ISSN 2658-5014(Print)
2019. № 1. С. 75–85
СИИТ
СИСТЕМНАЯ ИНЖЕНЕРИЯ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ
http://siit.ugatu.su

УДК 004.65

ФОРМАЛИЗАЦИЯ ЗАДАЧ В УСЛОВИЯХ ДЕДУКТИВНОГО СИНТЕЗА

А. П. Бельтюков

belt.udsu@mail.ru

ФГБОУ ВО «Удмуртский государственный университет» (УдГУ)

Поступила в редакцию 21 января 2019 г.

Аннотация. Описывается построение и трансформации формулировок конструктивных задач в парадигме дедуктивного синтеза их решений. Предлагается подход к построению формализации конструктивной задачи, когда уже примерно известны и неформальная постановка задачи, и неформальная идея ее решения. В этой ситуации из неформальной постановки задачи и неформальной идеи ее решения вырабатывается система понятий, положенная в основу онтологии: типы объектов в наиболее общем для данной задачи виде, конструктивные свойства-атрибуты, конструктивные отношения и спецификации исходных и целевых функций и операторов. Под дедуктивным синтезом в настоящей работе понимается логический вывод решений конструктивных задач из их формальных постановок в определенных системах вывода. Под решениями конструктивных задач здесь понимается построение алгоритмов, структур данных и объектов, содержащих как данные, так и алгоритмы. Более того, задача может быть расширена так, что частями решений могут оказаться как физические объекты, так и компетенции людей. Такие решения можно назвать физико-антропно-техническими. При этом сама система построения таких решений может оказаться физико-антропно-технической, на что и рассчитывает настоящая работа. Вместо традиционных языков логических формул рассматриваются языки спецификации задач, более подходящие для описания предметной области в информатике. Роль языков доказательств здесь играют специальные языки дедуктивного программирования. Наиболее близкое к ним понятие в логике – доказательство как терм – выражение, которое, вообще говоря, можно вычислять (выполнять). Постановке задачи предшествует изучение предметной области и выявление проблем, связанных с ней. Для формулирования задач решения выявленных проблем строится определённая терминологическая система. Для успешного использования имеющимся традиционным аппаратом конструктивного дедуктивного синтеза терминологическая система должна иметь определенную структуру конструктивной онтологии. Она состоит в том, что все термины разбиваются на три класса: термины, обозначающие типы значений, термины, обозначающие предикаты и термины, обозначающие конструктивные функции.

Ключевые слова: дедуктивный синтез; физико-антропно-технические системы; онтологии; конструктивная логика; человеко-машинные системы.

ВВЕДЕНИЕ

Под дедуктивным синтезом в настоящей работе понимается логический вывод решений конструктивных задач из их формальных постановок в определённых системах вывода. Под решениями конструктивных задач здесь понимается построение алгоритмов, структур данных и объектов, содержащих как данные, так и алгоритмы. Более того, задача может быть расширена так, что частями решений могут оказаться как физические объекты, так и компетенции людей. Такие решения можно назвать физико-антропно-техническими. При этом сама система построения таких решений может оказаться физико-антропно-технической, на что мы и рассчитываем в настоящей работе. Исторически первыми идеями дедуктивного синтеза программ были идеи извлечения алгоритмов из конструктивных доказательств, конструктивно понимаемых логических, логико-математических или иных логико-предметных формул [1–3]).

Однако в настоящей работе мы не рассматриваем традиционные логические языки ввиду того, что они оперируют со слишком мелкими действиями, не с теми, к которым привык человек, работающий в области информатики. Это важно, потому что в физико-антропно-технических

системах человек является ведущим звеном. Удобство работы человека – ключевой фактор таких систем. Впервые автором такой подход был рассмотрен в работе [4].

Вместо традиционных языков логических формул мы рассматриваем языки спецификации задач, более подходящие для описания предметной области в информатике. Роль языков доказательств у нас здесь играют специальные языки дедуктивного программирования.

Наиболее близкое к ним понятие в логике – доказательство как терм – выражение, которое, вообще говоря, можно вычислять (выполнять).

ПРЕДШЕСТВУЮЩИЕ ИССЛЕДОВАНИЯ

Проблема автоматизации синтеза алгоритмов программ и других технических решений в последние годы широко исследуется. Отметим некоторые работы последних лет.

При автоматизации синтеза программ большое внимание уделяется ранжированиию используемых объектов, в частности вводится такой подход в работе [5].

Также была проведена работа по синтезу программ с использованием количественных целей [6, 7]. Идея сглаживания программы [7] состоит в том, чтобы свести задачу синтеза к последовательности задач численной оптимизации, где программа аппроксимируется в некотором непрерывном пространстве.

При дедуктивном синтезе используется теория типов уточнения [8, 22], которая изучает типы, снабженные предикатами из разрешимой логики.

Основанный на типах взгляд на дедуктивный нисходящий синтез интерпретирует проблему синтеза как проблему «обитаемости» типа синтезируемой функции, что исследуется в работе [9].

Идея ранжировать программы при синтезе вдохновлена работой по обучению методам ранжирования [10], используемым при поиске информации. Как выяснилось, проблема поиска оптимальной обучающей последовательности является NP-трудной [11].

