УДК 510.62:004.82

А. А. Малых, А. В. Манцивода, В. С. Ульянов

ЛОГИЧЕСКИЕ АРХИТЕКТУРЫ И ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

Дескриптивные логики имеют большие возможности для обработки данных и знаний в мировой информационной среде. Semantic Web (SW) преобразует этот потенциал в гармоничный подход, ориентированный на модернизацию всемирной паутины. К сожалению, ряд проблем затрудняет продвижение идей SW в интернете. В частности, логика слишком тяжела для решения большинства повседневных задач веб-разработки, и слишком трудна для основной массы разработчиков. В данной статье рассматривается подход, базирующийся на идее логических архитектур и ориентированный на преодоление этих трудностей.

Ключевые слова: онтология, ОО-проекция, логическая архитектура, дескриптивные логики, семантическое программирование, объектно-ориентированный подход.

Введение

Дескриптивные логики (DL) — одна из наиболее популярных и адекватных формальных систем для автоматизированного представления и обработки знаний. Они предлагают необходимый компромисс между выразительностью и сложностью, между универсальностью и способностью решать реальные практические задачи. Различные диалекты DL могут настраиваться под специфику предметных областей и задач. Тем не менее, все еще есть ряд проблем, который наносит ущерб продвижению этих понятий среди широкого круга разработчиков и пользователей, в частности, в рамках Semantic Web [1].

Проблема 1. Универсальные логические формализмы недостаточно чувствительны к структуре и сложности решаемых задач.

Проблема 2. Эти формализмы не учитывают человеческий фактор, связанный с высоким порогом понимания, резко ограничивающим круг их пользователей.

Проблема 3. Логики мало поддерживают принцип постепенности и этапности построения моделей сложных предметных областей.

Проблема 4. Возникают сложности с процедурной обработкой информации, описанной логическими средствами, которые обусловлены разной природой логических и императивных методов работы с данными и знаниями.

На практике недостаточно лишь накапливать данные и знания, что достигается с помощью их формализации логическими средствами. Необходимы практические инструменты обработки этих данных — поиска, редактирования, построения различных статистик, срезов, и т. д. и т. п. Несмотря на это, мы убеждены, что формальные методы математической логики могут внести по-настоящему существенный вклад в практику программистов и веб-разработчиков. Формальные методы дескриптивных логик продемонстрировали свою полезность при решении задач в ряде специализированных

ISSN 1818-7897. Вестник НГУ. Серия: Математика, механика, информатика. 2009. Т. 9, вып. 3. С. 64–85

областей (например, разработке больших онтологий в биологии и медицине). Однако для использования дескриптивных логик при решении универсальных задач построения продвинутых информационных ресурсов, и для привлечения к логикам внимания обычных разработчиков и пользователей, необходимо сделать еще ряд существенных шагов.

В качестве общего подхода к решению проблем, перечисленных выше, введем понятие логической архитектуры. Пусть $\mathcal{L} = \langle L, \vdash \rangle$ — логика языка L с отношением выводимости $\vdash$. Выделим в логике $\mathcal{L}$ некоторую внутреннюю структуру — иерархию:

$$\mathcal{A}_{\mathcal{L}} = \langle \mathcal{L}, \mathcal{L}_1, \ldots, \mathcal{L}_n; \subseteq \rangle$$

такую, что $\mathcal{L}_i \subseteq \mathcal{L}$ для каждой $\mathcal{L}_i = \langle L_i, \vdash_i \rangle$, $i = \overline{1,n}$ (т. е. L_i — подъязык L, и для любой $F \in L_i$ выполняется $\mathcal{L}_i \vdash_i F \Rightarrow \mathcal{L} \vdash F$).

Каждый элемент иерархии является подлогикой $\mathcal{L}$, которая может быть настроена на некоторую проблему, например, на решение определенного вида подзадач или на работу с определенным типом пользователей. Заметим, что такое «расслоение» логики является весьма стандартным и часто используется. В частности, язык описания онтологий OWL [2] реально является иерархией трех языков OWL-Lite $\subseteq$ OWL-DL $\subseteq$ OWL-Full.

Однако для наших целей построение такой иерархии недостаточно. Нас также крайне интересует ее взаимодействие с внешними факторами и агентами. Среди таких факторов мы в основном фокусируемся на следующих двух. Это интерфейсы (UI) и агенты для процедурного управления данными (Mod). Логические иерархии в сочетании с этими двумя типами агентов мы называем *логической архитектурой*.

Определение 1 (логическая архитектура). Пусть $\langle \mathcal{L}_1, \ldots, \mathcal{L}_n; \subseteq \rangle$ — некоторая иерархия логики $\mathcal{L}$. Тогда логическая архитектура логики $\mathcal{L}$ — это множество компонентов

$$\langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle,$$

где $\mathcal{C}_i = \langle \mathcal{L}_i, \mathrm{UI}_i, \mathrm{Mod}_i \rangle$. В этой тройке UI_i это интерфейс к данным, описываемым логикой $\mathcal{L}_i$, и Mod_i это библиотека модулей, взаимодействующих с данными, описываемыми логикой $\mathcal{L}_i$.

Очевидно, что это определение скорее концептуальное, чем математическое. На его основе мы будем строить некоторый общий подход. Начнем с анализа, каким образом логические архитектуры могли бы сделать вклад в решение проблем 1–4, обозначенных выше.

Проблема 1. Конкретная логическая архитектура может интерпретироваться как логическая структура, адаптированная к работе в конкретной предметной области. Адаптация включает сегментацию соответствующей логики, а также привязку к получившейся логической иерархии необходимого пакета интерфейсов и методов.

Проблема 2. Логическая архитектура может содержать компоненты с достаточно простыми логиками. Чем проще логика, тем легче разработка понятных интерфейсов, которые скрывают элитную природу логики и могут восприниматься обычными разработчиками и пользователями. В целом различные компоненты логической архитектуры могут быть ответственными за взаимодействие с разными типами пользователей, имеющих различные способности и задачи.

Проблема 3. Часть сегментов логической архитектуры может непосредственно отвечать за различные стадии разработки проекта.

Проблема 4. В общем случае ограничения, накладываемые на логические средства в рамках отдельного сегмента логической архитектуры, должны благотворно сказаться на решении проблем взаимодействия дескриптивной и процедурной составляющих. Хорошо известно, что чем сложнее логический формализм, тем мучительнее процесс скрещивания логических и процедурных средств. Конечно, следует учитывать, что, работая только с сегментом логики, процедурные методы все равно будут оказывать влияние на ситуацию в логической системе в целом. Поэтому универсальных рецептов работы здесь не существует. Однако в каждом конкретном случае могут быть найдены логически корректные эвристики, которые способны в существенной мере упростить взаимодействие между дескриптивным и процедурным блоками.

Концепция логической архитектуры лежит в основе разрабатываемого нами метода работы со знаниями и информационными ресурсами в распределенных информационных пространствах. Данный метод включает три ключевых компонента. Во-первых, это настраиваемая система логического вывода, осуществляющая поддержку подлогик разной сложности, а значит, позволяющая выбрать в конкретных случаях оптимальное соотношение выразительности и эффективности. Во-вторых, это общий механизм построения интерфейсов, работающих с конкретными компонентами логической иерархии. В-третьих, это механизм трансляции знаний, представленных в рамках конкретной подлогики, во внешние средства обработки данных (например, в программы на объектно-ориентированных языках программирования) и обратно. В качестве «главной» логики, на базе которой строятся логические архитектуры, нами выбрана дескриптивная логика $\mathcal{SHOIN}(D)$ [3].

Особую значимость имеют компоненты логической архитектуры, которые ориентированы на вовлечение в контекст нашего подхода широкого круга веб-разработчиков, создающих основную массу веб-ресурсов в интернете. Их вовлечение является ключевым условием продвижения логических средств в практику работы в информационных пространствах. Очевидно, что данные компоненты архитектуры должны обладать рядом определенных качеств. В частности, они должны (1) в большей степени ориентироваться на конкретные данные, чем на общие знания; (2) допускать очень эффективные реализации; (3) быть удобными для обычного веб-разработчика, обеспечивать привычный стиль работы, в частности, инкапсулировать незнакомые логические средства; (4) обеспечивать связь со стандартными средствами разработки, включая базы данных и объектно-ориентированные программы. Важно учесть также ряд методологических моментов, связанных с определенным несоответствием формализмов и практических задач. В частности, не очень удобным механизмом, реализованным в дескриптивных логиках, является ограниченная квантификация на множествах, что затрудняет моделирование упорядочений, хотя упорядоченные последовательности являются одной из базовых структур данных. Концепция семантического программирования [4], теоретически обосновывающая ограниченную квантификацию на списках, обеспечивает нас инструментами для решения указанной задачи.

В рамках данной работы строится компонент логической архитектуры, ориентированный на взаимодействие с «обычным» разработчиком информационных ресурсов и соответствующий вышеперечисленным требованиям. Решение данной задачи осуществляется через моделирование в рамках дескриптивной логики $\mathcal{SHOIN}(D)$ объектно-ориентированного подхода к представлению данных. Объектно-ориентированное программирование — одна из наиболее успешных парадигм, которая, в частности, очень активно используется в веб-разработке. Как известно, работа в рамках ООП основана на построении объектно-ориентированной модели предметной области или ее части. Мы исходим из того, что ОО модель может быть погружена в произвольное логическое описание $\mathcal{K}$ предметной области в качестве ее наиболее «твердой и осязаемой» части. Это означает, что даже если мы используем продвинутые и сложные логические инструменты для разработки логической модели $\mathcal{K}$, эта модель будет включать объектно-ориентированную подструктуру, отражающую объектную модель предметной области. ОО модель может играть и роль «каркаса», на котором формируется полное формальное описание предметной области.

В разделе 1 вводится объектно-ориентированный компонент логической архитектуры, который называется *OO-проекцией*. В разделе 2 вводится абстрактный объектно-ориентированный язык (AOOL), и обосновывается его система типов. Раздел 3 посвящен построению отображения из OO-проекции в систему типов языка AOOL и доказательству ключевых свойств их взаимодействия. Показывается идентичность поведения OO-проекций и структурных типов данных AOOL. В разделе 4 мы рассматриваем OntoBox — систему, которая реализует OO-проекции эффективным образом.

