

Г. М. Сергиевский

Ф15-91

О ПРИРОДЕ ЗНАНИЯ — РЕФЛЕКТОРНЫЙ ПОДХОД

Москва 1991

ГОСУДАРСТВЕННЫЙ КОМИТЕТ СССР ПО НАРОДНОМУ ОБРАЗОВАНИЮ

МОСКОВСКИЙ ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ
ИНЖЕНЕРНО-ФИЗИЧЕСКИЙ ИНСТИТУТ

Г.М.Сергиевский

О ПРИРОДЕ ЗНАНИЯ – РЕФЛЕКТОРНЫЙ ПОДХОД

Препринт ОI5-9I

УТВЕРЖДЕНО

РЕДСОВЕТОМ ИНСТИТУТА

Библиотека
МИФИ

МОСКВА I99I

УДК 519.71.

Сергиевский Г.М. О природе знания - рефлекторный подход.
М.:Препринт/МИФИ, 015-91,1991.-24 с.

Описан универсальный метод спецификации семантики языков представления знаний, приведены примеры спецификации.

© Московский инженерно-физический институт, 1991

ВВЕДЕНИЕ

Несмотря на очевидные успехи в развитии экспертных систем, следует признать, что они мало что дают для продвижения в таких проблемах искусственного интеллекта, как использование естественного языка как языка представления знаний для ЭВМ; реализация полномасштабного машинного перевода; построение решателей проблем, способных синтезировать программу по ее спецификации; разработка алгоритмов машинного обучения, способных решать перечисленные задачи по примерам их решения и т.п.

Одной из причин отсутствия значительного прогресса в перечисленных областях является, по мнению автора, отсутствие ясного представления о природе знания. Достаточно проанализировать типичные определения знания, чтобы понять, что мы не имеем конструктивной концепции знания, одинаково пригодной для определения знаний, представленных как в процедурной (например, в виде алгоритмов), так и в декларативной (например, в виде формул языка первого порядка) форме. Например, в [I] знаниями называют "хранимую в ЭВМ информацию, формализованную в соответствии со структурными правилами, которую ЭВМ может автономно использовать при решении проблем по таким алгоритмам, как логический вывод". Отмечая безусловную справедливость этих утверждений, следует признать, что они все-таки недостаточно конструктивны для использования в качестве аппарата работы со знаниями, представимыми во всем их реальном разнообразии. В виду необозримости проблемы в общей постановке в данной работе ограничимся рассмотрением ее детерминированного подмножества.

По мнению автора [2] , подобная концепция должна обеспечить решение трех фундаментальных задач.

I. Разработка универсального метода спецификации произвольных языков (от алгоритмических до естественных), используемых для представления знаний.

2. Разработка метода реализации языков по их спецификациям, пригодного для практического создания баз знаний в ЭВМ на них.

3. Разработка методов преобразования знаний, представленных на любом реализованном языке.

Остановимся подробнее на последней задаче. Преобразования знаний можно разделить на три группы: эквивалентные преобразования, преобразования специализации и преобразования обобщения. Разумеется, без обращения к формальной модели знаний невозможно строго определить эти преобразования. Однако на интуитивном уровне они достаточно прозрачны. Например, если имеются знания в виде общего метода решения некоторого класса задач, и дополнительно представлены знания, ограничивающие класс решаемых задач; то преобразование общего метода с учетом дополнительной информации в целях повышения его эффективности может рассматриваться как преобразование специализации. Выдвигается тезис, что без разработки подобных преобразований невозможна эффективная реализация существенно декларативных языков.

Преобразования обобщения решают обратную задачу – нахождение общего метода решения широкого класса задач, если известны частные методы для подмножеств задач, принадлежащих к этому классу. Без разработки подобных методов невозможен прогресс в машинном обучении.

Наконец, без выполнения эквивалентных преобразований знаний невозможно, решение проблем, понимаемых как решение задач в отсутствии известного алгоритма их решения. Решение проблем в этом случае есть эквивалентное преобразование базы знаний с целью приведения ее к виду, для которого существует алгоритм решения задачи. Основная задача здесь состоит в разработке методов автоматического поиска эквивалентных преобразований знаний, представленных на некотором языке по известной спецификации языка.

Необходимо напомнить, что все перечисленные задачи должны решаться для работы со знаниями, представленными на произвольном языке представления знаний (ЯПЗ).

Конструктивная концепция знаний должна также позволить формализовать процесс накопления знаний, исследование знаний на полноту, непротиворечивость и т.п.

Ключевой вопрос, на который необходимо ответить в самом начале построения конструктивной концепции знания – что мы

понимаем под знанием на неформальном уровне. Дальнейшее построение будет базироваться на следующем неформальном утверждении: знание это то, что позволяет решать задачи. Тем самым будем исходить из тезиса, что в основании знания лежат процедуры решения задач. Таким образом, с точки зрения предлагаемого подхода, знания всегда имеют операционную семантику, хотя на более высоких уровнях абстракции процедурный аспект знаний может быть замаскирован.