В работах [12] и [18] используется информация о типах в качестве основного фактора для решения проблемы синтеза фрагмента вызовов интерфейсов из частично построенной программы. Они решают ее с помощью поиска, дополненного функцией ранжирования и пространственными абстракциями. Как показано в [13], можно использовать средства метапрограммирования Racket для определения быстрого и компактного синтеза с исполняемой семантикой. Генетическое программирование использовалось для исправления ошибок в императивных программах [14].

Наиболее заметным недавним развитием в индуктивном синтезе является мета-интерпретативное обучение [15]. Оно расширяет классические методы обобщения эффективной реализацией изобретения предикатов: автоматическое введение полезных предикатов в вывод.

Синтез программ был применен для изучения модельной структуры в виде вероятностной программы [16].

Современные успешные применения методов синтеза используют умные ориентированные на области применения эвристики для прополки вывода [17].

Важное значения снова приобретает пополнение частично заданных выражений, управляемое типами [18]. Средства синтеза позволяют разделять области деятельности: разработчик приложения может сосредоточиться на своей задаче, не смешивая ее с деталями кодирования запросов, решаемых с помощью системы синтеза [19].

Для преодоления сложности задачи построения кода предложены, в частности, статистические модели [20]. Методы машинного обучения широко используются для синтеза и оптимизации программ. Их приложения включают в себя автоматический синтез программ из зашумленных данных [21].

Для синтеза программ рассматриваются «умные» алгоритмы оптимизации, устойчивые к задержкам на локальных оптимумах и ограниченные только их временными бюджетами [23].

Один из подходов к синтезу – использование шаблонов, которые намекают на син-

таксическую структуру строящегося артефакта [24]. Программирование с помощью систем синтеза может использовать символический оценщик, встраиваемый в виртуальную машину системы программирования Racket. Он следует стандартной стратегии оценки, отслеживая символические выражения и оценивая конкретные выражения с использованием семантики Racket. Виртуальная машина реализует эту технику для объединения символических выражений, созданных различными условными ветвями, в поток управления программы [25]. При исследовании сложностных ограничений на синтезируемые алгоритмы в работах [26] и [27] были найдены ограничения на структуру формул, выражающих постановки задач, при которых эффективность получаемых алгоритмов была достаточно высокой. Настоящая работа является продолжением работы автора и С. Г. Маслова [28].

ОСНОВНАЯ ИДЕЯ

Мы предполагаем, что постановке задачи предшествует изучение предметной области и выявление проблем, связанных с ней. Предполагаем также, что для формулирования задач решения выявленных проблем строится определенная терминологическая система. Для успешного использования имеющимся традиционным аппаратом конструктивного дедуктивного синтеза терминологическая система должна иметь определенную структуру конструктивной онтологии. Она состоит в том, что все термины разбиваются на три класса:

- термины, обозначающие исходные типы значений, все исходные значения должны быть конструктивными объектами, то есть однозначно задаваться конечными цепочками символов из заранее выбранного конечного алфавита, с каждым таким термином связано содержательное описание на естественном для человека языке и процедуры передачи соответствующих значений между исполнителями строящихся решений задач;

- термины, обозначающие конструктивные предикаты: конструктивные свойства и конструктивные отношения, конструктивность здесь означает, что свойства и отношения должны иметь реализации – специальные значения, конструктивные объекты, подтверждающие соответствующие свойства и отношения, таким образом свойства в сущности являются атрибутами, для передачи этих значений должны быть заданы соответствующие процедуры, а для предикатов – соответствующие неформальные описания, кроме того, для каждого предиката следует зафиксировать количество и типы аргументов; можно считать, что предикат это процедура, вырабатывающая по заданным аргументам определенный тип значений, можно считать, что с ложным значением предиката связан пустой тип; иногда при решении задач, связанных с обработкой ошибок, удобно считать, что тип ложного значения не пуст, но состоит исключительно из сообщений об ошибках;

- термины, обозначающие конструктивные функции (отображения, функционалы, операторы), функции могут быть как естественными объектами предметной области, так и искусственными объектами, строящимися при решении задачи, для каждой функции фиксируется число и типы аргументов и структура вырабатываемого значения (см. описание примера языка ниже), функция снабжается и неформальным описанием. Требуемые спецификации исходных типов, предикатов, операторов записываются на специальных языках (см. пример ниже). Формулировка конструктивной задачи как задачи построения конструктивных функций (функционалов, операторов) также записывается на таком языке и вдобавок снабжается неформальным описанием. Формальная постановка называется спецификацией конструктивной задачи.

Возможны и более сложные задачи, включающие построение не только новых операторов, но и новых типов и предикатов. Как правило, это задачи моделирования.

ТЕХНОЛОГИЯ УЧЕТА ПРЕДМЕТНЫХ ТЕРМИНОВ

Предметная область, прежде всего, содержит определённые типы предметов.

Это означает, что с каждым выявленным типом предметов требуется связать некоторое имя.

Поскольку мы разрабатываем человеко-машинный подход, то необходимо указать для человека, работающего с этой системой, содержательный смысл этого типа предметов. Например,

Item:Type – детали и сборочные единицы в производственном процессе.

При этом запись x:Item означает, что x принадлежит типу Item, т.е. x – некоторая конкретная деталь или сборочная единица.

Вообще запись вида A:B, где сущность A относится к классу B, будем называть формой спецификации. Для понимания такой записи человеком в человеко-машинной системе желательно комментирование таких форм на языке, естественном для человека:

A:B, - C.

