С. Ю. Скоробогатов, А. М. Чеповский

Язык Refal с функциями высшего порядка

В статье предлагается новый диалект языка Refal. Рассматриваются механизмы реализации функций высшего порядка. Путем ортогонализации основных синтаксических конструкций и семантических категорий языка Refal-5 разрабатывается его преобразование в новый, более мощный диалект, поддерживающий функции высшего порядка.

Введение

Функциональный язык Refal был разработан в нашей стране известным кибернетиком В.Ф.Турчиным в шестидесятых годах прошлого столетия [1]. Новый язык позиционировался В.Ф.Турчиным как средство для описания семантики других языков программирования. Затем язык был упрощен и получил свое название, которое расшифровывается как REcursive Functions Algorithmic Language. Язык Refal предназначен в первую очередь для решения задач, связанных с символьной обработкой.

Основным преимуществом языка Refal с точки зрения его автора является то обстоятельство, что с одной стороны он достаточно прост, чтобы написанные на нем программы могли выступать в роли объектов для теоретического анализа, а с другой стороны он обладает выразительной силой, позволяющей решать с его помощью реальные задачи. Достижение компромисса между двумя этими взаимоисключающими требованиями позволило исследователям экспериментировать с созданием на языке Refal самоприменимых программных инструментов для глубокого анализа и преобразования программ [2].

В настоящее время существуют несколько диалектов языка Refal. Это Refal-2, Refal-5, Рефал Плюс и Refal-6. Отметим характерную особенность большинства разработанных компиляторов языка Refal: они генерируют не машинный код, а код на так называемом языке сборки, который затем исполняется специальным интерпретатором.

В данной работе мы рассматриваем особенности нового диалекта, разработанного в 2005 году и получившего название Refal-7. Компилятор Refal-7 для платформы Microsoft .NET полностью реализован на языке C и работает в операционных системах Windows, Linux и FreeBSD. Он генерирует исполняемые файлы в формате Portable Executable, содержащие метаданные и код на промежуточном языке платформы .NET.

В тексте статьи мы будем использовать расширенную БНФ-нотацию для записи синтаксиса языка Refal-7. При этом мы будем придерживаться следующих соглашений:

- нетерминальные символы обозначаются русскими словами и словосочетаниями (слова в словосочетании разделяются знаком подчеркивания);
- терминальные символы записываются в двойных кавычках;
- знак `::=` разделяет левую и правую части правила;
- знак `|` обозначает альтернативу;
- квадратные скобки обозначают вхождение подвыражения 0 или 1 раз (другими словами, необязательность вхождения);
- фигурные скобки обозначают вхождение подвыражения 0 или более раз (итерация);
- фигурные и квадратные скобки обладают большим приоритетом, чем знак альтернативы;
- для изменения приоритета используются круглые скобки.

1. Функции высшего порядка

Функции высшего порядка (high order functions или higher order functions) – это функции, в качестве аргументов или возвращаемого значения которых могут выступать другие функции.

Использование функций высшего порядка, совершенно естественно вытекающее из ламбда-исчисления, можно рассматривать как один из механизмов абстракции, помогающих при проектировании программ.

Известно, что любой механизм абстракции можно реализовать на любом алгоритмически полном языке программирования. Например, разработанную в рамках объектно-ориентированной парадигмы иерархию классов можно реализовать на классическом Фортране, в синтаксисе которого отсутствуют какая-либо поддержка объектно-ориентированного программирования. Аналогично, функции высшего порядка можно использовать при программировании на любом языке, хотя чаще всего они применяются в функциональных языках.

Однако, несмотря на то, что функции высшего порядка доступны в любом языке программирования, интенсивность их использования существенно зависит от наличия в языке специальных средств поддержки. Поэтому, если сравнить стандартные библиотеки языков SML и C, можно заметить, что в SML функции высшего порядка применяются чрезвычайно активно, а в стандартной библиотеке языка C присутствует только одна функция высшего порядка `qsort`.

Для того чтобы определить, чем языки с развитой поддержкой функций высшего порядка (SML, Haskell, Scheme) отличаются от других языков программирования, нам

понадобится такое понятие, как *полноправные объекты данных* (first-class values). Объекты данных называются полноправными, если соблюдаются следующие условия: их можно присваивать локальным переменным; они могут являться компонентами составных структур данных; они могут быть переданы функциям в качестве параметров; их можно возвращать из функций; их можно создавать во время выполнения программы.

