Компьютерные инструменты в образовании, 2015
№ 6: 19–33
УДК: 004.825+004.43
http://ipo.spb.ru/journal

СРАВНЕНИЕ ЭФФЕКТИВНОСТИ РЕАЛИЗАЦИЙ АЛГОРИТМА ПОИСКА ВЫВОДА В СЕКВЕНЦИАЛЬНОМ ИСЧИСЛЕНИИ ВЫСКАЗЫВАНИЙ НА ЯЗЫКАХ РЕФАЛ-5 И HASKELL

Григорьев Валентин Дмитриевич

Аннотация

В статье предлагаются реализации на языках рефал-5 и Haskell алгоритма поиска вывода в секвенциальном исчислении высказываний. Реализации сравниваются по временной эффективности и удобочитаемости. В приложениях изложены базовые сведения об использованных языках программирования. Данную статью можно рассматривать как ориентирующую программиста при выборе языка программирования для реализации машины вывода в системах автоматического доказательства теорем и других интеллектуальных системах, основанных на логическом выводе.

Ключевые слова: *автоматическое доказательство теорем, пропозициональная логика, секвенциальное исчисление, рефал-5, Haskell, удобочитаемость программы, временная эффективность программы.*

1. ВВЕДЕНИЕ

Поиск вывода утверждения в заданной аксиоматической теории является основной задачей автоматического доказательства теорем (automated theorem proving) и важной задачей во многих интеллектуальных информационных системах [2, 10].

В основании машины вывода (inference engine) [2] лежит некоторая логика: пропозициональная, первого порядка, высших порядков или другая [10]. Вывод реализуется на основе исчисления, реализующего выбранную логику.

Подход к построению исчисления, позволяющего целенаправленно и эффективно осуществлять поиск вывода заданного утверждения, был предложен в 1934–1935 Г. Генценом в статьях [11, 12]. Исчисления, построенные по схеме, предложенной Генценом, называются исчислениями генценовского типа или секвенциальными исчислениями. Для пропозициональной логики и логики первого порядка секвенциальное исчисление с обратимыми правилами позволяет полностью автоматизировать процесс построения вывода [3, 6, 16].

Задача данной статьи — произвести сравнение предлагаемых реализаций по двум выбранным критериям. Так как реализации являются «естественными» для соответствующих языков, это сравнение показывает, насколько хорошо языки рефал-5 и Haskell подходят для построения машин вывода для секвенциального исчисления высказываний.

2. ОПРЕДЕЛЕНИЯ ПОНЯТИЙ СЕКВЕНЦИАЛЬНОГО ИСЧИСЛЕНИЯ ВЫСКАЗЫВАНИЙ С ОБРАТИМЫМИ ПРАВИЛАМИ

Ниже приводятся необходимые понятия из математической логики на основе [3, 6].

Под пропозициональной переменной понимается либо заглавная буква латинского алфавита, либо слово в нем, взятое в двойные кавычки.

Определение.

1. *Любая пропозициональная переменная является пропозициональной формулой.*
2. *Символы F и T являются пропозициональными формулами и называются истинностными константами.*
3. *Если A и B — пропозициональные формулы, то $(\neg A), (A \,\&\, B), (A \vee B), (A \to B), (A \leftrightarrow B)$ — пропозициональные формулы.*

Определение. *Секвенцией называется выражение вида $A_1, ..., A_m \vdash B_1, ..., B_n$, где $A_1, ..., A_m, B_1, ..., B_n$ — пропозициональные формулы. Список формул $A_1, ..., A_m$ называется антецедентом, список формул $B_1, ..., B_n$ называется сукцедентом.*

Определение. *Формульным образом секвенции $A_1, ..., A_m \vdash B_1, ..., B_n$ называется пропозициональная формула $A_1 \& ... \& A_m \to B_1 \vee ... \vee B_n$. Конъюнкция пустого списка формул (то есть при $m = 0$) полагается за T, дизъюнкция его (то есть при $n = 0$) — за F.*

Здесь приоритеты одной и той же логической связки сильнее у левого ее вхождения.

Определение. *Пусть $\Gamma, \Gamma_1, \Gamma_2, \Delta, \Delta_1, \Delta_2$ — списки пропозициональных формул, A, B — пропозициональные формулы. Секвенциальным исчислением высказываний (СИВ) называется исчисление, состоящее из следующих частей:*

1. *Алфавит:*

   $\{$*алфавит для записи пропозициональных переменных*$\} \bigcup \{T, F\} \bigcup \{\neg, \&, \vdash, (,)\}$

2. *Аксиомы: секвенции вида*

$$\Gamma_1 A \Gamma_2 \vdash \Delta_1 A \Delta_2\,, \quad \Gamma \vdash \Delta_1 T \Delta_2\,, \quad \Gamma_1 F \Gamma_2 \vdash \Delta.$$

