ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА, ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ

УДК 007:681.512.2

ОСОБЕННОСТИ ФОРМАЛЬНОГО ИССЛЕДОВАНИЯ ИНТЕЛЛЕКТУАЛЬНОГО ПРОГРАММНОГО ИНСТРУМЕНТАРИЯ

И. Ю. Каширин, д.т.н., профессор кафедры ВПМ РГРТУ; kashirin.i.yu@rsreu.ru
О. И. Каширина, студент РГРТУ, april.helga@mail.ru

Рассматриваются отличительные свойства формализации интеллектуального программного обеспечения. Особое внимание уделяется вопросам оптимизации при разработке интеллектуальных систем. В качестве базового математического аппарата предлагается использовать теоретическую концепцию формальных программных машин, представляющих собой упорядоченное множество прикладных универсальных алгебраических систем. Отдельно рассматриваются особенности интеллектуальных систем, связанные с уровнем знаний, упрощением предметной области, логическим программированием и нечетким представлением знаний. ***Целью работы*** *является анализ особенностей формального исследования известных концепций программных систем искусственного интеллекта с выявлением оптимизирующих возможностей соответствующих формальных программных машин.*

Ключевые слова: *искусственный интеллект, анализ программ, алгоритмические алгебры, инструментальные программные средства, формальные программные машины, оптимизация программ, уровень знаний, обучение и самообучение, метауровневые языковые средства.*

Введение

Наука об искусственном интеллекте [1], появившись в 40-х – 50-х годах прошлого века, имеет в настоящее время явные тенденции к росту своего значения в теории проектирования сложных программных систем. Причины таких тенденций объективны, поскольку искусственный интеллект, имея поначалу в качестве своего приложения упрощенные и ограниченные предметные области, быстро стал полигоном передовых идей в математическом и программном обеспечении ЭВМ. Эта область научно-технической деятельности позволила резко расширить класс задач, решаемых с помощью средств вычислительной техники, поставив в качестве цели исследований моделирование любой мыслительной деятельности человека.

Первыми практическими результатами в разработке методов проектирования интеллектуальных систем стали появление функционального [2] и логического [3] программирования, возникшие с разработкой таких языков, как LISP, Snobol, Refal или Prolog [2, 3, 4]. Инструментальная интеллектуальная система Smalltalk [5], также проектировавшаяся для решения интеллектуальных задач, сыграла решающую роль в становлении объектно-ориентированного проектирования. Идеи наследования свойств, положенные в основу многих современных алгоритмических языков, таких как ADA, CLOS, C++ [7], тоже имеют корни в работах, предопределивших возникновение программного инструментария, направленного на обработку фреймовых структур, представителями которого являются языки FRL, KRL, GUS [8] и др.

Современные исследования искусственного интеллекта затрагивают такие области, как разработка продукционных систем, систем параллельного логического вывода, проектирование нейрокомпьютеров, спецпроцессоров и ориентированных на них программных систем, генетические алгоритмы и онтологические модели представления знаний [5, 6].

Постановка задачи

Интеллектуальные программы имеют в своей основе различные теоретические концепции,

развивающиеся впоследствии в конкретные методы программирования, становление которых приводит к появлению нового программного или языкового инструментария. Вместе с тем в основе каждого такого инструментария находятся формально-языковые средства, предполагающие наличие собственного синтаксиса и семантики, а следовательно, системы объектов и отношений, имеющих свою систему оперирования. Любая система оперирования имеет алгебраические свойства и, следовательно, может быть подвергнута формальному анализу. Это свидетельствует о том, что интеллектуальные инструментальные и языковые средства можно исследовать на основе соответствующих формальных программных машин (ПМ).

Программная машина T_L языка L представляет собой следующий набор алгебраических систем:

$$T_L = < M_L, K_L, A_L, F_L, \Phi_L, I_L >.$$

Алгебраическая система M представляет собой алгебру элементов памяти с определенными на множестве ее элементов отношениями.

Элементы памяти являются фрагментами оперативной памяти, используемой в конкретном языке программирования (ЯП) для множества базовых и производных типов данных, с соответствующими адресами памяти. Так, например, для языков типа Си или Ява базовыми элементами M являются адресуемые области памяти, используемые для объектов целого, действительного и символьного типов, из которых синтезируются более сложные производные структуры, также имеющие единственный адрес начала соответствующей области памяти.

K представляет собой алгебраическую систему констант ЯП. Эта система жестко связана с системой M и соответствует прикладной составляющей модели данных ПМ алгоритмического языка.

Синтаксические конструкции любого из алгоритмических языков содержат в своем составе управляющие конструкции для организации алгоритмической схемы программы, а также конструкции, используемые для записи операций над базовыми типизированными объектами языка. Для исследования этих конструкций в ПМ используются соответственно алгебраические системы F и A. Для универсальных ЯП к системе F относятся такие синтаксические конструкции, как if-then-else, do-while, case и т.п., система A содержит операции для числовых и строковых типов данных, например +, *, /, = и т.д.

Алгебраические системы Ф и I предназначены для описания интерпретации соответствующих синтаксических конструкций, исследуемых в F и A. Система интерпретации некоторого ЯП представляет собой описание *семантики* (смысла) его синтаксических конструкций в некоторой более простой системе команд, оперирующей понятиями, заведомо более примитивными, чем понятия этого языка.

Задачей настоящей статьи является освещение разработки математического и программного обеспечения для интеллектуальных программных систем на основе программных машин.

Теоретическая часть

Алгебраическая система M_L элементов памяти представляет собой следующую систему множеств:

$$M_L = < A_m, \Omega_m, R_m >,$$

где пара множеств $<A_m, \Omega_m>$ – алгебра элементов памяти, в которой $A_m = \{m_1, m_2, ..., m_i, ..., m_n\}$ – множество адресов базовых и производных элементов памяти m_i, соответствующих определенным типам данных, $\Omega_m = \{ \omega_1, ..., \omega_k \}$ – сигнатура алгебры, где $\omega_i \in \Omega$ являются операциями над элементами памяти для синтеза производных элементов, $R_m = \{r_1, ..., r_t\}$ – множество отношений над элементами памяти, отражающими структуру памяти модели данных алгоритмического языка L.

Элементы множества A_m представляют собой базовые и производные элементы памяти для соответствующих типов, используемых в ЯП. Каждый из элементов имеет собственный ранг, характеризующий сложность структуры элемента памяти. Он представляет собой целое число, на единицу большее, чем ранг самого сложного элемента, включенного в его структуру. Элементы памяти, соответствующие базовым типам данных, имеют ранг 0. Ранг указывается верхним индексом элемента. Вместе с тем, если это не приведет к путанице, далее индекс ранга будет опускаться.

Для большинства ПМ можно использовать сигнатуру $\Omega_m = \{ + \}$, в которой единственная операция "+" соответствует синтезу двух элементов памяти в единый производный элемент более высокого ранга.