В языках программирования для некоторых типов объектов данных часть условий может не соблюдаться. Например, в языке С структуры не могут выступать в качестве возвращаемых из функций значений. При этом можно заметить следующий факт: чем больше язык дискриминирует такие объекты данных, как функции, тем меньше интенсивность использования функций высшего порядка при программировании на этом языке. Поэтому можно утверждать, что для полноценной поддержки функций высшего порядка требуется, чтобы функции были полноправными объектами данных.

В языках, в которых функции являются полноправными объектами данных, широко распространены следующие приемы программирования:

- применение обобщенных алгоритмов, реализованных в виде функций высшего порядка и позволяющих обрабатывать структуры данных различных типов;
- поддержка псевдобесконечных структур данных в языках с аппликативной (жадной) семантикой;
- создание синтаксических анализаторов на основе комбинаторов, реализованных в виде функций высшего порядка.

В разговоре о функциях, которые могут выступать в роли полноправных объектов данных, неизбежна терминологическая путаница. Дело в том, что мы привыкли называть функцией некоторый фрагмент программы, который содержит код, выполняющий вычисление значения функции на основе передаваемых ей параметров. С другой стороны, функции в качестве полноправных объектов данных могут храниться в переменных и передаваться в другие функции. При этом, как правило, в переменных сохраняются вовсе не фрагменты программы, хотя и такая возможность не исключена.

Другими словами, целесообразно отличать функции как фрагменты программы от функций как объектов данных. Поэтому мы будем называть функции, выступающие в роли объектов данных, *экземплярами функций*. В дальнейшем мы увидим, что такое название вполне оправдано, потому что механизм порождения новых функций подразумевает создание различных экземпляров одной и той же функции в программе. Тем самым можно провести аналогию между классом в объектно-ориентированном языке и функцией: класс – это шаблон для создания объектов (экземпляров класса), а функция – это шаблон для создания экземпляров функций.

Основной и, возможно, единственной операцией над экземплярами функций является *операция вызова экземпляра функции*. При вызове экземпляру функции передаются фактические параметры, на основе которых он вычисляет и возвращает некоторое значение. Таким образом, операция вызова экземпляра аналогична операции вызова функции.

Рассмотрим механизмы, которые используются в языках программирования для придания функциям статуса полноправных объектов данных. Условно разделим их на две группы:

1. механизмы обеспечения хранения и передачи экземпляров функций;
2. механизмы порождения экземпляров функций.

Хранение и передача экземпляров функций обычно реализуются в языках достаточно просто. Для этого мы разрешаем хранение экземпляров функций в переменных и в составе сложных структур данных. И, кроме того, обеспечиваем передачу экземпляров функций через параметры и возвращение экземпляров функций из других функций. Эти механизмы широко распространены во многих языках программирования. Например, в C разрешены указатели на функции, которые могут храниться где угодно и передаваться куда угодно, а в языке Pascal существуют специальные процедурные типы.

Что касается порождения экземпляров функций, то однозначного решения этой задачи, сочетающего универсальность и удобство применения и подходящего для любых языков, предложить чрезвычайно трудно. Поэтому мы ограничимся рассмотрением трех механизмов, реализованных в функциональных языках SML, Haskell и Scheme: это вложенные функции, анонимные функции и динамические окружения.

Вложенная функция (nested function) – это функция, объявленная внутри другой функции и имеющая доступ к ее локальным переменным. При создании экземпляра вложенной функции она параметризуется значениями локальных переменных объемлющей функции. Это означает, что на основе разных наборов значений локальных переменных объемлющей функции могут быть созданы различные экземпляры вложенной функции. В принципе, поддержка вложенных функций – это основной механизм порождения экземпляров функций.

Анонимная функция (anonymous function) – это вложенная функция, не имеющая имени. Анонимные функции во всем аналогичны обычным вложенным функциям, но объявляются внутри некоторых выражений языка.

Окружение (environment) – это отображение множества имен локальных переменных функции в множество значений этих переменных. Обычно окружения хранятся в составе фреймов функций на стеке вызовов. По аналогии с автоматическими

переменными языка C, которые как раз и размещаются в стеке, такие окружения можно называть *автоматическими окружениями*.