где C – комментарий на понятном для человека языке. Такая форма самодокументирования спецификаций используется как для передачи спецификации между людьми для дальнейшей обработки, так и для самодисциплины субъекта-разработчика.

УЧЕТ ПРЕДИКАТНЫХ ТЕРМИНОВ

Прежде всего, заметим, что, в соответствии с конструктивно-реализационной парадигмой, конструктивные свойства предметов должны иметь подтверждения – реализации этих свойств. Эти подтверждения являются, в свою очередь, снова предметами. Например, свойство детали быть покупной должно подтверждаться информацией о ее покупке. Таким образом, конструктивное свойство на самом деле – некоторый атрибут. Это же касается и конструктивных отношений: они также должны подтверждаться. Например, утверждение о том, что один продукт может быть переработан в другой продукт, должно подтверждаться некоторым описанием технологии переработки. Такой подход удобен тем, что экономит число сущностей, описываемых при постановке задачи: каждое свойство и отношение скрывают в себе еще одну сущность, их реализацию, которая будет обрабатываться в решении задачи. Поэтому при построении онтологии действует общая рекомендация: как можно больше предметов объявить подтверждениями свойств и отношений. Это упрощает логику решения задач. Свойства и отношения будем называть далее по традиции предикатами.

Далее используем следующие обозначения:

- запись P(x) означает, что предмет x обладает свойством P,

- запись p:P(x) означает, что предмет p – подтверждение того, что предмет x обладает свойством P (другими словами, p – одно из значений атрибута P предмета x),

- запись Q(x,y) означает, что предмет x находится с предметом y в отношении Q,

- запись q:Q(x,y) означает, что предмет q подтверждает то, что предмет x находится с предметом y в отношении Q, то же используем для трехместных отношений и для отношений большей вместимости.

Запись P:(x:X=>t:Type) означает, что P – одноместный предикат с аргументом типа X. Аналогично будем использовать записи

Q:(x:X,y:Y=>t:Type),

где X и Y – типы аргументов Q,

R:(x:X,y:Y,z:Z=>t:Type)

и т.д.

УЧЕТ КОНСТРУКТИВНЫХ ОПЕРАТОРОВ (ФУНКЦИЙ, ФУНКЦИОНАЛОВ)

При описании предметной области предполагается, что между ее элементами имеются функциональные связи. Это особые применяемые предметы. Ими могут быть, в частности, ранее созданные алгоритмы. Излагаемый здесь подход требует, чтобы свойства этих функциональных предметов могли быть описаны способами, описанными ниже.

Общий вид формы спецификации функционального предмета:

f : (A=>B),

где A – цепочка форм спецификаций аргументов функции f,

B – цепочка вариантов спецификаций результата применения f к своим аргументам.

Здесь предполагается, что число аргументов функционального предмета фиксировано, а результат будет иметь один из заранее заданного конечного списка видов. Следовательно, необходимо привести описание предметной области к этому виду.

Например, при наличии переменного числа аргументов нужно ввести понятие списка аргументов и т. п.

Формы спецификации в цепочке A будем разделять запятыми.

Например, запись A может выглядеть так:

n:I,c:C(i).

Это означает, что специфицируемый функциональный предмет применяется к упорядоченной паре предметов, первый из которых имеет тип I, а второй – подтверждение свойства C для первого предмета.

Варианты в B, если их несколько, будем разделять вертикальной чертой, каждый из этих вариантов, так же как и A, цепочка форм спецификаций. Например, B здесь может иметь вид:

d:S(i)|j:I,r:R(i,j).

Это означает, что в первом варианте получается подтверждение свойства S для предмета i, а во втором варианте – пара, состоящая из ещё одного предмета типа I и подтверждения того, что эти предметы находятся в отношении R.
B итоге всю форму спецификации будем записывать как

f:(n:I,c:C(i)=>d:S(i)|j:I,r:R(i,j)).

Будем также использовать и сокращённую запись, получающуюся по очевидным правилам:

f:(n:I,C(i)=>S(i)|j:I,R(i,j)).

Заметим, что здесь все используемые имена должны быть введены локально, как имена в формах спецификаций или известны из контекста (конструктивной онтологии). Предопределённым считается имя Type. Все остальные имена требуется определять.

Возможны и вложенные спецификации функциональных объектов, например,
f:(x:X,g:(y:X,r:P(y)=>s:Q(y)),p:P(x)=>q:Q(x)).

Последнюю запись можно представить, как запись задачи построения программы для некоторого исполнителя, который по предметам x, g и p указанных типов строит предмет q указанного типа.

Вообще говоря, онтологию для решения задачи можно представить как посылку одной большой спецификации функционального предмета. В итоге постановка задачи в целом представляется как форма спецификации одного функционального предмета – решения поставленной задачи.

Приведем простой пример задачи.

Требуется построить функциональный предмет t1 со следующей спецификацией:

t1:(

Node:Type, – имеется тип «узлы связи».

Base:(n:Node=>t:Type), – имеется свойство узла связи «быть базовой станцией».

Link:(m:Node,n:Node=>t:Type)

- Имеется отношение между узлами связи «можно установить связь».

