Реализация Рефал-компилятора. Представление данных и язык сборки

С.Ю. Скоробогатов

10 июля 2008 г.

Аннотация

В статье рассматриваются основные проблемы представления данных в виртуальной Рефал-машине, вырабатывается новое векторно-списковое представление объектных выражений, пригодное для хранения поля зрения и допускающее динамическое распараллеливание программ, и даётся описание языка сборки.

Содержание

1 Введение **4**

2 Списки и проблема копирования **5**

3 Векторы и проблема конкатенации **10**

4 Новое представление **14**
4.1 Векторная основа 14
4.2 Списковая надстройка 16
4.3 Состояния функций 17
4.4 Сеть конкретизаций 20

A Структуры данных **25**

B Набор инструкций **31**
B.1 Инструкции для организации вычислений 31
B.1.1 const-int 31
B.1.2 const-ptr 31
B.1.3 load 31
B.1.4 get 32
B.1.5 store 32
B.1.6 calc 32
B.1.7 duplicate 32
B.1.8 eliminate 33
B.2 Инструкции передачи управления 33
B.2.1 jump 33
B.2.2 branch 33
B.3 Инструкции потребления объектных выражений 33
B.3.1 failed 33
B.3.2 pop 34
B.3.3 dissolve 34
B.4 Инструкции сопоставления с образцом 34
B.4.1 compare 34
B.4.2 check 35
B.4.3 descend 35
B.5 Инструкции порождения замыканий 35
B.5.1 allocate 35
B.5.2 put 36
B.5.3 done 36
B.6 Инструкции создания объектных выражений 36

B.6.1 push-range . . . 36
B.6.2 push-empty . . . 37
B.6.3 push-call . . . 37
B.6.4 depend . . . 38
B.6.5 depend-this . . . 38
B.6.6 copy-dep . . . 38
B.6.7 transfer-dep . . . 39
B.6.8 link . . . 39
B.6.9 cut . . . 40
B.6.10 bracket . . . 40
B.6.11 concat . . . 41
B.7 Инструкции для взаимодействия с системой выполнения . 41
B.7.1 wait . . . 41
B.7.2 ret . . . 41
B.7.3 retf . . . 41
B.7.4 crash . . . 42

Список иллюстраций

1 Классическое представление объектного выражения. 5
2 Конечный автомат, передаваемый в функцию FA. 6
3 Компромиссное представление объектного выражения. . . . 8
4 Конкатенация с неготовым значением. 10
5 Компромиссное представление без головных элементов. . . 10
6 Векторное представление объектного выражения. 11
7 Хранение значений переменных в окружении. 15
8 Передача параметров через стек объектных выражений. . . 19
9 Организация поля зрения в классическом представлении. . 20
10 Схема поля зрения. 22

Список таблиц

1 Система зависимостей по данным. 23
2 Система зависимостей по побочным эффектам. 23
3 Полная система зависимостей. 24
4 Базовые структуры данных. 26
5 Структуры-перечисления. 27
6 Векторные структуры. 27
7 Структуры-записи. 28

1 Введение

Для разработчика любого Рефал-компилятора рано или поздно заканчивается период создания собственного диалекта, лексера, парсера, семантического анализатора и абстрактного синтаксиса. Наступает пора для разработки промежуточного языка, в который будут переводиться Рефал-программы. Такой язык традиционно называется *языком сборки* и, как правило, представляет собой систему команд абстрактной Рефал-машины.

Объектные выражения, с которыми работают Рефал-программы, весьма далеки от тех типов данных, на обработку которых ориентированы современные аппаратные средства. Поэтому приходится придумывать способ отображения объектных выражений в структуры, которые могут быть размещены в памяти ЭВМ и обработаны центральным процессором. Сорок лет истории развития языка Рефал показывают, что таких способов может быть множество, и что выбор способа представления объектных выражений оказывает заметное влияние на язык сборки.

В этой статье мы попытаемся разработать новый способ представления объектных выражений в памяти и систему команд языка сборки. Для того чтобы не отвлекаться на малозначащие детали, мы будем считать, что объектные выражения состоят из структурных скобок и символов неизвестной природы. Предполагается, что такое абстрагирование поможет сделать представление данных и язык сборки независимыми от диалекта.