Следующий принципиальный тезис состоит в том, что для формализации процессов, в которых используются знания, необходимо ввести, как обязательный компонент, систему, которая может воспринимать знания и использовать их для решения задач. Систему будем представлять в виде "черного ящика" с одним входом и одним выходом. Интерпретацию знаний будем проводить, наблюдая поведение системы.

Работа со знаниями всегда протекает во времени и состоит из упорядоченных по времени этапов: вводится программа, чтобы потом к ней обратиться; вводится описание схемы базы данных, чтобы потом вводить данные и далее обратиться к ним с запросами, и т.д. Это означает, что система, которую мы используем для описания работы со знаниями, является "динамической". Именно динамика поведения системы и даст возможность раскрыть сущность знания с точки зрения предлагаемого подхода.

Нас будет интересовать знание, представленное в знаковой форме, поэтому в качестве входных воздействий будем рассматривать сообщения в виде слов (цепочек) символов в алфавите $V_{вх}$, а реакцию системы в виде слов (цепочек) символов в алфавите $V_{вых}$.

Будем нумеровать входные сообщения и выходные реакции в соответствии с порядком их поступления (выработки) $x_1, x_2, \ldots$ ($y_1, y_2, \ldots$) соответственно.

Если при анализе сообщения $x_i = x_i^1 \ldots x_i^m$ нас интересует реакция системы на ввод каждого слова x_i^j, будем рассматривать x_i^j как отдельное сообщение, включая x_i^j, $j = \overline{1, m}$ в общую нумерацию и наоборот. Ограничимся в данной работе рассмотрением детерминированных систем.

Безусловно не всякое сообщение представляет собой знание. Попытаемся, постепенно усложняя структуру системы, выделить момент, когда появится возможность говорить о сообщении как о знании. Рассмотрение начнем с простейшего класса систем – –систем без памяти.

I. Системы без памяти

Систему без памяти можно описать функцией выходов $F(U,x)$ в соответствии с равенством

$$y_n = F(u_0, x_n), \quad n = 1,2,\dots,$$

где $u_0 \in U$ – заданное внутреннее состояние системы, остающееся неизменным в течение всего времени функционирования; U – множество внутренних состояний

$$F : U \times V_{вх}^{*} \to 2^{V_{вых}}$$

Тем самым за счет фиксированности u_0 $F(U,x)$ определяет отношение на $V_{вх}^{*} \times V_{вых}^{*}$, которое будет называться вопросно-ответным отношением. Множество значений $x \in V_{вх}^{*}$, при которых $F(u_0, x)$ определено, обозначим $L_{вх}$ и будем называть входным языком, соответственно $L_{вых}$ – выходным языком.

Очевидно в эту модель укладываются любые программные системы без памяти, т.е. такие, реакция в которых не зависит от предшествующих значений входных воздействий; например, интерпретаторы, в которые требуется одновременно подавать и текст интерпретируемой программы и данные для нее (ограничение, состоящее в рассмотрении систем с одним входом, легко преодалевается, если под входом понимаем n значений входных параметров, в которых разделитель между аргументами считается одним из символов алфавита $V_{вх}$).
Системы без памяти соответствуют в программировании строго функциональным языкам без побочных эффектов. О языке $L_{вх}$ можно говорить как о "нерасчлененном" языке, в котором и "программы" и "данные" должны составлять единый текст.

Программой (установкой, утверждением) формально можно назвать любую совокупность подцепочек, составляющих входное "сообщение", остающуюся неизменной для некоторого множества сообщений, соответственно " данными" для этой "программы "

являются остальные подцепочки сообщения. Практически в большинстве случаев деление входного сообщения на "программу" и "данные" может быть сделано вполне естественно. К этому вопросу мы еще вернемся в следующих разделах.

Нерасчлененный язык и реализующая его система без памяти обладают рядом существенных недостатков. Например, если сообщения x_n и x_{n+1} отличаются только " данными", но имеют "одинаковую программу"; мы обязаны x_n и x_{n+1} вводить полностью. Помимо невозможности сократить объем вводимой информации, мы не можем также оптимизировать процесс "исполнения программы", "скомпилировать" ее, например, в язык реализации функции выходов $F(U_0, x)$. Под "компиляцией " понимается формирование остаточной программы P_{rez} в результате частичных вычислений программы P при условии известных компонентов [3] . Тем самым исключается накопление знаний, например в виде библиотеки " скомпилированных" программ, составляющее важнейшее качество развитых программных систем. Однако это преобразование не реализуется исходной системой без памяти. Для обеспечения подобного преобразования мы должны перейти к рассмотрению следующего класса систем – систем с памятью.

2. Системы с памятью

Будем говорить, что система обладает памятью, если реакция на очередное сообщение x_n, и зависит от предшествующих сообщений $x_1, x_2, \ldots, x_{n-1}$ (в понятие реакции мы включаем отсутствие действительной реакции системы).

