ПРОГРАММИРОВАНИЕ, 2020, № 1, с. 63–74

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 004.432.4

ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ ПРОГРАММ ДЛЯ ОПИСАНИЯ ТИПОВ В ТЕРМИНАХ СОПОСТАВЛЕНИЯ ЗНАЧЕНИЙ С ОБРАЗЦОМ

© 2020 г. В. А. Васенин[a],*, М. А. Кривчиков[a],**

[a] *Московский государственный университет им. М.В. Ломоносова*
119991 Москва, ГСП-1, Ленинские горы, д. 1, Россия
**E-mail: vasenin@msu.ru*
***E-mail: maxim.krivchikov@gmail.com*

Поступила в редакцию 13.05.2019 г.
После доработки 08.06.2019 г.
Принята к публикации 18.06.2019 г.

В статье предлагается промежуточное представление для компактного и обобщенного описания особенностей спецификации системы типов в языках программирования с динамической типизацией, использующее вычисления на уровне типов и построенное на основе сопоставления с образцом. Побудительным мотивом для разработанного авторами промежуточного представления стало промежуточное представление языка Рефал-2 “Язык сборки”. Настоящее промежуточное представление относится не к байт-кодам, а к промежуточным представлениям на основе графа потока исполнения. Такое представление сохраняет информацию о типах значений, что отличает его от языка сборки. Представлена разновидность языка Рефал с поддержкой функций высшего порядка, замыканий и типа данных “ассоциативный массив”, а также транслятор в промежуточное представление. В терминах этого языка приведены примеры описания программ с простыми типами, а также полиморфизма по полям записей. Такие примеры представляют интерес для описания системы типов языков программирования с динамической типизацией.

DOI: 10.31857/S0132347420010070

1. ВВЕДЕНИЕ

В настоящее время предметно-ориентированные языки, как правило, реализуются с использованием динамической проверки типов. В качестве примеров приведем несколько достаточно распространенных предметно-ориентированных языков. Языки командных оболочек операционных систем (Bash) и системы сборки (Make) для описания аргументов и результатов вызываемых ими программ используют подстановку переменных в текстовой форме. Аналогичным образом реализуются языки описания шаблонов, например, Jinja2. Ряд предметно-ориентированных языков для описания конфигурации программных систем (Salt, Ansible) основаны на языках разметки (YAML, JSON) и не содержат средств статической проверки типов. Для некоторых предметно-ориентированных языков, например, языка описания пакетов программ Nix, вопрос о замене динамической проверки типов на статическую ставится уже после их разработки и внедрения в эксплуатацию [1].

Статическую типизацию поддерживают предметно-ориентированные языки следующих трех видов: ограниченные версии существующих языков программирования общего назначения (язык шейдеров GLSL как диалект языка C); встроенные в языки программирования общего назначения с достаточно мощными системами типов и выводом типов (различные eDSL в языке Haskell, например, на основе свободных монад); языки, реализованные с использованием систем макросов языков программирования общего назначения (например, макросы языка Rust).

Основным подходом к реализации проверки типов в языках программирования с динамической типизацией является постепенная типизация [2]. Для более строгой верификации свойств в таких языках системы постепенной типизации зачастую обладают особенностями, обусловленными сложившейся прагматикой (практикой применения) языка. Например, средство mypy, реализующее постепенную типизацию для языка Python, имеет встроенную поддержку так называемых протоколов для структурной типизации [3]. Такие протоколы представляют собой спецификацию, которой должно удовлетворять значение, для его использования в том или ином синтаксси-

ческом контексте (например, в качестве аргумента итерации в конструкции for ... in ... или в качестве аргумента для индексации с использованием квадратных скобок). Язык TypeScript, который интегрирует постепенную типизацию в язык JavaScript, поддерживает ключевое слово keyof в описании типов. Это ключевое слово преобразует объект (ассоциативный массив) в перечисление всех известных ключей, значения которых заданы в этом объекте.

В рамках теории типов наиболее выразительный подход к описанию типов поддерживает исчисление конструкций (Calculus of Constructions), которое позволяет компактно и однородно описать типы с использованием вычислений на уровне типов. В исчислении конструкций типы задаются, фактически, как термы типизированного лямбда-исчисления, т.е. гарантированно завершимые программы. В настоящей статье предлагается промежуточное представление для компактного и обобщенного описания особенностей спецификации системы типов в языках с динамической типизацией, также использующее вычисления на уровне типов. А именно, авторы настоящей статьи предлагают использовать для описания системы типов языков промежуточное представление программ на этих языках на основе сопоставления с образцом. Представленные далее результаты получены в рамках научно-исследовательской работы “Методы и средства разработки верифицируемого программного обеспечения с использованием предметно-ориентированных языков, имеющих заданную формальную семантику”. Вопрос применения этого представления для описания предметно-ориентированных языков более подробно будет рассмотрен в следующих публикациях.

2. ЯЗЫК ДЛЯ РАЗРАБОТКИ СРЕДСТВ ОПИСАНИЯ ПРЕДМЕТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

На настоящее время наиболее мощные (эффективные) механизмы для сопоставления с образцом в динамически типизированных языках программирования реализованы в языках семейства Рефал [4] (в частности, Рефал-5 [5]). В этой связи в качестве основы для промежуточного представления авторами была разработана разновидность языка Рефал с расширениями. Представленная далее разновидность языка Рефал поддерживает функции высшего порядка и тип данных – неизменяемый ассоциативный массив. Последняя модификация является, насколько известно авторам, новой для языков этого семейства. Синтаксис языка модифицирован с целью более близкого его соответствия языкам программирования, распространенным в настоящее время, что также является новым результатом. Настоящий раздел посвящен описанию особенностей разработанной авторами разновидности языка. Детали процесса разработки и реализации такой разновидности представлены в следующих разделах.

Основным видом постоянных значений в языке являются списки, в которые могут входить элементы перечисленных далее видов, разделенные пробелами.

