphi[\vec{x}]|^{y}_{z} \leftrightsquigarrow \vec{A}}$$

Сливане. Последното правило ни дава възможността да слеем ограниченията за две позиции в една:

$$(C)\ \frac{\varphi(\vec{u}\cdot v\cdot w) \in CON_{\vec{u}\cdot v\cdot w} \qquad \varphi(\vec{u}\cdot v\cdot w)[\vec{x}\cdot y\cdot y] \leftrightsquigarrow \vec{C}\cdot A\cdot B}{\varphi(\vec{u}\cdot v\cdot v) \in CON_{\vec{x}\cdot v} \qquad \varphi(\vec{u}\cdot v\cdot v)[\vec{x}\cdot y] \leftrightsquigarrow \vec{C}\cdot(A\wedge B)}$$

4.4 Логически правила

Логическите правила се базират на преминаването на $(\!|\ |\!)$ и $[\vec{x}]^{-1}$ през различните логически операции. Понеже $[\vec{x}]$ е инективно, взимането на прообрази преминава свободно през всички от тях. Това далеч не е така за $(\!|\ |\!)$ и в повечето случаи ще са необходими допълнителни условия над модалните формули.

Конюнкция.

$$(\wedge)\ \frac{\varphi[\vec{x}] \leftrightsquigarrow \vec{A} \qquad \psi[\vec{x}] \leftrightsquigarrow \vec{B} \qquad var(\vec{A}) \cap var(\vec{B}) = \varnothing}{(\varphi \wedge \psi)[\vec{x}] \leftrightsquigarrow \vec{A} \wedge \vec{B}\text{–поелементно}}$$

Тук разчитаме на факта, че ако две модални формули A и B нямат общи променливи, то $(\!|A|\!) \cap (\!|B|\!) = (\!|A \wedge B|\!)$, което се пренася поелементно до списъците с равна дължина $\vec{A}$, $\vec{B}$.

$$[\vec{x}]^{-1}[\![\varphi \wedge \psi]\!] = [\vec{x}]^{-1}\left[[\![\varphi]\!] \cap [\![\psi]\!]\right] = [\vec{x}]^{-1}[\![\varphi]\!] \cap [\vec{x}]^{-1}[\![\psi]\!] =$$
$$= [\vec{x}]^{-1}(\!|\vec{A}|\!) \cap [\vec{x}]^{-1}(\!|\vec{B}|\!) = [\vec{x}]^{-1}\left[(\!|\vec{A}|\!) \cap (\!|\vec{B}|\!)\right] = [\vec{x}]^{-1}(\!|\vec{A} \wedge \vec{B}|\!)$$

∎

Дизюнкция.

$$(\vee)\ \frac{\varphi[\vec{x} \cdot y] \leftrightsquigarrow \vec{C} \cdot A \qquad \psi[\vec{x} \cdot y] \leftrightsquigarrow \vec{C} \cdot B}{(\varphi \vee \psi)[\vec{x} \cdot y] \leftrightsquigarrow \vec{C} \cdot (A \vee B)}$$

Проблем при дизюнкцията е, че равенството $(\!|A|\!) \cup (\!|B|\!) = (\!|A \vee B|\!)$ не се разширява поелементно за списъци. Разполагаме само с включването $(\!|\vec{A}|\!) \cup (\!|\vec{B}|\!) \subseteq (\!|\vec{A} \vee \vec{B}|\!)$, понеже елемент на лявата част задължително удовлетворява всичките $\vec{A}$ или всичките $\vec{B}$, докато елемент от дясната може да удовлетворява само част от $\vec{A}$ и част $\vec{B}$, прескачайки между елементите на $\vec{A}$ и елементите на $\vec{B}$. Ако обаче изискаме $\vec{A}$ и $\vec{B}$ да се различават най-много в едно условие, очевидно възстановяваме равенството, откъдето и:

$$[\vec{x}]^{-1}[\![\varphi \vee \psi]\!] = [\vec{x}]^{-1}\left[[\![\varphi]\!] \cup [\![\psi]\!]\right] = [\vec{x}]^{-1}[\![\varphi]\!] \cup [\vec{x}]^{-1}[\![\psi]\!] =$$
$$= [\vec{x}]^{-1}(\!|\vec{C} \cdot A|\!) \cup [\vec{x}]^{-1}(\!|\vec{C} \cdot B|\!) = [\vec{x}]^{-1}\left[(\!|\vec{C} \cdot A|\!) \cup (\!|\vec{C} \cdot B|\!)\right] = [\vec{x}]^{-1}(\!|\vec{C} \cdot (A \vee B)|\!)$$

∎

Дизюнкция на $\vec{S} \cdot A$ списъци.

$$(\vee\text{-SA})\ \frac{\varphi[\vec{x} \cdot y] \leftrightsquigarrow \vec{S}_1 \cdot A \qquad \psi[\vec{x} \cdot y] \leftrightsquigarrow \vec{S}_2 \cdot B}{(\varphi \vee \psi)[\vec{x} \cdot y] \leftrightsquigarrow \vec{S}_1 \wedge \vec{S}_2 \cdot (A \vee B)}$$

Това правило донякъде компенсира недостатъците на $(\vee)$ понеже не изисква списъците $\vec{S}_1$ и $\vec{S}_2$ да съвпадат. Своя страна обаче, поставя ограничението те да бъдат съставени само от силно позитивни формули. По същество правилото е прилагане на (W), $(\wedge)$, $(\vee)$ и свойствата на силно позитивните формули:

$$\top[\vec{x} \cdot y] \leftrightsquigarrow \vec{S}_1 \cdot 1 \qquad\qquad \varphi[\vec{x} \cdot y] \leftrightsquigarrow (\vec{S}_1 \wedge \vec{S}_2) \cdot A$$
$$\top[\vec{x} \cdot y] \leftrightsquigarrow \vec{S}_2 \cdot 1 \qquad\qquad \psi[\vec{x} \cdot y] \leftrightsquigarrow (\vec{S}_1 \wedge \vec{S}_2) \cdot B$$

$$(\varphi \vee \psi)[\vec{x} \cdot y] \leftrightsquigarrow (\vec{S}_1 \wedge \vec{S}_2) \cdot (A \vee B)$$

∎

Ромбче.

$$(\Diamond)\ \frac{y \notin \vec{x} \qquad \varphi.[\vec{x} \cdot y] \leftrightsquigarrow \vec{A} \cdot B}{((\exists y \triangleright_i z)\varphi[\vec{x} \cdot y])[\vec{x} \cdot \not{y} \cdot z] \leftrightsquigarrow \vec{A} \cdot \not{B} \cdot \Diamond_i B}$$

Правилото $(\Diamond)$ е може би най-непретенциозно и неговата проверка също ще пропуснем.