С помощью операции синтеза можно описывать сложные композиции элементов памяти, представленные на рисунке 1. Алгебраические соотношения между элементами памяти в этом случае могут быть выражены следующими равенствами:

$$m_0^1 = m_0^0 + m_i^0; \; m_1^1 = m_0^0 + m_n^0; \; ... \; m_k^1 = m_i^0 + m_n^0;$$

$$m_0^2 = m_0^1 + m_i^0; \; m_1^2 = m_1^1 + m_i^0; \; ... \; m_m^2 = m_2^1 + m_i^1 + m_k^1;$$

или более детально через базовые элементы па-

мяти, например:

$$m_i^2 = m_0^0 + m_1^0 + m_i^0 \ .$$

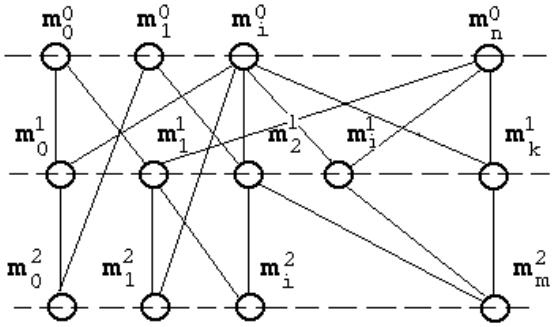

Рисунок 1 – Графическое представление алгебраической композиции

Множество отношений над элементами памяти состоит, как правило, из бинарных отношений и позволяет описывать ситуации взаимного пересечения областей памяти, что влияет на свойства операций алгебраической системы F. Можно привести примеры наиболее распространенных отношений, используемых в алгебраических системах элементов памяти для различных ЯП:

$$R_m = \{ =_m, >_m, \geq_m, \leftrightarrow_m, \perp_m \} \ .$$

В дальнейшем для удобства изложения нижний индекс отношения m, характеризующий его принадлежность к алгебраической системе элементов памяти, будет опускаться. Для приведенного множества отношений – первые два соответствуют равенству и неравенству двух элементов памяти по совпадению составляющих их элементов. Отношение ">" говорит о строгом включении всех составляющих элемента памяти, заданного вторым членом отношения, в элемент памяти, заданный первым членом. $\leftrightarrow$, $\perp$ $\leftrightarrow_m$, $\perp$ соответственно отношения общности и различия структуры элементов памяти.

Элементы памяти могут представлять собой не только пространство для хранения каких-либо данных, но и пространство для хранения операторов программы. Для различения таких элементов памяти можно соответственно использовать следующие конструкции: m (x) и m (f), где x – некоторая переменная или другой объект, относящийся к данным, а f – некоторый оператор программы. Такая запись дает возможность описывать более сложные композиции структур памяти, например списки, стеки или очереди.

Приведем пример формального представления списка. Пусть некоторый односвязный список m_s содержит n элементов. Тогда его можно представить следующим выражением:

$$m_s = m_1 (x_1 , m_2) + m_2 (x_2, m_3) + ... + (x_i, m_{i+1}) + ...+ m_n (x_n, 0).$$

В этом выражении использована операция ",", являющаяся синонимом операции композиции "+".

Заметим, что предложенное представление элементов памяти имеет еще одно положительное качество. Оно заключается в одинаковом представлении различных по типу элементов памяти, но отличающихся лишь способом оперирования с ними. Например, представление очереди m_q и стека m_c будет выглядеть эквивалентно:

$$m_q = m_c = m_1 (x_1)+m_2 (x_2)+...+ (x_i) +... + m_n (x_n).$$

Тип производного данного, различаемый по нижнему индексу, определяется операциями помещения (f_c^t , f_q^t) и извлечения (f_c^p , f_q^p) элементов:

$f_c^t (m_x , m_c) = m_x + m_c$; $f_q^t (m_x , m_q) = m_x + m_q$;

$f_c^p (m_c = m_1 (x_1) + m_2 (x_2) + ... + (x_i) + ... + m_n (x_n)) = m_1$;

$f_q^p (m_q = m_1 (x_1) + m_2 (x_2) + ... + (x_i) + ... + m_n (x_n)) = m_n$.

С помощью алгебры элементов памяти можно определять и более сложные операции преобразования памяти, например операцию f^p включения элемента в середину списка:

$$f^p (m_s , x_x, i) = m_1 (x_1 , m_2) + m_2 (x_2, m_3) + ... + (x_i, m_{i+1}) + m_{i+1} (x_x, m_{i+2}) + ...+ m_n (x_n, 0),$$

где $m_s = m_1 (x_1 , m_2) + m_2 (x_2, m_3) + ... + (x_i, m_{i+1}) + ...+ m_n (x_n, 0)$.

Аналогичным образом могут быть определены арифметические операции с содержимым памяти, например сложение элементов:

$$f_+ (m(x_1), m(x_2), m(x_3)) = m(x_1 = x_2 + x_3).$$

Алгебраическая система констант представляет собой прикладную систему базовых типов данных, используемых в ЯП. В простейшем случае алгебраическая система констант представляется системой множеств:

$$K_L = < A_K, \Omega_K, R_K >,$$

где $<A_k, \ \Omega_k >$ – прикладная алгебра констант базовых типов данных, в которой $A_k = \{ k_1, k_2, \ldots, k_i, \ldots, k_n \}$ – множество констант, Ω_k – множество операций $\omega_1, \omega_2, \ldots, \omega_j, \ldots, \omega_m$, таких как, например, сложение, умножение, конкатенация, а также операции построения констант для производных типов данных.

Одним из важных видов операций в $\Omega_к$ являются операции построения сложных констант для производных типов данных по аналогии с системой M_L;

$R_k = \{r_1, r_2, \ldots, r_i, \ldots, r_t\}$ – множество отношений прикладной системы, например формальной арифметики или алгебры символьных строк.

Алгебра A_L, используемая для формального описания в ее рамках синтаксических конструкций языкового инструментария, имеет своим носителем цепочки некоторого подъязыка языка L.

Например, для арифметических выражений могут быть определены операции сложения, вычитания, умножения и деления со следующим синтаксисом, аналогичным синтаксису языка Lisp [2]:

(ADD List1 List2), (SUB List1 List2),

(MUL List1 List2), (DIV List1 List2).

Для этих операций можно определить соответствующие программные термы:

$f_{add}(x_1, x_2)$, $f_{sub}(x_1, x_2)$, $f_{mul}(x_1, x_2)$,

$f_{div}(x_1, x_2)$.

Трехаргументную операцию композиции σ можно представить выражением:

$f_x : f_x(x_1, ..., x_i, ..., x_n)$, $f_y: f_y(y_1, .., y_j, ..., y_m)$,

$\sigma(f_y, i, f_x) = f_x(x_1, ..., f_y, ..., x_n)$.

Такую композицию назовем *подстановочной*. Для синтаксических конструкций программного инструментария она имеет важное значение и позволяет образовывать сложные синтаксические выражения, например:

(SETQ X (ADD (MUL Y Z) X),

что в виде программного терма выглядит так:

$f_{setq}(x, f_{add}(f_{mul}(y, z), x))$.

Нетрудно видеть, что в этом выражении операция подстановочной композиции использована дважды:

$\sigma(\sigma(f_{mul}(y, z), 1, f_{add}(x_i, x)), 2, f_{setq}(x, x_j))=$

$\sigma(f_{add}(f_{mul}(y, z), x), 2, f_{setq}(x, x_j)) =$

$= f_{setq}(x, f_{add}(f_{mul}(y, z), x))$.