§ 1. Объектно-ориентированные проекции

В этом разделе мы определяем компонент логической архитектуры, ориентированный на работу с объектными моделями и данными. Конструкция этого компонента основывается на объектно-ориентированном подходе. Как уже упоминалось выше, основной логикой для построения логических архитектур нами выбрана дескриптивная логика $\mathcal{SHOIN}(D)$. Этот выбор обусловлен рядом причин, и в первую очередь тем, что $\mathcal{SHOIN}(D)$ лежит в основе языка описания онтологий OWL-DL [2], что обеспечивает совместимость подхода, разрабатываемого нами с базовыми конструкциями SW.

Пусть

$$\mathcal{D} = \langle D_1, \ldots, D_k; \Omega \rangle$$

некоторая область данных. Примерами D_i могут служить целые и вещественные числа, строки, даты и т. д. Ω содержит имена «встроенных» операций и отношений, работающих на данных D_i. Мы считаем, что все элементы типов данных — выделенные, поэтому Ω содержит константы для каждого элемента каждого D_i. В дальнейшем через v, v_i будем обозначать константы из Ω, а через w, w_i — основные термы (без переменных) на Ω.

Словарь дескриптивной логики включает символы концептов, ролей, атрибутов и объектов. Концепты выделяют в множестве объектов предметной области совокупности объектов, обладающих определенными свойствами. В дальнейшем концепты обо-

значаются через A, B, C, E, причем A, B, C обозначают именованные (атомарные) концепты, а E — произвольные концепты (формулы). Роли (о-свойства) определяют отношения между объектами предметной области и обозначаются через R, R_i. Атрибуты (т-свойства), обозначаемые через P, P_i, привязывают элементы области данных к объектам. Через O, O_i обозначаем имена объектов предметной области. Словарь — это четверка $\langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{O} \rangle$, где $\mathcal{C}$ — множество именованных концептов, $\mathcal{R}$ — множество ролей, $\mathcal{P}$ — множество атрибутов, $\mathcal{O}$ — множество имен объектов. Область данных $\mathcal{D}$ считается фиксированной, поэтому ее компоненты в словари не входят.

Основные структуры и семантика дескриптивной логики $\mathcal{SHOIN}(D)$ представлены на рис 1. Описание предметной области в $\mathcal{SHOIN}(D)$, как и в любой другой дескрип-

Концепт	Пример	Семантика	
C	Woman	$C(x)$	имен. концепт
$E_1 \sqcap E_2$	Woman $\sqcap$ Manager	$E_1(x) \,\&\, E_2(x)$	конъюнкция
$E_1 \sqcup E_2$	Woman $\sqcup$ Man	$E_1(x) \vee E_2(x)$	дизъюнкция
$\neg C$	$\neg Man$	$\neg C(x)$	отр. имен. концептов
$\{o\}$	$\{john\} \sqcup \{paul\}$	$x = o$	объекты
$\forall R.E$	$\forall hasChild.Woman$	$\forall y.(R(x,y) \to E(y))$	огр. $\forall$
$\exists R.E$	$\exists hasChild.Woman$	$\exists y.(R(x,y) \,\&\, E(y))$	огр. $\exists$
$\forall R^-.E$	$\forall hasChild^-.Man$	$\forall y.(R(y,x) \to E(y))$	обрат. $\forall$
$\exists R^-.E$	$\exists hasChild^-.Man$	$\exists y.(R(y,x) \,\&\, E(y))$	обрат. $\exists$
$\forall P.D_i$	$\forall age.Integer$	$\forall v.(P(x,v) \to v \in \lvert D_i \rvert)$	$\forall$ на данных
$\forall P.w$	$\forall age.20$	$\forall d.(d = w \to P(x,d))$	$\forall 2$ на данных
$\exists P.w$	$\exists age.20$	$\exists d.(d = w \,\&\, P(x,d))$	$\exists$ на данных
$\geq n.R$	$\geq 1.hasChild$	$\exists y^{\geq n}.R(x,y)$	кардинальность R1
$\leq n.R$	$\leq 2.hasChild$	$\exists y^{\leq n}.R(x,y)$	кардинальность R2
$\geq n.P$	$\geq 1.hasChild$	$\exists y^{\geq n}.P(x,y)$	кардинальность P1
$\leq n.P$	$\leq 2.hasChild$	$\exists y^{\leq n}.P(x,y)$	кардинальность P2
$\top$	$\exists hasChild.\top$	true	все объекты
$\bot$	$\forall hasChild.\bot$	false	пусто

Рис. 1. Логика $\mathcal{SHOIN}(D)$

тивной логике, состоит из двух блоков: А-бокса и Т-бокса. TBox содержит описание знаний концептуального (терминологического) уровня и состоит из аксиом включения:

$E_1 \sqsubseteq E_2, \quad \forall x.(E_1(x) \to E_2(x))$

$R_1 \sqsubseteq R_2, \quad \forall x \forall y.(R_1(x,y) \to R_2(x,y))$

$P_1 \sqsubseteq P_2, \quad \forall x \forall v.(P_1(x,v) \to P_2(x,v))$

$E_1 \equiv E_2$ — сокращенная запись двух аксиом $(E_1 \sqsubseteq E_2)$ и $(E_2 \sqsubseteq E_1)$. ABox описывает предметную область на уровне конкретных данных. ABox может содержать аксиомы следующих типов: $C(O)$ — принадлежность объекта концепту, $R(O_1, O_2)$ — отношение между объектами, $P(O, v)$ — значение атрибута объекта.

Наша первая задача — в рамках $\mathcal{SHOIN}(D)$ построить «объектно-ориентированную» подсистему, ответственную за построение объектных подмоделей в рамках произ-

вольных логических моделей предметных областей. Такую подсистему назовем *OO-проекцией*. OO-проекция представляет собой модификацию и расширение простой дескриптивной логики $\mathcal{F}L_0$ [5].

Язык $\mathcal{F}L_0$ включает атомарные концепты C_i, роли R_i, конъюнкцию $\sqcap$ и квантор всеобщности $\forall R.C$. Также у нас имеются две логические константы $\top$ (top) и $\bot$ (bottom). Так как $\mathcal{F}L_0$ это дескриптивная логика, любая интерпретация I этих конструкций на множестве объектов Δ должна удовлетворять правилам

$$\top^I = \Delta,$$
$$\bot^I = \emptyset,$$
$$C_i^I \subseteq \Delta,$$
$$R_i^I \subseteq \Delta \times \Delta,$$
$$(F \sqcap G)^I = F^I \cap G^I,$$
$$\forall R.C^I = \{o \mid o \in \Delta \text{ и } \forall o'.(R(o, o') \to C(o'))\}.$$

Мы расширяем базовый формализм $\mathcal{F}L_0$ средствами для привязки объектов к данным из области данных $\mathcal{D}$. Это означает, что вместе с ролями R_i, которые описывают связи между объектами (как объектные роли `spouse` — отношение «быть супругом», и `hasChild` — отношение «иметь ребенка»), мы можем определять атрибуты объектов (например, целое число возраста `age` и строку имени `name`). Для этого используются атрибуты $P_1, \ldots, P_k$.

Введем теперь оператор типизации θ, который типизирует роли и атрибуты:

(1) привязывает роли $R \in \mathcal{R}$ к парам $\theta(R) = [C_d, C_r]$, где $C_d, C_r \in \mathcal{C}$;

(2) привязывает атрибуты P к парам $\theta(P) = [C_d, D_r]$, где $C_d \in \mathcal{C}$ и $D_r \in \mathcal{D}$.

C_d называется областью определения роли R (атрибута P), C_r (D_r) называется областью значений роли R (атрибута P).

Будем говорить, что интерпретация I согласована с θ, если

$$\forall R \in \mathcal{R}.(\theta(R) = [C_d, C_r] \ \to \ R^I \subseteq C_d^I \times C_r^I),$$
$$\forall P \in \mathcal{P}.(\theta(P) = [C_d, D_r] \ \to \ P^I \subseteq C_d^I \times D_r^I).$$

Интерпретация I согласована со словарем $\langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{O} \rangle$, если для любого объекта $o \in \top^I$ существует имя $O \in \mathcal{O}$ такое, что $o = O^I$ (т. е. все объекты выделенные). В дальнейшем будем рассматривать только интерпретации, согласованные с θ и словарем. $\mathcal{K} \models E$ будет обозначать, что E истинна в любой согласованной интерпретации $\mathcal{K}$. В отличие от этого $\mathcal{K} \vdash E$ означает, что E выводима из $\mathcal{K}$ в некотором полном исчислении (например, с помощью табличных алгоритмов), а значит, истинна в любой интерпретации.

Пусть $\mathcal{V} = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{O} \rangle$ — словарь.

Определение 2 (OO-концепт). Концепт вида

$$C_1 \sqcap \ldots \sqcap C_f \sqcap \forall R_1.E_1 \sqcap \ldots \sqcap \forall R_g.E_g \sqcap \forall P_1.D_1 \sqcap \ldots \sqcap \forall P_h.D_h$$

называется OO-концептом, где E_i — OO-концепты, $C, C_i \in \mathcal{C}$, $R_i \in \mathcal{R}$, $P_i \in \mathcal{P}$ и $D_i \in \mathcal{D}$. Если все $E_i \in \mathcal{C}$, то OO-концепт называется примитивным.

Определение 3 (ОО-определение). Аксиома

$$C \equiv E, \tag{1}$$

где E — примитивный ОО-концепт, называется объектно-ориентированным определением (ОО-определением).