Квадратче.

$$(\Box)\ \frac{y \notin \vec{x} \qquad \varphi[\vec{x} \cdot y] \leftrightsquigarrow \vec{S} \cdot N}{((\forall y \triangleright_i z)\varphi[\vec{x} \cdot y])[\vec{x} \cdot \not{y} \cdot z] \leftrightsquigarrow \vec{S} \cdot \not{N} \cdot \Box_i N}$$

Ако помислим за частния случай, в който списъкът $\vec{S}$ е празен, верността на условието от първи ред:

$$X \vDash \forall y \triangleright_i a.\varphi(y)$$

просто ни казва че:

$$R_i[a] \subseteq [\![\varphi]\!]$$

което комбинирано с предпоставката на правилото дава:

$$R_i[a] \subseteq (\!|N|\!)$$

Затова нека най-напред видим, че за негативни формули N е в сила следната:

Лема 1.

$$R_i[a] \subseteq (\!|N|\!) \iff a \in (\!|\Box_i N|\!)$$

Всъщност лявата посока $(\Leftarrow)$ е валидна за всякакви формули. За да покажем дясната трябва да намерим оценка $b \in \mathbb{B}$, за която:

$$a \in (\Box_i N)_b \iff a \in \Box_i N_b \iff R_i[a] \subseteq N_b$$

Допускането $R_i[a] \subseteq (\!|N|\!)$ ни дава индексирана фамилия:

$$\begin{aligned} f : \; & R_i[a] \to \mathbb{B} \\ & a' \in N_{f(a')} \end{aligned}$$

чрез която определяме b:

$$b \rightleftharpoons \wedge\{ f(a') \}$$

От антимонотонността на N следва

$$N_{f(a')} \subseteq N_b$$

откъдето

$$R_i[a] \subseteq \cup\{ N_{f(a')} \} \subseteq N_b$$

което доказва желаното свойство.

Лема 2. За общото правило към горната лема трябва да добавим възможност за силно позитивни формули $\vec{S}$:

$$\{\vec{a}\} \times R_i[a_n] \subseteq (\!|\vec{S} \cdot N|\!) \iff \vec{a} \cdot a_n \in (\!|\vec{S} \cdot \Box_i N|\!)$$

Условието за търсената оценка b е

$$\begin{aligned} \vec{a} \cdot a_n \in (\vec{S} \cdot \Box_i N)_b \iff \vec{a} \cdot a_n \in \vec{S}_b \times (\Box_i N)_b \iff \\ \iff \vec{a} \cdot a_n \in \vec{S}_b \times \Box_i N_b \iff \{\vec{a}\} \times R_i[a_n] \subseteq \vec{S}_b \times N_b \end{aligned}$$

което се разбива на:

$$\begin{aligned} \vec{a} &\in \vec{S}_b \\ R_i[a] &\subseteq N_b \end{aligned}$$

Оценката b е същата както в предишния случай. Допълнителното условие $\vec{a} \in \vec{S}_b$ се гарантира понеже силно позитивните формули $\vec{S}$ запазват сеченията.

Доказателство на правилото. В него следваме доказателството на лемите. Новото тук е, че трябва да вземем предвид $[\vec{x} \cdot y]^{-1}$. Както и в предишните случаи, лявата посока ($\rightsquigarrow$) е валидна за всякакви модални формули. В другата посока имаме структура X и елементите $\vec{a} \cdot a_n \in X^n$. Верността на условието:

$$X \vDash \forall y \triangleright_i a_n . \varphi[\vec{a} \cdot y]$$

заедно с предпоставките на правилото дават включването:

$$\{\vec{a}\} \times R_i[a_n] \subseteq [\vec{x} \cdot y]^{-1} (\!|\vec{S} \cdot N|\!)$$

Тук идва ролята на допускането $y \notin \vec{x}$, което ни позволява да разцепим $[\vec{x} \cdot y]$ на $[\vec{x}]$ и идентитета $[y]$, за да получим условията за оценката b:

$$\begin{aligned} \vec{a} &\in [\vec{x}]^{-1} \vec{S}_b \\ R_i[a] &\subseteq N_b \end{aligned}$$

откъдето без проблем можем да продължим аналогично на Лема 1 и Лема 2. ∎

Всъщност, в сила е и по-общо правило, което към $\vec{S}$ позволява и негативни формули. В този случай се налага да разделим $[\vec{x} \cdot y]$ на три части, отделяйки новите негативни позиции. Това показва, че аргументите $\vec{x}$ трябва да се състоят от два дизюнктни списъка $\vec{x_+} \cap \vec{x_-} = \varnothing$, като на $\vec{x_+}$ съответстват силно позитивни, а на $\vec{x_-}$ – негативни модални формули. Друг, по-лесен начин да видим това е първо да приложим правилото за сливане (C), така че на всяка променлива да съответства точно по една модална формула. Ако променливите имат смесени участия, тогава тази формула няма да е нито силно позитивна, нито негативна.

4.5 Примери

Ще се спрем на няколко прости примера, демонстриращи употребата на правилата. Целта ни ще е да стигнем до формула от първи ред с точно една свободна променлива, на която съответства точно една модална формула. Всъщност, ще изведем отрицанието на тези формули, понеже правилата се отнасят за дуалната релация $\rightsquigarrow$. При вземането на отрицания, ограничените квантори се превръщат един в друг, също като при обикновените.

Рефлексивност.

1. $? \neg(R_i x x)$
2. $(A)\ \neg R_i x x[x x] \rightsquigarrow \Box_i P \cdot \neg P$
3. $(C)\ \neg R_i x x[x] \rightsquigarrow \Box_i P \wedge \neg P$
4. $R_i x x$ съответства на $\Box_i P \to P$

Симетричност.

1. $? \neg(\forall y. R_i x y \to R_i y x)$
2. $\equiv \exists y \triangleright_i x \neg R_i y x$
3. $(A)\ \neg R_i y x[x y] \rightsquigarrow P \cdot \Box_i \neg P$
4. $(\Diamond)\ \exists y \triangleright_i x. \neg R_i y x[x x] \rightsquigarrow P \cdot \Diamond_i \Box_i \neg P$
5. $(C)\ \exists y \triangleright_i x. \neg R_i y x[x] \rightsquigarrow P \wedge \Diamond_i \Box_i \neg P$
6. $\forall y. R_i x y \to R_i y x$ съответства на $P \to \Box_i \Diamond_i P$.

Транзитивност.

1. $? \neg(\forall y z. R_i x y \wedge R_i y z \to R_i x z)$
2. $\equiv \exists y \triangleright_i x. \exists z \triangleright_i y. \neg R_i x z$

3. $(A)\ \neg R_i xz[xz] \leftrightsquigarrow \Box_i P \cdot \neg P$
4. $(\Diamond)\ \exists z \triangleright_i y.\neg R_i xz[xy] \leftrightsquigarrow \Box_i P \cdot \Diamond_i \neg P$
5. $(\Diamond)\ \exists y \triangleright_i x.\exists z \triangleright_i y.\neg R_i xz[xx] \leftrightsquigarrow \Box_i P \cdot \Diamond_i \Diamond_i \neg P$
6. $(C)\ \exists y \triangleright_i x.\exists z \triangleright_i y.\neg R_i xz[x] \leftrightsquigarrow \Box_i P \wedge \Diamond_i \Diamond_i \neg P$
7. $\forall yz.R_i xz \wedge R_i yz \to R_i xz$ съответства на $\Box_i P \to \Box_i \Box_i P$

Гъстота.