В примерах программ на Рефале мы будем обозначать символы, из которых состоят объектные выражения, греческими буквами. Кроме того, мы будем записывать индексы переменных подстрочным шрифтом.

При описании структур данных и инструкций языка сборки мы будем давать им имена, составленные из латинских букв.

Мы будем обозначать объектное выражение литерой $\mathcal{E}$, объектный терм – литерой $\mathcal{T}$, а объектный терм, представляющий одиночный символ, – литерой $\mathcal{S}$. При этом пустое объектное выражение мы будем записывать как $\diamondsuit$.

Если дано объектное выражение $\mathcal{E}{=}\,\mathcal{T}_1\,\mathcal{T}_2\,\ldots\,\mathcal{T}_{\mathrm{n}}$, то число $|\mathcal{E}| = n$ мы будем называть *длиной* выражения $\mathcal{E}$. При этом *размером* выражения $\mathcal{E}$ мы будем называть количество символов и скобок, из которых оно состоит.

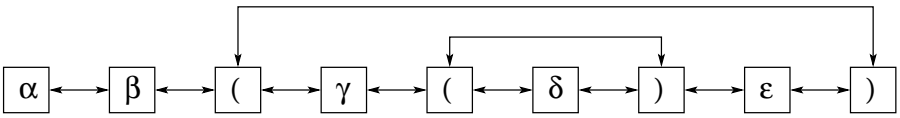

Рис. 1: Классическое представление объектного выражения.

2 Списки и проблема копирования

До середины 80-х годов среди реализаций языка Рефал доминировал *классический* способ представления объектного выражения в виде двусвязного списка. Рис. 1 демонстрирует классическое представление на примере выражения $\alpha\,\beta\,(\gamma\,(\delta)\,\varepsilon)$.

В классическом представлении каждое звено списка соответствует либо символу, либо структурной скобке и представляет собой структуру, состоящую из четырёх полей:

prev и next содержат указатели на предыдущее и следующее звенья, соответственно. Конкатенация двух непустых выражений A и B благодаря наличию этих полей сводится к изменению двух указателей, а именно: в поле next последнего звена выражения A записывается адрес первого звена выражения B, а в поле prev первого звена выражения B помещается адрес последнего звена выражения A.

tag задаёт тип звена, то есть показывает, соответствует ли звено открывающей скобке, закрывающей скобке или одному из видов символов.

info содержит информацию, смысл которой определяется значением поля tag. В случае, когда звено обозначает структурную скобку, в поле info записывается адрес парной структурной скобки. Если же звено соответствует символу, поле info некоторым образом кодирует его значение.

Хранение выражения в виде списка порождает известную *проблему копирования*, которая выражается в том, что более эффективно выполняются Рефал-программы, написанные в императивном, а не в функциональном стиле.

Чтобы проиллюстрировать проблему копирования, напишем на Рефале несложную функцию FA, выполняющую интерпретацию недетерминированного конечного автомата. Эта функция будет получать на

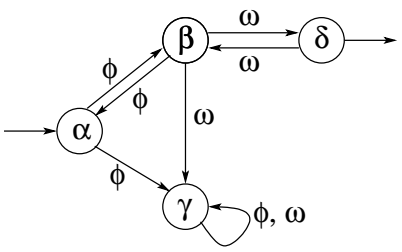

Рис. 2: Конечный автомат, передаваемый в функцию FA.

вход описание конечного автомата, входное состояние и цепочку символов, а возвращать список выходных состояний, достигнутых в результате распознавания цепочки. Этот список может оказаться пустым, если цепочка не распознаётся данным конечным автоматом. Детали представления конечного автомата в виде объектного выражения можно понять из вызова функции FA:

```
$ENTRY Go {
    = ⟨Prout
        ⟨FA
          /* Конечный автомат (см. рис. 2) */
          (
            ( α (ϕ β) (ϕ γ) )
            ( β (ϕ α) (ω γ) (ω δ) )
            ( γ (ϕ γ) (ω γ) )
            ( δ ( ) (ω β) ) /* Выходное состояние */
          )
          α /* Входное состояние */
          ϕ ϕ ϕ ω ω ω ω ω /* Распознаваемая цепочка */
        ⟩
      ⟩
    };
```

Отметим, что пустая пара скобок в описании состояния δ говорит о том, что это состояние – выходное.