Пример ОО-определения:

$$\texttt{Person} \equiv \texttt{Mammal} \sqcap \forall \texttt{hasChild}.\texttt{Person} \sqcap \forall \texttt{name}.\texttt{String}$$

Неформальная интерпретация этих конструкций с помощью объектно-ориентированных концептов (что оправдывает использование префикса «ОО–») прямая. В ОО-определении

$$C \equiv C_1 \sqcap \ldots \sqcap C_n \sqcap \forall R_1.B_1 \sqcap \ldots \sqcap \forall R_k.B_k \sqcap \forall P_1.D_1 \sqcap \ldots \sqcap \forall P_l.D_l$$

атомарные концепты C, C_i, B_i обозначают ОО-классы, $C_1 \sqcap \ldots \sqcap C_n$ обозначает наследование, и если $n > 1$, тогда мы имеем множественное наследование. Каждое выражение $\forall R_i.B_i$ обозначает поле R_i, значения которого являются объектами класса B_i. Каждое выражение $\forall P_i.D_i$ соответствует полю, значения которого являются элементами предопределенного типа данных D_i.

Мы говорим, что концепт C в ОО-определении (1) *непосредственно наследует* от $C_1, \ldots, C_n$. Пусть $\mathcal{A} = \{C_i \equiv E_i\}$ — множество ОО-определений. C_i *наследует* от C_j, если он непосредственно наследует от C_j, или существует C_k такой, что C_i наследует от C_k и C_k наследует от C_j. $\mathcal{A}$ называется ацикличным, если оно не содержит C_i, наследующих от самих от себя.

Схожим образом, если концепт C определен с помощью ОО-определения (1) в множестве ОО-определений $\mathcal{A}$, мы говорим, что концепт C *непосредственно зависит от* ролей $R_1, \ldots, R_k$ и атрибутов $P_1, \ldots, P_l$. C *зависит* от роли R (атрибута P), если он непосредственно зависит от роли (атрибута), или существует C' в $\mathcal{A}$, который непосредственно зависит от R (P), и C наследует от C'.

Теперь обратимся к определению ОО-проекции. Пусть $\mathcal{L}$ — дескриптивная логика, содержащая $\mathcal{SHOIN}(D)$ как подлогику, и $\mathcal{K} = TBox_{\mathcal{K}} \cup ABox_{\mathcal{K}}$ — $\mathcal{L}$–описание предметной области в словаре $\mathcal{V} = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{O} \rangle$ (в дальнейшем такие описания мы называем онтологиями). Пусть $\mathcal{V}_o = \langle \mathcal{C}_o, \mathcal{R}_o, \mathcal{P}_o, \mathcal{O}_o \rangle$ — подсловарь $\mathcal{V}$, $\mathcal{O}_o = \{O_1, \ldots, O_l\}$, а θ — оператор типизации на $\mathcal{V}_o$.

Определение 4 (ОО-проекция).

$$\mathcal{K}_o = TBox_o \cup ABox_o$$

— ОО-проекция словаря $\mathcal{V}_o$ и оператора θ, если

(1) $TBox_o$ — ацикличное множество ОО-определений, любая интерпретация которого согласована с θ;

(2) для каждой аксиомы $Ax \in TBox_o$ выполняется $\mathcal{K} \vdash Ax$;

(3) любая роль и атрибут из словаря $\mathcal{V}_o$ входит в ОО-определения $TBox_o$ не более одного раза;

(4) $ABox_o \subseteq ABox_{\mathcal{K}}$ и

- для любого $C(O) \in ABox_o$: $O \in \{O_1, \ldots, O_l\}$ и $C \in \mathcal{C}_o$, причем для каждого O такой C является единственным;
- для любого $R(O, O') \in ABox_o$: $O, O' \in \{O_1, \ldots, O_l\}$, $R \in \mathcal{R}_o$ такие, что $C(O), C'(O') \in ABox_o$, причем $\mathcal{K}_o \models C \sqsubseteq C_1$, $\mathcal{K}_o \models C' \sqsubseteq C'_1$ и $\theta(R) = [C_1, C'_1]$;
- для любого $P(O, v) \in ABox_o$: $O \in \{O_1, \ldots, O_l\}$, $P \in \mathcal{P}_o$, $C(O) \in ABox_o$ и $v \in |D|$, где $\theta(P) = [C_1, D]$ и $\mathcal{K}_o \models C \sqsubseteq C_1$.

Прокомментируем это определение. Поскольку ОО-проекции призваны моделировать объектно-ориентированный подход в дескриптивных логиках, необходимо сверить ее основные конструкции с хорошо знакомыми объектно-ориентированными конструкциями. $TBox_o$ несет ответственность за описание иерархий концептов, аналогичных описаниям иерархий классов в ОО-программировании. В частности, требование ациклич-ности $TBox_o$ очевидным образом вытекает из ацикличности иерархий объектно-ориентированных типов. Требование уникального вхождения ролей и атрибутов в $ABox_o$ обусловлено тем, что два вхождения имени атрибута обозначают один и тот же атрибут, в то время как одноименные вхождения атрибутов в объектно-ориентированных программах, как правило, обозначают разные атрибуты-тезки. Получается, что в дескриптивных логиках и ОО-подходе такие ситуации интерпретируются по-разному, поэтому мы должны их избегать. Оператор θ используется для того, чтобы явно типизировать роли и атрибуты. Заметим, что $TBox_o$ должен содержать всю информацию, необходимую для обеспечения такой типизации (поскольку любая интерпретация $TBox_o$ должна быть согласована с θ).

Описания самих объектов в ОО-проекции задаются в $ABox_o$. Хотя объект O может принадлежать к различным концептам в $ABox_{\mathcal{K}}$, для ОО-описания мы выбираем только один концепт как базовый для O. Это соответствует объектно-ориентированному подходу, в котором принадлежность объекта к нескольким классам определяется, как правило, через наследование. С другой стороны, число связей, описываемых ролями и атрибутами объекта O, может быть произвольным. Более того, любая роль R может связывать объект сразу с несколькими объектами, что определяется наличием в $ABox_o$ нескольких утверждений вида $R(O, O^1), \ldots, R(O, O^i), i \geq 0$. Для обозначения этого мы будем использовать более компактную запись: $R(O, \langle O^1, \ldots, O^i \rangle)$. То же самое и с атрибутами. Если O имеет значения $v_1, \ldots, v_j$ атрибута P в $ABox_o$, то мы пишем $P(O, \langle v_1, \ldots, v_j \rangle)$.

Следующее утверждение иллюстрирует еще одно полезное свойство ОО-проекций:

Утверждение 1. *Пусть концепт C определен в $TBox_o$ с помощью ОО-определения вида* (1). *Тогда для любой интерпретации I ОО-проекции $\mathcal{K}_o$, выполняется $R_i^I \subseteq C^I \times B_i^I$ и $P_i^I \subseteq C^I \times D_i^I$.*

ДОКАЗАТЕЛЬСТВО. следует из уникальности вхождения R_i и P_i в $TBox_o$, и из условия 1 определения ОО-проекции. □

Пусть $\mathcal{K}_o$ — ОО-проекция, и E_1 и E_2 — два ОО-концепта, определенных в словаре ОО-проекции. Говорим, что концепт E_1 поглощается концептом E_2 в $\mathcal{K}_o$, если для каждой согласованной интерпретации I, $E_1^I \subseteq E_2^I$ (т. е. $\mathcal{K}_o \models E_1 \sqsubseteq E_2$). Хорошо известно, что проблема классификации концептов для логики $\mathcal{FL}_0$ разрешима (см., например: [5]). В частности, для этого мы можем использовать алгоритм структурной классификации [6]. В ОО-проекциях иерархии, построенные на базе отношения поглощения, играют ключевую роль, поскольку аналогичны иерархиям классов в объектно-ориентированном подходе.

§ 2. Абстрактный объектно-ориентированный язык

Чтобы исследовать взаимодействие ОО-проекций с системой типов данных в объектно-ориентированном подходе, введем абстрактный объектно-ориентированный язык (AOOL) с множественным наследованием. Традиционно объектно-ориентированные языки описываются в рамках лямбда-исчисления с включением типов и рекурсией (см., например: [7–9]). Домены данных формируются из базовых типов данных с помощью конструктора записей. В некоторых языках также используется конструктор варианта (прямой суммы). Будем считать, что в качестве базовых типов данных в AOOL работают те же типы данных, что и в ОО-проекциях, т. е., принадлежащие области $\mathcal{D}$.

Определение 5 (Запись). Запись — это конечное отображение из имен полей в значения:

$$\texttt{value o = [a}_1 \mapsto \texttt{e}_1\texttt{, a}_2 \mapsto \texttt{e}_2\texttt{, ..., a}_n \mapsto \texttt{e}_n\texttt{]}$$

где $\texttt{a}_i$ — это поля записи и $\texttt{e}_i$ — значения.

Домен записи обозначаем через $\texttt{dom(o)} = \{\texttt{a}_1, \ldots, \texttt{a}_n\}$. Через точку определяем оператор взятия значения поля: $\texttt{o.a}_i = \texttt{e}_i$.

Определение 6 (композиция записей). Пусть o_1 и o_2 — две записи. Тогда их композиция $o = o_1 \oplus o_2$ определяется как запись, такая что $\texttt{dom(o) = dom(o}_1\texttt{)} \cup \texttt{dom(o}_2\texttt{)}$, и для каждого $\texttt{a} \in \texttt{dom(o)}$ выполняется:

$$\text{если } \texttt{a} \in \texttt{dom(o}_1\texttt{)}\text{, то } \texttt{o.a = o}_1\texttt{.a, иначе o.a = o}_2\texttt{.a.}$$

Некоторые примеры записей:

```
value john = [name ↦ 'John', surname ↦ 'Smith', age ↦ 40]
value marie = [name ↦ 'Marie', spouse ↦ john]
[a ↦ 1, b ↦ 2] ⊕ [b ↦ 3, c ↦ 4] = [a ↦ 1, b ↦ 2, c ↦ 4]
```

Например, `marie.name = 'Marie'` и `john.age = 40`.

Иногда значение поля записи неизвестно или не существует. Для отражения этого факта в AOOL добавляется специальное «пустое» значение `nil`. Пример использования `nil`: `[a ↦ 1, b ↦ nil]`.