Структура системы с памятью может быть представлена многими способами, эквивалентными друг другу. Определенную ниже структуру будем называть канонической. Система с памятью задается как шестерка $S = \langle U, X, Y, F, G, u_0 \rangle$,

U – мн-во внутренних состояний;
X – мн-во входных сообщений;
Y – мн-во выходных сообщений;
$F: U \times X \to 2^Y$ – функция выходов;
$G: U \times X \to U$ – функция переходов;
$u_0 \in U$ – начальное состояние,

а связь между параметрами определяется равенствами

$$y_n = F(u_n, x_n),$$
$$u_n = G(u_{n-1}, x_n), \quad n \geq 1.$$

Каждая система с памятью реализует некоторое отображение Ω входных последовательностей в выходные. Если обозначить $[x_1, \ldots, x_n]$ как последовательность входных сообщений $x_1, \ldots, x_n$, то функции $F(U, x)$ и $G(U, x)$ могут быть очевидным образом расширены до $F(u, [x_1, \ldots, x_n])$ и $G(U, [x_1, \ldots, x_n])$, имеющие своим значением выходное сообщение и внутреннее состояние после подачи на вход последовательности входных сообщений $[x_1, \ldots, x_n]$ при начальном значении внутреннего состояния U.

Сообщение, ввод которого изменяет реакцию на будущие сообщения, будем называть программой, установкой, утверждением, БЗ. Сообщение, ввод которого вызывает выработку реакции, — данным, стимулом, запросом. Стимул, не являющийся установкой, будем называть строгим стимулом (с-стимулом), аналогично, установка, не являющаяся стимулом, строгой установкой (с-установкой).

$$x_n - \text{с-стимул} \Leftrightarrow F(u_n, x_n) \neq \emptyset \;\&\; G(u_n, x_n) = u_n$$

$$x_n - \text{с-установка} \Leftrightarrow F(u_n, x_n) = \emptyset \;\&\; G(u_n, x_n) \neq u_n.$$

В каждом состоянии системы u_n существует некоторое множество стимулов $L_S(u_n)$ (область определения функции выходов в состоянии u_n). Множество реакций на множество $L_S(u_n)$ в состоянии u_n будем обозначать $L_R(u_n)$:

$$L_R(u_n) = \{ y \mid F(u_n, x) = y \;\&\; x \in L_S(u_n) \}.$$

Таким образом, состояние u_n определяет вопросно-ответное отношение, заданное функцией выходов на

$$L_S(u_n) \times L_R(u_n).$$

С рассматриваемой точки зрения сообщение x_n представляет собой знание, если это сообщение меняет будущее поведение системы (внутреннее состояние системы), т.е. является установкой.

Данное определение знания универсально и одинаково пригодно для любых языков (и не только языков), используемых как языки представления знаний (от алгоритмических до

естественных). Этот подход может использоваться и для спецификации языков, определяя язык посредством определения системы.

Очевидно, что в терминах программирования системе с памятью соответствует программирование с побочным эффектом. Поэтому программирование систем для работы со знаниями не может быть выполнено целиком в рамках чисто функционального стиля без побочных эффектов.

Дальнейшим развитием структуры систем с памятью являются системы, допускающие самообращение, и системы с автономным преобразованием внутреннего состояния.

Система, допускающая самообращение, для которой $V_{вх} = V_{вых}$, имеет возможность обращаться к самой себе в ходе интерпретации любого внешнего или внутреннего сообщения.

Система с автономным преобразованием внутреннего состояния содержит трансформационную машину (ТМ), которая может автономно преобразовывать внутреннее состояние системы (в качестве ТМ может выступать и сама система). Данное расширение позволяет реализовать преобразования знаний, отображенных во внутреннее состояние системы, в частности смешанные вычисления, повышающие вычислительную эффективность системы; а также преобразования покрытия, реализующие механизм индуктивного обучения.

Будем говорить, что система (в состоянии u) монотонна по установке x, если ввод этой установки расширяет область определения вопросно-ответного отношения, не меняя его на старой области.

Установка("программа") и стимул("данные") не обязательно должны подаваться последовательно друг за другом (как это необходимо в системе без памяти). Совокупность всех введенных установок будем называть текущим контекстом. Он отображается во внутреннее состояние и определяет интерпретацию стимулов (и будущих установок).

Множество запросов (стимулов), интерпретируемых системой в контексте, созданном установками $[x_1, \dots, x_n]$ образует язык $L_s([x_1, \dots, x_n]) = L_s(u_n)$, где u_n – внутреннее состояние, в которое перешла система после ввода $[x_1, \dots, x_n]$. Языком запросов будем называть язык L_s.

Множество утверждений (установок), интерпретируемых системой в контексте $[x_1, \ldots, x_n]$, образует язык

$$L_Y([x_1, \ldots, x_n]) = L_Y(u_n), \text{ где } u_n$$

определена выше.

Аналогично языком утверждений (языком представления знаний) будем называть язык L_Y:

$$L_Y = \{ L_Y(u_n) \mid u_n \in U \}.$$

Запрос x_n, принадлежащий языку запросов в данном состоянии u_n $x_n \in L_S(u_n)$, будем называть корректным относительно контекста $[x_1, \ldots, x_n]$. Объединение языков L_S и L_Y будем называть входным языком системы L_{SY}.