1. $?\ \neg(\forall y.R_i xy \to \exists z.R_i xz \wedge R_i zy)$
2. $\equiv \exists y \triangleright_i x.\forall z \triangleright_i x.\neg R_i zy$
3. $(A)\ \neg R_i zy[zy] \leftrightsquigarrow \Box_i \neg P \cdot P$
4. $(\Box)\ \forall z \triangleright_i x.\neg R_i zy[xy] \leftrightsquigarrow \Box_i \Box_i \neg P \cdot P$
5. $(\Diamond)\ \exists y \triangleright_i x.\forall z \triangleright_i x.\neg R_i zy[xx] \leftrightsquigarrow \Box_i \Box_i \neg P \cdot \Diamond_i P$
6. $(C)\ \exists y \triangleright_i x.\forall z \triangleright_i x.\neg R_i zy[x] \leftrightsquigarrow \Box_i \Box_i \neg P \wedge \Diamond_i P$
7. $\forall y.R_i xy \to \exists z.R_i xz \wedge R_i zy$ съответства на $\Diamond_i P \to \Diamond_i \Diamond_i P$

5 Алгоритъм на Крахт

По дадена входна формула от първи ред $\varphi[x]$, алгоритъмът на Крахт намира изходна модална формула A, такава че $\varphi[x] \leftrightsquigarrow A$. Както вече споменахме, взимайки отрицание на входа и на изхода, решаваме задачата за намиране на модално съответна формула.

Самата процедура може схематично да се даде чрез рекурсивната дефиниция:

$$\begin{aligned} f(\neg R^\alpha xy) &\rightleftharpoons g_R(\alpha, x, y) \\ f(\varphi \wedge \psi) &\rightleftharpoons g_\wedge(f(\varphi), f(\psi)) \\ f(\varphi \vee \psi) &\rightleftharpoons g_\vee(f(\varphi), f(\psi)) \\ f(\forall y \triangleright_i x.\varphi) &\rightleftharpoons g_\forall(i, x, y, f(\varphi)) \\ f(\exists y \triangleright_i x.\varphi) &\rightleftharpoons g_\exists(i, x, y, f(\varphi)) \end{aligned}$$

където подпрограмите g_R, $g_\wedge$, $g_\vee$, $g_\forall$, $g_\exists$, комбинират получените модални $\leftrightsquigarrow$ съответности, следвайки изложените правила (A), $(\wedge)$, $(\vee\text{-sa})$, $(\Box)$, $(\Diamond)$. Непосредствено от вида на рекурсивната схема се забелязва, че f следва структурата на входа, разбивайки проблема рекурсивно на по-прости и после комбинира решенията на тези по-прости проблеми.

Подформулите на входа може да имат повече от една свободна променлива, заради което вече разширихме дефиницията на $\leftrightsquigarrow$ да включва

списъци от модални формули. Правилата за комбиниране на подформули обаче имат някои сериозни ограничения за вида на съответствията, които можем да комбинираме. Тези ограничения определят и вида на входните формули, за които алгоритъмът е коректен. Ще ни трябват следните две дефиниции.

Наследствено екзистенциална променлива ще наричаме променлива, която е свободна във входната формула или свързана от екзистенциален квантор, който не е в областта на действие на универсален квантор. Понятието наследствено универсална променлива е дуално.

Основната променлива $pv(\varphi)$ на конкретна подформула φ е свободната променливата y, свързана от най-вложения квантор $\ldots Qy\triangleright_i x(\ldots \varphi \ldots)\ldots$

Част от ограниченията на правилата ще се пренесат върху входната формула, а други ще удовлетворим с по-нататъшни преобразувания или специфичен избор при прилагане на базовото правило (A). Ето някои наблюдения относно това:

1. Базовото правило (A) се отнася само за негативни литерали. Затова във входната формула всеки литерал, който не е част от ограничен квантор, трябва да бъде негативен.

2. Нямаме правило за отрицание, което означава че с изключение на литералите, други отрицания не може да има.

3. ($\wedge$) комбинира само списъци $\vec{A}$ и $\vec{B}$, за които $var(\vec{A}) \cap var(\vec{B}) = \varnothing$. Затова при всяко прилагане на (A) ще избираме нови съждителни променливи.

4. Нека φ е подформула на началната формула. Ако областта на действие на всеки квантор е минимизирана, то ни се гарантира, че в рамките на φ функцията pv се променя само при въвеждане на нов квантор. Това значи, че φ се разделя на нива според стойността на pv. Например, ако $\psi_1 \vee \psi_2$ е подформула на φ, то $pv(\psi_1 \vee \psi_2) = pv(\psi_1) = pv(\psi_2)$. Това равенство ще наричаме *условие за кохерентност.*

5. Подформулите на входната формула се разделят на два вида, в зависимост от това дали са в областта на действие на универсален квантор, т.е. дали имат свободни променливи, които не са наследствено екзистенциални. Ако съдържат такива променливи ще ги наричаме *(под)формули от тип 1*, а в противен случай *(под)формули от тип 2*.

6. Във формулата $\varphi[\vec{x} \cdot y]$ правилата ($\Diamond$) и ($\Box$) изискват $y \notin \vec{x}$. Това налага, преди прилагането им, да съберем всички участия на y със (C).

7. Прилагането на (C) може да елиминира негативни модални формули от съответните списъци. Тези формули са необходими за прилагането на правилото ($\Box$). Затова ще поискаме във всеки литерал $\neg R^{\alpha} x y$ поне една от променливите да бъде наследствено екзистенциална.

Събирайки тези изисквания, входната формула трябва да е изградена от $\top$, $\bot$, $\wedge$, $\vee$, ограничени квантори и литерали от вида $\neg R^{\alpha} xy$, където поне една от променливите x и y е наследствено екзистенциална. Читателят може да се увери че такива формули са дуалните на Крахтовите, дефинирани във въведението.

Чрез допълнителна обработка можем да минимизираме областта на действие на кванторите, затова без ограничение на общността ще считаме, че това е направено. Нещо повече, формулата ще приведем в дизюнкция $\vee\{\varphi_j\}$, където всяка φ_j е изградена от прилагането на операциите $\wedge$ и $\exists\triangleright$ към литерали или формули $\forall y \triangleright_i x.\psi_j$ от тип 2, със самите ψ_j от тип 1. След това алгоритъмът на Крахт може да бъде разделен на следните основни стъпки:

1. За всеки литерал въвеждаме нови съждителни променливи и прилагаме правилото (A) така, че на основната променлива да съответства негативна формула.

2. Условието за кохерентност е изпълнено и затова можем свободно да комбинираме подформули от тип 1 чрез правилата $(\Box)$, $(\Diamond)$, $(\wedge)$, $(\vee\text{-sa})$. Получават се съответствия от вида $\varphi[\vec{x} \cdot y] \leftrightsquigarrow \vec{S} \cdot N$, където на негативната формула N съответства основната променлива $y = pv(\varphi[\vec{x} \cdot y])$.

3. Стигаме до подформули от тип 2 на различните φ_j от най-външната дизюнкция $\vee\{\varphi_j\}$. За тях са достатъчни правилата (C), $(\Diamond)$ и $(\wedge)$, чиито предпоставки очевидно са удовлетворени.

4. Накрая, понеже всяка φ_j има най-много една свободна променлива, след прилагане на (C) можем да комбинираме всичките съответствия в една финална дизюнкция чрез правилото $(\vee)$.