Характерной особенностью автоматических окружений является то, что при выходе из функции ее окружение стирается со стека вместе с фреймом. Эта особенность препятствует использованию автоматических окружений в языках, поддерживающих функции высшего порядка. Поясним сказанное на примере. Пусть существуют функции A и B, причем B вложена в A и использует ее локальные переменные. И пусть функция A осуществляет передачу экземпляра функции B вовне (например, сохранив его в глобальной переменной или, что более характерно для функциональных языков, вернув экземпляр функции B в качестве значения). Если мы используем автоматические окружения, то при выходе из функции A ее окружение стирается со стека, что делает локальные переменные функции A недоступными из переданного вовне экземпляра функции B. Тем самым экземпляр B становится неработоспособным, так как при попытке его вызова он станет обращаться к недоступным переменным.

Эту проблему можно решить, если для хранения окружений использовать не стек, а динамическую память (кучу). Это решение является принципиальным, хотя на первый взгляд может показаться, что оно относится к реализации языка. На самом деле оно напрямую влияет на семантику языка, обеспечивая доступность локальных переменных после выхода из функций.

Окружения, размещенные в динамической памяти, мы будем называть *динамическими окружениями*. В случае использования динамических окружений при выходе из функции значения ее локальных переменных не стираются вместе с фреймом. Они остаются в динамической памяти и могут быть использованы существующими экземплярами вложенных функций. Естественно, использование динамических окружений подразумевает наличие какого-либо автоматического менеджера памяти (сборщика мусора).

Описанные выше механизмы поддержки функций высшего порядка в наиболее популярных функциональных языках программирования не обеспечиваются ни одним из диалектов языка Refal, что наглядно продемонстрировано в сводной таблице (Таблица 1). Поэтому решалась задача добавления в Refal недостающих возможностей, причем не путем простого добавления новых элементов синтаксиса, а, главным образом, через переосмысление уже существующих синтаксических конструкций

2. Передача, хранение и порождение экземпляров функций в Refal-7

Основной задачей, которая ставилась при разработке языка Refal-7, была задача реализации в языке Refal недостающих возможностей, необходимых для поддержки функций высшего порядка. Для создания Refal-7 в качестве базового диалекта был выбран Refal-5, так как в последние годы для решения наиболее сложных задач именно этот диалект пользуется наибольшей популярностью. Полное описание Refal-5 можно найти в [3].

Необходимость поддержки функций высшего порядка потребовала внедрения в язык механизмов передачи, хранения и порождения экземпляров функций.

Программы, написанные на языке Refal, оперируют символами, представляющими атомарные значения. В Refal-5 используются три вида символов: символы-слова, символы-литеры и символы-макроцифры. Передача и хранение экземпляров функций в Refal-7 осуществляется за счет добавления нового типа символов, а именно: символов-функций. Символы-функции наряду с другими символами могут входить в состав Refal-выражений. При этом через сопоставление объектных выражений с образцами символы-функции могут быть связаны с переменными.

Отметим, что каждой функции, объявленной в программе, соответствует свой символ-функция, который записывается как имя этой функции.

Операция вызова экземпляра функции полностью идентична операции вызова функции. Другими словами, для вызова экземпляра функции используются скобки активации. Например, если значением некоторой переменной `s.1` является символ-функция, то для вызова символа-функции мы можем записать

```
<s.1 аргумент>
```

При этом на месте переменной `s.1` мы можем записать любое выражение, результатом вычисления которого является символ-функция. Например, если некоторая функция `F` возвращает символ-функцию, то этот символ можно вызвать следующим образом:

```
< <F аргумент1> аргумент2 >
```

Нагромождение открывающих скобок активации выглядит не очень красиво, поэтому синтаксис вызова функций в Refal-7 расширен таким образом, что вышеприведенное выражение можно переписать как

```
<F аргумент1, аргумент2>
```

А в общем случае выражение вида

```
<выражение1, выражение2, … , выражениеN>
```

является сокращенной записью выражения

```
< … < <выражение1> выражение2 > … выражениеN >
```

Возможность порождения экземпляров функций появилась в Refal-7 благодаря добавлению вложенных и анонимных функций. Такие функции могут входить в состав Refal-выражений как термы, для чего используется синтаксическая конструкция вида

```
[ "$FUNC" Имя_функции ] Блок,
```

где `Блок` – это заключенная в фигурные скобки последовательность предложений.