Недетерминированный конечный автомат допускает переход из одного состояния по одному и тому же символу сразу в несколько других состояний. Поэтому алгоритм интерпретации такого автомата должен так или иначе ветвиться. В первом варианте программы, написанном в функциональном стиле, это ветвление осуществляется в первом предложении функции FA2:

```
FA {
    t_T s_S , t_T : (e_L (s_S ( ) e_V) e_R) = s_S ;
    t_T  s_S  e_P , t_T : (e_L (s_S e_V) e_R) = ⟨FA2 t_T (e_V) e_P⟩ ;
    }

FA2 {
    t_T (e_L (s_X s_Q) e_R) s_X e_P = ⟨FA t_T s_Q e_P⟩ ⟨FA2 t_T (e_R) s_X e_P⟩ ;
    e_1 = ;
    }
```

Можно заметить, что при осуществлении ветвления значения переменных t_T и e_P передаются как в функцию FA, так и в функцию FA2, что приводит к созданию копий этих значений.

Второй вариант программы написан в императивном стиле:

```
FA { t_T s_O t_X e_P = ⟨FA2 ( ) t_T ( ) (s_O) t_X e_P ( )⟩ ; }

FA2 {
    (e_A) ((s_S ( ) e_V) e_B) (e_Q) (e_1 s_S e_2) ( ) =
        ⟨FA2 (e_A (s_S ( ) e_V)) (e_B) (e_Q s_S) (e_1 e_2) ( )⟩ ;
    (e_A) ((s_S e_V) e_B) t_Q (e_1 s_S e_2) e_P =
        ⟨FA3 (e_A) (s_S) (e_V) (e_B) t_Q (e_1 e_2) e_P⟩ ;
    (e_A) (t_1 e_B) e_2 = ⟨FA2 (e_A t_1) (e_B) e_2⟩ ;
    t_T ( ) (e_Q) t_O ( ) = e_Q ;
    t_T ( ) t_Q ( ) t_X e_P = ⟨FA2 ( ) t_T ( ) t_Q e_P⟩ ;
    }

FA3 {
    t_A (e_S) (e_L (t_X s_W) e_R) t_B (e_Q) t_O t_X e_P =
        ⟨FA3 t_A (e_S e_L (t_X s_W)) (e_R) t_B (e_Q s_W) t_O t_X e_P⟩ ;
    t_A (e_S) (e_V) (e_B) t_Q t_O t_X e_P =
        ⟨FA2 t_A ((e_S e_V) e_B) t_Q t_O t_X e_P⟩ ;
    }
```

Несмотря на ужасный вид, второй вариант программы обладает важной особенностью: количество вхождений переменной в правую часть предложения всегда меньше или равно количеству вхождений этой переменной в левую часть предложения. Эта особенность позволяет при выполнении программы избежать копирования выражений, ограничившись только их конкатенацией. Учитывая, что конкатенация выражений выполняется за константное время, а время копирования выражений линейно зависит от их размера, можно придти к выводу, что второй вариант программы эффективнее первого варианта.

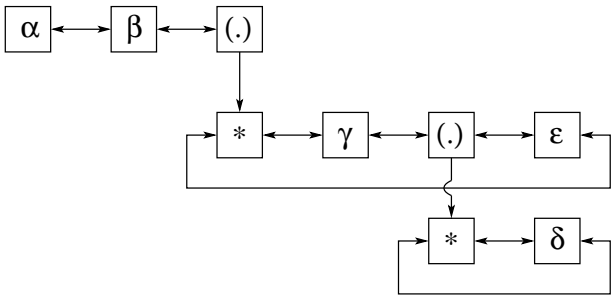

Рис. 3: Компромиссное представление объектного выражения.