- Имеются четыре функциональных предмета для осуществления следующих действий.

bs:(x:Node=>y:Node,b:Base(y),l:Link(x,y)) – связать узел с базовой станцией.

bc:(x:Node,y:Node,Base(x),Base(y)=>l:Link(x,y)) – связать базовые станции.

rc:(x:Node,y:Node,Link(x,y)=>Link(y,x)) – обратить связь,

tr:(x:Node,y:Node,z:Node,Link(x,y), Link(y,z)=>Link(x,z)) – соединение каналов,

m:N – дан узел.

n:N – дан еще один узел.

=>

Link(m,n) – требуется связать эти узлы.

)

ЗАПИСЬ РЕШЕНИЙ ЗАДАЧ

Мы предлагаем для записи решений задач чисто аппликативный подход. Это значит, что запись любого алгоритма состоит только из записей применений одних предметов к другим и записей построений новых искусственных предметов (алгоритмов) для решения частных задач. При этом не будем придерживаться широко распространенного минималистского лямбда-языка, а из соображений удобства работы человека в человеко-машинной системе предлагаем использовать немного более богатые средства. Тем не менее такие традиционные средства, как организация циклов и рекурсий, работа со сложными структурными объектами, будем считать опциональными. Это значит, что если нам нужны такие средства (например – организация цикла), то считаем соответствующие инструменты частью предметной области, описываемой онтологией. Вопросы

логической корректности этих средств в настоящей работе не рассматриваются и являются предметом отдельных исследований.

Рассмотрим каноническую запись наших алгоритмов. Все другие варианты записи, вводимые для удобства, будем считать сокращениями этой канонической записи. В свою очередь, конечно, и саму каноническую запись можно перевести на лямбда-язык, но это сделает ее менее удобной для человека.

Любой алгоритм будем записывать состоящим из 4 частей:

1) цепочка имен локальных предметов, обозначающих следующее:

●сначала идут соответственно аргументы алгоритма (входы),

●затем идут соответственно варианты завершения алгоритма (выходы);

2) имя действия, которое необходимо выполнить первым;

3) в скобках через запятую – аргументы этого действия, каждый

●аргумент – либо имя предмета, либо в скобках снова запись алгоритма (для наглядности будем использовать здесь фигурные скобки), аргументом также может быть и спецификация в случае, если нужно передать тип значения;

4) если выполненное действие – не выход, то алгоритмы, продолжающие выполненное действие в каждом из вариантов его завершения, каждый из алгоритмов заключается в (фигурные) скобки, последний алгоритм в скобки можно не заключать; заметим, что имена выходов в этих алгоритмах не нужны, так как используются выходы объемлющего алгоритма; если выполненное действие – выход, то эта часть пустая: у выхода нет своих выходов и продолжения ему не требуются. Заметим, что обрабатываемые предметы могут быть как обычными предметами области, так и типами или предикатами. В последнем случае исполнитель (будь то автомат или человек) получает описание этого типа, которое, в частности, должно обязательно содержать правила работы с предметами этого типа (как их хранить, передавать, утилизировать) на языке, обрабатываемом этим исполнителем.

В соответствии с парадигмой дедуктивного синтеза любое решение задачи, согласованное с ней по типам данных, является в сущности доказательством постановки задачи, если ее понимать как конструктивное утверждение, т.е. логическую формулу.

В качестве примера проиллюстрируем часть описанных средств для записи алгоритма t1, задача построения которого была описана выше. Текст снабдим двумя видами комментариев. Одни комментарии начинаются с двоеточия, это формальные комментарии, указывающие тип предмета для только что определенного имени. Такой комментарий заканчивается запятой или точкой с запятой. Другие комментарии – неформальные, начинаются со знака "-" и заканчиваются точкой.

Итак, решение задачи t1 можно записать следующим образом:

t1={

Node:Type, – исполнитель получает правила работы с узлами.

Base:(n:Node=>Type), – исполнитель получает правила работы с информацией о базовых станциях.

Link:(m:Node,n:Node=>Type), – исполнитель получает правила работы с информацией о соединениях.

bs:(x:Node=>y:Node,b:Base(y),l:Link(x,y)) , – искатель базы.

- Здесь предполагается, что приведенной информации достаточно, чтобы исполнителю корректно работать с функциональным предметом.

bc:(x:Node,y:Node,Base(x),Base(y)=>l:Link(x,y)), – связь баз.

rc:(x:Node,y:Node,Link(x,y)=>Link(y,x)), – обращение связи.

tr:(x:Node,y:Node,z:Node,Link(x,y),Link(y,z)=>Link(x,z)), – транзитивность связи.

m:N, – первый узел.

n:N, – второй узел.

return:(lmn:Link(m,n)); – выход для соединения этих узлов.

- Выходы специфицируются в скобках, чтобы отличаться от входов.

bs(m)p:Node,bp:Base(p),lmp:Link(m,p); – поиск первой базы.

bs(n)q:Node,bq:Base(q),lnq:Link(n,q); – поиск второй базы.

bc(p,q,bp,bq)lpq:Link(p,q); – соединение баз.

rc(n,q,lnq)lqn:Link(q,n); – обращение одной из связей.

tr(m,p,q,lmp,lpq)lmq:Link(m,q); – соединение каналов.

tr(m,q,n,lmq,lqn)lmn:Link(m,n); – соединение каналов.

return(lmn) – результат готов.