Например, анонимная функция, умножающая число на 2, может быть записана внутри Refal-выражения как

```
{ s.1 = <"*" s.1 2> }
```

Мы можем вызвать ее в программе следующим образом:

```
< { s.1 = <"*" s.1 2> } 8>
```

Если мы захотим дать вложенной функции имя, мы должны использовать ключевое слово `$FUNC`:

```
$FUNC MultiplyByTwo { s.1 = <"*" s.1 2> }
```

Рассмотрим несколько примеров, демонстрирующих использование функций высшего порядка в Refal-7.

Пример: функция Fxy

Составим функцию `Fxy`, выполняющую замену всех вхождений одного терма в выражении на другой терм. Без использования функций высшего порядка функцию `Fxy` можно записать следующим образом:

```
Fxy {
    t.X t.Y t.X e.1 = t.Y <Fxy t.X t.Y e.1>;
    t.X t.Y t.Z e.1 = t.Z <Fxy t.X t.Y e.1>;
    t.X t.Y = ;
    }
```

Вызов функции `Fxy` для замены всех букв `'a'` в выражении `'aaabbbzaaa'` на буквы `'b'` выглядит как

```
<Fxy 'a' 'b' 'aaabbbzaaa'>
```

Перепишем функцию таким образом, чтобы на основе информации о том, какой терм на какой надо заменить, она возвращала экземпляр функции, выполняющей замену:

```
Fxy {
    t.X t.Y =
        $FUNC F {
            t.X e.1 = t.Y <F e.1>;
            t.Z e.1 = t.Z <F e.1>;
```

```
            = ;
            };
    }
```

Из примера видно, что функция `F` вложена в функцию `Fxy` и имеет доступ к ее локальным переменным `t.X` и `t.Y`. Для вызова функции `Fxy` мы можем использовать выражение

```
<Fxy 'a' 'b', 'aaabbbzaaa'>
```

Теперь обобщим функцию `Fxy` таким образом, чтобы она могла выполнять более сложные замены. Для этого мы будем в качестве параметра передавать ей экземпляр функции, которая принимает терм и определяет, на какой терм его надо заменить:

```
Fxy s.G =
    $FUNC F {
        t.X e.1 = <s.G t.X> <F e.1>;
        = ;
        };
```

Характерная особенность приведенного примера: тело функции `Fxy` не заключено в фигурные скобки. Если функция содержит только одно предложение, то в этом случае синтаксис Refal-7 позволяет фигурные скобки не писать (при этом, однако, требуется, чтобы предложение заканчивалось точкой с запятой).

При вызове этого варианта функции `Fxy` мы используем объявление анонимной функции:

```
<Fxy { 'a' = 'b'; s.1 = s.1; }, 'aaabbbzaaa'>
```

Пример: композиция функций

Запишем функцию **"@",** возвращающую композицию функций, список которых передается ей в качестве аргумента:

```
"@" {
    s.F = s.F;
    s.F e.List =
        { e.Arg = <s.F <"@" e.List, e.Arg>>; };
    }
```

В качестве примера использования композиции функций рассмотрим последовательное применение нескольких функций, соответствующих фазам работы некоторого компилятора, к тексту компилируемой программы:

```
<CodeGen <Optimize <Chk <Parse <Lex e.Prog>>>>>
```

С помощью функции `"@"` это выражение можно записать более элегантно:

```
<"@" CodeGen Optimize Chk Parse Lex, e.Prog>
```

Пример: функция Map

Функция `Map` вызывает некоторую функцию для каждого терма в выражении и конкатенируют полученные результаты:

```
Map {
    s.F t.X e.1 = <s.F t.X> <Map s.F e.1>;
    s.F = ;
    }
```

Теперь мы можем использовать функцию `Map` для вычисления декартова произведения двух выражений:

```
Pair t.a = { t.b = (t.a t.b); };

Cartprod e.xs (e.ys) =
    <Map { t.X = <Map <Pair t.X> e.ys > } e.xs >;
```

Пример: функция Filter

Функция `Filter` оставляет в выражении только те термы, которые удовлетворяют некоторому предикату:

```
Filter {
    s.P t.X e.1, <s.P t.X> = t.X <Filter s.P e.1>;
    s.P t.X e.1 = <Filter s.P e.1>;
    s.P = ;
    };
```

3. Расширение семантики сопоставления с образцом

В языке Refal-7 нами была расширена из соображений симметрии семантика сопоставления с образцом.