Естественный язык естественно разделяется на язык запросов и язык представления знаний. К первому относятся все вопросительные и повелительные предложения, ко второму – – повествовательные. В искусственных языках такое деление иногда замаскировано, однако для спецификации языка предлагаемым методом его необходимо в подобных случаях расширить добавлением новых конструкций, относящихся к запросам (см. разд. 4).

Будем называть систему расширяемой, если вводом соответствующих установок она может реализовать вопросно-ответное отношение, описываемое произвольной частично рекурсивной функцией.

3. Рефлекторное определение знаний

Определение системы с внутренними состояниями позволяет описывать эффект ввода знаний через изменение этого состояния. Данный подход достаточно универсален, так как в его основе лежит весьма общая модель системы. Однако прямое его использование требует определения множества внутренних состояний U, что представляет собой самостоятельную задачу. Во многих случаях можно избежать этого, в определенной степени искусственного шага. Действительно, если ввести понятие системной функции f_n, описывающей в каждый мо-

мент времени связь между входом и выходом, то при программной ее реализации сама совокупность инструкций, реализующих f_n (будем обозначать ее $\langle f_n \rangle$), может рассматриваться как внутреннее состояние системы, роль которой сводится к интерпретации инструкций. Обозначив операцию аппликации функции f_n, описываемой набором инструкций $\langle f_n \rangle$, к исходным данным x в виде $\langle f_n \rangle (x)$, можем записать

$$y_n = \langle f_n \rangle (x_n) ,$$
$$\langle f_{n+1} \rangle = G (\langle f_n \rangle, x_n) .$$

Если включить набор инструкций, описывающих функцию G, в набор инструкций функции f_n, которая, например, вырабатывает (при ее применении x_n) "пару" значений – новые инструкции для f_{n+1} и выходную реакцию y_n, то основное соотношение можно записать

$$\langle f_n \rangle (x_n) = \langle f_{n+1} \rangle : y_n ,$$

где ":" – операция образования "пары".

По другому

$$y_n = t (\langle f_n \rangle (x_n)) ,$$
$$\langle f_{n+1} \rangle = h (\langle f_n \rangle (x_n)) ,$$

где h и t – функции выделяющие голову и хвост "пары".

Если мы включим операцию замещения $\langle f_n \rangle$ на $\langle f_{n+1} \rangle$ после ввода сообщения x_n в набор инструкций системной функции f_n, то основное соотношение еще упрощается

$$y_n = \langle f_n \rangle (x_n) ,$$

где $\langle f \rangle$ – обозначает набор инструкций системной функции, который теперь зависит от номера n. Следует отметить, что такая форма описания динамической системы увеличивает выразительные возможности, так как сами инструкции по модификации системной функции становятся доступными для модификации как любые другие инструкции.

С конкретной техникой реализации данного подхода познакомимся в следующем разделе, а в заключение приведем обоснование бихевиористской терминологии. Системная функция, поскольку она определяет отображение множества стимулов в множество реакций (а также установок в тут же исполняемые прав[illegible]

фикации этого отображения), может рассматриваться как определение совокупности"рефлексов", которые реализует система. Отсюда, способ определения знания посредством указания системной функции (совокупности рефлексов), которая ему сопоставляется, будем называть " рефлекторным определением знания" (РОЗ).

4. Язык описания расширяемых систем ESDL

Сформулируем требования к языку описания f .

Первое: в виду того что на один вход может быть выработано более одного выхода, язык должен позволять описывать многозначные функции. Второе: язык должен обладать средствами описания модификаций определенных с помощью него функций, т.е. обладать свойствами объектного языка и метаязыка. В [5] такие языки называются амальгамируемыми.

Для указанных целей может быть использован традиционный аппарат лямбда-исчислений, применяемый для задания функциональных языков [6] , расширенный в соответствии с перечисленными требованиями.

Определение f задается упорядоченной последовательностью клозов, каждый из которых описывает вариант выполнения функции f . Клоз представляет собой лямбда-выражение.

Для большой выразительной мощности в качестве связанной переменной лямбда-выражения разрешим использовать образец O^i, содержащий в общем случае неозначенные переменные. Таким образом определение имеет вид

$$f = \lambda O^1 .\ body^1$$
$$f = \lambda O^2 .\ body^2$$
$$\dots$$
$$f = \lambda O^n .\ body^n$$

Подача на вход системы сообщения x_n интерпретируется как аппликация f к x_n .Для реализации многозначности используется техника бектрейкинга, что приводит к расширенной интерпретации аппликации. Приведем ее операционное описание. Оно вполне традиционно.

Тело лямбда-выражения $body^k$ имеет вид

$$body^{\kappa} = c_1^{\kappa}, c_2^{\kappa}, \ldots, c_{m_\kappa}^{\kappa} \# e^{\kappa} \quad (m_\kappa \geq 0),$$

где c_i^{κ} – называется целью $i = \overline{1, m_\kappa}$, а e^{κ} – возвращаемым выражением.

Пусть a – цепочка, принадлежащая $L_{вх}$.