Остана да разгледаме процедурата по минимизация областта на действие на кванторите. Тя се базира на факта, че ограничените квантори могат да преминават през конюнкции или дизюнкции:

$$\exists y \triangleright_i x.(\varphi \vee \psi) \equiv (\exists x \triangleright_i y.\varphi) \vee (\exists x \triangleright_i y.\psi)$$
$$\forall y \triangleright_i x.(\varphi \wedge \psi) \equiv (\forall x \triangleright_i y.\varphi) \wedge (\forall x \triangleright_i y.\psi)$$

Процедирайки от най-вложените подформули към по-външни, минимизираме подформулата на всеки квантор. За $\exists y \triangleright_i x.\varphi$ първо привеждаме φ в дизюнктивна нормална форма, а после вкарваме квантора навътре. При $\forall\triangleright$ действаме дуално. Поради необходимостта от прилагане на нормални форми, в най-лошият случай алгоритъмът има двойно експоненциална сложност.

6 Бележки по реализацията

Алгоритъма на Крахт реализирахме на езика за програмиране Haskell [12]. Haskell е модерен, статично типизиран, чисто функционален език от високо ниво и с лениво пресмятане (lazy evaluation). Типовата система на Haskell е значително разширение на тази на Hindley-Milner, използвана в езика ML и съответно до доста голяма степен позволява автоматично отгатване на типовете (type inference). Спряхме се на Haskell точно поради достойнствата на типовата му система и най-вече заради параметричния полиморфизъм и алгебричните типове данни. Haskell, както и езиците от фамилията на ML, са традиционно силни в имплементацията на системи за автоматично доказване на теореми, писането на компилатори и въобще работа с формални логически обекти.

Предоставеният интерфейс на нашата реализация е към правия проблем за модално съответсвие, а не към релацията $\leftrightsquigarrow$, въпреки че тя се използва вътрешно. Това значи че във входната формула литералите $R^{\alpha} x y$ са без отрицания и се изисква поне една от променливите да бъде наследствено универсална. При неизпълнението на тези условия възниква една от следните възможни грешки:

1. `OneFreeVariableError` – Указаната формула има повече от една свободна променлива.
2. `InherentlyUniversalVariableError` – В някой от литералите липсва поне една наследствено универсална променлива.
3. `BoundVariableEqualsItsRestrictorError` – Някой ограничен квантор въвежда променлива, която съвпада с ограничителната.

Предобработката на формулата не представлява особен интерес, затова само ще спомена двете основни стъпки:

1. Преименуване на входните променливи, така че всеки квантор да свързва уникална променлива.
2. Минимизиране областта на действие на кванторите

6.1 Организация

Основната част на нашето приложение е библиотека, имплементираща алгоритъма на Крахт. Библиотеката също включва и импровизиран синтаксис за формули от първи ред с ограничени квантори. Парсър за този синтаксис е реализиран с автоматичния парсър-генератор Happy [8].

Самата библиотека е разделена в няколко основни модула:

1. `Fo2m` – Основен публичен интерфейс

2. `Fo2m.FirstOrder` – Формули от първи ред и тяхната предобработка
3. `Fo2m.FirstOrder.Parser` – Парсър за импровизирания синтаксис
4. `Fo2m.Modal` – Модални формули и основни операции с тях
5. `Fo2m.Rules` – Смятане с модални съответности (g_R, $g_\wedge$, …)
6. `Fo2m.NormalForm` – Нормални форми

6.2 Основни типове

След началната обработка на входната формула, алгоритъмът на Крахт е доста праволинеен и е просто рекурсивно обхождане на дървото на формулата. Ключовото при него е че типовете, с които се представят модалните съответствия, се конструират по същия начин като подформулите на входа.

За преставяне на формулите е естествено да използваме дървета. Например, логическите оператори $\wedge$ и $\vee$ приемат като аргумент списък от подформули $\vec{\varphi}$ и дават дърветата $\wedge\vec{\varphi}$ и $\vee\vec{\varphi}$. В частния случай, когато списъкът е празен, това са формулите за истина $\top = \wedge\langle\rangle$ и лъжа $\bot = \vee\langle\rangle$.

Във всяко съответствие $\varphi[\vec{x}] \leftrightsquigarrow \vec{A}$ на практика имаме асоциация на списъка от променливи $\vec{x}$ и списъка от модални формули $\vec{A}$. Тази асоциация представяме чрез списък от двойки $\vec{A} = \langle(x_1, A_1), \ldots, (x_n, A_n)\rangle$. Основното предимство на такова представяне е, че няма нужда да поддържаме списъци с една и съща дължина – конюнкцията се превръща в конкатенация на асоциативни списъци (по правила (W) и $(\wedge)$). Единственото нещо, за което трябва да се погрижим, е при съответствия на формули от тип 1 в асоциативния списък да слагаме негативната формула на първо място. Така можем да я отделим от силно позитивните за необходимите ни операции. Типа на такива списъци ще бележим с *List*.

Нека означим индексното множество за релациите с *Ind*. Правилата за смятане се представят с вече споменатите комбинатори $g_{\ldots}$, опериращи директно над съответните асоциативни списъци. Аргументите на тези комбинатори плътно следват дървовидната дефиниция на формулите от първи ред и имат следните сигнатури:

$$
\begin{aligned}
g_R &: \mathit{Ind}^* \times \mathit{VAR}_P \times \mathit{VAR}_P \to \mathit{List} \\
g_\wedge &: \mathit{List}^* \to \mathit{List} \\
g_\vee &: \mathit{List}^* \to \mathit{List} \\
g_\forall &: \mathit{Ind} \times \mathit{VAR}_P \times \mathit{VAR}_P \times \mathit{List} \to \mathit{List} \\
g_\exists &: \mathit{Ind} \times \mathit{VAR}_P \times \mathit{VAR}_P \times \mathit{List} \to \mathit{List}
\end{aligned}
$$

По този начин, след предобработката на входната формула, алгоритъмът на Крахт на практика е преконструиране на дървото ѝ, използващо тези комбинатори вместо логическите операции от първи ред. Това е възможно, понеже асоциативните списъци от типа *List* съдържат цялата необходима информация за модалното съответствие.

Литература

[1] J.A.F.K. van Benthem. “Correspondence theory”. In: *Handbook of Philosophical Logic*. Ed. by D.M. Gabbay and F. Guenthner. Vol. 2. Dordrecht: Reide, 1984, pp. 167–247.

[2] J.F.A.K. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, 1983.

[3] P. Blackburn, M. De Rijke, and Y. Venema. *Modal logic*. Cambridge Tracts in Theoretical Computer Science 53. Cambridge University Press, 2001.

[4] A. Chagrov and M. Zakharyaschev. *Modal Logic*. Oxford: Clarendon Press, 1997.

[5] L. Chagrova. “An undecidable problem in correspondence theory”. In: *J. Symbolic Logic*. Vol. 56. 1991, pp. 1261–1272.

[6] L. Chagrova and A. Chagrov. “The Truth About Algorithmic Problems in Correspondence Theory”. In: *Advances in Modal Logic*. Vol. 6. 2006.

[7] W. Conradie, V. Goranko, and D. Vakarelov. “Algorithmic Correspondence and Completeness in Modal Logic, I. The Core Algoritm SQEMA”. In: *Logical Methods in Computer Science*. Vol. 2. 2006, pp. 1–26.

[8] *Happy: The Parser Generator for Haskell*. URL: `www.haskell.org/happy`.

[9] M. Kracht. “How completeness and correspondence theory got married”. In: *Diamonds and Defaults: Studies in Pure and Applied Intensional Logic*. Ed. by M. de Rijke. Springer, 1993, pp. 175–214.