Рекурсивные функции и их семантика определяются через стандартный оператор неподвижной точки `Fix(.)` [10], который для каждого λ-выражения `F` возвращает `f = = Fix(F)`, такой что `f = F(f)`. Классический пример: для определения функции факториала, вместо

$$\texttt{fact} = \lambda\texttt{n. if n = 0 then 1 else n} \times \texttt{fact(n - 1)}$$

можно написать

$$\texttt{FACT} = \lambda\texttt{self.}\lambda\texttt{n. if n = 0 then 1 else n} \times \texttt{self(n - 1)}$$

и затем определить `fact = Fix(FACT)`. Внешняя переменная λ-выражения (в случае `FACT` это `self`) интерпретируется при вычислении неподвижной точки как ссылка на сам объект, определенный λ-выражением. Это дает еще одну возможность использовать оператор неподвижной точки — для моделирования переменных, ссылающихся на сам объект. Ссылки из объекта на него самого играют важную роль в объектно-ориентированных языках (например, в Java это `this`). Для определения семантики такой «само-ссылки» также используется `Fix`, например:

$$\texttt{Fix}(\lambda\texttt{self.[val} \mapsto \texttt{2, sq} \mapsto \texttt{self.val}^2\texttt{]}) = \texttt{[val} \mapsto \texttt{2, sq} \mapsto \texttt{4]}$$

Данные переменные требуют специального подхода при определении композиции двух записей:

$$\lambda\texttt{self.o}_1 \oplus \lambda\texttt{self.o}_2 = \lambda\texttt{self.(o}_1 \oplus \texttt{o}_2\texttt{)}$$

С неформальной точки зрения композиция двух записей означает, что они описывают один и тот же объект, который должен обозначаться одной и той же переменной `self`.

Теперь рассмотрим, как в AOOL определяются классы и наследование. Начнем с примера — определения класса `Point` [8]:

$$\begin{aligned}
&\texttt{class Point(X:Float, Y:Float)} = \lambda\texttt{self.[}\\
&\quad \texttt{x:Float} \mapsto \texttt{X,}\\
&\quad \texttt{y:Float} \mapsto \texttt{Y,}\\
&\quad \texttt{distFromOrig:Float} \mapsto \texttt{sqrt(self.x}^2 \texttt{ + self.y}^2\texttt{),}\\
&\quad \texttt{closerToOrig:Boolean} \mapsto \lambda\texttt{p.(self.distFromOrig < p.distFromOrig)]}
\end{aligned}$$

Благодаря оператору неподвижной точки, это определение может быть использовано как *генератор объектов*, например:

$$\begin{aligned}
&\texttt{Fix(Point(3, 4))} = \texttt{[}\\
&\quad \texttt{x} \mapsto \texttt{3,}\\
&\quad \texttt{y} \mapsto \texttt{4,}\\
&\quad \texttt{distFromOrig} \mapsto \texttt{5,}\\
&\quad \texttt{closerToOrig} \mapsto \lambda\texttt{p.(5 < p.distFromOrig)]}
\end{aligned}$$

Мы говорим, что этот объект *сгенерирован* классом `Point`.

Чтобы научить классы наследовать от других классов, введем операцию суперкласса *superclass* $\triangleright$ *class*:

```
λself.b ▷ λself.λsuper.a = λself.((λsuper.a)(b)) ⊕ λself.b
```

Операция суперкласса $\triangleright$ интуитивно достаточно понятна. Если у нас есть описание класса `λself.λsuper.a`, включающее ссылку объекта на себя (переменная `self`) и ссылку на суперкласс (переменная `super`), и если этот суперкласс определен с помощью выражения `λself.b`, тогда `λself.((λsuper.a)(b))` говорит, что `b` должна восприниматься как описание суперкласса в `a`. Также нам необходимо иметь доступ в `a` к полям, определенным в записи суперкласса. Это достигается добавлением `...⊕ λself.b`.

Общий шаблон для определения класса в нашем абстрактном языке имеет следующий вид:

Определение 7 (ОО-класс и ОО-программа). Определение класса `c` есть выражение вида

$$\begin{array}{l}\texttt{class c}(\bar{x})\ \texttt{=}\\ \quad \texttt{sc}(\bar{x})\ \triangleright\ \lambda\texttt{self.[a}_1\texttt{:c}_1 \mapsto \texttt{e}_1\texttt{, a}_2\texttt{:c}_2 \mapsto \texttt{e}_2\texttt{, ..., a}_n\texttt{:c}_n \mapsto \texttt{e}_n\texttt{]}\end{array} \tag{2}$$

ОО-программа — это конечная последовательность определений классов и генераторов объектов вида

```
value o = c(ē),
```

удовлетворяющая стандартному условию на ацикличность отношения наследования. Замкнутая ОО-программа содержит определения всех классов и объектов, чьи имена входят в нее.

В (2) `sc` обозначает суперкласс, $\texttt{a}_i\texttt{:c}_i$ неформально означает, что значение поля $\texttt{a}_i$ должно принадлежать классу или типу данных с именем $\texttt{c}_i$. Здесь это $\texttt{c}_i$ может быть либо базовым типом данных (т. е. одним из D_i), либо именем класса. Множественное наследование вводится через композицию суперклассов, так что $\texttt{sc}(\bar{x})\ \texttt{=}\ \texttt{sc}_1(\bar{x})\ \oplus \ldots \oplus\ \texttt{sc}_k(\bar{x})$ (выражение $e(\bar{x})$ означает, что все параметры выражения e находятся среди $\bar{x}$). Рассмотрим пример множественного наследования. Определим класс `ColoredCircle`, который наследует от класса `Point`, введенного выше, и класса `Colored`:

```
class Colored(C:String) = λself.[color:String ↦ C]
class ColoredCircle(X:Float, Y:Float, R:Float, C:String) =
   Colored(C) ⊕ Point(X, Y) ▷
   λself.[
      radius:Float ↦ R
      distFromOrig:Float  ↦ max(0, super.distFromOrig - self.radius)]
```

Заметим, что здесь функция `distFromOrig` переопределяется для кругов с использованием `distFromOrig`, определенного для точек. Вычисление `Fix(ColoredCircle(3, 4, 1, «красный»))` генерирует новый красный круг с радиусом 1.

Теперь мы готовы ввести систему типов абстрактного языка. Если взглянуть на конкретный объект, определенный в нем (скажем, `Point(3, 4)`, приведенный выше), то видно, что он состоит из полей двух типов. Некоторые поля содержат значения, характеризующие сам объект (как `x`, `y` и `distFromOrig` в `Point(3, 4)`). Другие поля

определяют, что может быть сделано с данными объектами, и содержат функциональные определения более высокого порядка (как `closerToOrig` в `Point(3, 4)`). В объектно-ориентированном подходе такие функции называются *методами*. Так как наша цель — найти отображение между системами типов в OO-проекциях и AOOL, то ясно, что методы не могут быть частью этой интеграции, поскольку они имеют функциональную либо процедурную природу, тогда как в дескриптивных логиках для манипуляции данными используются системы логического вывода. Таким образом, нам следует сфокусироваться большей частью на полях, которые имеют сходство со свойствами/ролями в онтологиях. Для этого введем понятие *структурного типа*.

Определение 8 (Структурный тип). Пусть $\mathcal{P}$ — OO-программа.

(1) Имена классов, а также имена типов данных $D_1, \ldots, D_k$ — структурные типы.

(2) Если $\mathtt{c}_1, \ldots, \mathtt{c}_k$ имена классов и $\mathtt{a}_1, \ldots, \mathtt{a}_n$ имена полей в $\mathcal{P}$, и $\tau_1, \ldots, \tau_n$ структурные типы, тогда $\tau = \{\mathtt{c}_1, \ldots, \mathtt{c}_k, \mathtt{a}_1{:}\tau_1, \ldots, \mathtt{a}_n{:}\tau_n\}$ — структурный тип с $\mathtt{dom}(\tau) = \{\mathtt{a}_1, \ldots, \mathtt{a}_n\}$ и $\mathtt{cls}(\tau) = \{\mathtt{c}_1, \ldots, \mathtt{c}_k\}$.

Композиция структурных типов определяется следующим образом:

$$\tau_1 \oplus \tau_2 = \tau_1 \cup \mathtt{cls}(\tau_2) \cup \{\mathtt{a} : \tau \mid \mathtt{a} \in \mathtt{dom}(\tau_2)/\mathtt{dom}(\tau_1) \text{ и } \mathtt{a} : \tau \in \tau_2\}.$$

Структурный тип $\tau_{\mathtt{c}}$ класса

$$\begin{array}{l}\mathtt{class\ c}(\bar{x})\ = \\ \quad \mathtt{sc}(\bar{x}) \rhd \lambda \mathtt{self}.[\mathtt{a}_1{:}\mathtt{c}_1 \mapsto \mathtt{e}_1,\ \mathtt{a}_2{:}\mathtt{c}_2 \mapsto \mathtt{e}_2,\ \ldots,\ \mathtt{a}_n{:}\mathtt{c}_n \mapsto \mathtt{e}_n]\end{array}$$

определяется следующим образом:

$$\tau_{\mathtt{c}} = \{\mathtt{c}, \mathtt{a}_1 : \mathtt{c}_1, \mathtt{a}_2 : \mathtt{c}_2, \ldots, \mathtt{a}_n : \mathtt{c}_n\} \oplus \tau_{\mathtt{sc}}$$

где $\tau_{\mathtt{sc}}$ есть структурный тип суперкласса `sc`.

Пусть $\mathcal{P}$ — OO-программа. Чтобы проверить, имеет ли объект `o` структурный тип τ в $\mathcal{P}$ (обозначается $\mathtt{o} : \tau$), введем следующее исчисление. Обозначим через $|D_i|$ множество всех элементов типа данных D_i. Правила вывода τ-исчисления представлены на рис. 2.