Аппликация f к аргументу a (обозначается fa) означает, что ищется первое по порядку лямбда-выражение, образец которого совместим с a. После этого делается попытка выполнить последовательно цели, составляющие тело данного лямбда-выражения. Не раскрывая пока структуру цели, отметим только, что ее выполнение может завершиться успехом и неуспехом. В случае, если выполнение всех целей завершается успехом, вычисляется значение возвращаемого выражения, которое объявляется результатом аппликации. Если выполнение какой-либо цели завершается неуспехом, то происходит "возврат" к выполнению предыдущей цели (делается попытка получить другой результат ее выполнения), и в случае успеха вычисления продолжаются. Если неуспехом завершается выполнение первой цели, то делается попытка найти следующий клоз, образец которого совместим с a. Если такого клоза нет, то фиксируется неуспех аппликации f к аргументу a. Успешно завершенная аппликация f к a вырабатывает одно (первое) значение функции f на аргументе a. Для получения следующего значения fa необходимо зафиксировать "возврат" к данной аппликации. При его поступлении восстанавливается история вычислений, приведшая к выработке предыдущего результата, и "возврат" передается в последнюю успешно выполненную цель. Выработка неуспеха при очередном "возврате" к аппликации fa интерпретируется как исчерпание значений данной аппликации.

Рассмотрим конструкцию "цель". Простая цель представляет собой конструкцию вида

выражение $\Rightarrow$ образец

где выражение содержит не более одной аппликации функции f (самообращение), встроенной функции или лямбда-выражения к данным, построенным с применением операций, определенных над структурами данных. Образец – традиционно представляет собой выражение, построенное только с применением операций, определенных над структурами данных.

Цель выполняется успешно, если найдется значение выражения, сопоставимое с образцом. Дополнительным эффектом является означивание переменных, входящих в образец.

Правая часть в виде одной анонимной переменной может не записываться, т.е. цель может иметь вид

выражение

Данная конструкция используется в тех случаях, когда важен сам факт успешного выполнения аппликации или происходит аппликация встроенной функции, не вырабатывающей значения.

Кроме "простой цели", разрешим использовать "составные цели", имеющие вид

(последовательность целей I ; ...; последовательность целей K). Семантика ее традиционна [4] – при очередном обращении к данной конструкции делается попытка выбрать первую по порядку последовательность целей, все цели которой выполняются успешно.

Конструкция "альтернативное завершение" в виде

(последовательность целей# I # выражение_I ;...;
последовательность целей _ K# выражение_ K) реализует альтернативные варианты завершения вычислений клоза.

К числу важнейших встроенных операций относятся две – – добавление альтернативы (в виде лямбда-выражения) числу альтернатив f и удаление альтернативы в двух вариантах: по "адресу" добавленной альтернативы и по тексту удаляемой альтернативы.

Первая операция имеет вид

$$+ \ (\lambda 0.\ body) \Rightarrow \text{переменная}$$

Ее результатом является добавление альтернативы $\lambda 0.\ body$ и в начало списка альтернатив функции f (при возврате данное действие отменяется). Отметим, что добавляемый клоз

$$\lambda 0.\ body$$

может содержать вхождения глобальных переменных, значения которых должны быть определены к моменту выполнения операции. Значением данной операции является "адрес" добавленной альтернативы. С этим значением возможно единственное действие – удаление добавленной альтернативы по ее адресу. Например, если была выполнена цель

$$+(\lambda 0.\ body) \Rightarrow x_1$$

то после выполнения операции

$$-(x_1)$$

данная альтернатива будет удалена.

Второй вариант удаления альтернативы реализуется операцией

$$-(\lambda 0.\ body)$$

При ее выполнении произойдет удаление альтернативы

$$\lambda 0.\ body$$

если она была добавлена операцией

$$+(\lambda 0.\ body)$$

с теми же значениями глобальных переменных.

Именно эти операции в сочетании с встроенной функцией "отсечение" [4], изображаемой " ! ", обеспечивают средства для описания модификации системной функции, достаточные для систем, работающих со знаниями.

Структуры данных, с которыми работает абстрактный язык ∈SDL, могут быть определены независимо от рассмотренной выше операционной семантики. В качестве них могут использоваться списки, структуры, строки и т.п. Имея в виду ориентацию языка на работу с текстами на языках выберем в качестве структур данных строки символов с единственной операцией конкатенацией. Будем использовать синтаксические образцы. В простейшем случае это означает, что:

1) задана КС-грамматика $G = \langle V_T, V_N, S, R \rangle$;

2) образец 0 имеет вид

$$0 = z_1 T_1 \ \ldots \ T_n z_{n+1}, \quad n \geq 1,$$

где $z_i \in V_{0x}^*$ – цепочки терминальных символов,

$T_i \in V_N$ – нетерминальные символы ;

3) сопоставление успешно, если цепочка x может быть представлена в виде $z_1 x_1 \ \ldots \ x_n z_{n+1}$ таким образом, что x_i выводима из T_i и G или, что то же самое, успешно сопоставлена x_i с какой-либо правой частью правила грамматики для T_i, $i = \overline{1,n}$;

4) значения x_i означают значения связанных переменных образца T_i .

Очевидно, что в общем случае существует множество означиваний, если грамматика недетерминированна, в том числе пустое. Считается, что на этом множестве определен линейный порядок, что дает возможность перебрать все означивания за счет механизма бектрейкинга.