[10] M. Kracht. *Tools and techniques in modal logic*. Studies in logic and the foundations of mathematics. Elsevier, 1999.

[11] H. Sahlqvist. “Completeness and correspondence in the first and second-order semantics for modal logic”. In: *Proc. Of the third Scandinavian Logic Symposium, Uppsala 1973*. Ed. by S. Kanger. Amsterdam: North-Holland, 1975, pp. 110–143.

[12] *The Haskell Programming Language*. URL: `www.haskell.org`.

Използвани означения

A,B,C	–	модални формули
P,Q	–	съждителни променливи
S	–	силно позитивни формули
N	–	негативни формули
$[\![A]\!]$	–	предикат, определен от A
$(\!(A)\!)$	–	предикат, ко-определен от A
φ,ψ,χ	–	предикатни формули, формули в контекст
x,y,z	–	индивидни променливи или елементи на структури
i,j,k	–	индекси
m,n	–	естествени числа, множества $\{0,\dots,m-1\}$
α,β	–	редици от индекси
$\vec{A},\vec{P},\vec{\varphi},\vec{x},$	–	списъци от съответнитеи видове ($\vec{A}$ и A нямат общо)
$\vec{x}\cdot y$, $\vec{x}\cdot\vec{y}$	–	конкатенация
$\lvert\vec{x}\rvert$	–	дължина на списъка
R	–	релации
$R^{i_1\dots i_n}$	–	$R_{i_1}\circ\dots\circ R_{i_n}$
$\Box^{i_1\dots i_n}$	–	$\Box_{i_1}\dots\Box_{i_n}$
$\Diamond^{i_1\dots i_n}$	–	$\Diamond_{i_1}\dots\Diamond_{i_n}$
$var(A)$	–	променливите на A
$free(\varphi)$	–	свободните променливи на φ
$\vec{x}\subseteq V$	–	понякога на редиците ще гледаме като на множества
$\varphi\lvert_{\vec{y}}^{\vec{x}}$	–	субституция $\vec{x}\mapsto\vec{y}$ ако в $\vec{x}$ няма повторения
$\varphi(\vec{y})$	–	субституция $\vec{x}\mapsto\vec{y}$ в контекста $\vec{x}$ на φ
$!\,\vec{y}$	–	списък от първите срещания в $\vec{y}$
$[\vec{y}]$	–	специалното изображение, породено от $\vec{y}$ [1]
$\varphi[\vec{y}]$	–	формулата $\varphi(\vec{y})$ с маркирани участия на променливите

[1] $\vec{y}$ поражда изображение $d:\ \lvert\vec{y}\rvert\to\lvert!\,\vec{y}\rvert$, слепващо повторенията в $\vec{y}$. Това изображение може да се „повдигне“ до включване $X^{\lvert!\vec{y}\rvert}\to X^{\lvert\vec{y}\rvert}$ за произволно множество X.

Азбучен указател

A Текст на реализацията

Fo2m/Common.hs

```
module Fo2m.Common
    ( Rel )
where

type Rel = String
```

Fo2m/FirstOrder/Parser.y

```
{
module Fo2m.FirstOrder.Parser
  ( parse )
where

import Data.Char
import Control.Monad
import qualified Fo2m.FirstOrder as Fo
}

%name parser
%tokentype { Token }
%error { parseError }

%monad { Maybe }

%token
    '(' { Pun '(' }
    ')' { Pun ')' }
    '[' { Pun '[' }
    ']' { Pun ']' }
    '&' { Pun '&' }
    '|' { Pun '|' }
    '/' { Pun '/' }
    '?' { Pun '?' }
    '=' { Pun '=' }
    '#' { Pun '#' }
    ':' { Pun ':' }
    '@' { Pun '@' }
    '!' { Pun '!' }
    sym { Sym $$ }
%%

Exp : Dis                       { $1 }

Dis : Dis '|' Con               { dsj [$1,$3] }
    | Con                       { $1 }

Con : Con '&' Qnt               { cnj [$1,$3] }
    | Qnt                       { $1 }

Qnt : '[' Var '?' Var ']' Qnt           { exi $2 $4 "" $6 }
    | '[' Var '?' Var ':' Rel ']' Qnt   { exi $2 $4 $6 $8 }
    | '[' Var '/' Var ']' Qnt           { uni $2 $4 "" $6 }
    | '[' Var '/' Var ':' Rel ']' Qnt   { uni $2 $4 $6 $8 }
    | Lit                       { $1 }

Lit : Var '=' Var               { lit $1 $3 [] }
    | Var '#' Var               { lit $1 $3 [""] }
```

```
    | Var '#' Var ':' Lst       { lit $1 $3 (reverse $5) }
    | '@'                       { nil }
    | '!'                       { one }
    | '(' Exp ')'               { $2 }

Var : sym                       { $1 }

Rel : sym                       { $1 }

Lst : {- empty -}               { [] }
    | Lst Rel                   { $2:$1 }

{
-- Lexer
data Token = Pun !Char
           | Sym !String
             deriving (Eq, Show)

lexer [] = return []
lexer (c:cs) | isSpace c = lexer cs
lexer ('(':cs) = lexer cs >>= return . (Pun '(' :)
lexer (')':cs) = lexer cs >>= return . (Pun ')' :)
lexer ('[':cs) = lexer cs >>= return . (Pun '[' :)
lexer (']':cs) = lexer cs >>= return . (Pun ']' :)
lexer ('&':cs) = lexer cs >>= return . (Pun '&' :)
lexer ('|':cs) = lexer cs >>= return . (Pun '|' :)
lexer ('/':cs) = lexer cs >>= return . (Pun '/' :)
lexer ('?':cs) = lexer cs >>= return . (Pun '?' :)
lexer ('=':cs) = lexer cs >>= return . (Pun '=' :)
lexer ('#':cs) = lexer cs >>= return . (Pun '#' :)
lexer (':':cs) = lexer cs >>= return . (Pun ':' :)
lexer ('@':cs) = lexer cs >>= return . (Pun '@' :)
lexer ('!':cs) = lexer cs >>= return . (Pun '!' :)
lexer cs | length sym > 0 = lexer cs' >>= return . (Sym sym :)
         | otherwise      = mzero
    where (sym, cs') = span isAlphaNum cs

parseError _ = mzero

-- Constructors
lit u v r = Fo.Atom () u v r (-1)
nil = dsj []
one = cnj []
dsj = Fo.Junc () False
cnj = Fo.Junc () True
exi = Fo.Bind () False
uni = Fo.Bind () True

-- Parser
parse :: String -> Maybe (Fo.Formula () String)
parse = parser <=< lexer
}
```