Управляющие конструкции алгоритмического языка служат для разветвления линейного вычислительного процесса с целью выполнения наиболее сложной алгоритмической части программ. Соответствующая алгебраическая система описывает возможные композиции синтаксических конструкций ЯП или другого программного инструментария, используемых для задания структуры алгоритма. Она представляет собой следующую систему множеств:

$$F_L = <A_m, \Omega_m, R_m>,$$

где пара множеств $< A_F, \Omega_F >$ – алгебра управляющих конструкций, в которой $A_F = \{ f_1, \ldots, f_i, .., f_n \}$ – множество командных конструкций, описываемых порождающей грамматикой G_L некоторого языка L и характеризующих управление вычислениями в программе. Каждая конструкция f_i имеет произвольную местность $(f_i) = n$, говорящую о возможности выполнения n подстановок в те места f_i, которые соответствуют нетерминальным символам грамматики G_L.

Сами же подстановки являются элементами множества операций Ω_F, необходимыми для формирования более сложных и более конкретных программ. Так, например, конструкцию

if x_0 then x_1 else x_2

можно представить как

$f_1(x_0, x_2, x_3)$, $(f_1) = 3$,

а конструкцию while x_0 do x_1 – как

$f_2(x_0, x_1)$, $(f_2) = 2$,

составной оператор

Begin x_1, ..., x_n End

как n-местную конструкцию

$f_3(x_1, ..., x_n)$.

Множество R_F состоит из бинарных отношений над управляющими конструкциями $=, \neq, \geq, \leq, >, <$, имеющими следующий смысл:

$=$ – эквивалентность конструкций,

$\neq$ – неэквивалентность конструкций,

$\geq$ – нестрогое вхождение одной конструкции в другую,

$\leq$ – обратное предыдущему отношение,

$>, <$ – строгие отношения вхождения.

Интерпретация синтаксических конструкций – есть способ задания семантики конструкций через систему команд какой-либо более простой ПМ. Такой машиной может быть, например, процессор некоторой ЭВМ или базовый Ассемблер вычислительной машины.

Алгебраические системы, описывающие интерпретацию, имеют носители с элементами – n-местными термами, в которых на местах аргументов используются следующие типы конструкций:

$m(x_i)$ – адрес первого элемента объекта (переменной) с именем x_i,

$m(I_i)$ – адрес операции с именем I_i,

m(Stop) – адрес завершающей выполнение пустой операции,

$m(\varphi_i)$ – адрес операции φ_i.

Символ "Stop" означает пустую заключительную операцию интерпретации какой-либо синтаксической конструкции и может опускаться при последовательной композиции двух интерпретационных последовательностей операций.

Алгебра интерпретации прикладной системы

операций I_L представляет собой следующую систему множеств:

$$I_L = < A_I, \Omega_I, R_I >,$$

где A_I – множество микропрограмм, интерпретирующих выражения прикладной системы операций. Для этого множества существует некоторое подмножество $A_I^0 \subset A_I$, называемое базисом интерпретирующей системы, т.е. множество элементарных операций, из композиции которых получаются все остальные микропрограммы.

Так например, для формальной арифметики, используемой в универсальных алгоритмических языках, это могут быть операции "+", "-" с присоединенными к ним операциями работы со стековой памятью и безусловными переходами:

$$A_I = \{ I^+, I^-, I^v \},$$

где $I^+(m(x_i), m(I_j^x))$ – сложение x_i с содержимым сумматора и помещение результата в сумматор, далее переход к операции I_j^x; $I^- (m(x_i), m(I_j^x))$ – операция уменьшения содержимого сумматора на величину x_i, $I^v(m(x_i), m(I_j^x))$ – перенос содержимого сумматора в переменную x_i.

Множество Ω состоит из единственной операции последовательной композиции "$\circ$" интерпретирующих выражений в производные выражения. Например, обнуление сумматора можно описать так:

$$[I^v (m(x), m(I^-))] \circ [I^- (m(x), m(Stop)] = I^{\Sigma} (m(x),m(Stop)),$$

где m(x) – память, отведенная под произвольную временную переменную.

Последовательности композиций позволяют описывать интерпретацию арифметических выражений в ЯП. В качестве примера опишем арифметическое выражение языка Си $x = a + b$.

В этом выражении использованы две разноприоритетные операции Си: { =, + } или { $f^=$, f^+}. В форме термов алгебры конструкций синтаксиса прикладной системы операций рассматриваемое выражение можно записать в форме

$f^= (x, f^+ (a, b))$.

Интерпретация этого выражения выглядит следующим образом:

$$I^{=+}(x,a,b) = I^{\Sigma}(m(x),\ m(Stop)) \circ I^{+}(m(a), Stop) \circ \circ I^{+}(m(b), Stop) \circ I^{v}(m(x), Stop) .$$

Управляющие синтаксические конструкции, описывающиеся системой F_L , имеют следующую систему интерпретации Φ_L:

$$\Phi_L = < A_\Phi, \Omega_\Phi, R_\Phi >.$$

Здесь A_Φ – множество интерпретирующих элементов (микропрограмм, используемых для интерпретации управляющих конструкций инструментального средства L) , Ω_Φ – сигнатура, состоящая из операций композиции (как правило, единственная операция последовательной композиции "$\circ$"), R_Φ – отношения эквивалентности и вхождения на интерпретирующих последовательностях из A_Φ.

Интеллектуальные программы имеют свои особенности, которые необходимо принимать во внимание при формальном исследовании. Перечислим основные из них:

– отход от строгого понимания алгоритма при разработке интеллектуальных программ;

– использование нечетких рассуждений при выборе решения;

– рассмотрение внутренних элементов программной системы как данных и процедур нового уровня, называемого уровнем знаний;

– использование неклассических видов логик;

– возможность применения в рассуждениях индуктивных утверждений;

– упрощения при представлении моделей предметных областей;

– наличие метауровневых языковых средств;

– возможность обучения и самообучения интеллектуальных программ.

Рассмотрим перечисленные особенности подробнее, учитывая их влияние на формализмы программных машин.

Нестрогое понимание алгоритма

В классической теории алгоритмов основными признаками алгоритма считаются следующие известные признаки:

1) дискретность алгоритма,

2) детерминированность алгоритма,

3) элементарность шагов,

4) направленность алгоритма,

5)универсальность (массовость) алгоритма.

Почти все перечисленные признаки алгоритмов для интеллектуальных систем подвергаются изменению. Так, например, если свойство дискретности в основном сохраняется (кроме, может быть, новых типов спецпроцессоров), то детерминированность считается необязательной, т.е. выбор пути решения может осуществляться иногда с использованием случайных или псевдослучайных факторов. Элементарность шагов также претерпевает изменения, поскольку это могут быть ряд параллельных процессов или сложное действие, неделимое с точки зрения элементарности, но имеющее побочные эффекты. Направленность алгоритма интеллектуальной программы рассматривается на более абст-

рактном уровне, т.е. программа не должна обеспечивать строгое функциональное соответствие «вход – выход», но должна обеспечивать перечень предъявляемых к ней требований. На один и тот же вопрос различным пользователям могут быть выданы разные данные. Свойство универсальности программ с интеллектом, как правило, сохраняются. Наиболее яркими примерами, подтверждающими высказанные утверждения, являются продукционные экспертные системы [8].