В целях большей наглядности введем "синтаксический сахар" для записи на языке ESDL.

Прежде всего, учитывая что средствами ESDL определяется единственная функция, не будем явно записывать имя этой функции при ее аппликации. Так, вместо $f x_1$ будем записывать x_1 . Отсутствие аппликации – цитированием с помощью " . Так, "abc означает само abc , а не fabc . Конструкцию $\lambda 0.\ body$ будем записывать в виде

$$0 \rightarrow body$$

Переменные записываются с прописной буквы, объектные символы – со строчной. Для обозначения аппликации лямбда-выражения или переменной к операнду для однозначности чтения введем специальный символ "$*$ ".

Описание системной функции на некоторый момент времени определяет входной язык, интерпретируемый системой, хотя это и не значит, что описание машины полностью определяет вопросно-ответное отношение в любой момент времени, которое зависит от введенной информации. Таким образом, описание машины может использоваться для спецификации языка. Рассмотрим применение данного подхода для этой цели.

5. Рефлекторная семантика

Определение языка через определение системы, интерпретирующий этот язык, путем сопоставления каждой конструкции языка правила выработки реакции и/или модификации системной функции называется рефлекторной семантикой данного языка.

Приведем примеры спецификаций некоторых языков или их фрагментов.

5.1. Простой язык программирования SL

SL – простейший язык программирования без блочной структуры, содержащей основные типы операторов, присваивания,

цикла, условный. Будем считать, чть программа на SL представляет собой определение процедуры, к которой можно (не рекурсивно) обратиться по имени, задав ее фактические параметры. Результат вычислений в виде числа выводится на печать обращением к встроенной функции print(x).

Спецификация язнка на языке ESDL содержит описание синтаксиса в терминах КС-грамматик и описание семантики в виде клозов. Описание синтаксиса очевидно.

Синтаксис SL

```
Program → Id '(' Formparlist ')' = Row'.' .
Row     → Operator | Operator ';' Row.
Operator → Id := Ex | while Bex cl Row lc |
                if Bex then Row else Row.
Bex     → Ex = Ex | Ex > Ex
Ex      → Term + Ex | Term | Term - Ex
Term    → Mult * Term | Mult | Term | Ex
Mult    → Id | Const | '(' Ex ')' | print '(' Ex ')'
Formparlist → Id | Id ',' Formparlist
Factparlist → Const | Const ';' Factparlist
```

Семантика SL

```
/*1*/ Id '(' Formparlist ')' = Row → + (Id '(' Factparlist ')' →
        as (Formparlist ';' Factparlist), Row #) #.
/*2*/ as '(' Id ';' Const ')' → Id := Const #.
/*3*/ as '(' Id ';' Formparlist ';' Const ';' Factparlist ')' →
        Id := Const, as(Formparlist ';' Factparlist)#.
/*4*/ Oper ';' Row → Oper, Row #.
/*5*/ Id := Ex → + (? Id →! # ? Ex) #.
/*6*/ While Bex cl Row lc → new ⇒ E1,
        + (E1 → Bex ⇒ T, Row, E1#; # true ), E1#.
```

```
/*7*/  if Bex then Row1 else Row2 →
           (Bex ⇒ T, Row1 #; Row2 # ) #.
/*8*/  ? Const → # " Const.
/*9*/  ?(Term + Ex) → ? Term ⇒ E1, ? Ex ⇒ E2,
           E3 is E1 + E2 # " E3.
/*10*/ ?(Mult * Term) → ? Mult ⇒ E1, ? Term ⇒ E2,
           E3 is E1 * E2 # " E3.
/*11*/ ? '(' Ex ')' → # ? Ex.
/*12*/ ?(Ex1 = Ex2) → ? Ex1 ⇒ E1, ? Ex2 ⇒ E2,
           E1 ⇒ E2 # " T.
/*13*/ ?(Ex1 > Ex2) → ? Ex1 ⇒ E1, ? Ex2 ⇒ E2,
           E1 > E2 # " T.
```

Первый клоз описывает действия системы при вводе текста на языке SL , представляющего собой конструкцию (образец) вида

```
Id '(' Formparlist ')' = Row
```

Любая введенная программа сопоставится с этим образцом, означив синтаксические переменные Id, Formparlist и Row.

Действие состоит в формировании реакции системы на запрос, образец которого имеет вид

```
Id (Factparlist)
```

где Id (глобальная переменная для добавляемого клоза) означена, а Factparlist переменная в образце добавляемого клоза – нет.

Реакция на образец указанного вида состоит в присваивании фактических параметров формальным (операция as , описываемая клозами /*2*/ и /*3*/) и выполнении тела процедуры (заданного глобальной переменной Row), правила описания которого заданы клозами /*4*/ ÷ /*93*/ .