3. *Основные правила вывода секвенций:*

$$(\neg \vdash) \frac{\Gamma_1 \Gamma_2 \vdash \Delta_1 A \Delta_2}{\Gamma_1 (\neg A) \Gamma_2 \vdash \Delta_1 \Delta_2}\,, \qquad (\vdash \neg) \frac{\Gamma_1 A \Gamma_2 \vdash \Delta_1 \Delta_2}{\Gamma_1 \Gamma_2 \vdash \Delta_1 (\neg A) \Delta_2}\,,$$

$$(\& \vdash) \frac{\Gamma_1 A B \Gamma_2 \vdash \Delta}{\Gamma_1 (A \& B) \Gamma_2 \vdash \Delta}\,, \qquad (\vdash \&) \frac{\Gamma \vdash \Delta_1 A \Delta_2 \quad \Gamma \vdash \Delta_1 B \Delta_2}{\Gamma_1 \Gamma_2 \vdash \Delta_1 (A \& B) \Delta_2}\,.$$

4. Допустимыми в СИВ являются правила:

$$(\vdash \vee)\frac{\Gamma \vdash \Delta_1 AB\Delta_2}{\Gamma \vdash \Delta_1 (A \vee B)\Delta_2}, \qquad (\vee \vdash)\ \frac{\Gamma_1 A\Gamma_2 \vdash \Delta \quad \Gamma_1 B\Gamma_2 \vdash \Delta}{\Gamma_1 (A \vee B)\Gamma_2 \vdash \Delta},$$

$$(\vdash \to)\frac{\Gamma_1 A\Gamma_2 \vdash \Delta_1 B\Delta_2}{\Gamma_1\Gamma_2 \vdash \Delta_1 (A \to B)\Delta_2}, \qquad (\to \vdash)\ \frac{\Gamma_1\Gamma_2 \vdash \Delta_1 A\Delta_2 \quad \Gamma_1 B\Gamma_2 \vdash \Delta_1\Delta_2}{\Gamma_1 (A \to B)\Gamma_2 \vdash \Delta_1\Delta_2},$$

$$(\vdash \leftrightarrow)\ \frac{\Gamma_1 A\Gamma_2 \vdash \Delta_1 B\Delta_2 \quad \Gamma_1 B\Gamma_2 \vdash \Delta_1 A\Delta_2}{\Gamma_1\Gamma_2 \vdash \Delta_1 (A \leftrightarrow B)\Delta_2}, \qquad (\leftrightarrow \vdash)\ \frac{\Gamma_1 AB\Gamma_2 \vdash \Delta_1\Delta_2 \quad \Gamma_1\Gamma_2 \vdash \Delta_1 AB\Delta_2}{\Gamma_1 (A \leftrightarrow B)\Gamma_2 \vdash \Delta_1\Delta_2}.$$

Определение. *Последовательность секвенций $S_1, ..., S_n$ называется выводом в исчислении, если каждая секвенция либо является аксиомой, либо может быть получена из предыдущих по одному из правил вывода. Последняя секвенция вывода называется выводимой в исчислении.*

3. РЕАЛИЗАЦИИ АЛГОРИТМА ПОИСКА РАЗМЕЧЕННОГО ВЫВОДА СЕКВЕНЦИИ

3.1. Обоснование выбора языков программирования

Реализация алгоритма поиска вывода в секвенциальном исчислении высказываний на языке рефал-5 была разработана профессором кафедры информатики математико-механического факультета СПбГУ Косовским Н.К. и в течение многих лет представлялась студентам в рамках курса «Рекурсивно-логическое программирование». Она демонстрирует удобочитаемость и предельную краткость, которых позволяют добиться ключевые принципы языка рефал-5 при решении классической задачи, возникающей при написании интеллектуальных систем.

Основным принципом, влияющим на простоту реализации, является принцип сопоставления с образцом, на котором базируется рефал-5. Сопоставление с образцом позволяет писать код в терминологии, близкой к предметной области, что делает его удобным и легким для чтения.

Для проведения сравнения с реализацией Н.К. Косовского автором данной статьи был выбран язык Haskell, также широко использующий сравнение с образцом, хотя и в несколько более ограниченной форме. Кроме того, существенным фактором при выборе языка Haskell послужила его гибкая система типов, позволяющая естественным для языка и простым для восприятия программистом образом ввести все необходимые сущности из области математической логики.

Сведения о языках рефал-5 и Haskell, достаточные для понимания написанных программ, приведены в приложениях А и В соответственно. Для более полного знакомства с языком рефал-5 можно порекомендовать официальную документацию [19] и статью Б. Банчева [1], а для языка Haskell — официальный сайт [13] и учебник А. Холомьева [9].

3.2. Краткое описание алгоритма

Под размеченным выводом понимается пронумерованная последовательность секвенций, для каждой из которых указывается, для какой формулы она выводится и по

какому правилу выводится сама.

На каждом шаге вычисления функция поиска размеченного вывода проверяет, не является ли текущая секвенция аксиомой, и в случае отрицательного ответа пытается применить какое-либо однопосылочное правило. В случае невозможности применения правила такого типа выполняется попытка применения двупосылочного правила. Если ни одно из них также невозможно применить, то секвенция невыводима.

3.3. Технические замечания

Для логических связок при выводе обе реализации используют следующие обозначения — ! для отрицания, & для конъюнкции, | для дизъюнкции, –> для импликации и <–> для эквивалентности. В качестве знака секвенции используется '|–', символ перевода строки — '\n'.

Для экономии места в листингах ниже приводится лишь по одному правилу вывода для однопосылочных и двупосылочных правил. Остальные конструируются аналогично в соответствии с определением секвенциального исчисления высказываний.