1. Целочисленные неотрицательные константы (например: 0 12 12345). В настоящей реализации поддерживаются 32-битные целые числа со знаком.

2. Строковые (многострочные) константы в одинарных или двойных кавычках (аналогично языку Python, виды кавычек не различаются) с возможностью экранирования специальных символов (например: “ABC” “A BC\” “ 'A\nB\tC'). Строковые константы интерпретируются как последовательность индивидуальных букв (code-points в терминологии стандарта Unicode; термин “буква” используется в смысле английского слова character для различения с термином “символ”, который используется аналогично английскому symbol).

3. Символы – идентификаторы, которые могут быть ассоциированы с функцией (например: ABC, Fn1). С позиций реализации символы можно охарактеризовать как интернированные строки.

4. Вложенные списки отделяются парами круглых скобок. Например, список “A (B C) D” состоит из трех элементов, первым из которых является символ A, вторым – список из двух символов B и C, и третьим – символ D.

5. Ассоциативные массивы записываются внутри фигурных скобок. Ключами и значениями ассоциативного массива могут быть единичные значения. Ключ отделяется от значения двоеточием, пары “ключ-значение” разделяются запятыми. Например, ассоциативный массив “{abc: DEF, (“12”): (“34”), 12: 34, (1 2): (3 4)}” содержит следующие 4 записи вида “ключ-значение”: символ abc – символ DEF, строка (список букв) “12” – строка “34”, число 12 – число 34, список чисел (1 2) – список чисел (3 4), соответственно. Другие реализации языков семейства Рефал не поддерживают ассоциативные массивы и их сопоставление с образцом.

6. Вложенные функции, объявление которых состоит из ключевого слова fn, опционального имени и тела функции в фигурных скобках.

Отдельным классом выражений в языках семейства Рефал являются образцы. Образцы – это списки, в которых в качестве элементов, в дополнение к отдельным значениям, могут встречаться свободные переменные. Свободные переменные записываются в виде “тип.имя”, как это принято

```
fn Palindrome {
   empty => True;
   s.1 => True;
   s.1 e.2 s.1 => <Palindrome e.2>;
   _ => False;
}
```

Рис. 1. Функция “Паллиндром” на языке из настоящей статьи.

```
Palindrome {
   = True;
   s.1 = True;
   s.1 e.2 s.1 = <Palindrome e.2>;
   e.1 = False;
}
```

Рис. 2. Функция “Палиндром” на языке Рефал-5.

в других языках семейства. В настоящей реализации поддерживаются следующие основные типы свободных переменных:

• “s” – единичное атомарное значение (число, символ, буква);

• атомарное значение заданного типа (“i” – число, “S” – символ, “c” – буква);

• “t” – единичное произвольное значение (атомарное значение или список или ассоциативный массив);

• “d” – ассоциативный массив;

• “e” – произвольная последовательность значений (подсписок).

Образцы для вложенных списков записываются в круглых скобках. Образцы для ассоциативных массивов записываются в фигурных скобках. Для сопоставления ассоциативных массивов с образцом вводятся следующие далее дополнительные синтаксические конструкции на основе оператора “многоточие” (“...”).

• Сохранить остаток ассоциативного массива в новую переменную. Например, следующая конструкция задает образец ассоциативного массива, в котором под ключом “a” должно находиться значение “b”, а остальные пары “ключ-значение”, входящие в ассоциативный массив будут сохранены в новый ассоциативный массив – переменную d.rest: {a: b, ...d.rest}.

• Игнорировать остаток ассоциативного массива. Например, ассоциативный массив {a: b, c: d} не пройдет сопоставление с образцом {a: b}, т.к. содержит дополнительную пару “ключ-значение”, но пройдет сопоставление с образцом {a: b, ...}.

Для удобства чтения программ на языке, пустой список обозначается ключевым словом empty как в значениях, так и в образцах. Кроме того, прочерк (“_”) обозначает безымянную произвольную последовательность значений.

Основным элементом программ на языках Рефал-семейства являются функции. Объявление функции в настоящей реализации начинается с ключевого слова fn, затем следует имя функции, затем в фигурных скобках – последовательность предложений – ветвей сопоставления с образцом, разделенных точкой с запятой (“;”). Предложение начинается с образца, с которым будет сопоставлен аргумент функции, и продолжается с использованием комбинации выражений и связующих операторов, перечисленных далее. Последнее выражение в комбинации определяет результат выполнения предложения. В дополнение к константам и переменным, выражения могут содержать вызовы других функций. Вызов функций, как и в других языках семейства, обозначается угловыми (активационными) скобками, а первый элемент в скобках обозначает вызываемую функцию (например, вызов функции сложения Add для двух чисел 1 и 2 записывается следующим образом: <Add 1 2>).

Классический пример функции “Палиндром” можно представить следующим образом. На рис. 1 показан код на предлагаемом авторами расширении языка Рефал, а на рис. 2 – пример из документации языка Рефал-5 [5].

В этом примере используется связующий оператор “отобразить” (“⇒”), который отделяет выражение слева от результата – выражения справа. В языке поддерживаются представленные далее связующие операторы.

• Операторы безусловного перехода на следующий шаг “запятая” (“,”) или “отобразить” (“⇒”). В других языках семейства, как правило, в качестве такого связующего элемента используется знак равенства. Если при вычислении выражений, следующих за этим оператором, произойдет неуспешное сопоставление с образцом или неуспешный вызов функции, выполнение функции завершится с неуспехом.

• Оператор условного перехода на следующий шаг “амперсанд” (“&”). Если при вычислении выражения, непосредственно следующего за этим оператором, произойдет неуспешное сопоставление с образцом или неуспешный вызов функции, будет рассмотрена следующая альтернатива сопоставления с образцом в рамках активной функции.

• Оператор простого дополнительного сопоставления с образцом “let” (let образец = выражение). Вычисляет значение выражения и сопоставляет его с образцом.