Итак, в языке Refal, как и в любом другом чистом функциональном языке, отсутствует понятие присваивания переменной значения. Вместо него существует понятие *связывания* переменной со значением. Специфика связывания заключается в том, что переменная, будучи связана с некоторым значением, уже никак не может это значение поменять: оно остается с переменной до конца ее времени жизни. Сопоставление объектного выражения с образцом является единственным доступным в языке Refal способом связывания значений с переменными. Для того чтобы представить результат связывания переменных в образце в результате его сопоставления с объектными выражением, нам понадобится понятие подстановки.

Пусть $V = \{ v_i \}$ – множество переменных, входящих в некоторый образец p, а E – множество всех возможных объектных выражений. Будем называть *подстановкой* отображение sub: $V \rightarrow E$, которое каждой свободной переменной $v_i \in V$ ставит в соответствие объектное выражение такое, что:

- $sub(v_i)$ – символ, если v_i – s-переменная;
- $sub(v_i)$ – объектный терм, если v_i – t-переменная;
- $sub(v_i)$ – непустое объектное выражение, если v_i – v-переменная;
- $sub(v_i)$ – произвольное объектное выражение, если v_i – e-переменная.

Применение подстановки sub к образцу p заключается в замене каждого вхождения в образец p каждой переменной $v_i \in V$ на объектное выражение $sub(v_i)$. Будем обозначать эту операцию, как p/sub. Несложно показать, что в результате применения подстановки к образцу получается некоторое объектное выражение.

Пусть даны объектное выражение e и образец p. Тогда результатом сопоставления выражения e с образцом p является конечное множество S подстановок, применение которых к образцу p дает объектное выражение e:

$$S = \{ sub_i \mid p/sub_i = e \}.$$

Если множество S – пустое, то говорят, что сопоставление неуспешно.

Рассмотрим пример. Пусть дан образец

```
p = e.1 '+' e.2
```

и объектное выражение

```
e = 10 '+' 20 '+' 30 '+' 40
```

Тогда возможны три варианта успешного сопоставления объектного выражения e с образцом p:

```
1. e.1 -> 10,                 e.2 -> 20 '+' 30 '+' 40
2. e.1 -> 10 '+' 20,          e.2 -> 30 '+' 40
3. e.1 -> 10 '+' 20 '+' 30,   e.2 -> 40
```

Рассмотренный пример иллюстрирует тот факт, что сопоставление далеко не всегда бывает однозначным. А так как переменные образца не могут быть связаны сразу с несколькими значениями, то нам нужно уметь выбирать из нескольких вариантов один.

Разрешение неоднозначности сопоставления осуществляется путем введения отношения порядка на множестве подстановок.

Количество термов, из которых состоит некоторое выражение e, мы будем называть *длиной выражения* e и обозначать |e|. Тогда *отношение порядка на множестве подстановок*, полученном в результате сопоставления некоторого выражения

с образцом p, определяется следующим образом: подстановка sub_1 предшествует подстановке sub_2, если $|sub_1(v)| < |sub_2(v)|$, где v - такая переменная образца p, что для любой переменной $v' < v$ верно равенство $sub_1(v') = sub_2(v')$.

В языке Refal-5 принято следующее *отношение порядка на множестве переменных образца*: переменная v_i предшествует переменной v_j, если при просмотре образца слева направо переменная v_i встречается первой.

В качестве примера пронумеруем переменные в следующем образце согласно определенному нами отношению порядка:

```
Alpha e.1 'x' e.1 (e.3 e.3 e.2) t.X s.Y
      1           2       3    4   5
```

Отношение порядка на множестве переменных образца, принятое в Refal-5, не является единственно возможным. Действительно, выбор направления просмотра образца, при котором переменные нумеруются слева направо, ничем не обоснован. С тем же успехом можно нумеровать переменные справа налево и, как мы увидим в дальнейшем, даже более изощренным способом.