}

Одновременно это решение является доказательством утверждения, которым является спецификация задачи t1.

То, что названо формальными комментариями, хотя и восстанавливается однозначно, опускаться нами не будет. В сложных системах исполнения решений такой комментарий на самом деле описывает вполне конкретное действие – передачу описанного значения. Например, поиск первой базы можно развернуть следующим образом:

bs(m) – выполнение действия bs с передачей аргумента m.

p:Node, – импорт первого результата с помощью процедуры Node.

bp:Base(p), – импорт второго результата с помощью процедуры Base с параметром p.

lmp:Link(m,p); – импорт третьего результата с помощью процедуры Link с параметрами m и p.

Возможна и более подробная запись:

bs(m:Node) – выполнение действия bs с передачей аргумента m.

p:Node, - импорт первого результата с помощью процедуры Node.

bp:Base(p:Node), – импорт второго результата с помощью процедуры Base с параметром p.

lmp:Link(m:Node,p:Node); – импорт третьего результата с помощью процедуры Link с параметрами m и p.

В принципе можно использовать и еще более подробную запись. Для последней строки это будет

lmp:Link(m:Node:Type,p:Node:Type);

Здесь явно указывается, что сама процедура Node передается универсальной процедурой Type. Считается, что наша система должна изначально владеть процедурой Type и передавать ее не требуется. Заметим, что настолько подробная запись практически не нужна.

Другая строка программы:

bc(p,q,bp,bq)lpq:Link(p,q);

записывается так:

bc(p:Node,q:Node,bp:Base(p:Node),bq:Base(q:Node))lpq:Link(p:Node,q:Node);

Решение без комментариев выглядит следующим образом:

t1={Node,Base,Link,bs,bc,rc,tr,m,n,return;
bs(m)p,bp,lmp;
bs(n)q,bq,lnq;
bc(p,q,bp,bq)lpq;
rc(n,q,lnq)lqn;
tr(m,p,q,lmp,lpq)lmq;
tr(m,q,n,lmq,lqn)lmn;
return(lmn)
}

Возможна и совсем короткая запись, при которой имена, не используемые в качестве аргументов предикатов, опускаются:

t1={Node,Base,Link,bs,bc,rc,tr,m,n,return;
bs(m)p;bs(n)q;bc(p,q);rc(n,q);tr(m,p,q);tr(m,q,n);return()}.

ПРИМЕРЫ СПЕЦИФИКАЦИЙ ФУНКЦИЙ

Приведем пример спецификации оператора цикла динамического программирования. С логической точки зрения этот цикл – рекурсия по некоторому строгому частичному порядку R на множестве I. Описание части онтологии со спецификацией этого оператора для заданных I и R с комментариями выглядит следующим образом:

I:Type, – тип переменной цикла.

R:(i:I,j:I=>t:Type), – отношение на значениях этого типа.

C:(B:(i:I=>t:Type)=>u:Type), – ограничение на задачи, которые могут решаться в цикле.

for: – имя оператора цикла как функционального предмета области.

(- Начало спецификации оператора цикла.

B:(i:I=>t:Type), – решаемая циклом задача.

c:C(B), – ограничение, накладываемое на задачу.

n:I, – верхняя граница цикла.

do:(– начало спецификации тела цикла.

i:I, - параметр цикла.

mem:(– начало спецификации памяти цикла.

j:I, - индекс памяти цикла.

rij:R(i,j) – ограничение памяти цикла.

=> bj:B(j) – содержимое памяти цикла.

) - конец спецификации памяти цикла.

=> bi:B(i) – цель шага цикла.

) - конец спецификации тела цикла.

=> bn:B(n) – цель всего цикла.

) - конец спецификации всего цикла.

Заметим, что приведенный выше оператор for может быть явно выведен из некоторых ограничений, на задачи, решаемые в цикле. Например, ограниченность длины цепочек, связанных отношением R, может быть выражена следующим ограничением:

C=

{

B:(i:I=>t:Type), - решаемая задача.

return:(Type);

- возвращается конструктивное логическое значение.

return

((w:I,x:I,y:I,z:I,rwx:R(w,x),rxy:R(x,y),ryz:R(y,z)=>bw:B(w)))

- если найдена слишком длинная цепочка в R, то считаем, что задача решена.

}

Здесь в качестве примера приведено ограничение 3 на длину цепочки предметов, связанных отношением R. Аналогично можно ввести любое другое ограничение.

Пример задачи, решаемой с помощью оператора for:

t2:(

I:Type, – пусть I – обозначение для некоторых продуктов.

R:(I,I=>Type), – R(i,j) – технологии получения i из j.

C:((I=>Type)=>Type),

- ограничение на решаемые в цикле задачи.

B:(I=>Type), – B(i) – стоимость продукта i.

c:C(B), – ограничение на решаемую задачу

for:

(B:(I=>Type),c:C(B),n:I,(i:I,(j:I,R(i,j)=>B(j))=>B(i))=>B(n))

- спецификация оператора цикла.

db:(i:I=>B(i)|j:I,R(i,j)), – база данных, в которой записано, что каждый продукт – покупной с известной стоимостью или получается из другого продукта.

cn:(i,j:I,R(i,j),B(j)=>B(i)),

- процедура вычисления стоимости производимого продукта по стоимости предшественника и способу производства.

n:I - данный продукт.

