
С.А. Романенко

# Рефал и Агда как воплощения идеи "метаалгоритмического языка"

***Рекомендуемая форма библиографической ссылки***

Романенко С.А. Рефал и Агда как воплощения идеи "метаалгоритмического языка" // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18-23 сентября 2017 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2017. — С. 417-424. — URL: http://keldysh.ru/abrau/2017/63.pdf doi:10.20948/abrau-2017-63

***Размещена также презентация к докладу***

# Рефал и Агда как воплощения идеи "метаалгоритмического языка"

**С.А. Романенко**[1]

*1 ИПМ им. М.В. Келдыша РАН*

**Аннотация.** Сравниваются возможности языков Рефал и Агда. Показывается, что они основаны на одних и тех же принципах, но Агда предоставляет дополнительные возможности для записи утверждений о программах и доказательств этих утверждений.

**Ключевые слова:** алгоритмический язык, метаалгоритмический язык, метасистемный переход, статическая типизация, зависимые типы.

## 1. Введение

Язык Рефал ("РЕкурсивно-Функциональный Алгоритмический язык") был предложен В.Ф.Турчиным в 1966 году [1,2] и привлек внимание тем, что он резко отличался от других языков программирования, находившихся в то время в поле зрения общественности (таких как Фортран, Алгол-60 и Кобол). При этом Рефал отличался не только внешне, но и внутренне, ибо сама система понятий, на которых он был основан, выглядела как нечто странное и "ортогональное" по сравнению с тем, что считалось "респектабельным" и "естественным".

Кратко особенности Рефала (и его отличия от других языков) можно сформулировать следующим образом.

- Обработка **символьной информации** (цепочек и деревьев).
- **Декларативность**. Описываем не последовательность действий, а то, что нам хочется получить (и из чего).
- **Управление** в программе организуется не с помощью переходов и циклов, а с помощью *вызовов функций* и *рекурсии*.
- **Сопоставление с образцами** используется для организации разборов случаев и ветвлений в программе (вместо условных переходов и других подобных управляющих конструкций).

Вопрос: "экзотичность" Рефала – это было хорошо или плохо? По этому поводу мнения разделились. Кто-то воспринял появление Рефала как проявление "сектантства" и уклонение в сторону от "столбового пути", а кому-то необычность Рефала понравилась.

Но, в действительности, необычность Рефала была обусловлена его предназначением, тем, как его предполагалось использовать. Кстати, Рефал не

сразу получил своё нынешнее имя! Первоначально он (просто и скромно) назывался "метаалгоритмическим языком" [2].

Замысел состоял в том, что Рефал будет использоваться не в качестве "обыкновенного" языка программирования, а в качестве инструмента "мета-деятельности", связанной с языками программирования. Например, для описания семантики языков программирования и быстрого изготовления реализаций языков программирования.

Частично, этот первоначальный замысел реализовался. Например, В.Ф.Турчиным был написан на Рефале компилятор с Алгола-60 в язык ассемблера [3]. Но, с другой стороны, быстро обнаружилось, что Рефал можно использовать и "нецелевым образом", т.е. не в качестве невиданного "метаалгоритмического", а как обыкновенный "алгоритмический" язык. В особенности для задач, где требоваласть работа не с числами, а с какой-то "символьной" информацией. Конечно, тексты программ являются символьной информации, но ведь есть и многие другие примеры информации такого рода. Например – алгебраические выражения. И соблазн спуститься с "мета-небес" на землю оказался слишком силен. И это искушение первым не выдержал сам же В.Ф.Турчин, занявшись задачами, связанными с преобразованиями алгебраических выражений [4].

И, в итоге, получилось так, что Рефал начал, использоваться не для каких-то "возвышенных" целей, а для решения различных задач "имеющих большое народнохозяйственное значение" [5,6,7], постепенно превращаясь из "метаязыка" в "заурядный" язык программирования.