Переменная в языке SL является объектом двух операций – присваивания ей значения и обращения за ее значением. Если первая операция имеет в специфицируемом языке явное обозначение в виде оператора присваивания, то вторая операция явного обозначения не имеет, и обращение к ней идентифицируется по контексту – помещением идентификатора переменной в выражении. Точно также отсутствует явное обозначение операции вычисления значения выражения. При спецификации методом рефлекторной семантики удобно ввести эти операции явно. Обращение за значением выражения Ex (и переменной Id) обозначим $?Ex$ ($?Id$) соответственно.

Тогда рефлекторная семантика оператора присваивания

$$Id := Ex$$

состоит в модификации реакции на запрос

$$?Id$$

сводящейся к выдаче значения выражения Ex (клоз /*5*/). Поскольку реакция на запрос $?Id$ единственная, в целях оптимизации целесообразно использовать встроенную функцию "!".

Цикл реализуется определением клоза с уникальным для каждого цикла образцом в виде константы (создаваемой операцией *new*). Семантика остальных конструкций очевидна.

После ввода утверждения (базы знаний) в виде, например, определения процедуры

```
fact (x) = y:= 1; while x>0 cl y:= y*x;
           x:= x-1 cl; print (y).
```

будет сформирован клоз, обеспечивающий интерпретацию запросов вида $fact(N)$,

где N – натуральное число. При вводе запроса

$$fact(5)$$

будет напечатано 120.

Данный пример демонстрирует синтаксическую ориентированность метода рефлекторной семантики, состоящую в сопоставлении описаниям конструкций входного языка соответствующих действий.

5.2. Язык равенств

Приведем пример спецификации языка равенств арифметических выражений, которые:

допускают разрешение любого равенства относительно каждой входящей в него переменной;

при задании значений некоторых переменных, значения подлежащих определению переменных могут быть выражены через значения последовательным разрешением некоторого числа равенств.

Синтаксическое описание

Равенство Eq понимается как

$$Ex1 = Ex2 .$$

Последовательность равенств $Eq\,list$ как

$$Ex1 = Ex2 \quad или \quad Ex1 = Ex2\ ';'\ Eq\,list .$$

Равенства вводятся конструкцией

$$do\ Eq\,list .$$

Синтаксическое описание Ex совпадает с приведенным выше, дополненным при необходимости описанием базовых функций:

sin, cos, $arcsin$ и т.д. Запрос о значении искомой переменной (целевой запрос) записывается $?\,Id$, начальные присваивания $Id := const$.

Семантика

Семантика языка равенств состоит в описании метода их разрешения относительно любой, входящей в них переменной и представляет собой вариант метода "изоляции" . . Для исключения повторного использования равенства каждому равенству сопоставляется уникальный номер, генерируемый функцией *new*. При поиске равенств, позволяющих выразить значение необходимой переменной через выражение, содержащее другие переменные, запрещается использовать равенство, номер которого содержится в списке использованных номеров.

$/*1*/$: $?\,Id \rightarrow \#\,?\,Id(L1)$.

$/*2*/$ $do(Eq\ ';'\ Eq\,list) \rightarrow do\,Eq, do\,Eq\,list\ \#$.

$/*3*/$ $do(Ex1 = Ex2) \rightarrow new \Rightarrow N,\ Ex1 == Ex2(N),$
$Ex2 == Ex1(N)\ \#$.

```
/*4*/ (Term + Ex) == Ex1 (N) → Term == (Ex1 - Ex)(N),
                    Ex == (Ex1 - Term)(N) #.
/*5*/ (Term - Ex) == Ex1 (N) → Term == (Ex1 + Ex)(N),
                    Ex == (Ex1 + Term)(N) #.
/*6*/ (Mult * Term) == Ex1 (N) → Mult == (Ex1 / Term)(N),
                    Term == (Ex1 / Mult)(N) #.
/*7*/ (Mult / Term) == Ex1 (N) → Mult == (Term * Ex1)(N),
                    Term == (Mult / Ex1)(N) #.
/*8*/ (Sin (Ex)) == Ex1 (N) → Ex == (arc sin (Ex1))(N) #.
/*9*/ Id == Ex (N) → + (? Id (L) → N ∉ L # ? Ex ([NIL]) #.
/*10*/ ?(Term + Ex)(L) → ? Term (L) ⇒ X1, ? Ex (L) ⇒ X2,
                    X3 is X1 + X2 # X3.
/*11*/ ?(Term - Ex)(L) → ? Term (L) ⇒ X1, ? Ex (L) ⇒ X2,
                    X3 is X1 - X2 # X3.
/*12*/ ?(Mult * Term)(L) → ? Mult (L) ⇒ X1, ? Term (L) ⇒ X2,
                    X3 is X1 * X2 # X3.
/*13*/ ?(Mult / Term)(L) → ? Mult (L) ⇒ X1, ? Term (L) ⇒ X2,
                    X3 is X1 / X2 # X3.
/*14*/ ?(Sin (Ex))(L) → ? Ex ⇒ X1, X2 is Sin (X1) # X2.
```

Клоз /*1*/ формирует начальное обращение за значением целевой переменной с пустым списком использованных номеров равенств.