Для того чтобы снизить остроту проблемы копирования, можно использовать так называемый *компромиссный* вариант спискового представления объектных выражений. В этом варианте каждому терму верхнего уровня отводится ровно одно звено. Если некоторый терм представляет собой выражение в структурных скобках, то в поле info звена, соответствующего этому терму, записывается ссылка на *головное звено* другого списка. В головном звене хранятся адреса первого и последнего звеньев выражения, содержащегося внутри структурных скобок, и счётчик ссылок. На рис. 3 показано выражение $\alpha\,\beta\,(\gamma\,(\delta)\,\varepsilon)$ в компромиссном представлении, причём головные звенья помечены звёздочками.

Копирование выражения в компромиссном представлении сводится к копированию термов верхнего уровня и увеличению счётчиков в головных звеньях, соответствующих подвыражениям в скобках. Тем самым время копирования оказывается прямо пропорционально длине, а не размеру выражения.

Если мы попытаемся сформулировать основное свойство спискового представления объектных выражений, обусловливающее проблему копирования, у нас получится следующее утверждение: *так как объектное выражение, представленное в виде списка, в процессе выполнения программы может подвергаться изменениям, то в случае размножения выражения мы не можем ограничиться размножением ссылок на список, но вместо этого во избежание возможных побочных эффектов должны либо сразу создавать полную копию списка, либо использовать какую-нибудь технологию наподобие подсчёта ссылок для отслеживания необходимости создания полной копии.* Это утверждение представляет собой фундаментальное свойство изменяемых данных и поэтому справедливо как для классического, так и для компромиссного представления объектных выражений.

Проблема копирования – наиболее критикуемая особенность спискового представления, потому что борьба с копированием серьёзно ухудша-

ет стиль Рефал-программ. Однако, наряду с этим недостатком, списковое представление имеет важные достоинства.

Во-первых, списковое представление оптимально для конкатенации выражений: конкатенация выполняется за время, не зависящее от размера конкатенируемых выражений.

Во-вторых, если разрешить хранение в списке скобок конкретизации, то становится возможным использование списка для представления поля зрения Рефал-программы. При этом подразумевается, что во время выполнения программы из поля зрения добываются аргументы вызываемых функций, а возвращаемые функциями значения подставляются в поле зрения. Хранение поля зрения в списке позволяет обойтись без стандартного стека вызовов функций, который имеет тенденцию быстро переполняться, обеспечивает элегантную реализацию хвостовых вызовов и, кроме того, превращает часть вызовов в хвостовые за счёт поддержки *конкатенации с неготовым значением*.

Поясним на примере, что же такое конкатенация с неготовым значением. Для этого рассмотрим функцию Rev, переставляющую термы, из которых состоит выражение, в обратном порядке:

```
Rev {
    t_X e_1 = ⟨Rev e_1⟩ t_X;
    = ;
    }
```

На рис. 4 показано вычисление выражения

```
⟨Prout ⟨Rev α β⟩⟩.
```

Строго говоря, вызов функции Rev в примере не является хвостовым, потому что возвращаемое ею значение должно быть конкатенировано со значением переменной t_X. Однако благодаря тому, что поле зрения хранится в виде списковой структуры, мы можем скобки конкретизации, обозначающие вызов функции, соединить со значением t_X. Тем самым мы выполняем конкатенацию заранее, ещё до вызова функции, и фактически делаем вызов функции хвостовым.