3.4. Реализцация на языке рефал-5

В программе используются функция *Prove*, инициализирующая стек и запускающая вычисления, функция *Analyse*, в которой собственно происходит построение вывода, а также вспомогательная функция *IncN*, используемая для увеличения на 1 счетчика номеров секвенций, расположенного в стеке с именем *N*.

Строки, начинающиеся с символа '*' — комментарии.

```
* Функция, увеличивающая на 1 счетчик номеров секвенций
IncN {
  * Берем значение счетчика из стека, увеличиваем его на 1 и кладем обратно в стек
  = <Br N'='<Add(<Dg N>) 1>>;
}

* Функция, инициализирующая используемые стеки и запускающая поиск вывода севенции
e.1
Prove {
   * Инициализация стека номеров секвенций N
   e.1 = <Br N'='*'>
      * Кладем 1 в N
      <Br N'='1>
      * Инициализация стека S номеров секвенций, предшествующих при поиске вывода
      <Br S'='*'>
      * Кладем 0 в S
      <Br S'='0>
      * Начинаем поиск вывода e.1
      <Analyse e.1>;
}

* Функция, выполняющая поиск вывода
Analyse {
* Аксиомы:
   e.0, e.0: e.1 '|–' e.4 'T' e.6 =
      <Cp N>') ' e.0 ' – axiom, for '<Dg S> <IncN> '\n';
```

```
   e.0, e.0: e.1 'F' e.2 '|-' e.3 =
      <Cp N>') ' e.0 ' - axiom, for '<Dg S> <IncN> '\n';
   e.0, e.0: e.1 t.2 e.3 '|-' e.4 t.2 e.5 =
      <Cp N>') ' e.0 ' - axiom, for ' <Dg S> <IncN> '\n';

* Однопосылочные правила:

* Проверяем, не содержит ли e.0  отрицания в антецеденте
   e.0, e.0: e.1 ('!'t.1) e.2 '|-' e.3 =
      * Собираем строку-отчет
      <Cp N>') ' e.0 '    - by (! |-) for ' <Dg S> '\n'
      * Помещаем в стек S номер текущей секвенции
      <Br S'='<Cp N>>
      * Инкрементируем счетчик номеров секвенций
      <IncN>
      * Ищем вывод секвенции, полученной из e.0
      <Analyse e.1 e.2 '|-' e.3 t.1>;

* Другие однопосылочные правила опущены для экономии места
* Реализуются аналогично, в соответствии с определением
* секвенциального исчисления высказываний

* Two-premises rules:
   e.0, e.0: e.1 '|-' e.2 (t.1 '\&' t.2) e.3 =
      <Cp N>') ' e.0 '    - by (|- \&) for ' <Dg S> '\n'
      <Br S'='<Cp N>>
      <Br S'='<Cp N>>
      <IncN>
      <Analyse e.1 '|-' e.2 t.1 e.3>
      <Analyse e.1 '|-' e.2 t.2 e.3>;

* Другие двупосылочные правила опущены для экономии места
* Реализуются аналогично, в соответствии с определением
* секвенциального исчисления высказываний

      * Если секвенция не является аксиомой и к ней нельзя
      * применить никакое обратное правило секвенциального
      * исчисления высказываний, то она невыводима
      e.1 = e.1 ' - unprovable ( '<Dg S>')''\n';
}
```

3.5. Реализация на языке Haskell

В отличие от языка рефал-5 в языке Haskell можно ввести следующие типы данных:

```
-- Определяем тип для пропозициональных формул
data BooleanExpression =
      T | F |                                    -- Логические константы
      Var String |                               -- Пропозициональная переменная
      Not BooleanExpression |                    -- Отрицание
      BooleanExpression :\&: BooleanExpression | -- Конъюнкция
      BooleanExpression :|: BooleanExpression |  -- Дизъюнкция
      BooleanExpression :->: BooleanExpression | -- Импликация
      BooleanExpression :<=>: BooleanExpression  -- Эквивалентность
```

```
-- Секвенция - два списка пропозициональных формул
data Sequention = [BooleanExpression] ::- [BooleanExpression]
```

Ниже представлен алгоритм проверки выводимости секвенции. Функция *infer* инициирует проверку выводимости секвенции, функции *prove*, *tryOnePremise*, *tryTwoPremise* реализуют логику поиска вывода.

Функция *strForRule*, конструирующая строку-отчет для заданного правила, ниже опущена как несущественная для понимания алгоритма.

Функции *leftNegation* и *rightConjuction* проверяют наличие отрицания в антецеденте и конъюнкции в сукцеденте. Аналогичные функции реализованы и для других логических связок.

Для того, чтобы построить из этих простых функций функции *tryOnePremise* и *tryTwoPremise*, реализующих поиск вывода, используется функция *build*, которая принимает список пар вида (*function*, *rule*), где *function* — функция проверки наличия связки в антецеденте или сукцеденте, *rule* — строковое представление соответствующего правила, а также номер секвенции, из которой была получена текущая секвенция, номер предыдущей секвенции в списке и еще одну функцию, которая выполняется, если ни одно из сопоставлений не привело к успеху.