• Оператор дополнительного сопоставления с образцом-функцией “match” (match выражение with функция). Вычисляет значение выражения и

```
fn PreAlph {
  s.1 s.1;
  s.1 s.2 & let e.A s.1 e.B s.2 e.C = <Alphabet>;
}

fn Order {
  (e.1)e.2 & <Pre(e.1)(e.2)> => (e.1)(e.2);
  (e.1)e.2 => (e.2)(e.1)
}
```

Рис. 3. Предикаты на основе успешного/неуспешного завершения вычислений на языке из настоящей статьи.

вызывает функцию с аргументом – значением выражения.

• Новый оператор отрицания “except” (except {предложение}). Если вычисление предложения в исходном контексте значений переменных завершается успешно, результатом оператора отрицания является неуспех. Если же вычисление предложения завершается неуспешно, оператор отрицания возвращает пустой список.

В целом, операторы дополнительного сопоставления с образцом аналогичны условиям и блокам из языка Рефал-5, а разделение перехода на следующий шаг на условный и безусловные аналогично Рефал-6 или Рефал-плюс. Синтаксические отличия настоящей реализации от традиционных обусловлены вопросами эргономики использования языка, в частности, знакомства с языком разработчиков, которые ранее не изучали языки семейства Рефал.

Ограничение распространения сигнала о неуспешном завершении вычислений в рамках одной функции позволяет определять предикаты на значениях в виде функций, вычисление которых завершается успешно для значений, удовлетворяющих предикату, и неуспешно для значений, не удовлетворяющих предикату. В следующем далее примере эту особенность иллюстрирует функция PreAlph, которая проверяет, что первый аргумент (буква) предшествует второму в некотором заданном алфавите. Реализация на языке Рефал-5 возвращает значения-символы T и F, которые обозначают истинное и ложное значение, соответственно. Символы T и F не являются специальными (в отличие, например, от значений True и False в языке Python), их использование обусловлено исключительно соглашением об обозначении логических значений. Реализация в настоящей разновидности языка в качестве истинного значения использует успешное завершение вычислений (возможно, с пустым результатом), а в качестве ложного – неуспешное завершение вычислений, которое в дальнейшем может быть перехвачено с использованием связующего оператора “&”.

```
PreAlph {
  s.1 s.1 = T;
  s.1 s.2, <Alphabet>: e.A s.1 e.B s.2 e.C
          = T;
  e.1 = F;
}

Order {
  (e.1)e.2, <Pre (e.1)(e.2)>:
       { T = (e.1)(e.2):
         F = (e.2)(e.1);
       };
}
```

Рис. 4. Предикаты на основе возвращаемых значений на языке Рефал-5.

Таким образом два примера из главы 4 руководства Рефал-5 [5] в настоящей реализации могут быть записаны следующим образом, с учетом отмеченных выше синтаксических и семантических особенностей (на рис. 3 представлен код на языке, разработанном авторами, а на рис. 4 – код на языке Рефал-5):

Оператор отрицания необходим для адекватного описания нетривиальных условий в системах типов, что необходимо для применения языка по его основному назначению. Пример на рис. 5 показывает применение оператора отрицания для определения предиката “последовательность” из пяти атомарных значений, третье из которых не равно 1”.

Следует также отметить, что в функции, объявленной в операторе match, можно использовать переменные из контекста того предложения внешней функции, в котором был объявлен оператор match. Кроме того, такая функция может быть объявлена с именем, которое может быть использовано внутри нее для рекурсивного вызова или возврата замыкания. Таким образом, настоящая реализация поддерживает функции высшего порядка и замыкания. Пример на рис. 6 демонстрирует применение отмеченных функциональных возможностей языка. Функция Map является функцией высшего порядка, которая применяет пер-

```
ThirdOfFiveNot1 {
  s.1 s.2 s.3 s.4 s.5,except {let 1 = s.3}
}
```

Рис. 5. Оператор отрицания: предикат для последовательности из пяти атомарных значений, третье из которых не равно 1.

вый аргумент-функцию к каждому из следующих аргументов. Функция AddOne определяет переменную s.add, в которой хранится некоторое значение, и безымянную функцию, которая сохраняется в переменной t.addOne и которая добавляет к своему аргументу значение s.add из замыкания. Эта безымянная функция передается как первый аргумент в функцию Map. Результатом выполнения является список, элементы которого увеличены на единицу от исходного (список 1 2 3 4 5 становится списком 2 3 4 5 6).

3. ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ ДЛЯ ОПИСАНИЯ ПРЕДМЕТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Как правило, для анализа программ используется не их исходный код, а некоторое промежуточное представление с бóльшим, чем у исходного кода, уровнем детализации (уникальность идентификаторов, раскрытие синтаксического сахара), а также с бóльшим уровнем абстракции от оборудования, по сравнению с машинным кодом. В статье [6] приведена следующая далее классификация промежуточных представлений для описания предметно-ориентированных языков программирования.

1. Низкоуровневые промежуточные представления, используемые в компиляторах в машинный код. Например, представления LLVM и Typed Assembly Language.

2. Виртуальные машины уровня приложений на основе байт-кода, ориентированные на императивные языки программирования. Например, машина JVM для языка Java, спецификация CLI для языка C#.

3. Виртуальные машины уровня приложений на основе байт-кода, ориентированные на функциональные языки программирования. Например, машина ZINC для OCaml, машина WAM для Prolog, машина BEAM для Erlang, байт-код языка Python.

4. Внутренние промежуточные представления на основе графа потока исполнения, сохраняющие информацию о типах значений. Например, представление continuation-passing style (CPS).

5. Высокоуровневые, отражающие специфику языка программирования промежуточные представления на основе канонических форм синтаксического дерева. Например, представление STG для Haskell и представление Cminor для компилятора CompCert языка C.