Естественно, что все перечисленные «ослабления» свойств алгоритма являются скорее псевдонарушениями свойств, поскольку большая часть интеллектуальных программ проектируется на известных алгоритмических языках. В то же время, если рассматривать эти программы как инструментальные средства нового уровня, то логичнее не рассматривать их отображение (редукцию) в средства более простых уровней, поскольку такое отображение будет сложным и не позволит непосредственно исследовать задачу программы.

Использование нечетких рассуждений

Применение нечеткой логики [9] в конкретных программах приводит к тому, что по входным данным заранее сложно определить конкретную трассу решения. Многие из условных операторов интеллектуальных программ могут использовать различной сложности критерии означивания логических условий. Кроме того, эти критерии могут изменяться при повторном прохождении операторов разветвления структуры программы. Другим фактором, влияющим на свойства программных машин, является возможность использования элементов данных с множественными значениями, выбираемыми различным образом в зависимости от контекста оператора и приписанными значениям элемента памяти вероятностям принадлежности [9]. Факторы уверенности определяют возможность выдачи множественного результата, т.е. нескольких результатов разной степени вероятности. Примером таких ПМ является язык инструментальной системы GURU [10].

Уровень знаний

Данные и процедуры интеллектуальных ПМ рассматриваются как знания. Основными признаками, отличающими данные от знаний, можно считать следующие:

– интенсиональность, т.е. возможность вывода одних знаний из системы других, а также их синтез на основе соответствующих семантических описаний;

– активность знаний, т.е. они могут выступать в качестве декларативной составляющей, или процедурной, т.е. могут сами выступать и в роли данных, и в роли алгоритмов;

– отображение в знаниях причинно-следственных, родовидовых и ассоциативных отношений, адекватных представлению человека о конкретной предметной области.

С точки зрения формализации знаний в ПМ сложность заключается в определении в каждом конкретном случае, какие знания выступают в роли данных, а какие – в роли системы оперирования.

Использование неклассических видов логик

К неклассическим видам логик можно отнести:

– псевдофизические логики, описывающие свойства пространственно-временных, причинно-следственных, ассоциативных и других отношений и ситуаций предметной области [11];

– многозначные логики, предполагающие наличие дополнительных значений, кроме «истина» и «ложь», например «неопределено»;

– паранепротиворечивые логики, допускающие временное существование противоречий в аксиоматических утверждениях, разделяемых по локальным классам [12];

– немонотонные логики, более широко, чем в классических логиках, трактующие правила вывода;

– модальные логики, построенные на семантике возможных миров;

– логики умолчаний, формализующие лишь выполнимые рассуждения;

– другие логики, выходящие за рамки классических исчислений.

Следует заметить, что в реальных интеллектуальных программах редко встречаются случаи использования сразу нескольких видов неклассических логик, поскольку в большинстве случаев это может привести к неустранимой противоречивости. В то же время при использовании какой-либо из перечисленных логик программные машины, соответствующие таким инструментальным интеллектуальным средствам, существенно изменяют свойства своих операций. Так, например, паранепротиворечивые или многозначные логики могут привести к параллельным взаимоисключающим трассам выполнения программ. В этом случае логические условия алгоритмов могут либо временно не получать означивания, либо получить одновременное противоречивое означивание. Таким образом, семантику операций алгоритмических алгебр, составляющих программную машину, необхо-

димо задавать как семантику возможных миров, а при исследовании графов трасс выполнения программ рассматривать «второе измерение», вводимое категорией возможности.

Применение индуктивных утверждений

Формирование индуктивных утверждений имеет в основном структурно-статистический характер. В этом случае интеллектуальные программы используют большие массивы экспериментально полученной информации. Алгоритмы, работающие на множестве таких массивов, могут существенно изменяться во времени. В некоторых случаях необходимо предусматривать изменение алгоритма кардинальным образом. При этом появляется существенно новое понятие, ранее не используемое при проектировании самого алгоритма. Это понятие *функциональной целостности* алгоритма. Оно предполагает существование функционально-зависимых подмножеств алгоритма, которые должны быть жестко согласованны между собой. Изменение в каком-либо одном из алгоритмов подмножества должно повлечь соответствующие изменения во всех других алгоритмах этого подмножества.

Упрощение предметной области

Методы искусственного интеллекта используются в тех областях программирования, где невозможно или чрезвычайно сложно получить точное математическое описание модели предметной области. В этом случае приходится использовать упрощение этой модели до того уровня, который характерен для решения человеком эвристических задач с неполной информацией.

Эта особенность интеллектуальных систем является одной из основных, обеспечивающих эффективность таких систем по сравнению с другими. Кроме того, это, по-видимому, единственная особенность, не сказывающаяся на усложнении формальных программных машин интеллектуальных инструментальных средств.

Метауровневые языковые средства

Наличие метауровней предполагает возможность изменения текста программы самой этой же программой в процессе своего функционирования. Это явление исследуется не только для языкового инструментария, но и для формальных систем, получивших название *семиотических* [13].

Семиотические системы допускают изменение аксиом, правил вывода и даже базового синтаксиса логического исчисления. Сложность формального исследования программ с метауровнями весьма велика, проектирование и верификация таких систем также весьма трудоемки. Одним из способов, позволяющих упростить эти процессы, является строгое разделение метауровней с гарантированием условия их пустого пересечения во времени по шагам работы интеллектуального алгоритма. Эти условия должны подвергаться отдельному исследованию, т.е. их формулированию и последующему доказательству выполнимости на всех множествах входных данных.

Обучение и самообучение

Обучение программной системы – это способность программы частично изменять свой алгоритм на основе примеров пар данных вида (входные данные, выходные данные), предоставляемых человеком. *Самообучение* программной системы – это способность программы частично изменять свой алгоритм на основе анализа собственных предыдущих результатов. Самообучение можно разделить на два типа:

– самообучение с целью оптимизации алгоритма;

– самообучение с усилением целевой функции программы.

Как следует из рассмотренных особенностей интеллектуальных систем, каждая из особенностей определенным образом влияет на формализм программной машины, в рамках которого можно исследовать интеллектуальные программы. Выделим явно те изменения, которым должен быть подвергнут этот формализм:

– необходимо использование множественных значений констант для элементов памяти;

– способы соотнесения значений элементам памяти должны опираться на нечеткие меры соответствия;

– необходимо отражение изменения программных термов в конструкциях самих термов, т.е. использование переменных, определенных на множестве программных подтермов;

– способы соотнесения значений переменным программных термов могут быть также основаны на теории нечетких множеств;

– необходимы формальные средства для исследования вариантов работы программы с одним и тем же набором входных данных.

При внимательном рассмотрении перечисленных изменений видно, что все они так или иначе направлены на *внесение эвристической составляющей* во все основные элементы формализма программных машин. Это является главным формальным признаком интеллектуальных инструментальных и языковых систем. С одной стороны, внесение этих изменений усложняет программную машину, переводя соответствующие ей аксиоматики в неразрешимые

теории высоких порядков, с другой – это необходимая плата за допущения в упрощении предметной области, позволяющем решать задачи повышенной интеллектуальной сложности. Связь между этими взаимопротивоположными категориями вполне непосредственная и определяется тем, что недостатки в точности алгоритма решения приходится компенсировать перебором вариантов в совершенно другом измерении данных, а именно, в *пространстве возможностей* для получения множественных значений алгоритма.