В целях оптимизации для представления строковых данных используется тип *ByteString*, а не стандартный *String*.

```
-- Функция, запускающая вычисление. Извлекает строку-отчет из результат выполнения
prove
infer :: Sequention -> ByteString
infer s = toLazyByteString $ snd $ prove s 0 0

-- Проверяем, является ли секвенция аксиомой
-- То есть имеются ли одинаковые элементы в антецеденте и сукцеденте,
-- F в антецеденте или T в сукцеденте
checkAxioms (a ::- s) = (not $ null (intersect a s)) || (elem F a) || (elem T s)

-- Функция поиска вывода секвенции. На вход подается секвенция s, номер секвенции,
-- для которой она выводится for и номер предыдущей секвенции в порядке поиска
pred.
-- Если s -- аксиома возвращается пара (номер, строка-отчет).
-- Иначе выполняется попытка применения однопосылочного правила.
prove s for pred = if checkAxioms s then (num, strForRule s " - axiom" num for)
                   else tryOnePremise s for pred
        where num = pred + 1

-- Функция, осуществляющая последовательные попытки применения функий из списка, пока
-- какая-нибудь не вернет значение, отличное от Nothing. Если такое находится, то
-- для полученных секвенций выполняется рекурсивный вызов prove. Если не находится, то
-- вызывается функия continuation.
build s (f:funcs) for pred continuation =
    if isJust fs then
    case fromJust fs of
       [x]    -> let (new, str) = prove x num num
                 in (new, mconcat[strForRule s (snd f) num for, stringUtf8 "\n", str])
       [x, y] -> let
                   (new1, str1) = prove x num num
```

```
                  (new2, str2) = prove y num new1
                in (new2, mconcat[strForRule s (snd f) num for, stringUtf8 "\n",
                                  str1, stringUtf8 "\n", str2])
    else build s funcs for pred continuation
    where fs = (fst f) s
          num = pred + 1

build s [] for pred continuation = continuation s for pred

-- Функция, пытающаяся применить однопосылочные привила в порядке расположения в
списке,
-- передваваемом на вход функции build. В случае неудачи переходит к попытке
применения
-- двупосылочных правил.
tryOnePremise s for pred =
        build s [(leftNegation, " by (! |-)"), (rightNegation, " by (|- !)"),
                 (leftConjuction, " by (& |-)"), (rightDisjunction, " by (|- |)"),
                 (rightImplication, " by (|- ->)")] for pred tryTwoPremise

-- Функция, пытающаяся применить двупосылочные привила в порядке расположения в
-- списке, передваваемом на вход функции build. В случае неудачи возвращает пару
-- (номер, строка-отчет о невыводимости s)
tryTwoPremise s for pred =
        build s [(rightConjuction, " by (|- &)"), (leftDisjunction, " by (| |-)"),
                 (leftImplication, " by (-> |-)"), (leftEquivalence, " by (<=> |-)"),
                 (rightEquivalence, " by (|- <=>)")]
                 for pred (\_ _ _ -> (num, strForRule s " - unprovable" num for))
   where num = pred + 1

-- Однопосылочные правила

-- Проверка на наличие отрицания в антецеденте
leftNegation  s@(antecedent ::- succeedent) =
   -- выполняем вычисление в монаде Maybe. Если в антецеденте найдется отрицание, то
   -- соответсвующая пропозициональная формула будет помещена в переменную
expr. Иначе
   -- будет возвращен Nothing.
   do
      expr <- find isNot antecedent
      -- Если отрицание было найдено, то возвращаем список из секвенций,
      -- из которых выводится s при прямом порядке вывода (в данном случае
      -- список одноэлементный)
      return $ case expr of
                 (Not x) -> [((delete expr antecedent) ::- (x:succeedent))]

-- Другие однопосылочные правила опущены для экономии места
-- Реализуются аналогично в соответствии с определением
-- секвенциального исчисления высказываний

-- Двупосылочные правила

-- Проверка на наличие конъюнкции в сукцеденте.
-- Аналогично leftNegation, но список двухэлементный.
```

```
rightConjuction s@(antecedent ::- succeedent) =
    do
    expr <- find isConjuction succeedent
    return $ let new_succeedent = delete expr succeedent
             in case expr of
                (a :&: b) -> [antecedent ::- (a:(new_succeedent)),
                              antecedent ::- (b:(new_succeedent))]

-- Другие двупосылочные правила опущены для экономии места
-- Реализуются аналогично в соответствии с определением
-- секвенциального исчисления высказываний.
```

4. СРАВНЕНИЕ СТЕПЕНИ УДОБОЧИТАЕМОСТИ ТЕКСТА ПРОГРАММ

Если отвлечься от деталей, связанных с выводом, то хорошо видно, что подход, используемый в языке рефал-5, позволяет реализовать алгоритм предельно элегантным образом — по сути просто переписываются правила вывода (см. определение понятия секвенциального исчисления высказываний с обратимыми правилами) в соответствующие рефал-выражения.

Ограничение возможностей сопоставления с образцом в языке Haskell не позволяет реализовать логику проверки соответствия секвенции заданному правилу столь элегантным образом, как в языке рефал-5, однако возникающее усложнение весьма незначительно.

Кратко резюмируя, можно сделать вывод о том, что оба языка позволяют реализовать алгоритм достаточно естественным образом, в обоих случаях возникающая многословность связана в основном со сбором результата в единую строку и нумерацией.