Однако, через некоторое время, В.Ф.Турчин продолжил деятельность, ради которой создавался Рефал. При этом, у него возникла мысль, что можно использовать Рефал, в качестве метаязыка, применительно к нему самому [7,8,9,10].

В качестве курьеза отметим, что сборник [10] был издан анонимно, т.е. как бы от лица "неизвестных авторов". В этом проявилась специфика эпохи, на которой, за недостатком места, мы задерживаться не будем, а сразу перейдем к ситуации в настоящий момент (опуская много другого интересного).

Можно ли считать Рефал "живым" языком, или же он относится к категории "древних"? Наверное, можно сказать, ситуация с Рефалом примерно такая же, как и с классической латынью, на которой сейчас говорит только некоторые специалисты, но не "простой народ" на базаре. С другой стороны, как известно, на базаре в Италии сейчас говорят на итальянском языке, в котором слово "вода" пишется "acqua", а не "aqua", как в латыни, звучание-то осталось то же самое.

Так же и с Рефалом! Когда Рефал только появился, он воспринимался как нечто необычное и экзотическое, ибо был основан на декларативности, функциональной форме записи программ, рекурсии и использовании образцов для распознавания ситуаций и разбора случаев. Что резко отличало Рефал от Фортрана, Алгола-60 и Кобола. Но, за прошедшее время, все эти идеи

"расползлись" по множеству языков программирования, и ныне Рефал выглядит уже как нечто "вполне естественное".

Одним из ныне здравствующих языков, близких по духу к Рефалу, является Агда [11,12]. При этом, если использовать Рефал и Агду в качестве "метаалгоритмических" языков, Агда предлагает некоторые дополнительные полезные возможности.

В следующих разделах мы рассмотрим пример "метаалгоритмического" применения Рефала и Агды и сравним возможности этих языков.

## 2. Пример применения Рефала

Данные, которые обрабатывают Рефал-программы являются *выражениями*, состоящими из *символов* и круглых скобок (в котором скобки расставлены "правильно"). А символ – это либо число, либо идентификатор (начинающийся с заглавной буквы), либо последовательность знаков, вроде +*&%?! (с некоторыми ограничениями). Символы и выражения вида ( ... ) называются *термами*. Таким образом, каждое выражение является последовательностью термов (которая может быть пустой).

Например, 1966, Add, + – символы, Add, (2 + 3) – термы, а Push 5 – выражение из двух термов: Push и 5.

Допустим, что мы хотим описать семантику арифметических выражений, определив "интерпретатор", т.е. функцию, вычисляющую значение арифметического выражения.

Простоты ради, будем считать, что каждое арифметическое выражение является либо числом, либо термом вида (t1 + t2), где t1 и t2 – арифметические выражения. Тогда интерпретатор таких выражений записывается на Рефале следующим образом:

```
<Eval sN> = sN
<Eval (t1 + t2)> = <Add <Eval t1> <Eval t2>>
```

Как это следует понимать? Во-первых, угловые скобки в Рефале изображают вызовы функций. Например, <Eval sN> – это вызов функции Eval с аргументом sN. При этом, sN – это переменная, значением которой может быть произвольный символ, а t1 и t2 – это переменные, значением которых могут быть произвольные *термы*.

Программа на Рефале представляет собой набор *предложений*. Левая часть каждого предложения описывает ситуацию, которая может возникнуть при попытке вычислить вызов функции, а правая часть – на что следует заменить вызов функции, если правило применимо. Например, выражение (1 + 2) будет вычисляться так:

```
<Eval (1 + 2)> → <Add <Eval 1> <Eval 2>> →
<Add 1 <Eval 2>> → <Add 1 2> →3
```