Авторами разработано новое промежуточное представление уровня 4 по представленной классификации с достаточно выразительной для решения задачи системой типов. Побудительным мотивом для настоящего промежуточного представления стало промежуточное представление языка Рефал-2 “Язык сборки” [7].

Основные отличия промежуточного представления в настоящей реализации от языка сборки можно сформулировать в виде следующих тезисов. Язык сборки относится к третьему классу промежуточных представлений, то есть, к виртуальным машинам уровня приложений на основе байт-кода, ориентированным на функциональные языки программирования. Программы на языке сборки – это последовательности команд, не имеющие глубокой внутренней структуры. Предлагаемое промежуточное представление относится к четвертому классу, то есть, к внутренним промежуточным представлениям на основе графа потока исполнения, сохраняющим инфор-

```
fn Map {
   t.fn => emply;
   t.fn s.1 e.rest => <t.fn s.1> <Map t.fn e.rest>;
}

fn AddOne {
   _ =>
     let s.add = 1,
     let t.addOne = fn{s.val => <Add s.add s.val>},
     <Map t.addOne 1 2 3 4 5>
}
```

Рис. 6. Функция высшего порядка и замыкания в языке из настоящей статьи.

мацию о типах значений. Язык сборки описан для языка “Рефал-2”, в предлагаемом промежуточном представлении поддерживаются функции высшего порядка. Как было отмечено выше, сам язык и, как следствие, его промежуточное представление были расширены новыми типами значений и конструкциями для описания системы типов. В качестве достаточно близкого представления можно отметить язык рефал-графов, который используется в суперкомпиляторе SCP-4 [8].

Модель исполнения промежуточного представления имеет следующий вид. При исполнении функции в каждый момент времени задан список значений – аргумент, список значений – результат, последовательность распознанных образцов – значений переменных и набор значений переменных из замыкания, которые индексируются парой целых чисел (порядковый номер внешнего контекста замыкания, номер переменной во внешнем контексте). Контекст исполнения функции содержит также стек обработчиков неуспешных результатов распознавания или вызова функции. Результат может содержать команды вызова (активационные скобки). Перед его сохранением в переменную или перед анализом образцов выполняется стадия активации. На стадии активации все команды вызова в списке заменяются на результаты вызова.

Промежуточное представление хранится в виде дерева в форме s-выражений в текстовом виде (следует отметить, что при необходимости изменить формат хранения, например, на компактную нетекстовую кодировку или XML не составит труда). Дерево содержит узлы-выражения перечисленных далее основных видов.

1. Объявление функции (Function (Name имя функции) выражения ...) и ссылка на функцию (FunctionRef имя функции).

2. Список или структурные скобки: (Structural выражения ...).

3. Вызов или активационные скобки: (Eval выражения ...).

4. Отрицание (Except выражения ...).

5. Ветвление (Branch (выражения ветви 1) (выражения ветви 2) ...).

6. Значение-ассоциативный массив (Dictionary ((выражение-ключ) (выражение-значение)) ...).

7. Значение-константа (Value тип-значения значение).

8. Команды без аргументов: распознавание пустого аргумента (Check), успешное завершение выполнения функции (Return), неуспешное завершение ветки вычислений (Fail), очистка стека обработчиков (PopHandlers), поиск произвольной последовательности значений (Expand), распознавание остатка (Close), активация и сохранение в последовательности переменных текущего результата (Push).

9. Команды распознавания фрагмента аргумента (Recognize направление распознаватель). В качестве направлений поддерживаются: начало и конец аргумента (Left и Right, соответственно); включение в ассоциативный массив в начале аргумента (Dictionary); ключ и значение в ассоциативном массиве (Key и Value номер переменной-ключа, соответственно). Специальные направления (Dictionary и Key) поддерживаются не для всех распознавателей. В качестве распознавателей поддерживаются вложенные списки (Structural), произвольные значения (Term), атомарные значения (Literal), значения заданного примитивного типа (ValueLiteral), константы (Exact), значения ранее распознанных переменных (Old Expression уровень номер).

10. Команда поиска произвольной последовательности значений (Extend).

11. Команда загрузки в аргумент значения переменной для дополнительных условий (Range номер-переменной).

12. Команда вывода в результат постоянных значений (Value тип значение) и значений ранее распознанных переменных (Emit уровень номер).

13. Команды проверки типов (Typecheck IsType имя-функции) и (Typecheck Correct тип-предусловие имя-функции тип-постусловие).

Поле “уровень” в распознавателе OldExpression и комане Emit равно нулю в случае, когда номер соответствует переменной в контексте текущей функции. В противном случае для замыканий оно отражает уровень вложенности лексического контекста, из которого загружается переменная с заданным номером.

Такое промежуточное представление достаточно для трансляции в него программ, написанных на языке, описанном в предыдущем разделе. В качестве одного из направлений дальнейшей работы предполагается добиться большей однородности команд представления. Как уже было отмечено ранее, направления распознавания Dictionary, Key и Value с позиций семантики существенно отличаются от направлений распознавания по списку Left и Right. В частности, с их использованием невозможно эффективное представление перебора пар “ключ-значение”, входящих в ассоциативный массив, для распознавания ассоциативных массивов со сложной структурой. В настоящее время в случаях, когда необходимо решить эту задачу, используются относительно менее эффективные функции стандартной библиотеки, которые выполняют преобразование между ассоциативным массивом и списком пар “ключ-значение”, из которых составлен массив.

Поскольку промежуточное представление предназначено также для описания семантики

программ, написанных на предметно-ориентированных языках программирования, команды промежуточного представления могут содержать дополнительную информацию отладочного характера. Такая информация позволяет установить связь между элементами исходного кода и командами промежуточного представления. Формат дополнительной информации в настоящее время строго не определен, его уточнение является одним из направлений дальнейшей работы.