Вместе с тем перечисленные сложности формализации не означают, что проблема формального исследования интеллектуальных систем в части доказательства их корректности, завершимости, оптимизации, автоматического проектирования программ не имеет решения. Эти задачи не только актуальны, но и инициируют исследования нового уровня, в частности доказательство непротиворечивости, полноты, разрешимости и концептуальной целостности интеллектуальных систем и знаний, на которых они основаны.

Таким образом, задачей формального исследования интеллектуальных систем является анализ интеллектуальных программных машин для инструментальных и языковых средств, основанных на эвристическом расширении этого формализма, а также исследование зависимости между свойствами программных машин и особенностями построенных в их рамках системах искусственного интеллекта.

Для изучения принципов формального исследования интеллектуальных систем рассмотрим подходы к анализу следующих типов программ:

– продукционные системы;

– системы эвристического программирования;

– программы логического проектирования;

– средства проектирования интеллектуальных систем, основанные на алгоритмах сопоставления;

– естественно-языковые системы.

Программные машины продукционных систем

Продукцией может считаться некоторая четверка вида N: Q, S $\rightarrow$ P , где N – имя продукции, Q – *область применимости* продукции, S – ее *антецедент*, а P – *консеквент*. Имя является уникальным идентификатором. Область применимости означает обычно некоторый слабый предикат, характеризующий предметную область задачи, для фрагмента решения которой предназначена продукция. Антецедент – сильный предикат (логическое выражение), принимающее истинность на множестве элементов памяти продукционной программы. Истинность антецедента указывает на необходимость выполнения действия S по преобразованию элементов памяти. Такая простая форма продукции, тем не менее, может предполагать весьма сложный синтаксис и семантику для составляющих Q, S и P, которые могут включать в себя сложные алгоритмы в качестве составляющих.

В наиболее поздних алгоритмических языках, основанных на продукциях, каждая продукция может представлять собой несколько ограничивающих применение предикатов разной степени точности и несколько действий, направленных также на различные уровни решения. Некоторые элементы продукций могут иметь характер метапреобразований, например удаления какой-либо продукции или изменения приоритета при выборе. Вместе с тем какой бы сложной не была структура N: Q, S $\rightarrow$ P, ее всегда можно подвергнуть декомпозиции, получив несколько более простых продукций.

Эвристическая сложность программных систем, построенных на основе продукций, заключается в том, что в базовом продукционном формализме на каждом из шагов решения может быть использовано любое правило из всех имеющихся. Это приводит к большому числу переборов вариантов для определения условия применимости правила. Кроме того, допускаются ситуации, когда могут быть применимы сразу же несколько правил. В этом случае приходится руководствоваться различными *стратегиями* выбора продукций, в том числе *стратегиям разрешения* правил [8].

Рассмотрим применение формализма программных машин для продукционного инструментария. Для алгебры элементов памяти таких машин используют, как правило, какую-либо из *алгебр фреймов* [14, 15]. Фреймы можно представлять как сложные элементы памяти, состоящие из строковых ячеек – *слотов*. Для этих элементов существуют отношения наследования и "часть-целое", как и для инструментария при объектно-ориентированном проектировании. Так же, как и деление на классы и объекты, в продукционных фреймовых системах существует соответствующее деление на фреймы и *экземпляры*. Множество экземпляров фреймовой системы задает область определения и значений для продукций.

Алгебраической системой управляющих синтаксических конструкций в продукционных системах являются алгебры продукций, относящиеся по типу к решеткам. Этот тип позволяет довольно просто достраивать и объединять различные продукционные программы, используя аддитивную операцию. В то же время для сокращения *пространства состояний* эвристического перебора продукций используют также мультипликативную операцию, которая в этом случае является операцией последовательной композиции продукций. Действительно, рассмотрим некоторое ограниченное множество продукционных правил $P = \{ p_1, p_2, ..., p_n \}$. Это множество является множеством-носителем алгоритмической алгебры-решетки. Если предположить, что в алгебре используется лишь одна аддитивная операция объединения множества продукций, то алгебра продукций образует полугруппу. Пусть также есть еще одно множество продукций $Q = \{ q_1, q_2, ..., q_m \}$, также образующее полугруппу. Будем рассматривать вариант, при котором необходимо выбирать на каждом шаге решения все продукции, которые могут быть применены. Тогда, если алгоритм P решает какую-либо задачу за k шагов, то общее число продукций, которое ему необходимо будет просмотреть на применимость, будет равным k*n, поскольку на каждом из шагов приходится рассматривать n продукций. Для программы Q это число для k шагов решения соответственно составит величину k*m. Если необходимо объединить программы P и Q, в алгебре-полугруппе приходится использовать аддитивную операцию объединения множеств правил:

$$P + Q = \{ p_1, p_2, ..., p_n, q_1, q_2, ..., q_m \}.$$

При отсутствии в объединяемых множествах эквивалентных продукций общее число элементов результирующего множества станет равным n+m. Теперь для решения той же самой задачи в k шагов результирующая программа выполнит (n + m) * k сопоставлений. Если предположить, что никакое из правил множества Q не может выполняться ранее, чем правила из P, то приходится признать неэффективность аддитивной операции композиции, поскольку выполнение последовательной мультипликативной композиции оказывается более целенаправленным. Мультипликативная композиция P*Q предполагает, что после выполнения шагов на основе множества P они больше не рассматриваются, а сопоставление происходит лишь для правил множества Q. Из этого следует, что общее число просмотров антецедентов для k шагов решения станет равным x*n + y*m = k, где x + y = k .

Исследование корректности продукционных программ также может рассматриваться с использованием формализма программных машин. В этом случае завершимость работы программных систем формально определяется до составления программы на уровне программной машины. Этот факт основан на том, что в случае возникновения бесконечного цикла интерпретатор продукционных правил, следуя алгоритмам поиска эквивалентных последовательностей продукций, отыскивает повторяющийся фрагмент, после чего останавливает интерпретацию с результатом "неудача" или выбирает следующую альтернативную продукцию. Поиск бесконечно повторяющихся последовательностей можно произвести и в тексте готовой программы для выдачи соответствующего предупреждения пользователю (поскольку в общем случае это может и не быть логической ошибкой). Для этого исследуются КНФ термов текста программы и выявляются транзитивные замыкания последовательных композиций правил. При структурном подходе к продукционному проектированию это сделать несложно, поскольку в КНФ уже заранее локализованы фрагменты программы, где может произойти повтор последовательностей продукций.

Рассмотренное сравнение операций композиции делает возможным анализ продукционных программ с целью их оптимизации.

Эвристическое программирование появилось в методологии интеллектуальных систем значительно ранее, чем теории представления знаний, экспертных, расчетно-логических и естественно-языковых систем [16]. Благодаря времени появления и эффективности методов эвристическая составляющая присутствует во всех классах интеллектуальных программ.

Наиболее известными эвристическими методами, которые можно использовать при поиске решения задач с большими пространствами, являются методы, построенные на основе переборных алгоритмов поиска в глубину и в ширину:

– метод поиска по градиенту;

– алгоритм равных цен;

– метод от наилучшего частичного пути;

– игровые минимаксные стратегии с α/β-отсечениями;

– методы локально-углубленного поиска и т.п.