Fo2m.hs

```
{-# LANGUAGE FlexibleContexts #-}

module Fo2m
    ( translate )
where
```

```
import Control.Monad
import Control.Monad.Error
import qualified Fo2m.FirstOrder as Fo
import qualified Fo2m.Modal as Mo
import qualified Fo2m.Rules as Ru
import qualified Fo2m.FirstOrder.Negation as FoNeg
import qualified Fo2m.Modal.Negation as MoNeg

stage3 :: [[Fo.Formula Int Int]] -> Mo.Formula
stage3 = extract . Ru.t2Junc 0 False . map (Ru.t2Junc 0 True . map stage2)
    where extract [x] = snd x
          extract  _  = error "bug: stage3 incoherence"

stage2 :: Fo.Formula Int Int -> Ru.Type2 Int
stage2 (Fo.Atom p u v r n)       = Ru.t2Atom p u v r n
stage2 (Fo.Junc p b xs)          = Ru.t2Junc p b (map stage2 xs)
stage2 (Fo.Bind p False u v r x) = Ru.t2Bind p False u v r (stage2 x)
stage2 (Fo.Bind p True u v r x)  = Ru.ttUniv p u v r (stage1 x)

stage1 :: Fo.Formula Int Int -> Ru.Type1 Int
stage1 (Fo.Atom p u v r n)   = Ru.t1Atom p u v r n
stage1 (Fo.Junc p b xs)      = Ru.t1Junc p b (map stage1 xs)
stage1 (Fo.Bind p b u v r x) = Ru.t1Bind p b u v r (stage1 x)

translate :: (Ord u, MonadError Fo.Reason m) => Fo.Formula () u -> m Mo.Formula
translate = (liftM $ MoNeg.emerge . MoNeg.deep . stage3)
          . Fo.preprocess False . FoNeg.deep
```

Fo2m/FirstOrder.hs

```
{-# LANGUAGE FlexibleContexts #-}

module Fo2m.FirstOrder
    ( Reason, Formula(..), atom, junc, bind, preprocess, flatten )
where

import Data.Maybe (fromJust)
import Data.List (group, sort, partition)
import Control.Monad (liftM)
import Control.Monad.State
import Control.Monad.Error
import qualified Fo2m.NormalForm as NormalForm
import Fo2m.Common (Rel)


data Reason = OneFreeVariableError
            | InherentlyUniversalVariableError
            | BoundVariableEqualsItsRestrictorError
              deriving (Show, Eq, Ord)

data Formula p v = Atom p v v [Rel] Int      -- TODO parameterize by
                 | Junc p Bool [Formula p v] -- this last Int
                 | Bind p Bool v v Rel (Formula p v)
                   deriving (Show, Eq, Ord)

instance Error Reason where
    noMsg = undefined

instance NormalForm.View (Formula p v) where
    view x@Atom {} = NormalForm.Atom x
    view x@Bind {} = NormalForm.Atom x
```

```
    view x@(Junc _ b xs) = NormalForm.Junc b xs


alloc :: (Enum a, MonadState a m) => m a
alloc = do n <- get
           modify succ
           return n

uniq :: Ord a => [a] -> [a]
uniq = map head . group . sort

atom :: Eq v => p -> v -> v -> [Rel] -> Int -> Formula p v
atom p u v [] n | u == v = Junc p False []
atom p u v r n = Atom p u v r n

junc :: (Ord p, Ord v) => p -> Bool -> [Formula p v] -> Formula p v
junc _ _ [x] = x
junc p b xs  = Junc p b (uniq $ concatMap unpack xs)
    where unpack (Junc _ c xs) | c == b = xs -- NOTE here we do not
          unpack x = [x]                     -- unpack singletons

bind :: p -> Bool -> v -> v -> Rel -> Formula p v -> Formula p v
bind p b _ _ _ (Junc _ c []) | c == b = Junc p c []
bind p b u v r x = Bind p b u v r x

occurs :: Eq v => v -> Formula p v -> Bool
occurs w (Atom _ u v _ _) = w == u || w == v
occurs w (Junc _ _ xs) = any (occurs w) xs
occurs w (Bind _ _ u v r x) = w == u || (w /= v && occurs w x)

free :: Ord v => Formula p v -> [v]
free = uniq . go []
    where go bv (Atom _ u v _ _) = [ u | u `notElem` bv ] ++ [ v | v `notElem` bv ]
          go bv (Junc _ _ xs) = concatMap (go bv) xs
          go bv (Bind _ _ u v _ x) = [ u | u `notElem` bv ] ++ go (v:bv) x

clean :: (Ord u, Enum v, MonadError Reason m) => Formula p u -> m (Formula p v)
clean fml =
    case free fml of
      []  -> return (go [] fml `evalState` toEnum 1)
      [u] -> return (go [(u,toEnum 0)] fml `evalState` toEnum 1)
      _   -> throwError OneFreeVariableError
    where go kv (Atom p u v r n) = return (Atom p (remap u kv) (remap v kv) r n)
          go kv (Junc p b xs) = liftM (Junc p b) (mapM (go kv) xs)
          go kv (Bind p b u v r x) = do n <- alloc
                                        y <- go ((v,n):kv) x
                                        return (Bind p b (remap u kv) n r y)
          remap v kv = fromJust $ lookup v kv

boundCheck :: (Eq v, MonadError Reason m) => Formula p v -> m (Formula p v)
boundCheck fml = if ok fml
                 then return fml
                 else throwError BoundVariableEqualsItsRestrictorError
    where ok (Atom _ _ _ _ _) = True
          ok (Junc _ _ xs) = all ok xs
          ok (Bind _ _ u v _ x) = u /= v && ok x

krachtCheck :: (Eq v, MonadError Reason m) => Formula p v -> m (Formula p v)
krachtCheck fml = if ok fml
              then return fml
              else throwError InherentlyUniversalVariableError
    where ok (Atom _ _ _ _ _) = True
```

```
          ok (Junc _ _ xs) = all ok xs
          ok (Bind _ False _ _ _ x) = ok x
          ok (Bind _ True _ v _ x) = inScope [v] x
          inScope vs (Atom _ u v _ _) = u `notElem` vs || v `notElem` vs
          inScope vs (Junc _ _ xs) = all (inScope vs) xs
          inScope vs (Bind _ _ u v _ x) = inScope (v:vs) x

numerate :: [[Formula p v]] -> [[Formula p v]]
numerate xss = mapM (mapM go) xss `evalState` 0
    where go (Atom p u v r _) = liftM (Atom p u v r) alloc
          go (Junc p b xs) = liftM (Junc p b) (mapM go xs)
          go (Bind p b u v r x) = liftM (Bind p b u v r) (go x)

annotate :: (Ord v, Enum v, MonadError Reason m) => Formula () v -> m (Formula v v)
annotate fml =
    case free fml of
      []  -> return (go (toEnum 0) fml)
      [u] -> return (go u fml)
      _   -> throwError OneFreeVariableError
    where go p (Atom _ u v r n) = Atom p u v r n
          go p (Junc _ b xs) = Junc p b (map (go p) xs)
          go p (Bind _ b u v r x) = Bind p b u v r (go v x)

distribute :: Ord v => Bool -> v -> v -> Rel -> [[Formula () v]] -> Formula () v
distribute b u v r xss = junc () b (map inner xss)
    where inner xs = let (as, bs) = partition (occurs v) xs
                     in junc () (not b) ((bind () b u v r $ junc () (not b) as) : bs)

miniscope :: Ord v => Formula () v -> Formula () v
miniscope (Atom _ u v r n) = atom () u v r n
miniscope (Junc _ b xs) = junc () b (map miniscope xs)
miniscope (Bind _ b u v r x) = distribute b u v r $ NormalForm.convert b $ miniscope x

preprocess :: (Ord u, Ord v, Enum v, MonadError Reason m)
              => Bool -> Formula () u -> m [[Formula v v]]
preprocess b = liftM (numerate . NormalForm.convert b)
             . annotate <=< liftM miniscope
             . krachtCheck <=< clean <=< boundCheck

flatten :: Ord v => Formula () v -> Formula () v
flatten (Junc _ b xs) = junc () b (map flatten xs)
flatten (Bind _ b u v r x) = bind () b u v r (flatten x)
flatten x = x
```