4. ИНТЕРПРЕТАТОР ПРОМЕЖУТОЧНОГО ПРЕДСТАВЛЕНИЯ

Для исполнения программ, записанных в промежуточном представлении, авторами разработан интерпретатор промежуточного представления. Основными требованиями к интерпретатору являются расширяемость и производительность, достаточная для адекватной поддержки цикла разработки в исследовательском режиме. В этой связи было принято решение о реализации исполнителя языка именно в форме интерпретатора, а не в форме компилятора в машинный код. Интерпретатор реализован на языке Rust, его объем составляет порядка 5 тысяч строк кода.

Интерпретатор использует расширенную (исполнимую) версию промежуточного представления для более эффективной интерпретации. Преобразование хранимой версии промежуточного представления в исполнимую выполняется на стадии предобработки. На этой стадии интерпретатор считывает дерево промежуточного представления из исходного файла и сохраняет его в виде арены – набора линейных последовательностей команд, индексируемых 32-битным беззнаковым целым числом. Таким образом, в памяти такие команды, как Structural или Function хранят индекс последовательности команд в арене. Основная часть процедуры предобработки заключается в выполнении двух следующих процедур. Во-первых, устанавливается соответствие имен функций в командах FunctionRef с индексами их определений в соответствии с лексическим контекстом. Во-вторых, для замыканий выполняется выделение номеров переменных, которые используются замыканием, и команда Function заменяется на специфичную для интерпретатора команду захвата переменных в замыкание CaptureClosure. В перспективе предполагается перенести эту команду в хранимую часть промежуточного представления. Однако на данном этапе отмеченное выше преобразование с технической точки зрения проще выполнять в рамках предобработки в интерпретаторе, а не в трансляторе в промежуточное представление. Кроме того, при предобработке выполняется подгрузка внешних модулей и связывание импортированных функций. Этот аспект реализации носит в большей степени технический характер, его описание выходит за рамки настоящей статьи.

Следует отметить следующие две особенности описываемого в настоящей статье языка, которые нашли свое отражение в промежуточном представлении, и которые оказывают существенное влияние на структуру интерпретатора. Первая особенность связана с процессом исполнения отдельных функций. Семантику команды ветвления Branch в значительной части языков программирования, распространенных в настоящее время, можно достаточно адекватно описать условным оператором if. А именно, если какое-то условие не выполняется, происходит переход к следующей ветви условного оператора. Однако команда поиска произвольной последовательности Extend имеет нетривиальную семантику, которая свойственна в большей степени логическим языкам программирования, и которая обусловлена семантикой переменных типа “e” в языках семейства Рефал. Все случаи неуспешного сопоставления с образцом или вызова функций, которые происходят после выполнения команды Extend возвращают управление обработчику команды Extend, который выполняет “откат” вычислений. Обработчик восстанавливает исходное значение аргумента и переносит следующий элемент аргумента в переменную, после чего возобновляет выполнение операций после команды Extend с новым оставшимся значеним аргумента. Для адекватного описания такого поведения в терминах структурного программирования можно использовать оператор цикла while. Наконец, семантика команды очистки стека обработчиков PopHandlers, которая соответствует связующему оператору безусловного перехода, в совокупности с описанными выше командами Extend и Branch, может быть адекватно описана только в терминах прямых переходов к блокам команд.

Вторая особенность связана с семантикой обработки неуспешных вызовов функций. В классических реализациях языков Рефал-2 и Рефал-5 неуспешный вызов функции приводит к аварийному завершению программы. В настоящей реализации используется принцип, аналогичный обработке программных исключений в языках программирования C++, C#, Java. А именно, неуспешный вызов функции возвращает сигнал неуспешного сопоставления с образцом на уровне вызывающей функции. Как следствие, этот сигнал может быть обработан стандартным образом с помощью обработчиков Branch и Extend. Такую семантику можно сравнить с откатными функциями языка Рефал-Плюс [9]. Разница заключается в той особенности, что решение об обработке или необработке неуспешного вызова функции принимается на уровне вызывающей функции, а не на уровне определения. За счет этой особенности семантика такого аспекта язы-

ка ближе именно к обработке исключений, чем к откатным функциям. В зависимости от последовательности команд вызывающей функции, активация результата может проходить как в конексте вызывающей функции, так и на один уровень выше в стеке вызовов. Последняя ситуация, в частности, имеет место для возвращаемого значения функции, если оно указано после оператора безусловного перехода. Текущая реализация интерпретатора промежуточного представления поддерживает эту особенность в форме частичной хвостовой рекурсии.

Следует также отметить особенности реализации интерпретатора с позиций управления памятью. Интерпретатор представленного в настоящей статье промежуточного представления использует автоматическое управление памятью. Списки значений хранятся с использованием подсчета ссылок. Для повышения эффективности, список значений может храниться как в виде непрерывной последовательности значений (или ссылки на фрагмент другого списка), так и в виде последовательности фрагментов. При добавлении значений распознанных переменных к списку командой Emit используется представленная далее эвристическая схема.

1. Если и последнее имеющееся, и вновь добавляемое значение являются последовательными фрагментами одного и того же списка, вместо пары таких фрагментов записывается один фрагмент, содержащий все необходимые значения.

2. В противном случае, если длина вновь добавляемого фрагмента превышает пороговое значение, фрагмент добавляется в виде ссылки. Размер порогового значения определяется экспериментально из соображений эффективного использования кэш-памяти современных процессоров.

3. Если длина вновь добавляемого фрагмента не превышает пороговое значение, то его элементы копируются в новый список.

На ранних стадиях разработки интерпретатора для автоматического управления памятью использовался алгоритм сборки мусора mark and sweep, причем как для значений, так и для кода промежуточного представления, но его накладные расходы оказались слишком велики. После этого реализация алгоритма сборки мусора была переписана на использование схемы с двумя поколениями. Эта модификация дала ускорение в несколько раз, но производительность интерпретатора все равно была недостаточной при необоснованно большом расходе памяти. В итоге замена сборки мусора на статическую арену для хранения кода и подсчет ссылок для фрагментов значений дала ускорение на 2 порядка, по сравнению с mark and sweep с двумя поколениями, и достаточную, на настоящее время, производительность.