Эвристическое проектирование является более общей концепцией, чем, например, продук-

ционное проектирование, которое относится к одному из его частных случаев.

Можно сформулировать общие требования к достаточно эффективному алгоритму поиска:

– при известном решении задачи этот алгоритм должен выдавать абсолютно точное решение, не используя перебора;

– при абсолютно неизвестной задаче алгоритм должен использовать полный перебор на основе алгоритмов поиска в ширину или в глубину;

– при решении задачи, фрагменты которой являются ранее решенными, для них необходимо использовать готовое решение, в то время как для оставшихся фрагментов нужно использовать перебор;

– при существовании в решаемой задаче аналогий с другими ранее решенными необходимо пользоваться аналогичной оценочной функцией.

Перечисленные требования исчерпывают все возможности, которые могут быть использованы для получения максимально точного и универсального алгоритма поиска решения

Приведем далее элементы формальных программных машин, позволяющих исследовать системы, основанные на эвристическом подходе.

Главной составляющей программных машин является алгебраическая система управляющих синтаксических конструкций. Если рассмотреть эту систему с точки зрения алгоритмических языков, которые используются для проектирования эвристических систем, она мало отличается по набору операций и множеству-носителю от универсальных языков программирования. Вместе с тем при проектировании эвристических программ, основанных на планировании решения, необходимо использование качественно новых операций, в частности *операций выделения наиболее общих задач* и *последовательной или альтернативной композиции общих задач*. Эти операции в общем случае неприменимы непосредственно к синтаксическим конструкциям алгоритмического языка, но могут быть рассмотрены на множестве термов алгебры изменения состояний.

Логическое программирование в большей части связано с использованием алгоритмических языков семейства Prolog [3]. Для этих языков характерно множественное использование операции унификации логических термов со связными переменными. Программу на базовом языке Prolog можно представить следующей формальной архитектурой:

$$f_a(x_1, ..., x_n) \Rightarrow f_{a+1}(x_1, ..., x_n),$$
$$f_b(x_1, ..., x_n) \Rightarrow f_{b+1}(x_1, ..., x_n),$$
$$\ldots$$
$$f_t(x_1, ..., x_n) \Rightarrow f_{t+1}(x_1, ..., x_n);$$
$$f_r(x_1, ..., x_n), f_{r+1}(x_1, ..., x_n), \ldots,$$
$$f_m(x_1, ..., x_n).$$

Эта архитектура построена из двух различных частей. Первая из них, *база правил*, содержит знаки логического следования "=>", вторая, *база фактов* – лишь список программных термов. В приведенной архитектуре все термы программы имеют n аргументов. В реальности число аргументов различно, но может быть несложно сведено к одному и тому же числу, если использовать фиктивные переменные.

Унификация как основная операция алгебры интерпретации синтаксических конструкций заключается в сопоставлении с базой знаний некоторого входного терма-вопроса $f_y(x_1, ..., x_n)$, в котором на месте некоторых переменных находятся константы. Результатом работы всей программы является нахождение таких значений переменных терма, при которых он считается истинным высказыванием с точки зрения логических правил и фактов, содержащихся в программе. При анализе логических программных систем с точки зрения их синтеза можно выделить операцию *причинно-следственной композиции*, позволяющую синтезировать основу правила, операцию *теоретико-множественной композиции фактов* и операции синтеза термов с помощью *логических связок*. К булеву кольцу можно отнести лишь алгебру, построенную на основе сигнатуры логических связок.

Если формально описать алгебраическую систему для управляющих синтаксических конструкций языка Prolog, то можно получить следующую систему:

$$F_{Prolog} = < F, \ \Omega, \ R \ >,$$

где $F = F_0 \cup F_1$ состоит из множества фактов F_0 и множества правил F_1, Ω содержит операции причинно-следственной (“*”) и теоретико-множественной (“+”) композиций, R включает отношения равенства, неравенства, а также строгие и нестрогие отношения теоретико-множественного включения. Операции из Ω можно определить следующим образом:

$$\forall f_x, f_y \in F, \quad f_x + f_y = f_x \ \& \ f_y;$$
$$f_x * f_y = f_x \Rightarrow f_y,$$

а следовательно, результаты этих операций можно опять представить логическим термом, входящим в множество F.

Замечательным свойством логических про-

грамм является то, что, представимые в форме продукций (логических правил), они остаются полностью логическими выражениями. Это свойство позволяет рассматривать для ПМ логического инструментария отдельно левые и правые части правил с точки зрения их возможной оптимизации.

Рассмотрим два произвольных правила:

$f_x (x_1, ..., x_n) => f_a (x_1, ..., x_n)$,
$f_y (x_1, ..., x_n) => f_b (x_1, ..., x_n)$.

В общем случае левые и правые их части могут быть представлены как результаты некоторых логических операций, например конъюнкций:

$f_x (x_1, ..., x_n) = f_{1x} (x_1, ..., x_n) \& f_{2x} (x_1, ..., x_n) \& ... \& f_{mx} (x_1, ..., x_n)$,
$f_y (x_1, ..., x_n) = f_{1y} (x_1, ..., x_n) \& f_{2y} (x_1, ..., x_n) \& ... \& f_{hx} (x_1, ..., x_n)$,
$f_a (x_1, ..., x_n) = f_{1a} (x_1, ..., x_n) \& f_{2a} (x_1, ..., x_n) \& ... \& f_{ka} (x_1, ..., x_n)$,
$f_b (x_1, ..., x_n) = f_{1b} (x_1, ..., x_n) \& f_{2b} (x_1, ..., x_n) \& ... \& f_{mr} (x_1, ..., x_n)$.

Для оптимизации можно использовать следующие выражения, справедливые для термов логических ПМ:

$f_x \& f_y => f_z , f_x \& f_t => f_k \rightarrow f_x => f_p, f_p \& \& f_y => f_z, f_p \& f_t => f_k$,
$f_x \& f_y => f_z , f_x \& \neg f_y => f_z \rightarrow f_x => f_z$,
$f_x => f_z, f_x \& f_y => f_z \& f_t , f_x \& f_h => f_m \rightarrow f_y => => f_t , f_h => f_m$,

где символ $\rightarrow$ обозначает вывод производного утверждения. Эти выражения могут быть переписаны с использованием алгебраических операций ПМ следующим образом:

$f_x \& f_y * f_z + f_x \& f_t * f_k = f_x * f_p + f_p \& f_y * f_z + f_p \& f_t * f_k$,
$f_x \& f_y * f_z + f_x \& \neg f_y * f_z = f_x * f_z$,
$f_x * f_z + f_x \& f_y * f_z \& f_t + f_x \& f_h * f_m = f_x * f_z + f_y * f_t + f_h * f_m$.

Рассмотренные выражения для эквивалентных оптимизирующих трансформаций не исчерпывают всех возможностей оптимизации логических программ, даже если при этом использовать алгоритмы резолюции для хорновских дизъюнктов.

К инструментальным средствам интеллектуальных систем, основанным на алгоритмах сопоставления и унификации, можно класс программ, связанных с унификацией. Этому классу принадлежат системы продукционного и логического программирования. В то же время существуют другие инструментальные системы с нестандартной семантикой, основанные на унификации. Наиболее известными представителями этих инструментальных языковых систем являются *языки представления знаний*, такие как FRL, KRL, GUS [15] и др.