Однако удобочитаемость программы на языке рефал — максимально возможная.

5. ПРИМЕРЫ СРАВНЕНИЙ ВРЕМЕНИ РАБОТЫ ПРОГРАММ

Измерения производились на ноутбуке Lenovo IdeaPad Z510, с 4-х ядерным процессором Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz и 8 ГБ RAM, операционная система Ubuntu 14.04.3 LTS.

Компилятор refc и интерпретатор refgo из сборки Refal-5 VERSION-PZ от 28 октября 2004 были использованы при работе с рефал-5, компилятор ghc 7.6.3 был использован при работе с Haskell.

К сожалению, стандартная функция *TimeElapsed* в использованной сборке для работы с Refal-5 оказалась нереализованной, поэтому измерения производились при помощи стандартной в Linux утилиты *time*.

Результаты измерений представлены в следующей таблице:

Секвенция	Длина вывода	Refal-5	Haskell
$\vdash A\&A \leftrightarrow A$	6	3 ms	4 ms
$\vdash (A \leftrightarrow B) \leftrightarrow (\neg A \leftrightarrow \neg B)$	25	4 ms	5 ms
$(C \vee A)\&B \rightarrow C\&D \vdash (A\&B \vee C\&B \leftrightarrow B)\&A$	43	4.5 ms	6 ms
$\vdash ((A \leftrightarrow B) \leftrightarrow C) \leftrightarrow (A \leftrightarrow (B \leftrightarrow C))$	47	5 ms	6 ms
$\vdash ((((A \leftrightarrow B) \leftrightarrow C) \leftrightarrow D) \leftrightarrow E) \leftrightarrow (A \leftrightarrow (B \leftrightarrow (C \leftrightarrow (D \leftrightarrow E))))$	319	18 ms	19 ms

В таблице опущены некоторые скобки, приоритет логических связок определяется следующим образом: сначала идет отрицание (not), затем конъюнкция (&), дизъюнкция ($\vee$), импликация ($\rightarrow$) и эквивалентность ($\leftrightarrow$). В программе на языке рефал-5 необходимо расставлять все скобки, в программе на языке Haskell можно задать приоритет операций, тогда часть скобок может быть опущена.

Из таблицы видно, что временная эффективность программы на языке рефал-5 несколько выше, чем программы на Haskell. При этом с ростом длины вывода не наблюдается роста разницы времен исполнения. По всей видимости эта разница связана с представлением строковых данных, которое в языке рефал-5 реализовано более эффективно.

6. СРАВНЕНИЕ С ПРЕДШЕСТВУЮЩИМИ РАБОТАМИ

Других реализаций алгоритма поиска вывода в исчислении высказываний на языке рефал-5 автору неизвестно.

На языке Haskell было реализовано немалое количество алгоритмов поиска вывода и проверки выводимости, но большинство из них ориентированы на логику первого порядка. Производить сравнение с ними представляется не корректным, так как алгоритм для логики первого порядка заведомо более сложный и сравнение получилось бы не равноценным.

Было найдено два пакета на языке Haskell, имеющих отношение к поиску вывода для исчисления высказываний — Proper [15] и WangsAlgorithm [17].

Proper представляет собой достаточно крупный пакет, однако совершенно недокументированный. В нем есть функционал для проверки общезначимости пропозициональной формулы, но, к сожалению, не реализовано построение вывода. Проверка общезначимости в нем основывается не на секвенциальном исчислении. Для сравнения с данным пакетом представленные выше программы, были модифицированы таким образом, чтобы осуществлялась лишь проверка выводимости секвенции, без построения вывода. Сравнение на наиболее длинном из рассматриваемых выводов (см. таблицу выше) показало, что реализация Proper несколько быстрее модифицированной реализации на языке Haskell, которая, в свою очередь, несколько быстрее модифицированной реализации на языке рефал-5. Косвенно это подтверждает, что выигрыш реализации на языке рефал-5 при немодифицированной версии происходит за счет более оптимальной работы со строковыми данными.

WangthAlgorithm представляет собой пакет, реализующий поиск вывода секвенции в секвенциальном исчислении высказываний с обратимыми правилами. К сожалению, в рассматриваемое исчисление не включены допустимые правила для эквивалентности. Сравнение с приведенными выше реализациями проводилось для севкенции $(C \vee A)\&B \rightarrow C\&D \vdash (A\&B \vee C\&B \rightarrow B)\&A$. Оно показало, что WangthAlgorithm заметно проигрывает по скорости. По-видимому, это связано с более распространенным текстовым отчетом, предлагаемым WangthAlgorithm.

7. ЗАКЛЮЧЕНИЕ

Итак, оба языка достаточно хорошо подходят для решения задачи поиска вывода в секвенциальном исчислении высказываний с обратимыми правилами.

Преимуществом языка рефал-5 является максимально возможная степень удобочитаемости и несколько большая эффективность, но в связи со слабым развитием языка могут возникнуть проблемы при интеграции с другими частями системы. Частично данная проблема решается использованием более новых диалектов, например рефал+.

Язык Haskell не позволяет элементарными средствами сделать код программы настолько удобочитаемым, как язык рефал-5, однако имеет развитое сообщество и большое количество библиотек. Кроме того, имеются API для встраивания кода на языке Haskell в программы на других языках (C, Java, Python, C++ и др.). Это позволяет легко интегрировать модуль в большую систему.