=>B(n) – требуется найти стоимость данного продукта.

)

Пример решения этой задачи:

t2=

{I,R,C,B,c,for,db,cn,n,

- комментарии к этим именам приведены выше.

return:(B(n)); – спецификация оператора возврата из программы.

for – запускаем цикл.

(B, – в цикле рашаем задачу B.

c, – задача B удовлетворяет условию C.

n, – цикл выполняем до достижения продукта n.

{ – начало тела цикла

i:I, – параметр цикла

mem: – память цикла.

(j:I, – индекс памяти цикла.

R(i,j) – ограничение памяти цикла

=>B(j) – содержимое памяти цикла

),

continue:(B(j)); – оператор завершения шага цикла.

db(i) – рассматриваемый в цикле продукт ищем в базе данных.

{ – первый случай: продукт – покупной.

bi:B(i), – его известная стоимость.

continue(bi) – эту стоимость подаём на завершение шага цикла.

} – второй случай: продукт производится из другого продукта

j:I, – продукт предшественник.

rij:R(i,j); – способ получения текущего из предшественника.

mem(j,rij) – ищем продукт-предшественник в памяти цикла.

bj:B(j); – найденная стоимость предшественника.

cn(i,j,rij,bj) – вычисляем стоимость текущего продукта.

bi:B(i); – вычисленная стоимость текущего продукта.

continue(bi) – эту стоимость подаём на завершение шага цикла.

} – конец тела цикла.

) – конец цикла.

bn:B(n); – результат работы цикла.

return(bn) – подаем результат на возврат из программы.

}

Решаемая в цикле задача может быть специфицирована сложной формулой. При этом эта сложная формула рассматривается как конструктивная логическая функция, которая подставляется на место предиката, определяющего задачу, решаемую в цикле.

ЗАКЛЮЧЕНИЕ

В работе представлен понятийный аппарат, предназначенный для использования в человеко-машинных и физико-антропно-технических системах.

Предложенные понятия позволяют формулировать и решать конструктивные задачи с помощью таких систем и для таких систем.

Понятия образуют иерархию типов и опираются на небольшие универсалии, включающие понятие передачи типа значений и базовые языки постановок и решений.

Предложенный аппарат предназначается для описания конструктивных физико-антропно-технических систем.

СПИСОК ЛИТЕРАТУРЫ

1. **Kleene S. C.** Introduction to metamathematics – North-Holland, 1951, 500 p.

2. **Шанин Н. А.** Об иерархии способов понимания суждений в конструктивной математике // Тр. Матем. ин-та АН СССР, 1973, т. 129, с. 203–266. [N. A. Shanin, "On the hierarchy of ways to understand judgments in constructive mathematics", in: "Tr. Matem. in-ta AN SSSR", 1973, V. 129, pp. 203-266.]

3. **Марков А. А.** Попытка построения логики конструктивной математики // В сб.: Исследования по теории алгорифмов и математической логике, М., 1976, т. 2, С.3–31. **[**A. A. Markov, "An attempt to build a logic of constructive mathematics". in: Research in the theory of algorithms and mathematical logic, M, 1976, V. 2, pp. 3-31.**]**

4. **Бельтюков А. П.** Малые сложностные классы и автоматический дедуктивный синтез алгоритмов // Известия института математики и информатики УдГУ, Ижевск, 1995, вып. 2, С. 3–89. **[**A. P. Beltiukov, "Small complexity classes and automatic deductive synthesis of algorithms", in: Izvestia instituta matematiki i informatiki UdGU, Izhevsk,1995, 2, pp. 3-89.**]**

5. **Optimizing** synthesis with metasketches /J. Bornholt et al. // ACM SIGPLAN Notices, ACM, 2016, volume 51, pp. 775–788. [James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze "Optimizing synthesis with metasketches". In ACM SIGPLAN Notices, volume 51, pp. 775–788. ACM, 2016.]

6. **Measuring** and synthesizing systems in probabilistic environments / K. Chatterjee, et al, // J. ACM, 62(1):9:1–9:34, March 2015. ISSN 0004-5411. **[**Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh "Measuring and synthesizing systems in probabilistic environments". J. ACM, 62(1):9:1–9:34, March 2015. ISSN 0004-5411.**]**

7. **Chaudhuri S., Clochard M., and A. Solar-Lezama**. Bridging boolean and quantitative synthesis using smoothed proof search // ACM SIGPLAN Notices, January 2014, 49(1) pp. 207–220, ISSN 0362-1340. **[**S. Chaudhuri, M. Clochard, and A. Solar-Lezama "Bridging boolean and quantitative synthesis using smoothed proof search". ACM SIGPLAN Notices, 49(1):207–220, January 2014.**]** ISSN 0362-1340.