Интепретатор в его настоящей версии выполняет трансляцию транслятора, который описан в следующем разделе, из исходного языка в промежуточное представление за 1.1 секунды с использованием памяти 9.4 Мбайт. Последняя версия транслятора, которая могла выполняться внешней реализацией языка Рефал, которая использовалась в качестве отправной точки для разработки самоподдерживающейся реализации языка, была компилируемой в язык C. Она выполняет аналогичное преобразование за 1.57 секунды с использованием памяти порядка 1 Гбайт.

В перспективе есть возможность дальнейшего повышения эффективности использования кэш-памяти (и, как следствие, повышения производительности интерпретатора) за счет более компактного представления последовательностей значений и команд в памяти. Однако на настоящее время авторы оценивают производительность интерпретатора как адекватную и не считают целесообразным усложнять код интерпретатора с целью дальнейшего повышения его производительности.

Ранее в экспериментальных целях разрабатывались также следующие реализации интерпретатора. Простой рекурсивный интерпретатор деревьев команд оказался неработоспособен из-за отсутствия в распространенных языках программирования поддержки гарантированной хвостовой рекурсии. Поэтому на нетривиальных программах всегда оставалась вероятность столкнуться с переполнением стека. Транслятор в код на языке JavaScript был разработан из тех соображений, что современные реализации языка на основе трассирующих JIT-компиляторов (использовалась реализация V8) обеспечат достаточную производительность при относительно простой схеме трансляции программы. Но на практике такая реализация оказалась неработоспособна из-за особенностей алгоритма сборки мусора в машине V8 и высокого расхода памяти. Интерпретатор на языке Haskell разрабатывался в качестве исполнимой модели денотационной семантики языка. Однако адекватная для применения в дальнейших исследованиях производительность и сравнимый объем кода реализации на языке Rust показали, что в ближайшее время дорабатывать этот вариант нет необходимости.

5. РАЗРАБОТКА ТРАНСЛЯТОРА ИЗ ИСХОДНОГО ЯЗЫКА В ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

Промежуточное представление, показанное в настоящей статье, предназначено, как и любое другое промежуточное представление, для использования автоматизированными средствами, а не для написания программы вручную. Для проверки интерпретатора на работоспособность и для реализации представленной выше разновид-

ности языка авторы разработали транслятор из этого языка в промежуточное представление.

Структура транслятора близка к общепринятой. Исходный код проходит представленные далее по тексту стадии (преобразования).

1. Лексический анализ. Строка из букв разделяется на однородные последовательности – токены (константы, идентификаторы, знаки пунктуации и т.п.).

2. Выделение скобок. В последовательности токенов выделяется структура на основе вложенных пар скобок, соответствующих друг другу.

3. Синтаксический анализ. Структурированное скобками дерево токенов преобразуется в абстрактное синтаксическое дерево исходного языка.

4. Трансляция в промежуточное представление. Абстрактное синтаксическое дерево преобразуется в последовательности команд промежуточного представления. Основная часть трансляции заключается в преобразовании образцов в последовательность команд, аналогично алгоритму трансляции в язык сборки в [7] с некоторыми упрощениями. С учетом используемого способа управления памятью (подсчет ссылок на фрагменты), который позволяет вставлять новые копии фрагментов списков за постоянное время, использовать нетривиальные преобразования для оптимизации построения выходного значения на данной стадии нет необходимости.

5. Свертка ветвей распознавания. Последовательности команд, которые являются общими префиксами для всех ветвей распознавания в коде одной функции, выносятся за пределы команды ветвления.

6. Локальная оптимизация. Относительно простой алгоритм трансляции в промежуточное представление порождает код, в котором можно выделить некоторые шаблоны, допускающие дальнейшее упрощение. Например, в последовательности команд (Close) (Check) последняя команда всегда завершится успешно, поэтому ее можно исключить с сохранением семантики. На стадии локальной оптимизации такие шаблоны в дереве команд приводятся к более простому виду.

7. Сериализация промежуточного представления в выходной текстовый формат. На этом этапе выполняется экранирование строк и вывод промежуточного представления в виде текста с отступами, отражающими структуру кода.

Первая версия транслятора была разработана на языке Рефал-5 и запускалась на одной из реализаций, доступных в сети Интернет в свободном доступе в работоспособном состоянии. Затем транслятор с использованием метода раскрутки (bootstrapping) был переведен в режим самоподдержки (self-hosting) по следующей схеме.

Шаг 1. Транслятор T_0, написанный на языке Рефал-5 скомпилирован компилятором Рефал-5 в исполнимый код E.

Шаг 2. Итеративная доработка интерпретатора и транслятора (T).

a. Получено промежуточное представление и машинный код фрагмента транслятора (E(T') = I', Refal-5(T') = E'). В качестве таких фрагментов использовался код каждой из стадий трансляции, последовательно.

b. Промежуточное представление подается на вход интерпретатору для самоприменения (Interpreter(I')(T')). Результаты его выполнения сравниваются с результатом выполнения машинного кода (E'(T')).

c. При выявлении полученных ошибок и несоответствий производится доработка интерпретатора или транслятора.

d. Итерация завершается после того как достигается самоподдержка:

$E(T) = I$,

$\text{Interpreter}(I, T) = I_1$,

$\text{Interpreter}(I_1, T) = I_2$,

$\text{Interpreter}(I_2, T) = I_3$,

$I_2 \equiv I_3$

Итеративная доработка проводилась в ручном режиме в связи со спецификой задачи. Проверка сохранения самоподдержки по сценарию d добавлена в автоматический набор тестов и выполняется при каждом изменении в интерпретаторе или трансляторе. После достижения самоподдержки, исходная реализация Рефал-5 больше не требуется для запуска модифицированного транслятора, что позволяет перейти к следующему шагу.