Список литературы

1. *Банчев Б.* Язык Рефал — взгляд со стороны // Практика функционального программирования, 2011. Т. 7, № 1. С. 9–30. URL:http://fprog.ru/2011/issue7/ (дата обращения: 28.11.2015).
2. *Гаврилова Т.А., Хорошевский В.Ф.* Базы знаний интеллектуальных систем. СПб: Питер, 2000.
3. *Герасимов А.С.* Курс математической логики и теории вычислимости. СПб: Изд-во «Лема», 2011. С. 284 http://www.mccme.ru/free-books/gerasimov-3ed-mccme.pdf (дата обращения: 28.11.2015).
4. *Душкин Р.* Модель типизации Хиндли-Милнера и пример её реализации на языке Haskell // Практика функционального программирования, 2010. Т. 5, № 6. С. 158–178.http://www.fprog.ru/2010/issue5/ (дата обращения: 28.11.2015).
5. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ, 2-е издание. М.: «Вильямс», 2005.
6. *Косовский Н.К.* Элементы математической логики и теория субрекурсивных алгоритмов. Л.: Изд-во Ленинградского универститета, 1981.
7. *Косовский Н.К., Косовская Т.М.* Полиномиальный тезис Чёрча для рефал-5 функций, нормальных алгоритмов и их обобщений // Компьютерные инструменты в образовании, 2010. № 5.С. 12–21.
8. *Турчин В.Ф., Сердобольский В.И.* Язык Рефал и его использование для преобразования алгебраических выражений // Кибернетика, 1969. № 3. С. 58–62. http://pat.keldysh.ru/~roman/doc/Turchin/1969-Turchin_Serdobol'skij--Yazyk_Refal_i_ego_ispol'zovanie_dlya_preobrazovaniya_algebraicheskix_vyrazhenij.pdf (дата обращения: 28.11.2015).
9. *Холомьев А.* Учебник по Haskell. https://anton-k.github.io/ru-haskell-book/book/home.html.
10. *Bridge J.* Machine learning and automated theorem proving: Tech. Rep. 792: University of Cambridge, Computer Laboratory, 2010. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-792.pdf (дата обращения: 28.11.2015).
11. *Gentzen G.* Untersuchungen über das logische Schließen. I // Mathematische Zeitschrift, 1935. Bd. 39, H. 1. S. 176–210. http://dx.doi.org/10.1007/BF01201353 (дата обращения: 28.11.2015).
12. *Gentzen G.* Untersuchungen über das logische Schließen. II // Mathematische Zeitschrift, 1935. Bd. 39, H. 1. S. 405–431. http://dx.doi.org/10.1007/BF01201363 (дата обращения: 28.11.2015).
13. Haskell language home page. https://www.haskell.org/ (дата обращения: 28.11.2015).
14. *Hudak P., Hughes J., Peyton Jones S., Wadler P.* A History of Haskell: Being Lazy with Class // Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. HOPL III. New York, NY, USA: ACM, 2007. P. 12–1–12–55. http://doi.acm.org/10.1145/1238844.1238856 (дата обращения: 28.11.2015).
15. *Huff D.* Proper. GitHub repository. https://github.com/dillonhuff/Proper (дата обращения: 03.12.2015).
16. *Kleene S. C.* Mathematical logic. John Willey & Sons, Inc., 1967. Перевод на русский язык: Клини С.К. Математическая логика. М.: «Мир», 1973.
17. *Korkut J.* WangsAlgorithm. GitHub repository. https://github.com/joom/WangsAlgorithm (дата обращения: 03.12.2015).

18. *Lisitsa A.* Introduction to Refal. Meeting of the BCS Advanced Programming Specialist Group, 2014. http://www.bcs.org/upload/pdf/intro-to-refal-130314.pdf (дата обращения: 28.11.2015).
19. *Turchin V.* REFAL-5 Programming Guide & Reference Manual. http://refal.botik.ru/book/html/index.html (дата обращения: 28.11.2015).

ПРИЛОЖЕНИЯ

A. КРАТКОЕ ОПИСАНИЕ ЯЗЫКА РЕФАЛ-5

A.1. Общая информация

Рефал (Refal) — язык программирования функционального стиля, ориентированный на символьные вычисления: обработку строк, машинный перевод, решение задач искусственного интеллекта [8]. Язык рефал был создан В. Ф. Турчиным в 1960-х годах, позднее самим Турчиным и его учениками были созданы различные более удобные для программирования диалекты: рефал-5, рефал-6, рефал+. В данной статье используется язык рефал-5.

Язык рефал является одним из первых функциональных языков программирования, во многом он уникален. Отличительной чертой языка рефал является то, что в основе его лежит расширенная версия понятия алгоритма Маркова. Достаточно подробное рассмотрение математических оснований языка рефал-5 можно прочитать в [7].

Основными концептуальными подходами при программировании на языке рефал являются сопоставление с образцом, переписывание не вложенных друг в друга термов и рекурсия.

Подробное рассмотрение языка рефал с анализом его преимуществ и недостатков по сравнению с другими языками, а также с анализом перспектив развития можно найти в [1].