В $\tau_{\mathtt{class}}$ выражение `{...}` обозначает структурный тип, который содержит все элементы `{c,...}` за исключением имени класса `c`. Заметим также, что $\tau_{\mathtt{fields}}$ возвращает `true`, если $k = 0$. $\tau_{\mathtt{gfp}}$ отлавливает ситуации, когда кросс-ссылки на объекты формируют циклы. Это соответствует идее наибольшей неподвижной точки. Такие ловушки циклов необходимы, так как система объектов может представлять произвольный ориентированный граф.

τ-вывод $\mathtt{o} : \tau$ — это дерево вывода с корнем $\mathtt{o} : \tau$, в котором следующий шаг по любой ветке получается из предыдущего применением одного из τ-правил, и все листья являются либо `true`, либо `false`. Мы говорим, что `o` имеет структурный тип τ, если существует конечный τ-вывод $\mathtt{o} : \tau$, все листья которого `true`. Обозначим этот факт $\mathcal{P} \vdash \mathtt{o} : \tau$. Будем считать, что порождение объектов в программе $\mathcal{P}$ не зацикливается, т. е. для всех объектов `o` вычисление неподвижной точки $\mathtt{Fix}(\mathtt{c}(\bar{\mathtt{e}}))$ в правиле $\tau_{\mathtt{value}}$ завершается за конечное число шагов. В этом случае верно следующее предложение.

$$\tau_{\mathtt{D}}\ \frac{o : D_i \qquad o \in |D_i|}{\mathtt{true}} \qquad \tau_{\mathtt{class}}\ \frac{o : \{\mathtt{c}, \ldots\}}{o : \tau_{\mathtt{c}} \qquad o : \{\ldots\}}$$

$$\tau_{\mathtt{nil}}\ \frac{\mathtt{nil} : \tau}{\mathtt{true}} \qquad \tau_{\mathtt{value}}\ \frac{\mathtt{o} : \tau \qquad (\mathtt{value\ o} = \mathtt{c}(\bar{\mathtt{e}})) \in \mathcal{P}}{\mathtt{Fix}(\mathtt{c}(\bar{\mathtt{e}})) : \tau}$$

$$\tau_{\mathtt{fields}}\ \frac{[\mathrm{a}_1 \mapsto \mathrm{e}_1,\ \mathrm{a}_2 \mapsto \mathrm{e}_2,\ \ldots,\ \mathrm{a}_n \mapsto \mathrm{e}_n] : \{\mathtt{a}_{i_1} : \tau_{i_1}, \ldots, \mathtt{a}_{i_k} : \tau_{i_k}\}, \forall i_j \in [1..n]}{\mathtt{true} \qquad \mathtt{e}_{i_1} : \tau_{i_1} \quad \ldots \quad \mathtt{e}_{i_k} : \tau_{i_k}}$$

$$\tau_{\mathtt{gfp}}\ \frac{\dfrac{\dfrac{o : \tau}{\vdots}}{o : \tau}}{\mathtt{true}} \qquad \tau_{\mathtt{fail}}\ \frac{o : \tau \quad \text{нельзя применить никакое другое правило}}{\mathtt{false}}$$

Рис. 2. Правила τ-исчисления

Утверждение 2. *Пусть $\mathcal{P}$ — ОО-программа. Тогда любое $o : \tau$ имеет конечный τ-вывод в $\mathcal{P}$. Любой τ-вывод завершается за конечное число шагов при выполнении следующего ограничения: правила $\tau_{\mathtt{class}}$ и $\tau_{\mathtt{fields}}$ могут применяться только если $\tau_{\mathtt{gfp}}$ не применимо.*

ДОКАЗАТЕЛЬСТВО. Любая ОО-программа содержит определение не более чем конечного числа объектов. Если рассмотреть произвольный структурный тип τ, то он будет состоять из имен классов, имен полей и имен типов данных, входящих в программу — их число также конечно. Определим глубину $\mathrm{depth}(\tau)$ структурного типа τ следующим образом: глубина атомарных структурных типов $\mathrm{depth}(\mathtt{c}) = 0$, $\mathrm{depth}(\mathtt{D}) = 0$, где $\mathtt{c}$ — имя класса, а $\mathtt{D}$ — имя типа данных; и кроме того

$$\mathrm{depth}(\{\mathtt{a}_1 : \tau_1, \ldots, \mathtt{a}_k : \tau_k\}) = 1 + \max_{j=\overline{1,k}}(\mathrm{depth}(\tau_j)).$$

Заметим, что в выводе любого $\mathtt{o}{:}\tau$ глубина не может превышать $\max(\mathrm{depth}(\tau), 1)$, поскольку отсутствуют правила, увеличивающие глубину структурных типов, за исключением τ_{class}. Но в результате применения τ_{class} появляется структурный тип $\tau_{\mathtt{c}}$ класса $\mathtt{c}$, глубина которого, как легко проверить, всегда равна 1. Таким образом, в любом доказательстве может появиться только конечное число различных структурных типов и объектов. Следовательно, в любой достаточно длинной цепочке вывода некоторое выражение $\mathtt{o}{:}\tau$ обязательно повторится, а значит, будет применимо правило $\tau_{\mathtt{gfp}}$. □

Мы также можем установить естественный порядок на структурных типах.

Определение 9 (порядок на структурных типах). Пусть $\mathcal{P}$ — ОО-программа и τ_1 и τ_2 — два структурных типа. Говорим, что τ_1 меньше, чем τ_2 в программе $\mathcal{P}$ (обозначается $\tau_1 \leq_{\mathcal{P}} \tau_2$), когда для любого объекта $\mathtt{o}$ выполняется: если $\mathcal{P} \vdash \mathtt{o} : \tau_1$, то $\mathcal{P} \vdash \mathtt{o} : \tau_2$.

Заметим, что в общем случае определения класса и суперкласса могут иметь поля с совпадающими именами. В теле определения класса вида (2) мы можем иметь доступ к обоим полям-тезкам. Однако если $\mathtt{a}$ — такое имя, чтобы иметь доступ к полю $\mathtt{a}$, определенному в суперклассе, нам следует использовать переменную `super` и писать `super.a`. Это означает, что соответствующий структурный тип должен описывать оба поля $\mathtt{a}$ и `super.a`. В дескриптивных логиках такая ситуация невозможна, поскольку все атрибу-

ты и роли должны иметь уникальные имена в пространстве онтологии (на практике — в рамках одного пространства имен). Таким образом, прямое отображение таких полей в онтологии невозможно. Намного легче переименовать поля-тезки. Далее мы подразумеваем, что нет подобных конфликтов имен для тех классов, которые должны быть отображены в логическое описание. Вообще говоря, мы можем разрешать данные конфликты без переименования, но это бы неоправданно усложнило формальную систему.

§ 3. Отображение

В этом разделе будет определено отображение из OO-проекций в систему типов языка AOOL. Чтобы отобразить OO-проекции в конструкции AOOL, вводится оператор $E \overset{\mathtt{tr}}{\to} \mathtt{e}$, переводящий сущность E OO-проекции в соответствующую ей сущность `e` языка AOOL (также будем пользоваться обозначением $\mathtt{tr}(E) = \mathtt{e}$). Для множеств сущностей полагаем $\mathtt{tr}(\{E_1, \ldots, E_k\}) = \{\mathtt{tr}(E_1), \ldots, \mathtt{tr}(\mathtt{E_k})\}$. Необходимо принять во внимание, что объекты в онтологиях могут иметь произвольное число значений ролей и атрибутов, тогда как значения полей в записях AOOL единичны. Чтобы избавиться от данного несоответствия, будем пользоваться списками в качестве значений полей. Для этого введем AOOL-определение *списка* объектов класса `c`:

```
class list-c(Hd:c, Tl:list-c) =
      λself.[head-c : c ↦ Hd, tail-c : list-c ↦ Tl]
```

Аналогично можно определить списки элементов базовых типов данных. Если имя `list-c` занято, то можно использовать любое другое уникальное имя. Заметим, что здесь в данном определении используются специфичные (типизированные) версии `head` и `tail`, вместо полиморфных. Пример:

```
Fix(list-Person(john, list-Person(marie, nil))) =
      [head-Person ↦ john,
         tail-Person ↦ [head-Person ↦ marie, tail-Person ↦ nil]]
```

является списком двух персон. Мы используем следующую короткую форму для представления этого списка:

```
list-Person⟨john,marie⟩          list-Person⟨⟩ = nil
```

Пусть $\mathcal{K}_o = TBox_o \cup ABox_o$ — произвольная OO-проекция со словарем $\mathcal{V}_o = \langle \mathcal{C}_o, \mathcal{R}_o, \mathcal{P}_o, \mathcal{O}_o \rangle$, $\mathcal{O}_o = \{O_1, \ldots, O_l\}$. Оператор $\overset{\mathtt{tr}}{\to}$ отображает $\mathcal{K}_o$ в определенную программу языка AOOL.

Перевод имен. Для каждого $C_i \in \mathcal{C}_o$, $R_i \in \mathcal{R}_o$, $P_i \in \mathcal{P}_o$, и $O_i \in \mathcal{O}_o$ вводим имя класса `ci`, имена полей `ri` и `pi`, и имя объекта `oi`, соответственно, и определяем $C_i \overset{\mathtt{tr}}{\to} \mathtt{ci}$, $R_i \overset{\mathtt{tr}}{\to} \mathtt{ri}$, $P_i \overset{\mathtt{tr}}{\to} \mathtt{pi}$ и $O_i \overset{\mathtt{tr}}{\to} \mathtt{oi}$. Аналогично с типами данных : $D_i \overset{\mathtt{tr}}{\to} \mathtt{di}$.

Перевод OO-определений осуществляется следующим образом:

$C \equiv C_1 \sqcap \ldots \sqcap C_n \sqcap \forall R_1.B_1 \sqcap \ldots \sqcap R_m.B_m \sqcap \forall P_1.D_{i_1} \sqcap \ldots \sqcap P_g.D_{i_g} \overset{\mathtt{tr}}{\to}$

```
class c(Y:list-y, Xr1:list-b1,..,Xrm:list-bm,
```

```
Xp1:list-di1,..,Xpg:list-dig) =
   c1 (Ȳ) ⊕,...,⊕ cn(Ȳ) ▷
                λ self.[r1 : list-b1 ↦ Xr1, ..., pg : list-dig ↦ Xpg]
```

Здесь `list-bi` является списком объектов класса `bi`, соответствующего концепту B_i, `list-di` является списком значений типа данных D_i, $\overline{\mathtt{Y}}$, `Xri` и `Xpi` — переменные, причем $\overline{\mathtt{Y}}$ — кортеж переменных для передачи параметров определениям суперклассов.