Шаг 3. Итеративная модификация синтаксиса и семантики.

a. В интерпретатор добавляется поддержка переходного синтаксиса.

b. Код транслятора T переводится на переходный синтаксис.

c. Удаляется поддержка старого синтаксиса, добавляется поддержка нового синтаксиса.

d. Код транслятора T переводится на новый синтаксис.

На каждом этапе проверяется, что состояние самоподдержки сохранено, путем прямой проверки условия, указанного в пункте d второго шага.

6. ОПИСАНИЕ ПРОСТЫХ ТИПОВ И ПОЛИМОРФИЗМА ПО ПОЛЯМ ЗАПИСЕЙ С ИСПОЛЬЗОВАНИЕМ ПРОМЕЖУТОЧНОГО ПРЕДСТАВЛЕНИЯ

В настоящем разделе приведены примеры применения разработанного промежуточного представления для описания простых типов.

```
fn Bool {
   0;
   1;
}

fn Not {
   0 => 1;
   1 => 0;
}

fn IncompleteNot {
   0 => 1;
}
```

Рис. 7. Код функций над булевскими переменными.

```
(Function (Name Bool) (Branch
   ((Recognize Left Exact Int 0)
      (Check)(Return))
   ((Recognize Left Exact Int 1)
      (Check)(Return))
))

(Function (Name Not) (Branch
   ((Recognize Left Exact Int 0)
      (Check)(Value Int 1)(Return))
   ((Recognize Left Exact Int 1)
      (Check)(Value Int 0)(Return))
))

(Function (Name IncompleteNot) (Branch
   ((Recognize Left Exact Int 1)
      (Check)(Value Int 1)(Return))
))
```

Рис. 8. Промежуточное представление функций над булевскими переменными.

Пример для простых типов представлен на рис. 7, 8. Функция Bool задает спецификацию типа данных, который может принимать одно из двух значений – 0 или 1. Если функцию Not вызывать с аргументом, который удовлетворяет спецификации Bool, то вычисление этой функции обязательно успешно завершится, причем результат вычисления будет удовлетворять спецификации Bool. В свою очередь, функция IncompleteNot определена не для всех возможных значений аргумента, удовлетворяющего спецификации Bool. Для представленных функций эти результаты можно получить автоматически с использованием алгоритма частичного вычисления (прогонки), аналогичного [10]. Следует отметить, что в настоящее время алгоритм прогонки реализован не для всех конструкций промежуточного представления. Использованная для получения результатов настоящей статьи реализация алгоритма прогонки выполняет вычисление значения функции на заданной исходной параметризации аргумента и возвращает два набора параметризаций. Первый (положительный) набор содержит все возможные уточнения исходной параметризации, на которых вычисление функции завершается успешно, вместе с представлением результата вычисления в каждом из уточнений. Второй (отрицательный) набор является дополнением к первому, то есть, описывает все возможные уточнения исходной параметризации, на которых вычисление функции завершается неуспешно.

Полиморфизм по полям записей (row polymorphism) – это свойство системы типов, которое допускает определение функций на ассоциативных массивах, для которых система типов гарантирует невозможность возникновения исключительных ситуаций вида “ключ отсутствует в ассоциативном массиве” и “тип значения не соответствует ожидаемому” [11].

Для определения полиморфизма по полям записей достаточно ввести тип записи и три операции, а именно: проекция (получение значения по ключу, Select), добавление поля (Add), удаление поля (Remove), как показано на рис. 9. Для описания типа записи используется ассоциативный массив. Поведение функции Select описано с использованием аксиоматической семантики в терминах троек Хоара. Предусловие SelectPreCondition и постусловие SelectPostCondition характеризуют поведение функции Select, а именно, тот факт, что для любого (разрешимого) свойства s.predicate, если для некоторого набора значений выполнено условие SelectPreCondition s.predicate, то вычисление функции Select с этим аргументом завершится успешно, а для результата будет выполнено условие SelectPostCondition s.predicate.

В терминах алгоритма прогонки, с учетом его текущих ограничений, процесс проверки этого условия может быть описан в следующей форме. Предположим, что вычисление функции s.predicate всегда завершается с успешным или неуспешным результатом. Выполним прогонку для предусловия с произвольным аргументом: <SelectPreCondition s.predicate e.argument> и оставим только положительный набор параметризаций аргумента. Для всех положительных параметризаций P аргумента e.argument выполним прогонку для исходной функции: <Select P>. Результатом прогонки должны быть только положительные параметризации. В противном случае предусловие задает слишком слабые ограничения, при которых невозможно гарантировать успешное завершение функции Select. Для всех положительных параметризаций Q – результатов

```
fn Select {
   s.label {s.label: t.value, ...} => t.value
}

fn SelectPreCondition  {
   s.predicate s.label { s.label: t.value, ...},
      <s.predicate t.value>
}

fn SelectPostCondition {
   s.predicate t.value, <s.predicate t.value>
}

fn Add {
   {...d.dict} t.value s.label,
      excepr{let {s.label: _,...}=t.dict}
      => {...d.dict,s.label: t.value}
}

fn Remove {
   s.label{s.label: _,...d.rest} => {...d.rest}
}
```

Рис. 9. Реализация полиморфизма по полям записей.

второй прогонки выполним прогонку для постусловия: <SelectPostCondition s.predicate Q>. Результатом этой третьей прогонки, аналогично второй, должны быть только положительные параметризации.

В рамках дальнейшей работы предполагается расширить реализацию алгоритма прогонки для полной поддержки промежуточного представления. Кроме того, предполагается расширить исходный язык двумя конструкциями для проверки типов. Эти конструкции соответствуют инструкциям проверки типов в промежуточном представлении, их предварительный вариант представлен далее.

1. Конструкция “type выражение”, которая проверяет, что выражение является типом, то есть, запускает статическую проверку завершимости выражения для любых входных значений.

2. Конструкция “correct (предусловие) (функция) (постусловие)”, которая для типов “предусловие” и “постусловие” выполняет процедуру проверки, описанную выше.