ПМ, описывающая работу инструментария с фреймами в качестве главных составляющих, содержит алгебраическую систему элементов памяти как алгебру композиции и декомпозиции фреймов, алгебраическую систему управляющих синтаксических конструкций как систему, включающую оперирования с процедурной частью программ и алгебру интерпретации конструкций как набор процедур унификации на множестве фреймов. Множества отношений включают родовидовые отношения на фреймах, отношения "часть-целое" и отношения использования.

Если рассмотреть фрейм как элемент памяти некоторой формальной ПМ, его можно представить следующей конструкцией:

$N: IsA (N, pN), f_a (x_1, x_2, ..., x_n)$;
$f_b (x_r, x_{r+1}, ..., x_t), ..., f_c (x_h, x_{h+1}, ..., x_m)$,

где N – имя фрейма, pN – имя родовидового предка фрейма, связанного с ним соответствующим отношением, а f_a, f_b, ..., f_c – слоты фрейма, представляющие собой некоторые списки характеристик с возможными значениями, представленными собственными аргументами. Сама по себе фреймовая структура, согласно ее определению [14] как минимальной совокупности знаний, необходимых для точного определения понятия, должна обладать свойством оптимальности хранения знаний. Это должно обеспечиваться тем, что для любых двух фреймов, имеющих подмножества равных слотов, должен существовать общий фрейм, состоящий только из этих равных слотов.

С точки зрения синтеза фреймовой программы в соответствующей ПМ можно выделить две операции:

– теоретико-множественное объединение фреймов в единую программу (порядок их следования в программе в общем случае неважен);

– операцию родовидовой композиции фреймов.

Операция родовидовой композиции позволяет связать два фрейма в родовидовую иерархическую подпоследовательность, устанавливая для слотов этих фреймов отношение наследования. Эта операция является по своей природе мультипликативной и обладает свойством дистрибутивности по отношению к теоретико-множественному объединению.

Пусть в алгебре синтаксических конструкций фреймового инструментария существует

множество-носитель, представляющее собой множество возможных фреймов F = { f_1, ..., f_n }. Две операции { * , + } предназначены соответственно для формирования фреймов-наследников и независимых напрямую фреймов программы. Каждый фрейм f_i может содержать некоторое число m слотов. Будем обозначать это следующим образом: $f_i (x_1, ..., x_m)$. В этих обозначениях можно описать смысл предложенных операций:

$$f_a (x_{1a}, ..., x_{ma}) + f_b (x_{1b}, ..., x_{kb}) = = f_a (x_{1a}, ..., x_{ma}) + f_{1b} (x_{1b}, ..., x_{kb});$$

$$f_a (x_{1a}, ..., x_{ma}) * f_b (x_{1b}, ..., x_{kb}) = = f_c (x_{1c}, ..., x_{hc}) + + f_{ap} (x_{1ap}, ..., x_{map}) + f_{bp} (x_{1bp}, ..., x_{kbp}),$$

где ";" – синтаксическая конструкция разделения фреймов (разделитель), а $f_c (x_{1c}, ..., x_{hc})$ – фрейм, полученный в результате пересечения множеств слотов фреймов $f_a (x_{1a}, ..., x_{ma})$ и $f_b (x_{1b}, ..., x_{kb})$, фреймы $f_{ap} (x_{1ap}, ..., x_{map})$ и $f_{bp} (x_{1bp}, ..., x_{kbp})$ представляют собой преобразованные фреймы $f_a (x_{1a}, ..., x_{ma})$ и $f_b(x_{1b}, ..., x_{kb})$ путем удаления из них общих слотов.

Несложно заметить, что для операций "*" и "+" существует дистрибутивность справа и слева:

$$f_c (x_{1c},..., x_{hc})* (f_a (x_{1a}, ..., x_{ma}) + + f_b (x_{1b}, ..., x_{kb})) = (f_a (x_{1a}, ..., x_{ma}) + + f_b (x_{1b}, ..., x_{kb})) * f_c (x_{1c}, ..., x_{hc}) = = f_a (x_{1a}, ..., x_{ma}) * f_c (x_{1c}, ..., x_{hc}) + + f_b(x_{1b}, ..., x_{kb}) * f_c (x_{1c}, ..., x_{hc}) = = f_{ac} (x_{1ac}, ..., x_{mac}) + f_{bc} (x_{1bc}, ..., x_{kbc}) + + f_{ap} (x_{1ap}, ..., x_{map}) + f_{bp} (x_{1bp}, ..., x_{kbp}),$$

где $f_{ac} (x_{1ac}, ..., x_{mac})$ и $f_{bc} (x_{1bc}, ..., x_{kbc})$ – фреймы, полученные выделением общих частей фреймов $f_a (x_{1a}, ..., x_{ma})$ и $f_b (x_{1b}, ..., x_{kb})$ с фреймом $f_c (x_{1c}, ..., x_{hc})$, а $f_{ap} (x_{1ap}, ..., x_{map})$ и $f_{bp}(x_{1bp}, ..., x_{kbp})$ – фреймы, преобразованные из $f_a(x_{1a}, ..., x_{ma})$ и $f_b (x_{1b}, ..., x_{kb})$ удалением из них слотов, общих с $f_c (x_{1c}, ..., x_{hc})$.

Одновременная дистрибутивность операции * по отношению к + справа и слева говорит о коммутативности операций * и +. Для композиционной алгебры фреймов существует единичный *нуль-фрейм*, не содержащий слотов. Если этот фрейм обозначить символом 0, то для него будут справедливы следующие свойства:

$$\forall f_x (x_1, .., x_n), \quad f_x (x_1, .., x_n) * 0 = = f_x (x_1, .., x_n);$$

$$\forall f_x (x_1, .., x_n), \quad f_x (x_1, .., x_n) + 0 = = f_x (x_1, .., x_n);$$

$$\forall f_x (x_1, .., x_n), \exists f_x^{-1} (x_1, .., x_n) , f_x (x_1, .., x_n) * f_x^{-1} (x_1, .., x_n) = 0 .$$

Рассмотренные свойства операций дают возможность оптимизировать фреймовые структуры, приводя их к регулярным формам, чем обеспечивается принцип минимальности представления знаний.

Экспериментальные исследования

Цель эксперимента – оценить степень оптимизации программ в условиях интеллектуального инструментария при использовании оптимизирующих преобразований формализма программных машин.

Усредненные результаты 7 экспериментов применения формальных правил оптимизации для контрольных примеров программ (экспертные системы) приведены в таблице.

Результаты экспериментов

Параметр	Для Lisp-программ		Для Пролог-программ	
	Оптимзация		Оптимизация	
	до	после	до	после
Число операторов	420	358	58	51
Занимаемая память в Кбайтах	24	22	16	15

Кроме того, выявлена закономерность улучшения параметров сокращения программного кода при увеличении объема программ.

Заключение

Рассмотренные особенности исследования и оптимизации интеллектуальных программных систем дают возможность существенно повысить эффективность разработки таких систем на основе использования математической концепции программных машин.

Библиографический список

1. **Нильсон Н.** Принципы искусственного интеллекта. М.: Радио и связь, 1985. 372 с.