8. **Flanagan C.** Hybrid type checking // ACM Sigplan Notices, volume 41, ACM, 2006, pp. 245–256. **[**Cormac Flanagan "Hybrid type checking". In ACM Sigplan Notices, volume 41, pp. 245-256. ACM, 2006.**]**

9. **Example-directed** synthesis: a type-theoretic interpretation/J. Frankle, et al, // ACM SIGPLAN Notices, volume 51, ACM, 2016, pp. 802–815. **[**Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewi "Example-directed synthesis: a type-theoretic interpretation". In ACM SIGPLAN Notices, volume 51, pp. 802-815. ACM, 2016.**]**

10. **An efficient** boosting algorithm for combining preferences / Freund Y., et al, // The Journal of machine learning research, 2003, 4, pp. 933–969. **[**Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer "An efficient boosting algorithm for combining preferences". The Journal of machine learning research, 4:933-969, 2003.**]**

11. **On the** complexity of teaching/Goldman S. A. and Kearns M. J. // Journal of Computer and System Sciences, 50, 1992, pp. 303–314. **[**Sally A. Goldman and Michael J. Kearns "On the complexity of teaching". Journal of Computer and System Sciences, 50:303–314, 1992.**]**

12. **Complete** completion using types and weights / T. Gvero, et al, // In ACM SIGPLAN Notices, ACM, 2013, V. 48, pp. 27–38. **[**Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac "Complete completion using types and weights". In ACM SIGPLAN Notices, vol. 48, pp. 27-38. ACM, 2013.**]**

13. **Educational** pearl: Automata via macros / S. Krishnamurthi et al // Journal of Functional Programming, 2006, 16(03), pp. 253–267. **[**Shriram Krishnamurthi "Educational pearl: Automata via macros". Journal of Functional Programming, 16(03):253–267, 2006.**]**

14. **A generic** method for automatic software repair/ C. Le Goues, at al, // IEEE Transactions on Software Engineering, 2012, 38(1), pp. 54–72. **[**Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer "Genprog: A generic method for automatic software repair". IEEE Transactions on Software Engineering, 38(1):54–72, 2012.**]**

15. **Meta-interpretive** learning of higher-order dyadic Datalog: Predicate invention revisited / Muggleton S., Lin D., and Tamaddoni-Nezhad A. // Machine Learning, 2015, 100(1), pp.49–73. [Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad "Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited". Machine Learning, 100(1):49–73, 2015.]

16. **Efficient** synthesis of probabilistic programs / A. V. Nori, et all, // ACM SIGPLAN Notices, ACM, 2015, V. 50, pp. 208–217. [Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy “Efficient synthesis of probabilistic programs”. In ACM SIGPLAN Notices, vol. 50, pp. 208-217. ACM, 2015.]

17. **Automatically** improving accuracy for floating point expressions / P. Panchekha, at al. // ACM SIGPLAN Notices, 2015, 50(6), pp. 1–11. [Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock Automatically improving accuracy for floating point expressions. ACM SIGPLAN Notices, 50(6):1-11, 2015.]

18. **Type-directed** completion of partial expression / D. Perelman, at al. // In ACM SIGPLAN Notices, ACM, 2012, V. 47, pp. 275–286. [Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman “Type-directed completion of partial expressions”. In ACM SIGPLAN Notices, V. 47, pp. 275-286. ACM, 2012.]

19. **Chlorophyll**: Synthesis-aided compiler for low-power spatial architectures/ P. P. Mangpo, at al. // ACM SIGPLAN Notices, ACM, 2014, V. 49, pp. 396–407. [Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik “Chlorophyll: Synthesis-aided compiler for low-power spatial architectures”. In ACM SIGPLAN Notices, vol. 49, pp. 396-407. ACM, 2014.]

20. **Raychev V., Vechev M., and Yahav E.** Code completion with statistical language models // ACM SIGPLAN Notices, ACM, 2014, vol. 49, pp. 419–428. **[**Veselin Raychev, Martin Vechev, and Eran Yahav “Code completion with statistical language models”. In ACM SIGPLAN Notices, vol. 49, pp. 419-428. ACM, 2014.**]**

21. **Learning** programs from noisy data / Raychev V., at al // ACM SIGPLAN Notices, ACM, 2016, volume 51, pp. 761–774. [Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause “Learning programs from noisy data”. In ACM SIGPLAN Notices, vol, 51, pp. 761–774. ACM, 2016.]

22. **Rondon P. M., Kawaguci M., and Ranjit J.** Liquid types // ACM SIGPLAN Notices, ACM, 2008, vol. 43, pp. 159–169. [Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala “Liquid types”. In ACM SIGPLAN Notices, vol. 43, pp. 159–169. ACM, 2008.]

23. **Schkufza E., Sharma R., and Aiken A.** Stochastic superoptimization // ACM SIGARCH Computer Architecture News, ACM, 2013, vol. 41, pp. 305–316. [Eric Schkufza, Rahul Sharma, and Alex Aiken Stochastic superoptimization. In ACM SIGARCH Computer Architecture News, vol. 41, pp. 305–316. ACM, 2013.]

24. **Srivastava S., Gulwani S., and Foster J. S.** Templatebased program verification and program synthesis // International Journal on Software Tools for Technology Transfer (STTT), 2013, 15(5-6), pp. 497–518. [Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster “Templatebased program verification and program synthesis”. International Journal on Software Tools for Technology Transfer (STTT),

15(5-6):497–518, 2013]

25. **Torlak E. and Bodik R. A** lightweight symbolic virtual machine for solver-aided host languages // ACM SIGPLAN Notices, ACM, 2014, vol. 49, pp. 530–541. **[Emina Torlak and Rastislav Bodik** “A lightweight symbolic virtual machine for solver-aided host languages”, In ACM SIGPLAN Notices, ACM, 2014, vol. 49, pp. 530–541.**]**