7. ЗАКЛЮЧЕНИЕ

Одним из важнейших практических аспектов языка программирования является коммуникация между разработчиками. Исходный код содержит наиболее точную и актуальную информацию о внутреннем устройстве системы. Для новых языков программирования элемент схожести их синтаксиса с другими языками, с которыми знаком разработчик, способствует более быстрому пониманию кода на этом языке [12]. Кроме того, как следует из результатов [12], язык без ключевых слов или с активным использованием вместо них неочевидных спецсимволов представляется менее интуитивным, чем язык с ключевыми словами. Личный опыт работы авторов со студентами механико-математического факультета МГУ согласуется с этими результатами.

В части известных авторам аналогов предлагаемого подхода к проверке типов следует отметить работу В.И. Шелехова по предикатному программированию [13]. Языки семейства Рефал и суперкомпиляция использовались ранее для верификации свойств, которые имеют характер низкого уровня абстракции от аппаратного обеспечения [14, 15]. В статье [16] приводится пример использования суперкомпиляции в рамках теории типов Мартина-Лёфа. В общем контексте исследования, результаты которого приведены в настоящей статье, следует упомянуть язык исполняемых программных спецификаций [17], который предназначен для спецификации предметно-ориентированных языков. Описание типов с использованием функций используется также в системе PVS (Prototype Verification System) [18], которая используется в упомянутой выше статье [13].

В качестве направления дальнейшей трансформации синтаксиса с целью повышения его эргономичности авторы рассматривают возможность ис-

пользования более привычных обозначений для списков и переменных-образцов. В значительной части распространенных языков программирования литералы списков обозначаются квадратными скобками. Исключение составляют языки семейства LISP, которые можно причислить к относительно нераспространенным. Использование квадратных скобок для списков освободит круглые скобки для их общепринятой роли – конструкции группировки. Для обозначения переменных предполагается использовать только их имя с опциональной аннотацией типа в скобках. Переменные, обозначающие образцы распознавания произвольной последовательности, в этом случае будут обозначаться префиксом-многоточием (таким образом, например, образец (s.a e.b s.a) будет записан как [a ...b a] или [(a: atom) ...b a]). Вопрос записи, допускающей различие между символами и переменными-образцами в настоящее время требует дополнительной проработки. Одно из возможных решений этого вопроса заключается в использовании подхода на основе регистра первой буквы идентификатора, аналогичного используемому в языке Haskell: в переменных первая буква должна быть строчной, а в символах – заглавной.

8. БЛАГОДАРНОСТИ

Работа выполнена в рамках проекта РФФИ 16-07-01178а.

СПИСОК ЛИТЕРАТУРЫ

1. *Hufschmidt T.* A type-system for Nix [Электронный ресурс] // NixCon 2017. 2017. URL: http://nixcon2017.org/schedule.nixcon2017.org/system/event_attachments/attachments/000/0000/003/original/main.pdf
2. *Siek J., Taha W.* Gradual Typing for Objects // ECOOP 2007 – Object-Oriented Programming / под ред. Ernst E. Springer Berlin Heidelberg, 2007. P. 2–27.
3. *Levkivskyi I., Lehtosalo J., Langa Ł.* PEP 544 – Protocols: Structural subtyping (static duck typing) [Электронный ресурс] // Python.org. 2017. URL: https://www.python.org/dev/peps/pep-0544/(дата обращения: 10.01.2019).
4. *Турчин В.Ф.* Метаалгоритмический язык // Кибернетика. 1968. № 4. С. 45–54.
5. *Turchin V.F.* REFAL-5, Programming Guide and Reference Manual. Holoyke, MA: New England Publishing Co., 1989.
6. *Васенин В.А., Кривчиков М.А.* Методы промежуточного представления программ // Программная инженерия. 2017. Т. 8. № 8. С. 345–353.
7. *Романенко С.А.* Машинно независимый компилятор с языка рекурсивных функций: дисс. ... канд. физ.-мат. наук: 01.01.10. Москва: ИПМ АН СССР, 1978. 148 с.
8. *Немытых А.П.* Суперкомпилятор SCP-4: общая структура. М.: Изд-во ЛКИ, 2007. 150 с.
9. *Романенко С.А., Гурин Р.Ф.* Язык программирования Рефал Плюс. Переславль-Залесский: Ун-т города Переславля им. А.К. Айламазяна, 2006. 221 с.
10. *Турчин В.Ф.* Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ. Киев-Алушта. 1972. С. 31–42.
11. *Wand M.* Type inference for record concatenation and multiple inheritance // Information and Computation, 1991. V. 93. № 1. P. 1–15.
12. *Stefik A., Siebert S.* An Empirical Investigation into Programming Language Syntax // Transactions on Computing Education, 2013. V. 13. № 4. P. 19:1–19:40.
13. *Шелехов В.И.* Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная Инженерия. 2011. № 2. С. 14–21.
14. *Немытых А.П.* Верификация как параметризованное тестирование (эксперименты с суперкомпилятором SCP4) // Программирование. 2007. Т. 33. № 1. С. 22–35.
15. *Лисица А.П., Немытых А.П.* Об одном приложении вычислений с оракулом // Программирование. 2010. Т. 36. № 3. С. 43–53.
16. *Ключников И.Г., Романенко С.А.* Суперкомпиляция для теории типов Мартина-Лёфа // Программирование. 2015. № 3. С. 73–87.
17. *Новиков Ф.А., Новосельцев В.Б.* Язык исполняемых программных спецификаций // Программирование. 2010. Т. 36. № 1. С. 66–78.
18. *Shankar N., Owre S.* Principles and Pragmatics of Subtyping in PVS // Recent Trends in Algebraic Development Techniques, WADT'99 / под ред. Bert D., Choppy C., Mosses P.D. Toulouse, France: Springer-Verlag, 1999. V. 1827. P. 37–52.