Покажем на примере, что разрешение неоднозначности сопоставления с образцом, базирующееся на нумерации переменных образца слева направо, не всегда адекватно решаемым задачам. Для этого составим функцию Calc, вычисляющую значение скобочного арифметического выражения:

```
Calc {
    e.1 '+' e.2 = <Add <Calc e.1> <Calc e.2>>;
    e.1 '-' e.2 = <Sub <Calc e.1> <Calc e.2>>;
    e.1 '*' e.2 = <Mul <Calc e.1> <Calc e.2>>;
    e.1 '/' e.2 = <Div <Calc e.1> <Calc e.2>>;
    ( e.1 )     = <Calc e.1>;
    s.1         = s.1;
    }
```

Нетрудно заметить, что функция Calc не всегда правильно вычисляет выражения, содержащие операции вычитания и деления. Действительно, рассмотрим вызов функции Calc для вычисления значения выражения 20-10-5:

```
<Calc 20 '-' 10 '-' 5>
```

Сопоставление выражения 20 '-' 10 '-' 5 с образцом e.1 '-' e.2 дает две подстановки:

```
1. e.1 -> 20,          e.2 -> 10 '-' 5
2. e.1 -> 20 '-' 10,   e.2 -> 5
```

В силу того, что в Refal-5 принята нумерация переменных слева направо, неоднозначность будет разрешена в пользу первой подстановки, и вызов функции будет переписан как

```
<Sub <Calc 20> <Calc 10 '-' 5>>
```

Другими словами, скобки в выражении `20-10-5` будут расставлены так, как если бы операция вычитания была правоассоциативной, а именно: `20-(10-5)`.

Для того чтобы функция `Calc` работала правильно, в нее нужно добавить два предложения, которые значительно ухудшают читаемость кода и снижают его эффективность:

```
Calc {
    e.1 '+' e.2         = <Add <Calc e.1> <Calc e.2>>;
    e.1 '-' e.2 '-' e.3 = <Calc <Calc e.1 '-' e.2> '-' e.3>;
    e.1 '-' e.2         = <Sub <Calc e.1> <Calc e.2>>;
    e.1 '*' e.2         = <Mul <Calc e.1> <Calc e.2>>;
    e.1 '/' e.2 '/' e.3 = <Calc <Calc e.1 '/' e.2> '/' e.3>;
    e.1 '/' e.2         = <Div <Calc e.1> <Calc e.2>>;
    ( e.1 )             = <Calc e.1>;
    s.1                 = s.1;
    }
```

В Refal-7 направление просмотра образца при нумерации переменных может меняться в зависимости от модификаторов направления `$R` и `$L`. Соответственно, синтаксис образцов Refal-7 в расширенной БНФ-нотации выглядит как

```
Образец ::= [ "$L" | "$R" ] { Образцовый_терм }
Образцовый_терм ::= Символ | Переменная | "("Образец")"
```

Модификатор `$R` означает нумерацию переменных справа налево, а модификатор `$L` – слева направо. При этом модификаторы разрешено использовать как в начале образца, так и после любой открывающей структурной скобки. Если перед образцом не указан модификатор, то подразумевается `$L`. Если модификатор не указан после открывающей структурной скобки, то направление нумерации переменных наследуется с предыдущего уровня.

В качестве примера пронумеруем переменные в следующем образце:

```
Alpha e.1 'x' e.1 ($R e.3 e.3 e.2) t.X s.Y
      1                   3   2    4   5
```

В этом примере на верхнем уровне переменные нумеруются слева направо, а внутри структурных скобок – справа налево.

Используя модификаторы направления, мы можем убрать два лишних предложения из функции `Calc`:

```
Calc {
    e.1 '+' e.2    = <Add <Calc e.1> <Calc e.2>>;
    $R e.1 '-' e.2 = <Sub <Calc e.1> <Calc e.2>>;
    e.1 '*' e.2    = <Mul <Calc e.1> <Calc e.2>>;
    $R e.1 '/' e.2 = <Div <Calc e.1> <Calc e.2>>;
    ( e.1 )        = <Calc e.1>;
    s.1            = s.1;
    }
```

4. Синтаксис предложений Refal-7

Поддержка функций высшего порядка потребовала добавления в Refal новых элементов синтаксиса. В то же время оказалось, что эти новые элементы способны заменить некоторые существующие в Refal-5 синтаксические конструкции. Это дало возможность упростить язык и добавить в него дополнительные возможности.