26. **Beltiukov A. P.** Intuitionistic formal theories with realizability in subrecursive classes // Annals of Pure and Applied Logic, 89, 1997, pp. 3-15. **[**A. P. Beltiukov “Intuitionistic formal theories with realizability in subrecursive classes” In: Annals of Pure and Applied Logic, 89, 1997, pp. 3-15.**]**

27. **Beltiukov A. P.** A strong induction scheme that leads to polynomially computable realizations // Theoretical Computer Science, 322 (2004), P. 17-39. **[**A. P. Beltiukov “A strong induction scheme that leads to polynomially computable realizations” In: Theoretical Computer Science, 322 (2004), P. 17-39.**]**

28. **Beltiukov A. P., Maslov S. G.** Organizing understanding in the framework of ergatic system // CSIT’2015: Proceeding of the 17th International Workshop on Computer Science and Information Technologies, Rome, Italy, September 22-26, 2015, V.1, – Ufa.: Ufa State Aviation Technical Univercity, 2015, P. 60-64. **[**A. P. Beltiukov, S. G. Maslov “Organizing understanding in the framework of ergatic system”. In CSIT’2015: Proceeding of the 17th International Workshop on Computer Science and Information Technologies, Rome, Italy, September 22-26, 2015, V.1, Ufa.: Ufa State Aviation Technical Univercity, 2015, P. 60-64.**]**

ОБ АВТОРЕ

БЕЛЬТЮКОВ Анатолий Петрович, заведующий кафедрой теоретических основ информатики. Диплом: математик (ЛГУ, 1975). Доктор физико-математических наук по специальности: «Теоретические основы информатики» (СПбГУ, 1995.)

METADATA

Title: Formalization of tasks in conditions of deductive synthesis.

Author: A. P. Beltiukov

Affiliation:
Udmurt State University (UdGU), Russia.

Email: belt.udsu@mail.ru.

Language: Russian.

Source: SIIT, no. 1, pp.75-85, 2019. ISSN 2658-5014 (Print).

Abstract: The construction and transformation of formulations of constructive problems in the paradigm of the deductive synthesis of their solutions is described. An approach to constructing a formalization of a constructive problem is proposed, when both the informal formulation of the problem and the informal idea of its solution are already roughly known. In this situation, a system of concepts is developed from the informal formulation of the problem and the informal idea of its solution, which forms the basis of ontology: the types of objects in the most common form for this task, constructive attribute properties, constructive relations and specifications of the source and target functions and operators. By deductive synthesis in the present paper we understand the logical inference of solutions of constructive problems from their formal statements in certain systems of inference. By solving constructive problems, we mean the construction of algorithms, data structures, and objects containing both data and algorithms. Moreover, the task can be expanded so that parts of solutions can be both physical objects and people's competences. Such decisions can be called physic-anthropic-technical. At the same time, the very system for constructing such solutions may turn out to be physic-anthropic-technical, which is what the present work calcu-

lates. Instead of traditional languages of logical formulas, task specification languages are considered that are more suitable for describing the subject area in computer science. The role of evidence languages here is played by special deductive programming languages. The concept closest to them in logic is proof as a term — an expression that, generally speaking, can be calculated (carried out). Task setting is preceded by a study of the subject area and the identification of problems associated with it. To formulate problems of solving identified problems, a certain terminological system is built. For the successful use of the existing traditional apparatus of constructive deductive synthesis, the terminological system must have a definite structure of constructive ontology. It consists in the fact that all terms are divided into three classes: terms denoting types of values, terms denoting predictions, and terms denoting constructive functions. We consider terms denoting initial types of values, all initial values must be constructive objects, that is, uniquely defined by finite strings of characters from a pre-selected finite alphabet, each such term is associated with a meaningful description in a language natural to humans values between the performers of problem solving. We also consider terms denoting constructive predicates: constructive properties and constructive relations. Constructiveness here means that properties and relations must have realizations — special values, constructive objects confirming the corresponding properties and relations. Thus the properties are in essence attributes. For the transfer of these values, the corresponding procedures must be specified, and for the predicates - the corresponding informal descriptions. In addition, for each predicate, quantities should be fixed and the types of all its arguments. We can assume that a predicate is a procedure that produces a certain type of values for a given argument, we can assume that an empty type is associated with a false value of the predicate. Sometimes, when solving problems related to error handling, it is convenient to assume that the type of a false value is not empty, but it consists solely of error messages. At last we consider terms denoting constructive functions (mappings, functionals, operators), functions can be both natural objects of the domain, and artificial objects built when solving a problem value (see the description of the example language below), the function is supplied with a non-formal description. Required specifications of source types, predicates, operators are written in special languages. The formulation of the constructive task as the task of constructing constructive functions (functionals, operators) is also written in such a language and in addition is supplied with a non-formal description. The formal formulation is called the specification of a constructive task. More complex tasks are possible, including the construction of not only new operators, but also new types and predicates. As a rule, these are modeling tasks.

Key words: deductive synthesis; physical and anthropic-technical systems; ontologies; constructive logic; man-machine systems.

About authors:

BELTIUKOV, Anatoly Petrovich, Head of the Department of the Theoretical Foundations of Informatics. Diploma: mathematician (LSU, 1975). Doctor of Physical and Mathematical Sciences in the specialty: “Theoretical Foundations of Computer Science” (St. Petersburg State University, 1995.)