Fo2m/FirstOrder/Negation.hs

```
module Fo2m.FirstOrder.Negation
where

import Fo2m.FirstOrder

deep x@Atom {} = x
deep (Junc p b xs) = Junc p (not b) (map deep xs)
deep (Bind p b u v r x) = Bind p (not b) u v r (deep x)
```

Fo2m/Modal.hs

```
module Fo2m.Modal
    ( Formula(..), Kin(..), subkin, notSubkin, kin
    , atom, nega, junc, bind )
where
```

```
import Data.List (group, sort)
import Fo2m.Common (Rel)


data Kin = Spc | Spo | Neg | Amb
           deriving (Show, Eq, Ord)

data Formula = Nega Kin Formula
             | Atom Kin Int
             | Junc Kin Bool [Formula]
             | Bind Kin Bool [Rel] Formula
               deriving (Show, Eq, Ord)

subkin k Amb = True
subkin Amb k = k == Amb
subkin Spc k = k `elem` [Spc,Spo,Neg]
subkin Spo k = k `elem` [Spo]
subkin Neg k = k `elem` [Neg]

notSubkin j k = not $ subkin j k

lub j k | j `subkin` k = k
        | k `subkin` j = j
        | otherwise    = Amb

neg Spc = Neg
neg Spo = Neg
neg Neg = Amb
neg Amb = Amb

lim False Spc = Neg
lim True  Spc = Spc
lim False Spo = Amb
lim True  Spo = Spo
lim False Neg = Neg
lim True  Neg = Neg
lim False Amb = Amb
lim True  Amb = Amb

bindLim = lim

juncLim False [] = Neg
juncLim True  [] = Spc
juncLim b     xs = foldr1 f xs
    where f j k = lim b (lub j k)


kin (Atom k _) = k
kin (Nega k _) = k
kin (Junc k _ _) = k
kin (Bind k _ _ _) = k

atom n = Atom Spo n

nega (Junc _ b []) = junc (not b) []
nega x = Nega (neg $ kin x) x

junc _ [x] = x
junc b xs  = Junc (juncLim b (map kin xs)) b (uniq $ concatMap unpack  xs)
    where unpack (Junc _ c xs) | c == b = xs
          unpack x = [x]
```

```
bind b [] x = x
bind b _ (Junc _ c []) | c == b = junc b []
bind b r x = Bind (bindLim b $ kin x) b r x

uniq :: Ord a => [a] -> [a]
uniq = map head . group . sort
```

Fo2m/Modal/Negation.hs

```
module Fo2m.Modal.Negation
    ( deep, emerge )
where

import Data.List (partition)
import Fo2m.Modal

deep x@Atom {} = nega x
deep (Nega _ x) = x
deep (Junc _ b xs) = junc (not b) $ map deep xs
deep (Bind _ b r x) = bind (not b) r $ deep x

emerge x@Atom {} =
    x
emerge (Nega _ x) =
    case emerge x of
      Nega _ y -> y
      z -> nega z
emerge (Junc _ b xs) =
    separate b $ map emerge xs
emerge (Bind _ b r x) =
    case emerge x of
      Nega _ y -> nega $ bind (not b) r y
      z -> bind b r z

separate b xs = junc b ((nega $ junc (not b) (map erase ns)) : as)
    where (ns, as) = partition negated xs
          negated (Nega _ _) = True
          negated _ = False
          erase (Nega _ x) = x
          erase x = x
```

Fo2m/Rules.hs

```
module Fo2m.Rules
    ( Tuple, Type1, Type2, toType2
    , t1Atom, t1Junc, t1Bind
    , t2Atom, t2Junc, t2Bind, ttUniv )
where

import Data.List (partition, intercalate, groupBy)
import Data.Function (on)
import Fo2m.Common (Rel)
import qualified Fo2m.Modal as Modal
import Fo2m.Modal (Kin(Spo,Neg), subkin)


type Tuple v = (v, Modal.Formula)

type Type1 v = (Tuple v, [Tuple v])
```

```
type Type2 v = [Tuple v]


tvar :: Tuple v -> v
tvar = fst

tkin :: Tuple v -> Modal.Kin
tkin = Modal.kin . snd

posNeg :: v -> v -> [Rel] -> Int -> Type1 v
posNeg u v r n = ((v, Modal.nega $ Modal.atom n),
                  [(u, Modal.bind True r $ Modal.atom n)])

negPos :: v -> v -> [Rel] -> Int -> Type1 v
negPos u v r n = ((u, Modal.bind True r $ Modal.nega $ Modal.atom n),
                  [(v, Modal.atom n)])

t1Atom :: Eq v => v -> v -> v -> [Rel] -> Int -> Type1 v
t1Atom p u v
    | p == v = posNeg u v
    | p == u = negPos u v
    | otherwise = error "bug: type-1-atom incoherence"

t1Junc :: Eq v => v -> Bool -> [Type1 v] -> Type1 v
t1Junc p b xs
    | any (/=p) $ map (tvar . fst) xs = error "bug: type-1-junc incoherence"
    | otherwise = ((p, Modal.junc b (map snd as)), concat bs)
    where (as, bs) = unzip (map (preJunc b) xs)

t1Bind :: Eq v => v -> Bool -> v -> v -> Rel -> Type1 v -> Type1 v
t1Bind p b u v r ((w, x), xs)
    | p /= u = error "bug: type-1-bind incoherence (1) "
    | v /= w = error "bug: type-1-bind incoherence (2) "
    | otherwise = preBind b ((w, x), xs) ((u, Modal.bind b [r] x), xs)

contract :: Eq v => v -> [Tuple v] -> (Tuple v, [Tuple v])
contract p xs = ((p, Modal.junc True (map snd as)), bs)
    where (as, bs) = partition ((==p) . tvar) xs

toType2 :: (Tuple v, [Tuple v]) -> Type2 v
toType2 (x, xs) = x:xs

t2Atom :: Eq v => v -> v -> v -> [Rel] -> Int -> Type2 v
t2Atom p u v r n = toType2 (t1Atom p u v r n)

t2Junc :: Eq v => v -> Bool -> [Type2 v] -> Type2 v
t2Junc _ True xs = concat xs
t2Junc p False xs = [(p, Modal.junc False (map contract' xs))]
    where contract' x =
              case contract p x of
                (a, []) -> snd a
                (_, _) -> error "bug: type-2-disj incoherence"

t2Bind :: Eq v => v -> Bool -> v -> v -> Rel -> Type2 v -> Type2 v
t2Bind _ True _ _ _ _ = error "bug: type-2-univ inapplicable"
t2Bind p False u v r x
    -- p /= u = error "bug: tupe-2-exis incoherence"
    -- is it correct to assume this? TODO
    | otherwise = (u, Modal.bind False [r] (snd a)) : as
    where (a, as) = contract v x

ttUniv :: Eq v => v -> v -> v -> Rel -> Type1 v -> Type2 v
```

```
ttUniv _ u v r x = toType2 $ t1Bind u True u v r x

-- Kin preconditions

split :: Eq v => Type2 v -> [Type2 v]
split = groupBy ((==) `on` tvar)

allAre k xs = all ((`subkin` k) . tkin) xs

spone :: Eq v => Type2 v -> Bool
spone = all (either Spo Neg) . split
    where either k j xs = allAre k xs || allAre j xs

preBind :: Eq v => Bool -> Type1 v -> a -> a
preBind False _ y = y
preBind True (x,xs) y
    | tkin x `subkin` Neg && spone xs = y
    | otherwise = error "bug: box rule kin"

preJunc :: Eq v => Bool -> Type1 v -> Type1 v
preJunc True x = x
preJunc False (x,xs)
    | allAre Spo xs = (x,xs)
    | otherwise = error "bug: disj rule kin"
```