A.2. Основы синтаксиса языка рефал-5

В соответствии с официальной документацией [19] в языке Рефал-5 есть всего 4 типа констант:

- строка символов (characters) — строка, заключенная в апострофы; например ’refal’,
- составной символ (compound symbol) — строка, заключенная в двойные кавычки. Она рассматривается как единое целое и не анализируется транслятором,
- идентификатор (identifier) — строка символов со стандартными ограничениями для идентификаторов, которая либо не содержит кавычек, либо находится в двойных кавычках (то есть составной символ),
- макроцифра (macrodigit) — цифра в системе счисления по основанию 2^{32}, записанная в десятичном представлении;

и три типа переменных:

- s — переменная для одиночных символов;
- e — переменная для последовательности произвольной длины, возможно пустой, из символов или[1] термов;
- t — переменная для термов, то есть выражений, представляющих собой либо единственный символ, либо выражение, заключенное в круглые скобки, называемые структурными (или синтаксическими).

[1] Здесь и далее ”или” имеет значение, общепринятое в математике, то есть (A или B) $\leftrightarrow A \vee B \vee (A\&B)$.

Отметим, что выражение и терм индуктивно определяются друг через друга.

Целое число определяется как последовательность макроцифр, отделенных друг от друга пробелами, перед которой может быть расположен один из знаков '+' или '–'. Имеются встроенные целочисленные арифметические функции.

Единственный тип данных, которым оперирует программа на языке рефал-5 — рефал-выражение (Refal-expression), собственно последовательность термов. Рефал-выражение реализуется как дважды связанный список [5].

Имя функции представляет собой идентификатор.

Определение функции в языке рефал-5 состоит из имени функции и следующего за ним заключенного в фигурные скобки блока правил, разделенных точками с запятой. Каждое правило представляет собой выражение вида *pattern [, conditions] = expression*. Левая часть правила — шаблон, последовательность символов или переменных, структурированная скобками, с которой происходит сопоставление передаваемого функции аргумента. Также могут присутствовать дополнительные условия. Правая часть (после знака =) — последовательность из символов, термов или вызовов функций, возможно вложенных друг в друга.

Вызов функции представляет собой имя функции и её аргумент, заключенные вместе в угловые скобки, например <*Prout 'Hello, world!'*>.

Аргументом функции в языке рефал-5 всегда является рефал-выражение, возможно пустое, которое посредством сопоставления с образцом может быть разобрано на различные составляющие.

Процесс вычисления функции происходит следующим образом. Просматриваются сверху вниз правила данной функции. Если переданный параметр соответствует образцу и удовлетворяет дополнительному условию в случае его наличия, то в результате работы функции возвращается значение правой части правила (где вместо переменных подставлены соответствующие значения, полученные при сопоставлении, а все вызовы функций заменены на результаты их вычисления). В противном случае происходит переход к следующему правилу, если оно имеется. Если это было последнее правило, то происходит обрыв вычисления.

Сопоставление с образцом, вообще говоря, может быть осуществлено множеством различных образов. Но для разрешения неоднозначности в языке рефал-5 используется следующее правило: в качестве результата сопоставления выбирается вариант синтаксического разбора, при котором в рефал-выражении первая в порядке слева направо переменная *e*-типа имеет минимальное число термов, то есть в начале сопоставления полагается пустой и при каждой неудачной попытке сопоставления включает в себя еще один терм [18]. Сопоставление, разбивающее терм, невозможно. Например, при сопоставлении рефал-выражения ('ef')'erf' с шаблоном e.1 'e' e.2 будут получены следующие значения: e.1 = ('ef'), e.2 = 'rf', а не e.1 = (, e.2 = 'f')'erf'.

Программный модуль представляет собой список функций, по крайней мере одна из которых должна быть помечена как точка входа модуля — *$ENTRY*. Только такие функции могут быть вызваны из других модулей. Также могут присутствовать директивы импорта — *$EXTERNAL func-name*, функция *func-name* должна быть определена в другом модуле, совместно с которым выполняется компиляция. При совместной компиляции нескольких модулей в одном из них должна присутствовать функция с именем *Go*, которая является точкой входа для программы (аналогично *main* в C). Строки, начинающиеся с символа '*', являются комментариями.

A.3. Дополнительные условия (with-clause и where-clause)

Рассмотрим произвольное правило, содержащее одно условие. Оно имеет следующий вид: $pattern, conditional\_expression : conditional\_pattern = expression$ Конструкция, стоящая после запятой, была названа Турчиным with-clause, ниже для обозначения этой конструкции используется термин «дополнительное условие». *Conditional_expression* здесь — рефал-выражение, подобное тому, которое находится в правой части любого правила. То есть в него могут входить константы, вызовы функций или переменные, входящие в *pattern*. *Conditional_pattern* — шаблон, с которым сопоставляется результат вычисления *conditional_expression*. Он может содержать константы, переменные из *pattern* (по сути они выступают здесь в роли констант, подставляемых во время исполнения программы), а также новые переменные.

Большое удобство предоставляет использование where-clause. Правило с where-clause имеет следующую структуру:

```
pattern, conditional_expression:
{
        conditional_pattern1 = expression1;
        conditional_pattern2 = expression2;
        ...
}
```

Результат выполнения этого правила — значение первого *expression*, для которого успешно пройдет сопоставление *conditional_expression* с соответствующим *conditional_pattern*. Where-clause могут быть вложены друг в друга.