Перевод описания объекта. Сгруппируем всю информацию из $ABox_o$ об объекте O:

$$ABox_o^O = \{C(O),\ R_1(O, \langle O_1^1, \ldots, O_{e_1}^1\rangle), \ldots, R_k(O, \langle O_1^k, \ldots, O_{e_k}^k\rangle),\ P_1(O, \langle v_1^1, \ldots, v_{f_1}^1\rangle), \ldots, P_l(O, \langle v_1^l, \ldots, v_{f_l}^l\rangle)\}.$$

Пусть $\theta(R_i) = [C_i, B_i]$. Это означает (в соответствии с утверждением 1), что каждый объект O_j^i в любой интерпретации I ОО-проекции должен принадлежать B_i^I. Поэтому переводим последовательность $O_1^i, \ldots, O_{e_i}^i$ в $\mathtt{list\text{-}bi}\langle \mathtt{oi1}, \ldots, \mathtt{oiei}\rangle$, такой что $O_j^i \xrightarrow{\mathtt{tr}} \mathtt{oij}$.

Аналогично для P_i. Каждая последовательность $v_1^i, \ldots, v_{f_i}^i$ переводится в $\mathtt{list\text{-}di}\langle \mathtt{vi1}, \ldots, \mathtt{vifi}\rangle$. Теперь если $O \xrightarrow{\mathtt{tr}} \mathtt{o}$, то

```
ABox_o^O →tr
   value o = c(
      list-b1⟨o11,...,o1e1⟩, ..., list-bk⟨ok1,...,okek⟩,
      list-T1⟨v11,...,v1f1⟩, ..., list-Tl⟨vl1,...,vlfl⟩
   )
```

Последовательность, в которой списки входят в качестве аргументов в данный генератор объекта, должна соответствовать по типу последовательности аргументов в определении AOOL-класса, в который переводится концепт C. Заметим, что если $e_i = 0$ (т. е. $ABox_o$ не содержит ни одного утверждения вида $R_i(O, O')$), то соответствующий $\mathtt{list\text{-}bi}\langle \mathtt{oi1}, \ldots, \mathtt{oiei}\rangle$ эквивалентен `nil`. То же самое для f_j-х.

Перевод ОО-проекции. Пусть $\mathcal{K}_o$ — ОО-проекция. Тогда AOOL-программа $\mathcal{P}$, такая что $\mathcal{K}_o \xrightarrow{\mathtt{tr}} \mathcal{P}$, определяется следующим образом:

(1) Для каждого ОО-определения $C \equiv E \in TBox_o$, $\mathtt{tr}(C \equiv E)$ включается в $\mathcal{P}$. $\mathcal{P}$ также дополняется определением класса `list-c`, где $\mathtt{c} = \mathtt{tr}(\mathrm{C})$.

(2) Описание $\mathcal{A}_O \subseteq ABox_o$ каждого объекта $O \in \mathcal{O}_o$ переводится в $\mathtt{tr}(\mathcal{A}_O)$, которое включается в $\mathcal{P}$.

Перевод примитивного ОО-концепта. ОО-концепты переводятся в структурные типы. ОО-концепт вида

$$E = C_1 \sqcap \ldots \sqcap C_n \sqcap \forall R_1.B_1 \sqcap \ldots \sqcap \forall R_k.B_k \sqcap \forall P_1.D_1 \sqcap \ldots \sqcap \forall P_l.D_l$$

переводится в

$$E \xrightarrow{\mathtt{tr}} \mathtt{\{c1,\ldots,\ cn,\ r1:list\text{-}b1,\ \ldots,\ rk:list\text{-}bk,\ p1:list\text{-}d1,\ \ldots,\ pl:list\text{-}dl\}}$$

где $C_i \overset{\mathtt{tr}}{\to} \mathtt{ci}$, $R_i \overset{\mathtt{tr}}{\to} \mathtt{ri}$, $B_i \overset{\mathtt{tr}}{\to} \mathtt{bi}$, $P_i \overset{\mathtt{tr}}{\to} \mathtt{pi}$ и $D_i \overset{\mathtt{tr}}{\to} \mathtt{di}$.

Следующая теорема играет для нас ключевую роль, обосновывая гипотезу о том, что ОО-проекции ведут себя аналогично объектно-ориентированным конструкциям

Теорема 1. *Пусть $\mathcal{K}_o$ является ОО-проекцией и $\mathcal{K}_o \overset{\mathtt{tr}}{\to} \mathcal{P}$ для некоторой ОО-программы $\mathcal{P}$. Тогда для каждой пары примитивных ОО-концептов E_1 и E_2 в словаре $\mathcal{K}_o$,*

$$\mathcal{K}_o \models E_1 \sqsubseteq E_2 \quad \textit{тогда и только тогда, когда} \quad \tau_1 \leq_{\mathcal{P}} \tau_2,$$

где $E_1 \overset{\mathtt{tr}}{\to} \tau_1$ и $E_2 \overset{\mathtt{tr}}{\to} \tau_2$.

Рассмотрим основные этапы доказательства теоремы. Будем говорить, что ОО-концепт находится в канонической форме, если он имеет вид

$$\forall R_1.E_1 \sqcap \ldots \sqcap \forall R_g.E_g \sqcap \forall P_1.D_1 \sqcap \ldots \sqcap \forall P_h.D_h,$$

т. е. в нем отсутствует блок $C_1 \sqcap \ldots \sqcap C_f$. ОО-определение $C \equiv E$ находится в канонической форме, если E является примитивным каноническим ОО-концептом. В объектно-ориентированной интерпретации класс, обладающий каноническими ОО-определением, не зависит от суперклассов.

Лемма 1. *Для каждой ОО-проекции $\mathcal{K}_o = TBox_o \cup ABox_o$ существует логически эквивалентная ей ОО-проекция $\mathcal{K}'_o = TBox'_o \cup ABox_o$, в которой все ОО-определения из $TBox'_o$ находятся в канонической форме (такую ОО-проекцию также будем называть канонической).*

ДОКАЗАТЕЛЬСТВО. Доказываем индукцией по преобразованию ОО-определений из $TBox_o$ в каноническую форму. Основным шагом преобразования является подстановка следующего вида: если $C_i \equiv E_i \in TBox_o$, то заменяем ОО-определение $C_j \equiv C_i \sqcap E_j$ в $TBox_o$ на $C_j \equiv E_i \sqcap E_j$. Этот процесс обязательно завершится получением канонического $TBox'_o$, поскольку $TBox_o$ является ациклическим множеством. □

Аналогичную каноническую форму введем для AOOL-программ. Будем говорить, что AOOL-программа находится в канонической форме, если любой класс $\mathtt{c}$ имеет в ней определение вида

$$\begin{array}{l}\mathtt{class\ c}(\bar{x})\ \mathtt{=} \\ \quad \lambda\mathtt{self.[a_1{:}c_1} \mapsto \mathtt{e_1,\ a_2{:}c_2} \mapsto \mathtt{e_2,\ \ldots,\ a_n{:}c_n} \mapsto \mathtt{e_n]}\end{array}$$

не зависящее от суперклассов.

Лемма 2. *В любой AOOL-программе $\mathcal{P}$ иерархия классов по наследованию является ацикличной.*

ДОКАЗАТЕЛЬСТВО. Следует из того, что по определению ОО-проекции $TBox_o$ является ацикличным множеством, и из того, что отображение $\overset{\mathtt{tr}}{\to}$ не нарушает эту ацикличность. □

Лемма 3. *Для любой AOOL-программы $\mathcal{P}$, содержащей определения классов $\mathtt{c}_1, \ldots, \mathtt{c}_k$, существует каноническая AOOL-программа $\mathcal{P}'$, содержащая определения классов $\mathtt{c}'_1, \ldots, \mathtt{c}'_k$, такая, что $\mathcal{P} \vdash \mathtt{o} : \tau_{\mathtt{c}_i}$ тогда и только тогда, когда $\mathcal{P}' \vdash \mathtt{o} : \tau_{\mathtt{c}'_i}$, где $\mathtt{o}$ — произвольный объект, а $\tau_{\mathtt{c}_i}$ и $\tau_{\mathtt{c}'_i}$ — структурные типы классов $\mathtt{c}_i$ и $\mathtt{c}'_i$, соответственно.*

ДОКАЗАТЕЛЬСТВО. По лемме 2 система классов в $\mathcal{P}$ ациклична, т. е. представляет собой иерархию. Перестройка программы $\mathcal{P}$ в программу $\mathcal{P}'$ проводится индукцией по иерархии наследования. Ввиду ацикличности, в $\mathcal{P}$ обязательно должны присутствовать определения канонических классов, у которых нет суперклассов. На каждом следующем шаге для перестройки выбираем такие классы, которые зависят только от канонических классов. В каждом таком классе $\mathtt{c}_i$ удаляем блок с описанием суперклассов и расширяем запись в теле определения класса содержимым записей суперклассов. Получившийся канонический класс $\mathtt{c}'_i$ будет представлять $\mathtt{c}_i$ в программе $\mathcal{P}'$.