Fo2m/NormalForm.hs

```
{-# LANGUAGE NoMonomorphismRestriction, ViewPatterns #-}

module Fo2m.NormalForm
    ( ViewTyp(..), View, view, convert )
where

import Data.List (group, sort)


data ViewTyp a = Atom a
               | Junc Bool [a]

class View a where
    view :: a -> ViewTyp a


convert :: (Ord a, View a) => Bool -> a -> [[a]]
convert = go
    where go _   (view -> Atom x) = [[x]]
          go b x@(view -> Junc c xs)
              | b == c = minimal $ concatMap (go b) xs
              | b /= c = minimal $ sequence $ go (not b) x

minimal :: Ord a => [[a]] -> [[a]]
minimal = foldr extend [] . squash
    where extend x ms = if any (`subset` x) ms
                        then ms
                        else x : filter (not . subset x) ms

squash = uniq . map uniq
    where uniq = map head . group . sort

subset []    _  = True
subset (_:_) [] = False
```

```
subset (x:xs) (y:ys) = case compare x y of
                         LT -> False
                         GT -> subset (x:xs) ys
                         EQ -> subset xs ys
```

Main.hs

```
#!/usr/bin/env runhaskell

{-# LANGUAGE ScopedTypeVariables #-}

import System.IO (isEOF)
import Data.List (isPrefixOf)
import Data.Char (isSpace)
import Control.Monad

import Fo2m
import Fo2m.Util ( splitOnce )
import Fo2m.FirstOrder (Formula, flatten, preprocess)
import Fo2m.FirstOrder.Parser
import Fo2m.Html

endMarker line = "END" `isPrefixOf` line

uncomment line = (dropWhile isSpace s, dropWhile isSpace c)
    where s:c:_ = (splitOnce "--" line) ++ [""]

answer s =
    case parse s of
      Just a ->
          do putStr "<i>"
             putStr $ toHtml withIndexedVariables (flatten a)
             putStr "</i><br/>"
             case translate a of
               Right b ->
                   do putStr "<i>"
                      putStr $ toHtml (withVariablePrefix "P") b
                      putStr "</i>"
               Left e ->
                   do putStr "<b>Error: </b>"
                      putStr $ show e
      Nothing ->
          do putStr "<b>Syntax error: </b><pre>"
             putStr s
             putStr "</pre>"

handle line =
    do if null (s ++ c)
         then return ()
         else do
       putStr "<p>"
       do if null s
            then return ()
            else answer s
       do if null c
            then return ()
            else putStr ("&mdash;" ++ c)
       putStrLn "</p>"
    where (s, c) = uncomment line
```

```
mainLoop =
    do eof <- isEOF
       if eof
         then return ()
         else do
       line <- getLine
       if endMarker line
         then return ()
         else do
       handle line
       mainLoop

main = do putStrLn "<html><body>"
          mainLoop
          putStrLn "</body></html>"
```

Fo2m/Util.hs

```
module Fo2m.Util
    ( maybeSplit, splitOnce )
where

import Data.Maybe (listToMaybe, maybeToList)
import Data.List (inits, tails, stripPrefix)

strip :: Eq a => [a] -> [a] -> [[a]]
strip x y = maybeToList $ stripPrefix x y

maybeSplit :: Eq a => [a] -> [a] -> Maybe ([a], [a])
maybeSplit p s = listToMaybe
  [ (a, c) | (a, b) <- zip (inits s) (tails s), c <- strip p b ]

splitOnce :: Eq a => [a] -> [a] -> [[a]]
splitOnce p s = case maybeSplit p s of
                  Just (a, c) -> [a, c]
                  Nothing -> [s]
```

Fo2m/Html.hs

```
{-# LANGUAGE TypeFamilies #-}

module Fo2m.Html
    ( Render, Variable, toHtml
    , withIndexedVariables, withVariablePrefix )
where

import Data.List (intercalate)
import Data.Char (isNumber)
import Fo2m.FirstOrder as Fo
import Fo2m.Modal as Mo


class Render a where
    type Variable a
    toHtml :: (Variable a -> String) -> a -> String


withIndexedVariables x = reverse t ++ sub (reverse h)
    where (h, t) = span isNumber $ reverse x

withVariablePrefix s n = s ++ sub (show n)
```

```
paren s = "(" ++ s ++ ")"

sub "" = ""
sub s = "<sub>" ++ s ++ "</sub>"

indices xs = sub $ intercalate " " xs

foRecu var x@(Fo.Junc _ _ (_:_)) = paren $ toHtml var x
foRecu var x = toHtml var x

moRecu var x@(Mo.Junc _ _ (_:_)) = paren $ toHtml var x
moRecu var x = toHtml var x


instance Render (Fo.Formula p v) where
    type Variable (Fo.Formula p v) = v
    toHtml var (Fo.Atom _ u v [] _) =
        (var u) ++ "=" ++ (var v)
    toHtml var (Fo.Atom _ u v r _) =
        (var u) ++ "R" ++ indices r ++ (var v)
    toHtml _ (Fo.Junc _ False []) =
        "&perp;"
    toHtml _ (Fo.Junc _ True []) =
        "T"
    toHtml var (Fo.Junc _ _ [x]) =
        toHtml var x
    toHtml var (Fo.Junc _ b xs) =
        intercalate sign $ map (foRecu var) xs
        where sign | b == False = " &or; "
                   | b == True  = " &and; "
    toHtml var (Fo.Bind _ b u v r x) =
        sign ++ (var v) ++ "(" ++ (var u) ++ "R" ++ indices [r] ++ (var v)
                 ++ cosign ++ foRecu var x ++ ")"
        where sign | b == False = "&exist;"
                   | b == True  = "&forall;"
              cosign | b == False = " &and; "
                     | b == True  = " &rarr; "

instance Render Mo.Formula where
    type Variable Mo.Formula = Int
    toHtml var (Mo.Atom _ n) = var n
    toHtml var (Mo.Nega _ x) = "&not;" ++ moRecu var x
    toHtml _ (Mo.Junc _ False []) =
        "<b>0</b>"
    toHtml _ (Mo.Junc _ True []) =
        "<b>1</b>"
    toHtml var (Mo.Junc _ _ [x]) =
        toHtml var x
    toHtml var (Mo.Junc _ b xs) = intercalate sign $ map (moRecu var) xs
        where sign | b == False = " &or; "
                   | b == True  = " &and; "
    toHtml var (Mo.Bind _ _ [] x) = toHtml var x
    toHtml var (Mo.Bind _ b r x) = sign ++ indices r ++ moRecu var x
        where sign | b == False = "&loz;"
                   | b == True = "&#9633;"
```