В процессе разработки языка Refal-7 наиболее существенным изменениям подвергся синтаксис предложений, из которых состоят объявления функций. В Refal-5 предложения состоят из левой и правой частей. Левая часть начинается с образца, за которым следует последовательность условий. Правая часть представляет собой либо результатное выражение, либо блок:

```
Предложение ::= Левая_часть Правая_часть
Левая_часть ::= Образец { Условие }
Условие ::= "," Выражение ":" Образец
Правая_часть
  ::= "=" Выражение
    | "," Выражение ":" Блок
```

В основе преобразования синтаксиса предложений Refal-5 лежали две идеи: разбиение предложений на ортогональные действия и замена блоков на вложенные функции. Действительно, в предложении не обязательно различать левую и правую части. Вместо этого можно разбить синтаксические конструкции, образующие предложение, на некоторые элементарные составляющие и разрешить свободное комбинирование этих составляющих. Кроме того, можно заметить, что блоки Refal-5 аналогичны анонимным функциям и могут быть упразднены.

Предложение в Refal-7 начинается с образца, за которым следует непустая последовательность так называемых действий:

```
Предложение ::= Образец Действие { Действие }
```

Действие – это некоторая операция, на вход которой подается объектное выражение и окружение. В процессе выполнения предложения управление передается от одного действия к другому. Каждое действие преобразует переданное ему объектное выражение или окружение и передает их на вход следующему действию.

Всего существует три типа действий: действие-сборка, действие-сопоставление и действие-вызов. При этом каждое из этих действий может быть прозрачным или непрозрачным для неуспехов

Синтаксис действий в расширенной БНФ-нотации записывается как

```
Действие
  ::= ( ","  | "="  ) Выражение
    | ( ":"  | "::" ) Образец
    | ( "->" | "=>" ) Терм
```

При этом знаки `","`, `":"` и `"->"` обозначают прозрачные варианты сборки, сопоставления и вызова. Непрозрачные варианты этих действий обозначаются знаками `"="`, `"::"` и `"=>"`.

Запишем алгоритм вычисления предложения Refal-7.

1. Пусть A_1, ..., A_N – последовательность действий предложения.

   Пусть `Arg` – объектное выражение, переданное функции в качестве аргумента.

   Выполнение предложения начинается с сопоставления объектного выражения `Arg` с образцом в начале предложения. В результате сопоставления получается множество подстановок S_0.

2. Если множество S_0 – пустое, и данное предложение – не последнее в функции, то переходим к выполнению следующего предложения. Если же данное предложение – последнее, то функция завершается неуспехом.

   Если множество S_0 – непустое, то пусть окружение `Env` – минимальная подстановка, принадлежащая S_0. Удалим подстановку `Env` из S_0.

   Пусть `k = 1` – номер действия, которое будет выполняться.

3. Если A_k – сопоставление, то переход на 4.

   Если A_k – сборка, то переход на 6.

   Если A_k – вызов, то переход на 7.

4. A_k – действие, означающее сопоставление с некоторым образцом `P`.

Подставляем значения переменных из Env в образец P и выполняем сопоставление объектного выражения Arg с образцом P. В результате сопоставления получается множество подстановок S_k.

5. Если множество S_k – пустое, то переход на 8.

   Иначе пусть min_k – минимальная подстановка, принадлежащая S_k. Тогда удалим min_k из S_k и присвоим Env результат объединения Env с min_k.

   Переход на 9.

6. A_k – действие, означающее сборку некоторого выражения E.

   Подставляем значения переменных из Env в выражение E и осуществляем выполнение всех вызовов функций. В результате либо одна из вызванных функций возвратит неуспех, либо мы получим не содержащее скобок активации объектное выражение.

   В случае неуспеха переходим на 8.

   Иначе присваиваем полученное объектное выражение переменной Arg и переходим на 9.

7. A_k – действие, означающее вызов некоторого терма T.

   Подставляем значения переменных из Env в терм T и осуществляем выполнение всех вызовов функций.

   Если одна из вызванных функций возвратит неуспех, или в результате мы не получим символ-функцию, то переходим на 8.

   Вызываем полученный символ-функцию с аргументом Arg. Если это приведет к неуспеху, то переходим на 8. Иначе присвоим переменной Arg значение, возвращенное вызванным символом-функцией, и перейдем на 9.

8. Осуществление отката.

   Если A_k – непрозрачное действие, то функция завершается неуспехом.

   k = k – 1.

   Если k = 0, то переход на 2.

   Если A_k – сопоставление, то присвоим Env результат вычитания min_k из Env и перейдем на 5.

   Переход на 8.

9. Если k = N, то функция возвращает значение Arg.

   Иначе k = k + 1 и переходим на 3.