2. **Лавров С. С., Силагадзе Г. С**. Автоматическая обработка данных. Язык Лисп и его реализация. М.: Наука, 1978. 176 с.

3. **Стобо Дж.** Язык программирования ПРОЛОГ. М.: Радио и связь, 1993. 368 с.

4. **Lewis H. R., Statman R**., Unifiability is complete for co-LogSpace. – Info.Process.Lett., 1982, no 15, pp. 220-222.

5. **Каширин Д. И., Каширин И. Ю., Пылькин А. Н.** Полиморфическое представление знаний в Semantic Web. М.: Горячая линия–Телеком, 2009. 136 с.

6. **Каширин И. Ю., Медведев Р. Е.** Онтологическое накопление учебных знаний на основе сервиса информационных сетей // Информационные и теле-

коммуникационные технологии в образовании, СПГПТУ, 2013. С.75-84.

7. **Каширин И. Ю., Новичков В. С.** От С к С++. 2-е издание. М.: Горячая линия – Телеком, 2012. – 344 с.

8. **Брукинг А., Джонс П., Кокс Ф.** и др. Экспертные системы. Принципы работы и примеры. М.: Радио и связь, 1987. 224 с.

9. Нечеткие множества в моделях управления и искусственного интеллекта / под ред. Д.А. Поспелова М.: Наука, 1986. 312 с.

10. **Дюбуа Д., Прад А.** Теория возможностей. М.: Радио и связь, 1990. 288 с.

11**. Кандрашина Е. Ю., Литвинцева Л. В., Поспелов Д. А.** Представление знаний о времени и пространстве в интеллектуальных системах. М.: Наука, 1989. 328 с.

12. **Тей А.** и др. Логический подход к искусственному интеллекту: от классической логики к логическому программированию. М.: Мир, 1990. 432 с.

13. **Поспелов Д. А.** Логико-лингвистические модели в системах управления. М.: Энергоиздат, 1981. 231 с.

14. **Минский М.** Фреймы для представления знаний. М.: Энергия, 1979. 151 с.

15. **Каширин И. Ю., Каширин Д. И.** Модели представления знаний в системах искусственного интеллекта // Вестник Рязанского государственного радиотехнического университета. 2010. № 31. С. 36-43

16. **Каширин И. Ю., Крошилин А. В., Крошилина С. В.** Автоматизированный анализ деятельности предприятия с использованием семантических сетей. М.: Горячая линия – Телеком, 2011. 140 с.

UDC 007:681.512.2

FORMAL RESEARCH PECULIARITIES OF INELLIGENT PROGRAMMING TOOLS

I. Yu. Kashirin, PhD (technical sciences), full professor, RSREU, Ryazan, kashirin.i.yu@rsreu.ru
O. I. Kashirina, student, RSREU, Ryazan, april.helga@mail.ru

The problem of formal research peculiarities of intelligent programming tools is studied. The aim is to find main features of programming tools of artificial intelligence systems revealing the possibilities to optimize corresponding formal programming tools. In the article the analysis of programming tools research is proposed, distinctive features which determine its applicability for resolving different classes of formal research problems are considered,.

The most attention is devoted to optimization problem in formal expression of source programming code of intelligent soft systems. The following concepts of intelligent systems such as simplification of subject domain, logical programming, production systems, training and self-training, meta-level language tools, knowledge level, inductive reasoning are considered. First experiments of new formal approach to this problem showed efficiency in optimization expression of source programming code of intelligent soft systems. This approach is based on universal algebra sets called as programming machines.

Key words: *artificial intelligence, program analysis, algorithmic algebras, programming tools, formal programming machines, program optimization, knowledge level, training and self-training, meta-level language tools.*

References

1**. Nilson N**. *Principy iskusstvennogo intellekta* (Principles of artificial intelligence). Moscow: Radio and communication, 1985, 372 p. (in Russian).

2. **Lavrow S. S., Silagadze G. S**. *Avtomaticheskaja obrabotka dannyh. Jazyk Lisp i ego realizacija* (Automatic data processing. Lisp and his realization). Moscow: Science, 1978, 176 p. (in Russian).

3. **Stobo J.** *Jazyk programmirovanija PROLOG.* (Prolog programming language). Moscow: Radio I swyaz', 1993, 368 p. (in Russian).

4. **Lewis H. R., Statman R**., Unifiability is complete for co-LogSpace. – Info.Process.Lett., 1982, no 15, pp. 220-222.

5. **Kashirin D. I., Kashirin I. Yu., Pylkin A. N.** *Polimorficheskoe predstavlenie znanij v Semsntic Web.* **(**Polymorphic knowledge representation by Semantic Web). Moscow: Hotline–Telecom, 2009, 136 p. (in Russian).

6. **Kashirin I. Yu., Medvedev R. E.** Ontological the accumulation of educational knowledge on the base information network service/ *Information and telecommunication in education,* SPSPTU, 2013, pp. 75-84 (in Russian).

7. **Kashirin I. Yu., Novichkov V. S.** *Ot S k S++.* 2-e izdanie (From C to C++. 2-nd edition). M.: Moscow: Hotline–Telecom, 2015, 344 p. (in Russian).

8. **Brucking A., Jons P., Kox F.** etc. *Jekspertnye sistemy. Principy raboty i primery* (Expert systems. principles and examples). Moscow: Radio and communication, 1987, 224 p. (in Russian).

9. *Nechetkie mnozhestva v modeljah upravlenija i iskusstvennogo intellekta /* pod red. D.A. Pospelova (Fuzzy sets to control and artificial intelligence models. Edited bu. Pospelov D.A.). Moscow: Science, 1986, 312 p. (in Russian).

10. **Dyubua D., Prad A.** *Teorija vozmozhnostej* (Possibility theory). Moscow: Radio and communication, 1990, 288 p. (in Russian).

11. **Kandrashina E. Yu., Litvitseva L. V., Pospelov D. A.** *Predstavlenie znanij o vremeni i prostranstve v intellektual'nyh sistemah* **(**Knowledge presentation of time and space in intelligence systems). Moscow: Science, 1989, 328 p. (in Russian).

12. **Tey A.** etc. *Logicheskij podhod k iskusstvennomu intellektu: ot klassicheskoj logiki k logicheskomu programmirovaniju* (Logical approach to artificial intelligence from classical logic to logical programming). Moscow: World, 1990, 432 p. (in Russian).

13.**Pospelov D. A.** *Logiko-lingvisticheskie modeli v sistemah upravlenija* (Logical-linguistic model for control systems). Moscow: Energy publishing, 1981, 231 p. (in Russian).

14. **Minsky M.** *Frejmy dlja predstavlenija znanij* (Frames for knowledge representation). Moscow: Energy, 1979, 151 p. (in Russian).

15. **Kashirin D. I., Kashirin I. Yu.,** Knowledge representation model to artificial intelligence systems. *Vestnik Rjazanskogo gosudarstvennogo radiotehnicheskogo universiteta.* 2010, no. 31, pp. 36-43 (in Russian).

16. **Kashirin I. Yu., Kroshylin A. V., Kroshylina S. V.** *Avtomatizirovannyj analiz dejatel'nosti predprijatija s ispol'zovaniem semanticheskih setej* (Automated analysis of the company using semantic networks). Moscow: Hotline–Telecom, 2011, 140 p. (in Russian).