Клозы /*2*/ ÷ /*8*/ вводят набор равенств и выполняют подготовительную работу для изоляции переменных. Клоз /*9*/ "заносит" разрешенное равенство с проверкой возможности его применения в память системы. Клозы /*10*/ ÷ /*14*/ представляют собой просто расширенный вариант интерпретации выражений с учетом списка использованных переменных.

После ввода базы знаний, например,

do $a/\sin A = b/\sin B$;
$b/\sin B = c/\sin C$;

C:= 5 A:= 30 = 90.

на запрос ?α будет выработано значение 2.5.

6. Специализация знаний

Ввод знаний в систему в конечном счете выражается в расширении (в общем случае модификации) набора клозов, определяющих поведение системы. Каждый клоз описывает интерпретацию конструкции, совместимой с образцом в левой части клоза. При этом в ходе интерпретации примера конструкции могут добавляться (модифицироваться) новые клозы. Последнее происходит всегда, если определение некоторой конструкции выполняется через интерпретацию других конструкций, т.е. в режиме расширения. Такой режим работы при интерпретации запроса будет достаточно медленным. Однако он может быть значительно улучшен за счет специализации сформированного после ввода спецификации ЯПЗ, интерпретатора ЯПЗ и языка запросов. Для этого достаточно задать на вход обобщенный запрос с неопределенными значениями синтаксических переменных, и выполнить все вычисления, которые можно сделать без знания конкретных значений этих переменных. Подобный механизм специализации обеспечивается процедурой смешанных вычислений – простейшим типом преобразования знаний.

Например, если мы специфицируем общий вид запроса в разд. 5.I в виде

fact(X)

где X – неозначенная переменная, то после выполнения смешанных вычислений над текущим состоянием базы знаний получим эффективную программу вычисления факториала. Смешанный вычислитель выполнит в данном случае роль компилятора с ЯПЗ в язык реализации. Таким образом смешанный вычислитель будет играть роль планировщика, для которого в системе Приз [7] должна быть написана специальная программа. Отметим, что "автоматический синтез программы" при данном подходе выполняется в гораздо более общих моделях, чем в

системе Приз.

7. Состояние исследований

К настоящему времени получены следующие результаты:

1) разработана практическая версия языка описания расширяемых систем ESDL – язык SOL ;

2) выполнена прототипная реализация SOL в виде компилятора в Пролог на языке ARITY PROLOG ;

3) реализовано несколько стратегий смешанных вычислений для подмножества Пролога, в который компилируется описание на SOL ;

4) средствами SOL специфицирован ряд языков: простой язык программирования SL , язык равенств (с простейшей семантикой), язык продукции типа OPS –5, язык фреймов, фрагменты естественного языка;

5) проведена экспериментальная проверка реализаций перечисленных языков.

Как показал эксперимент, после выполнения смешанных вычислений программ, сформированных в результате ввода текстов на специфицированных языках, удается получить достаточно хорошие характеристики времени выполнения.

Выполненные к настоящему моменту разработки подтвердили практическую эффективность концепции рефлекторного определения знаний в традиционных областях искусственного интеллекта (создание оболочек экспертных систем, создание систем автоматического синтеза программ по моделям, создание систем интеллектуальных аналитических преобразований и др.), а также обоснованность предлагаемых путей решения перспективных задач искусственного интеллекта (реализация естественного языка как языка представления знаний, реализация решателей проблем, машинное обучение).

8. Направления дальнейших исследований

Приведенные результаты дают основания сформулировать следующие предложения по тематике исследований и разработок, реализующих концепцию рефлекторного определения знаний.

1. Промышленная реализация генератора оболочек интеллектуальных систем широкого назначения, позволяющего применять произвольный ЯПЗ.

2. Поисковые разработки для создания системы обработки знаний (СОЗ), предназначенной для человекомашинного решения проблем, способной работать со знаниями, представленными на любых ЯПЗ.

Благодарности

Автор выражает благодарность своим аспирантам Бумагину В.Ю., Чумаченко И.Б., активно участвующим в разработке концепции и программной реализации исследовательского проекта.

Литература

I. С.Осуга. Обработка знаний. Перевод с японского. М.: Мир, I989.

2. Сергиевский Г.М. Об одном подходе к описанию семантики языков представления знаний. Тезисы докл. I-й Всесоюзной конф. по ИИ, 2I-25 ноября, I988, Переславль-Залесский.

3. A. TAKEUCHI, H. FUJITA, Competitive Partial Evaluation, NJC, 6, 1988, p. 259-277.

4. У.Клоксин, К.Меллиш. Программирование на языке ПРОЛОГ М.: Мир, I987.

5. К.Хоггер. Введение в логическое программирование. М.: Мир, I988.

6. M. C. HENSON, Elements of Functional Language, Blackwell Scientific publications. OXFORD

7. Э.Х.Тыугу. Концептуальное программирование. М.: Наука, I984.

Рукопись поступила в издательский отдел 22.04.91.

Ответственный за выпуск Сергиевский Г.М.

Подписано в печать 21.05.91 Формат 60x84 I/I6

П.л.I,5 Уч.-изд.л.I,5 Тираж I00 экз.

Изд.№ 015-91 Заказ 1164 Цена 10 коп.

Типография МИФИ, Каширское шоссе, 3I