А теперь определим "компилятор", который превращает арифметическое выражение в "программу" состоящую из "команд" Push, Add и Seq. (Push n) помещает число n на вершину стека, Add забирает два числа с вершины стека, складывает их, и помещает результат на вершину стека, а (Seq c1 c2) изображает последовательность из двух команд c1 и c2, и означает, что нужно сначала выполнить c1, а потом – c2. (При этом c1 и c2 сами могут содержать Seq.)

```
<Compile sN> = (Push sN)
<Compile (t1 + t2)> = (Seq (Seq <Compile t1> <Compile t2>) Add)
```

А семантику "программ" можно формализовать, определив "интерпретатор", вычисляющий результат исполнения "программы" c по отношению к стеку s. При этом считаем, что пустой стек имеет вид (), непустой – вид (n s), где n – вершина стека, а s – остаток стека.

```
<Exec (Seq t1 t2) tS> = <Exec t2 <Exec t1 tS>>
<Exec (Push sN) tS = (sN tS)
<Exec Add (s2 (s1 tS)) = (<Add s1 s2> tS)
```

Таким образом, мы видим, что Рефал может использоваться для определения интерпретаторов и компиляторов других языков. Теперь посмотрим, как то же самое можно сделать с помощью языка Агда.

### 3. Пример применения Агды

Рассмотренный выше интерпретатор арифметических выражений можно переписать на Агде [11,12]. Получается следующее:

```
data Tm : Set where
  val : (n : ℕ) → Tm
  _⊕_ : (t1 t2 : Tm) → Tm

eval : Tm → ℕ
eval (val n) = n
eval (t1 ⊕t2) = eval t1 + eval t2
```

Главное отличие Агды от Рефала состоит в том, что Агда – язык со статической типизацией. Поэтому, прежде чем определять интерпретатор арифметических выражений, потребовалось определить структуру данных Tm. Заметим, что в Агде можно определять инфиксные операции, и, в данном случае, мы определили операцию _⊕_, конструирующую выражения вида t1 ⊕t2. А перед двумя предложениями, определяющими функцию eval, явно указан тип функции Tm → ℕ, который означает, что функция принимает на входе аргумент типа Tm и возвращает натуральное число.

Теперь определим на Агде структуру данных Code, изображающую программы, которые получаются в результате компиляции, а также сам компилятор.

```
data Code : (i j : ℕ) → Set where
  seq  : ∀ {i j k} (c1 : Code i j) (c2 : Code j k) → Code i k
  push : ∀ {i} (n : ℕ) → Code i (1 + i)
  add  : ∀ {i} → Code (2 + i) (1 + i)

compile : ∀ {i} (t : Tm) → Code i (1 + i)
compile (val n) = push n
compile (t1 ⊕t2) = seq (seq (compile t1) (compile t2)) add
```

Здесь мы сталкиваемся с понятием "зависимых типов", ибо тип Code имеет два целых неотрицательных параметра i и j. То есть мы имеем дело не с одним типом, а с целым семейством типов. При этом, если кусок кода c имеет тип Code i j, это означает, что исполнение кода c применительно к стеку s, имеющему глубину i, порождает стек глубины j. Заметим, что i, j и k заключены в data Code в фигурные скобки, что означает, что эти параметры являются "неявными", и их не требуется выписывать в явном виде при использовании seq, push и add, ибо компилятор неявно проставляет значения этих параметров в нужных местах.

У функции exec тоже есть неявный параметр i, и эта функция возвращает результат типа Code i (1 + i). Действительно, если задан стек глубины i, то результат вычисления арифметического выражения помещается на вершину стека, и получается стек глубины 1 + i.

И, наконец, теперь мы можем определить интерпретатор, который получает программу типа Code i j и стек глубины j, и выдает стек глубины j.

```
Stack : ℕ → Set; Stack i = Vec ℕ i

exec : ∀ {i j} (c : Code i j) (s : Stack i) → Stack j
exec (seq c1 c2) s = exec c2 (exec c1 s)
exec (push n) s = n ∷ s
exec add (n2 ∷ n1 ∷ s) = (n1 + n2) ∷ s
```

При этом Stack i представляется вектором длины i. Таким образом, пустой стек – это [], а непустой – выглядит как n ∷ s, где n – число на вершине стека, а s – остаток стека.