Рассмотрим несколько примеров, демонстрирующих использование действий в предложениях Refal-7.

Пример: функция Failed

Функция `Failed` инвертирует неуспех. Она получает в качестве параметра экземпляр функции и фактические параметры для вызова этого экземпляра. Если вызов порождает неуспех, функция `Failed` возвращает пустое выражение. В противном случае она сама порождает неуспех.

```
Failed {
    s.F e.arg, <s.F e.arg> = <>;
    e.1 = ;
    };
```

Этот пример демонстрирует использование прозрачного и непрозрачного вариантов действия-сборки. Обратите внимание, что пустые скобки активации в Refal-7 обозначают порождение неуспеха.

Пример: функция EqualTerms

Функция `EqualTerms` определяет, состоит ли объектное выражение из одинаковых термов:

```
EqualTerms {
    t.X e.1 : e.1 t.X = ;
    = ;
};
```

Пример: функция Sign

Функция `Sign` определяет знак целого числа. Она возвращает `-1` в случае отрицательного числа, `1` – в случае положительного числа и 0 – для нулевого числа. На Refal-5 эту функцию можно записать с использованием блока:

```
Sign {
    s.1, <Compare s.1 0>:
        {
            '<' = '-'1;
            '=' = 0;
            '>' = 1
        };
    };
```

В Refal-7 вместо блока используется вызов анонимной функции:

```
Sign s.1,
    <Compare s.1 0> =>
        {
```

```
        '<' = -1;
        '=' = 0;
        '>' = 1
    };
```

Заключение

В заключении приведем синтаксис Refal-7 в расширенной БНФ-нотации:

```
Модуль ::= Объявление_функции { Объявление_функции }
Объявление_функции ::=
    ["$ENTRY"] Имя_функции ( Блок | Предложение ";" )
  | "$EXTERN" Имя_функции {"," Имя_функции } ";"
Блок ::= "{" Предложение { ";" Предложение } [";"] "}"
Предложение ::= Образец Действие { Действие }
Действие ::=
    ( ","  | "="  ) Выражение
  | ( ":"  | "::" ) Образец
  | ( "->" | "=>" ) Терм
Выражение ::= { Терм }
Терм ::=
    Символ
  | Переменная
  | "(" Выражение ")"
  | "<" Выражение { "," Выражение } ">"
  | [ "$FUNC" Имя_функции ] Блок
Образец ::= ["$L" | "$R"] { Образцовый_терм }
Образцовый_терм ::=
    Символ
  | Переменная
  | "(" Образец ")"
```

Предложенный в данной статье язык Refal-7 находится в одном ряду с функциональными языками, обеспечивающими полноценную поддержку функций высшего порядка, и включает Refal-5 в качестве подмножества. Язык Refal-7 может претендовать на роль метаязыка для работы с символьными данными как в простейших

задачах обработки символьной информации, так и в сложных задачах искусственного интелекта.

В силу того, что Refal-7 сочетает простоту синтаксиса с широкими выразительными возможностями, которые обеспечиваются, в том числе, функциями высшего порядка и управлением порядком отождествления, он является идеальным языком для обучения функциональной парадигме программирования.

Литература

1. Турчин В.Ф. Метаалгоритмический язык. — Кибернетика № 4, 1968. С. 116–124.
2. Turchin V.F. Metacomputation: Metasystem Transitions plus Supercompilation — LNCS, Springer, 1996. V. 1110. P. 481-510.
3. Turchin V.F. Refal-5. Programming Guide and Reference Manual. — New England Publishing Co., Holyoke, 1989.

Таблица 1. Поддержка функций высших порядков в функциональных языках программирования

Язык	Функции как переменные и аргументы других функций	Функции в качестве возвраща-емых значений	Вложенные функции	Динамичес-кие среды	Анонимные функции
Scheme	да	да	да	да	да
SML	да	да	да	да	да
Haskell	да	да	да	да	да
Refal-(2,5,6,Плюс)	да	да	нет	нет	нет

Sergei Yu. Skorobogatov, Andrey M. Chepovskiy

Refal with High Order Functions

The new dialect of Refal language is offered. The mechanisms of implementation of the high order functions are considered. The orthogonalization of the basic syntactic constructs and semantic categories of Refal-5 language is given. The transformation of Refal-5 to new, more powerful dialect supporting the high order functions is developed.