Пусть $\mathcal{P} \vdash \mathtt{o} : \tau_{\mathtt{c}_i}$ для некоторого объекта $\mathtt{o}$. Это означает, что существует вывод $\mathtt{o} : \tau_{\mathtt{c}_i}$ в исчислении структурных типов. В этом случае вывод $\mathcal{P}' \vdash \mathtt{o} : \tau_{\mathtt{c}'_i}$ получается перестройкой вывода $\mathcal{P} \vdash \mathtt{o} : \tau_{\mathtt{c}_i}$ по следующей схеме. Поскольку все поля объектов класса $\mathtt{c}_i$, получаемые через наследование, присутствуют в определении $\mathtt{c}'_i$ явно, группы правил вывода $\tau_{\mathtt{fields}}$ для объектов класса $\mathtt{c}_i$ (по одному на каждый суперкласс) будут соответствовать в $\mathtt{c}'_i$ одному применению правила $\tau_{\mathtt{fields}}$ сразу для всех полей, относящихся к классу $\mathtt{c}_i$, а правило $\tau_{\mathtt{class}}$ будет применяться только к выражениям вида $\mathtt{o}_j : \mathtt{c}_j$ (за исключением самого $\mathtt{o} : \tau$, поскольку τ может быть произвольным структурным типом). Это будет происходить, потому что структурные типы канонических классов не содержат имен суперклассов. Остальные сегменты вывода остаются при перестройке неизменными.

В обратную сторону лемма также доказывается перестройкой вывода. Если имеется вывод $\mathcal{P}' \vdash \mathtt{o} : \tau_{\mathtt{c}'_i}$, и класс $\mathtt{c}_i$ имеет суперклассы $\mathtt{c}_i^1, \ldots, \mathtt{c}_i^h$ в программе $\mathcal{P}$, то единственное применение правила $\tau_{\mathtt{fields}}$ к полям класса $\mathtt{c}'_i$ будет соответствовать в общем случае $h+1$ применениям правила $\tau_{\mathtt{fields}}$ к полям класса $\mathtt{c}_i$. При этом правило $\tau_{\mathtt{class}}$ будет осуществлять переход от класса к суперклассам с актуализацией полей, определяемых в суперклассах. □

Пусть $\mathcal{V}_o = \langle \mathcal{C}_o, \mathcal{R}_o, \mathcal{P}_o, \mathcal{O}_o \rangle$ — словарь, $\mathcal{K}_o$ — OO-проекция в этом словаре, а $\mathcal{P} = \mathtt{tr}(\mathcal{K}_o)$ — AOOL-программа, соответствующая $\mathcal{K}_o$. Для произвольного OO-концепта E в словаре $\mathcal{V}_o$ и произвольного структурного типа τ программы $\mathcal{P}$ обозначим

$$\mathcal{O}_E = \{O \mid O \in \mathcal{O}_o, \mathcal{K}_o \models E(O)\},$$
$$\mathtt{O}_\tau = \{\mathtt{o} \mid O \xrightarrow{\mathtt{tr}} \mathtt{o} \text{ для некоторого } O \in \mathcal{O}_o \text{ и } \mathcal{P} \vdash \mathtt{o} : \tau\}.$$

Лемма 4. *Пусть $\mathcal{K}_o$ — каноническая OO-проекция и $\mathcal{P} = \mathtt{tr}(\mathcal{K}_o)$ — соответствующая ей AOOL-программа. Тогда $\mathcal{P}$ также каноническая, и для любого примитивного OO-концепта E выполняется $\mathtt{O}_{\mathtt{tr}(E)} = \mathtt{tr}(\mathcal{O}_E)$.*

ДОКАЗАТЕЛЬСТВО. Каноничность $\mathcal{P}$ проверяется непосредственно — через анализ правила $\xrightarrow{\mathtt{tr}}$ для OO-определений.

Приведем набросок доказательства второй части леммы. Первым шагом перейдем от OO-концепта E к эквивалентному ему каноническому концепту E', в котором вхождения концептов C_i в E заменены на определения C_i из $TBox_o$. Поскольку все определения из $TBox_o$ канонические, то E' будет иметь вид

$$E' = \forall R_1.B_1 \sqcap \ldots \sqcap \forall R_k.B_k \sqcap \forall P_1.D_1 \sqcap \ldots \sqcap \forall P_l.D_l.$$

Очевидно, что $\mathcal{K}_o \models E \equiv E'$. По лемме 3 структурные типы $\tau = \mathtt{tr}(E)$ и $\tau' = \mathtt{tr}(E')$ также эквивалентны в $\mathcal{P}$.

Доказательство леммы базируется на циклическом применении следующей процедуры, которая показывает, что если выполняется $\mathcal{K}_o \models \forall R.B(O)$, причем $ABox_o$ содержит утверждения $R(O, O_1), \ldots, R(O, O_k)$, то в соответствующем τ-выводе мы можем перейти от доказательства $\mathtt{o}\!:\!\{\mathtt{r}\!:\!\mathtt{list\text{-}b}\}$, где $\mathtt{o} = \mathtt{tr}(O)$, $\mathtt{r} = \mathtt{tr}(R)$ и $\mathtt{b} = \mathtt{tr}(B)$, к доказательству $\mathtt{oi}\!:\!\mathtt{b}$, где $\mathtt{oi} = \mathtt{tr}(O_i)$. Возможны два случая.

Случай $k = 0$. В этом случае по определению $\mathtt{tr}$ запись объекта $\mathtt{tr}(O)$ будет содержать поле $\mathtt{r} \mapsto \mathtt{nil}$. Тогда доказательство по этой ветке осуществляется применением правила $\tau_{\mathtt{nil}}$.

Случай $k > 0$. В этом случае по определению отображения $\mathtt{tr}$ запись, определяющая объект $\mathtt{o} = \mathtt{tr}(O)$, будет содержать поле $\mathtt{r} \mapsto \mathtt{list\text{-}b}\langle \mathtt{o1}, \ldots, \mathtt{ok} \rangle$. Чтобы явно получить эту запись, применяем к $\mathtt{o}\!:\!\{\mathtt{r}\!:\!\mathtt{list\text{-}b}\}$ правило τ_{value}. Затем последовательным применением правила τ_{class} к списку $\mathtt{list\text{-}b}$ мы придем (по нескольким веткам вывода) к необходимости доказать, что $\mathcal{P} \vdash \mathtt{oi} : \mathtt{b}$, $i = \overline{1,k}$.

Таким образом мы придем к ситуации, в которой к объекту O_i можно применить ту же процедуру, которая только что была применена к объекту O. Последовательным применением этой процедуры (с использованием при необходимости правила $\tau_{\mathtt{fields}}$) обеспечивается построение необходимого нам вывода в τ-исчислении. В случае если $ABox_o$ содержит замкнутые цепочки объектов вида $R_1(O_1, O_2), \ldots, R_k(O_k, O_1)$, то неограниченное применение процедуры, описанной выше, приведет к бесконечным веткам дерева вывода. Во избежание этого применяется правило τ_{gfp}, которое соответствует семантике наибольшей неподвижной точки, а значит допускает системы объектов предметной области, представляющие собой ориентированные графы с циклами. Это необходимо, поскольку в $ABox_o$ ОО-проекций объекты вместе с ролями формируют в общем случае конечные ориентированные графы с циклами.

Доказательство в другую сторону основывается на следующем рассуждении. Пусть для некоторого τ' выводится $\mathcal{P} \vdash \mathtt{o} : \tau'$, где $\mathtt{o} = \mathtt{tr}(O)$. Предположим, что τ' содержит компонент $\mathtt{r}\!:\!\mathtt{list\text{-}b}$. Это значит, что если τ' получен из канонического ОО-концепта E' применением отображения $\mathtt{tr}$, то E' должен содержать компонент $\forall R.B$. Далее для объекта $\mathtt{o}$ имеются две возможности: его запись содержит компонент $\mathtt{r} \mapsto \mathtt{nil}$ либо $\mathtt{r} \mapsto \mathtt{list\text{-}b}\langle \mathtt{o1}, \ldots, \mathtt{ok} \rangle$. В первом случае это означает, что $ABox_o$ не содержит ни одного утверждения вида $R(O, O')$, а значит, в силу согласованности интерпретаций со словарем и по определению ограниченного квантора всеобщности, $\mathcal{K}_o \models \forall R.B(O)$. Во втором случае, по определению $\mathtt{tr}$, для всех O_i, $O_i \overset{\mathtt{tr}}{\to} \mathtt{oi}, i = \overline{1,k}$, должно выполняться $R_i(O, O_{ij}) \in ABox_o$ и $\mathcal{K}_o \models B(O_i)$. □

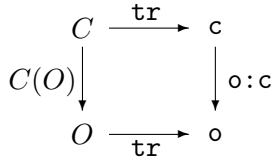
Доказательство теоремы 1 базируется на следующем утверждении, которое имеет и самостоятельное значение, поэтому также оформлено в виде теоремы. Пусть $\mathcal{K}_o$ — ОО-проекция в словаре $\mathcal{V}_o = \langle \mathcal{C}_o, \mathcal{R}_o, \mathcal{P}_o, \mathcal{O}_o \rangle$ и $\mathcal{K}_o \overset{\mathtt{tr}}{\to} \mathcal{P}$.

Теорема 2. *Для произвольного ОО-концепта E выполняется* $\mathtt{O}_{\mathtt{tr}(E)} = \mathtt{tr}(\mathcal{O}_E)$.

ДОКАЗАТЕЛЬСТВО. Следует из лемм 1, 3 и 4. □

Доказательство теоремы 1 является прямым следствием теоремы 2. Заметим, что из

теорем 1 и 2 следует коммутативность следующей диаграммы для любых C и O:



Теорема 1 оправдывает нашу общую идею о том, что OO-проекции со стандартной семантикой дескриптивной логики ведут себя таким же образом, что и система типов объектно-ориентированного языка.

§ 4. Реализация

Адекватность и эффективность подхода, рассмотренного в этой статье, была проверена практически — через реализацию OntoBox [12], системы, которая фокусируется на работу с данными на уровне OO-проекций. Благодаря специальному языку запросов OntoBox QL [12], OntoBox имеет обратную совместимость с гораздо более сложными дескриптивными логиками. Семантика языка запросов строится на базе дескриптивной логики $\mathcal{SHOIN}(D)$ [3]. Обоснованность использования $\mathcal{SHOIN}(D)$ мы обсуждали выше.