## 4. Утверждения и доказательства

И здесь мы подходим к самому интересному месту! Оказывается, что средствами Агды можно не только формально записать интерпретатор, компилятор и интерпретатор кода, но можно сделать больше: сформулировать некоторые утверждения об этих функциях и даже записать доказательства этих утверждений (и компилятор Агды проверит правильность этих доказательств).

Например, корректность compile и exec по отношению к eval означает, что если задано выражение t и стек s, то вычисление exec (compile t) s помещает eval t на вершину стека. Формулировка и доказательство соответствующей теоремы на Агде выглядит следующим образом.

```
correct : ∀ {i} (t : Tm) (s : Stack i) →
  exec {i} (compile t) s ≡ eval t :: s
correct (val n) s = refl
correct (t1 ⊕t2) s =
  exec (compile (t1 ⊕t2)) s
    ≡⟨ ⟩
  exec (seq (seq c1 c2) add) s
    ≡⟨ ⟩
  exec add (exec c2 (exec c1 s))
    ≡⟨  cong (exec add ∘ exec c2) (correct t1 s) ⟩
  exec add (exec c2 (n1 :: s))
    ≡⟨  cong (exec add) (correct t2 (n1 :: s)) ⟩
  exec add (n2 :: n1 :: s)
    ≡⟨ ⟩
  n1 + n2 :: s
    ≡⟨ ⟩
  eval (t1 ⊕t2) :: s ■
  where
  n1 = eval t1; n2 = eval t2;
  c1 = compile t1; c2 = compile t2
```

На первый взгляд, это "доказательство" выглядит весьма загадочно. По внешнему виду – это определение некоторой функции correct, а совсем не то, что мы привыкли видеть в качестве доказательств в учебниках математики. Как же так?

Но, в действительности, никакого противоречия тут нет. Да, correct – это функция. Но, одновременно, это и доказательство! Один из принципов, на которых построен язык Агда – *соответствие Карри-Ховарда* [13], которое заключается в том, что всякий тип τ – это некоторое "утверждение", а любое значение v, имеющее этот тип, является "доказательством" утверждения τ.

В частности, если имеется функциональный тип вида σ → τ, считается, что его следует истолковывать как утверждение "из σ следует τ". А соответствующее доказательство – это функция f, которая берет некое a типа σ и вырабатывает результат f a типа τ. Или, другими словами, f, получив на входе любое доказательство факта σ умеет превращать его в доказательство факта τ. Но это ведь как раз и соответствует интуитивному пониманию того, что "из σ следует τ".

Итак, мы видим определение типа функции correct (формулировку утверждения теоремы) и определение этой функции (доказательство).

Присмотревшись к определению функции correct, мы видим, что оно сотоит из двух предложений. Первое предложение рассматривает случай, когда t имеет вид val n. В этом случае, exec (compile (val n)) s тривиально (по-определению exec и compile) сводится к n :: s. Таким образом, получается, что нужно доказать n :: s ≡ n :: s, а доказательством этого является refl, в правой части предложения, означающее: "верно в силу рефлексивности равенства".

А второе предложение рассматривает случай, когда t имеет вид t1 ⊕ t2, и его правая часть – гораздо сложнее, чем у первого предложения. При этом, в

правой части correct дважды вызывает саму себя. Если correct рассматривать как функцию, это означает, что correct определена рекурсивно, а если рассматривать её определение как доказательство, это означает, что доказательство проводится индукцией по структуре арифметического выражения. При этом, вызвав correct t1 s, получаем доказательство факта exec c1 s ≡ n1 :: s, а вызвав correct t2 (n1 :: s), получаем доказательство факта exec (compile t2) (n1 :: s) ≡ n2 :: n1 :: s. Ну, а всё остальное в доказательстве – это "клей", который скрепляет эти факты, чтобы получить из них утверждение теоремы.