Вообще говоря, дополнительные условия и блоки не обязательны, так как их достаточно несложным образом можно заменить на вспомогательные функции. Однако их использование делает код существенно более читабельным, позволяя не засорять пространство имен и «не плодить лишние сущности». Кроме того, они позволяют концептуально новым образом организовать вычисление функции, которое, по сути, превращается в поиск по дереву, что делает язык рефал-5 в большой степени естественным для решения задач искусственного интеллекта.

A.4. Работа со стеком

В языке рефал-5 есть встроенные функции для работы с глобальным стеком. Эти функции имеют побочные эффекты и нарушают принцип чистоты (одна и та же функция на одних и тех же входных данных может выдавать различные результаты, если она использует функции работы со стеком) и не рекомендуются к использованию без необходимости [19], однако в некоторых ситуациях могут ощутимо упростить логику программы.

Для помещения выражения в стек используется функция *Br* (от слова bury — закапывать), для изъятия из стека — функция *Dg* (от dig — выкапывать). Для взятия верхнего элемента со стека без изменения содержимого стека используется функция *Cp* (от copy — копировать).

Например, чтобы поместить в стек под именем *N* выражение ’Refal’, будет использоваться <Br N ’=’ ’Refal’>, а чтобы достать его оттуда, — <Dg N> или <Cp N>.

B. КРАТКОЕ ОПИСАНИЕ ОСНОВ ЯЗЫКА HASKELL

Haskell — чистый функциональный язык программирования общего назначения. Он является строго статически типизированным языком с автоматическим выводом типов на основе алгоритма Хинди-Милнера [13].

Существует большое количество литературы о языке Haskell, список источников, рекомендованных для использования при изучении, представлен на официальном сайте [13]. Среди русскоязычных источников следует особо выделить замечательный учебник А. Холомьева [9].

Haskell является довольно сложным языком, поэтому в данной статье рассмотрены только конструкции и понятия, минимально необходимые для понимания кода приведенной программы.

Любое выражение в Haskell имеет тип. То, что *expr* имеет тип *T*, записывается как *expr :: T*. Для записи функциональных типов используется стрелка →, она разделяет типы аргументов между собой и типы аргументов от типа возвращаемого значения. Возвращаемое значение указывается последним. Например, функция *add*, складывающая целые числа, будет иметь тип Int → Int → Int. Такая запись связана с понятием карринга – все функции в Haskell фактически принимают один аргумент и могут возвращать другие функции. Но для понимания приводимой программы это не важно, поэтому подробно карринг не описывается.

Указывать тип выражения в Haskell в большинстве случаев необязательно — он может быть выведен на основе алгоритма Хинди-Милнера [4].

Нефункциональные выражения в Haskell (то есть не являющиеся функциями) бывают элементарными, например Int, Double, Char, и составными. Составные типы в Haskell реализуются через понятие алгебраического типа данных [9, 14] — объединения именованных произведений типов. Например, стандартный тип *Maybe*, представляющий выражения, которые могут иметь значение, а могут не иметь его, определяется как *data Maybe a = Just a | Nothing*. *Just* и *Nothing* называются конструкторами, *a* — параметр типа. Конкретные выражения могут иметь либо значение *Just <конкретный тип>*, либо *Nothing*. Так *Just Int* может иметь значение *Just 0* или *Nothing*.

Разбор выражений или декомпозиция осуществляется посредством механизма сопоставления с образцом [9, 14].

Определение функции представляет собой необязательное объявление типа вида *func-name :: type* и набор предложений (*clause*) вида *func-name patterns = expression*, каждое должно начинаться с новой строки и иметь такой же отступ, как и предыдущее. Приведем пример объявления функции на Haskell — функции вычисления факториала:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

В данном случае важно, что Haskell ориентирован на работу с алгебраическими типами данных и активно использует сопоставление с образцом.

Конструкция *let a=<expr1>in <expr2>* позволяет вводить локальные переменные. То есть если *a* входит в *expr2*, то вместо *a* будет подставлен результат вычисления *expr1*. *Where* выполняет ту же функцию, но, в отличие от *let*, является глобальным в рамках одного предложения. Данные конструкции используются для избежания повторного вычисления одинаковых выражений.

EFFICIENCY COMPARISON OF IMPLEMENTATIONS OF INFERENCE ALGORITHM FOR SEQUENT PROPOSITIONAL CALCULUS IN REFAL-5 AND HASKELL

Grigorev V. D.

Abstract

In this paper author describes implementations of inference algorithm for sequent propositional calculus in Refal-5 and Haskell programming languages. Time efficiency and readability of these implementations are compared. Also, there is a concise description of considered programming languages presented in appendix section. The aim of this paper is to provide the basis of the programming language selection for the implementation of inference engine for artificial intelligence systems, based on logical inference and automated theorem proving tools in particular.

Keywords: *automated theorem proving, propositional logic, sequent calculus, Refal-5, Haskell, readability of program, time efficiency of program.*

Григорьев Валентин Дмитриевич,
студент 3-его курса кафедры информатики математико-механического факультета СПбГУ по направлению «Фундаментальные информатика и информационные технологии»,
v.d.grigorev@mail.ru