Чтобы оценить эффективность OntoBox'а как реализации OO-проекций, мы использовали несколько задач реальной сложности, в частности, известную биологическую таксономию NCBI [11], которая описывает имена всех организмов, присутствующих в генетических базах данных и имеющих по крайней мере одну последовательность нуклеотидов или протеинов. Эта таксономия содержит 482 960 объектов. OntoBox отыскивал цепочки узлов определенной длины в дереве таксономии. Для сравнения те же запросы были выполнены в SQL в исходной базе данных, работающей под СУБД MySQL 5.0.51a. Некоторые результаты: OntoBox нашел 471 762 цепочки длины 5 за 6,62 с, тогда как MySQL затратил на это 42,34 с. Для длины 30: 30 736 цепочек были найдены за 18,94 с (OntoBox) и 148,56 с (MySQL). Мы также проверили (вручную) технику компиляции для OntoBox QL. Результаты компилируемого кода — 0,99 с для длины 5 и 2,36 с для длины 30. Более подробно язык рассматривается в [12].

Эти и другие результаты показывают, что на простом уровне логики могут работать чрезвычайно эффективно. В частности, на уровне OO-проекций у нас есть реализация очень быстрой системы управления объектно-ориентированными данными. Но это не просто объектная БД. Данные и знания, хранящиеся и обрабатывающиеся на уровне OO-проекций, корректно встраиваются в общий логический контекст и, таким образом, могут служить нижним уровнем сложных моделей знаний. В частности, OntoBox может быть использован для разработки «каркасов» баз знаний, в которых хранятся наиболее простые и существенные данные о предметной области.

И, конечно, технология, рассмотренная в данной статье, может работать во многих веб- и других информационных проектах, использующих данные и знания, представленные в объектно-ориентированных моделях. Мы также проверили, могут ли «обычные» пользователи (например, студенты) успешно работать с OntoBox'ом. Результаты этих

тестов оказались весьма многообещающими.

Поскольку много практических задач лучше всего решаются при помощи объектно-ориентированного подхода и через конструирование объектно-ориентированных моделей, возможности, предоставляемые ОО-проекциями, весьма перспективны. Хорошо известны трудности, с которыми сталкиваются разработчики баз данных, когда им приходится погружать в БД объектные модели. Иногда разработчики БД-проектов вынуждены создавать своего рода «обертки» поверх реляционных моделей и под конкретные реализации СУБД.

Более того, существует разнообразное программное обеспечение, ориентированное на общее решение этой проблемы. Здесь можно упомянуть Hibernate [13] и ADO.NET Entity Framework [14]. Например, Hibernate «позволяет отобразить объектно-ориентированные модели (в Java) в реляционные базы данных». Он предоставляет инструмент для создания адаптера от ОО модели к реляционной модели и обратно для более чем 20 различных диалектов СУБД. Во многих задачах такая возможность значительно упрощает разработку проектов, имеющих объектно-ориентированную природу. Но дополнительный слой между ОО моделью и системой хранения значительно усложняет общую схему. Пользователь не имеет полного контроля над ситуацией, и испытывает трудности при отладке. Несмотря на это, данная технология очень популярна.

С другой стороны, OntoBox имеет непосредственное и прямое отображение в модель данных объектно-ориентированного языка. И, что также существенно, OntoBox, как компонента логической архитектуры, является частью намного более общей формальной системы, базирующейся на дескриптивной логике $\mathcal{SHOIN}(D)$. Язык запросов OntoBox QL играет здесь две роли. На уровне OntoBox он играет роль, аналогичную той, что SQL играет в реляционных базах данных. На уровне же всей логической архитектуры OntoBox QL обеспечивает связи между компонентами архитектуры.

Мы также использовали OntoBox для разработки ряда естественно-научных исследований и проектов, связанных с озером Байкал. В частности, были разработаны междисциплинарные онтологии об озере (включая онтологию географических объектов Байкала, онтологию «Популяция планктона озера Байкал», 3D модель озера и т. д.).

Заключение

Исследование, представленное здесь, является частью общего проекта Meta2, имеющего целью разработку интегрированной среды по управлению данными/знаниями и веб-программированием, основанными на дескриптивных логиках. Вместе с [12] оно формирует базу для дальнейшей деятельности. На теоретическом уровне мы планируем продолжить структуризацию дескриптивных логик и исследовать взаимодействие логических архитектур с другими формальными методами и техниками. Взаимодействие между очень быстрой системой OntoBox и более сложными системами вывода также является для нас очень интересной темой. На экспериментальном уровне мы запустили проект клиент-серверной версии OntoBox'a с программным интерфейсом для таких популярных сред разработки, как Java, Javascript, Ruby, .NET и др. Мы планиру-

ем сформировать продвинутую среду для разработки распределенных информационных ресурсов, базирующуюся на наших технологиях. В ближайшем будущем также планируем сфокусироваться на приложениях, связанных с междисциплинарными исследования озера Байкал и некоторыми приложениями, работающими с «продвинутой» обработкой мультимедиа информации.

Список литературы

1. The Semantic Web // http://www.w3.org/2001/sw/

2. Web Ontology Language (OWL) // www.w3.org/2004/OWL

3. *Horrocks I., Patel-Schneider P. F.* Reducing OWL Entailment to Description Logic Satisfiability / Eds. D. Fensel, K. Sycara, J. Mylopoulos // Proc. of the 2003 Int. Semantic Web Conference (ISWC 2003). Number 2870 in Lecture Notes in Computer Science. P. 17–29. Springer.

4. *Goncharov S., Ershov Yu., Sviridenko D.* Semantic Programming // 10^{th} World Congress Information Processing'86. Dublin, 1986. P. 1093–1100.

5. *The Description* Logic Handbook: Theory, Implementation, and Applications / Eds. F. Baader, D. Calvanese, D. L. McGuinness et al. Cambridge University Press, 2003.

6. *Borgida A., Patel-Schneider P. F.* A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic // J. of Artificial Intelligence Research. 1994. No. 1 (June 1994). P. 277–308.

7. *Cardelli L.* A Semantics of Multiple Inheritance // Information and Computation. 1988. Vol. 76. No. 2–3 (February/March 1988). P. 138–164.

8. *Cook W., Palsberg J.* A Denotational Semantics of Inheritance and Its Correctness // ACM SIGPLAN Notices. 1989. Vol. 24. No. 10. P. 433–443.

9. *Hense A. V.* Denotational Semantics of an Object-Oriented Programming Language with Explicit Wrappers // Information and Computation. 1993. Vol. 5. No. 3 (May). P. 181–207.

10. *Scott D.* Data Types as Lattices // SIAM Journal on Computing. 1976. Vol. 5. No. 3. P. 522–586.

11. The NCBI Entrez Taxonomy // http://www.ncbi.nlm.nih.gov/sites/entrez?db=taxonomy

12. *Malykh A., Mantsivoda A.* A Query Language for Logic Architectures // Proc. of Int. Conf. on Perspectives of System Informatics'09. Novosibirsk, 2009. P. 212–220. http://meta2project.org/ru/psi09.pdf

13. Hibernate: Mapping an Object-Oriented Domain Model to a Relational Database // https://www.hibernate.org/

14. ADO.NET Entity Framework // http://www.microsoft.com/sqlserver/2008/en/us/ado-net-entity.aspx

Материал поступил в редколлегию 30.05.2009

Адреса авторов

МАЛЫХ Антон Александрович
РОССИЯ, 664003, Иркутск
пр. К. Маркса, 1
Иркутский государственный
университет, ИМЭИ
e-mail: malykh@baikal.ru

МАНЦИВОДА Андрей Валерьевич
РОССИЯ, 664003, Иркутск
пр. К. Маркса, 1
Иркутский государственный
университет, ИМЭИ
e-mail: andrei@baikal.ru

УЛЬЯНОВ Владимир Сергеевич
РОССИЯ, 664003, Иркутск
пр. К. Маркса, 1
Иркутский государственный
университет, ИМЭИ
e-mail: lenin@baikal.ru

The purpose of this work is to explore a first boundary value problem for equations of mixtures of compressible viscous fluids in the steady three-dimensional case.

UDC 512.816

Kyrov V. A., Muradov R. M. Some of Transformation Groups and Their Invarians // Vestnik, Quart. J. of Novosibirsk State Univ., Series: Math., mech. and informatics. 2009. Vol. 9. No. 3. P. 54–63.

In this paper we establish a connection between physical structure of rank $(n+1, 2)$ and transformation Lie groups. We find invarians of transformation Lie groups for 1-metric and 2-metric of physical structures.

UDC 510.62:004.82

Malykh A. A., Mantsivoda A. V., Ulianov V. S. Logic Architectures and the Object Oriented Approach // Vestnik, Quart. J. of Novosibirsk State Univ., Series: Math., mech. and informatics. 2009. Vol. 9. No. 3. P. 64–85.

Description logics have strong potential to enhance data and knowledge management on the World Wide Web. The Semantic Web has converted this potential into a harmonious approach. But some obstacles make its influence on common practice quite limited. In particular, logic is too heavy for solving the majority of 'everyday' web development tasks, and too complicated for conventional developers. In this paper we introduce some formal methods based on the idea of logic architectures, which are aimed at solving this problem.

UDC 517.925.54 + 517.929

Matveeva I. I., Popov A. M. On properties of solutions to one system modeling a multistage substance synthesis // Vestnik, Quart. J. of Novosibirsk State Univ., Series: Math., mech. and informatics. 2009. Vol. 9. No. 3. P. 86–94.

The Cauchy problem for a system of ordinary differential equations modeling a multistage substance synthesis is considered. We study properties of the last component of its solution, describing the concentration of the synthesis product, as a function of the parameter τ characterizing the total time of the synthesis process. The continuous dependence on τ is established, estimates for the continuity module are obtained. We prove the uniform convergence as $\tau \to 0$; moreover, the limit function is a solution to the Cauchy problem for one ordinary differential equation

UDC 519.716

Panteleyev V. I. The criteria of completeness for redefining boolean function // Vestnik, Quart. J. of Novosibirsk State Univ., Series: Math., mech. and informatics. 2009. Vol. 9. No. 3. P. 95–114.