Более сложный и интересный пример использования Агды можно найти в статье [14], в которой показывается эквивалентность нескольких подходов к суперкомпиляции. Сначала процесс суперкомпиляции описывается в виде "отношения суперкомпиляции", которое определяет, что считается правильным результатом суперкомпиляции при заданных начальных условиях. Затем определяются два *алгоритма*, порождающих множество возможных результатов суперкомпиляции и доказывается, что они эквивалентны и корректны с точки зрения отношения суперкомпиляции.

**Заключение**

По большому счету, можно утверждать, что идеи, на которых был основан Рефал (декларативность, функциональность, рекурсия и сопоставление с образцом) оказались полезными и плодотворными и, в настоящее время их можно обнаружить (в той или другой степени) и в других языках.

Язык Агда основан на тех же самых принципах, что и Рефал, но предоставляет ещё и возможность записывать утверждения о программах и доказательства этих утверждений. При этом проверка правильности доказательств выполняется автоматически.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 16-01-00813-а, и госзадания ФАНО России, регистрационный номер НИР: АААА-17-117040610375-5.

**Литература**

1. Турчин В.Ф. Метаязык для формального описания алгоритмических языков // В сборнике: Цифровая вычислительная техника и программирование. М.: Советское радио, 1966, с. 116–119.
2. Турчин В.Ф. Метаалгоритмический язык // Кибернетика № 4, 1968, с. 116−124.
3. Турчин В.Ф. Транслятор с АЛГОЛа, написанный на языке РЕФАЛ // В сб.: Труды 1-ой всесоюзной конференции по программированию. В. Процессоры с известных языков. – Киев: 1968, с. 134-151.

4. Турчин В.Ф., Сердобольский В.И. Язык Рефал и его использование для преобразования алгебраических выражений // Кибернетика № 3, 1969, с. 58−62.
5. Задыхайло И.Б., Котов Е.И., Мямлин А.Н., Поздняков Л.А., Смирнов В.К. Вычислительная система с внутренним языком повышенного уровня // М.: ИПМ АН СССР, 1975, препринт № 41. – 42 с.
6. Арсентьева Н.Г., Янова Э.К. Опыт программирования одной лингвистической задачи на языке РЕФАЛ // М.: ИПМ АН СССР, 1976, препринт № 113. – 37 с.
7. Наумов Н.А., Рубин А.Г., Смирнов В.К. Об одном способе реализации входных языков для символьного процессора // М.: ИПМ им.М.В.Келдыша АН СССР, 1981, препринт № 146. – 27 с.
8. Турчин В.Ф. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ // В сб.: Труды симпозиума "Теория языков и методы построения систем программирования". – Киев-Алушта: 1972, с. 31-42.
9. Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛе // В сб.: Автоматизированная система управления строительством. Труды ЦНИПИАСС, N 6. – Москва: ЦНИПИАСС, 1974, с. 36-68.
10. Турчин В.Ф. и др. Базисный Рефал и его реализация на вычислительных машинах // М.: ЦНИПИАСС, 1977, с. 92-95.
11. Norell U. Towards a practical programming language based on dependent type theory. Ph.D. thesis // Chalmers University of Technology and Göteborg University, 2007.
12. The Agda wiki. URL: http://wiki.portal.chalmers.se/agda/.
13. Пирс Б. Типы в языках программирования // Добросвет, 2012. – 680 с. – ISBN 978-5-7913-0082-9.
14. Grechanik S., Klyuchnikov I., Sergei Romanenko S. Staged Multi-Result Supercompilation: Filtering by Transformation. In *Fourth International Valentin Turchin Workshop on Metacomputation (Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, June 29 - July 3, 2014)*. A.V. Klimov and S.A. Romanenko, Ed. // Pereslavl-Zalessky: Publishing House "University of Pereslavl", 2014. – 256 с. – ISBN 978-5-901795-31-6, pages 54-78.