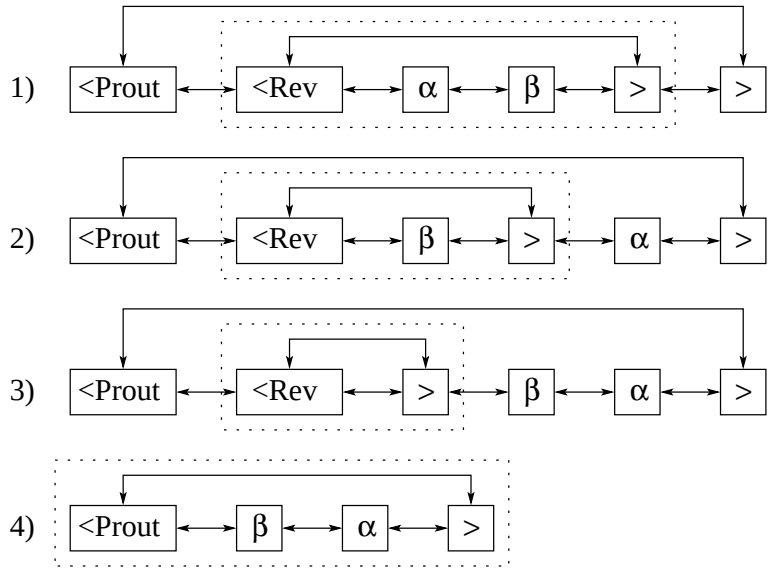

Рис. 4: Конкатенация с неготовым значением.

3 Векторы и проблема конкатенации

Рассмотрим дальнейшее развитие компромиссного представления объектных выражений.

Сначала уберём из компромиссного представления головные звенья. Это элементарно делается, если в поле info звена, соответствующего подвыражению в скобках, вместо адреса головного звена разместить ссылки на начало и конец этого подвыражения. В качестве примера на рис. 5 показано выражение $\alpha\,\beta\,(\gamma\,(\delta)\,\varepsilon)$ в компромиссном представлении, лишённом головных звеньев.

Затем избавимся от полей prev и next. Для этого будем хранить звенья в векторе в том порядке, в котором соответствующие им термы входят в выражение.

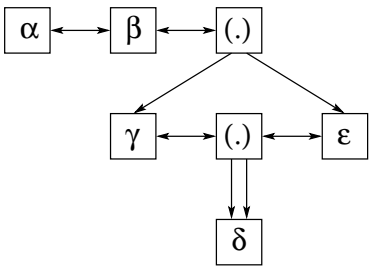

Рис. 5: Компромиссное представление без головных элементов.

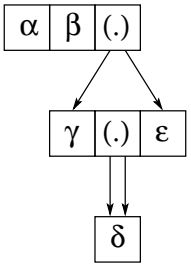

Рис. 6: Векторное представление объектного выражения.

И, наконец, компенсируем потерю счётчиков ссылок, которые были в головных звеньях, запретив изменять векторы звеньев. Тем самым данные, с которыми мы работаем, становятся неизменяемыми, и мы можем в дальнейшем не бояться побочных эффектов.

Получившееся в результате представление называется *векторным* *. Рис. 6 демонстрирует векторное представление на примере выражения $\alpha\,\beta\ (\gamma\ (\delta)\ \varepsilon)$.

Векторное представление значительно лучше решает проблему копирования, нежели компромиссное представление, потому что позволяет свободное выделение частей выражений и не требует заботы о счётчиках ссылок. Например, в предложении

$$e_1\,\alpha\ \texttt{=}\ \texttt{⟨F}\ e_1\texttt{⟩}\ \texttt{⟨G}\ e_1\texttt{⟩;}$$

компромиссное представление не позволяет обойтись без копирования значения e_1, потому что требует, чтобы термы верхнего уровня копировались всегда. Векторное представление свободно от этого ограничения.

Более сложный пример, иллюстрирующий превосходство векторного представления над компромиссным, даёт предложение

$$\alpha\ (e_1\,\beta\,e_2\,\gamma\,e_3)\ \texttt{=}\ (e_1\,\beta\,e_2)\ (e_2\,\gamma\,e_3)\texttt{;}$$

Компромиссное представление разрешает не копировать выражение в скобках, если оно взято целиком. Если же нам требуются перекрывающиеся части выражения в скобках, то компромиссное представление с задачей уже не справляется, в то время как для векторного представления это не представляет никаких трудностей.

К сожалению, за эффективное копирование выражений приходится расплачиваться неэффективной конкатенацией. Тем самым на смену

*Векторное представление было впервые предложено в [2].

проблеме копирования в векторном представлении приходит *проблема конкатенации*.

В общем случае для конкатенации выражений $\mathcal{E}_1$, $\mathcal{E}_2 \ldots \mathcal{E}_\mathrm{n}$ в векторном представлении требуется создать вектор размера $|\mathcal{E}_1|+|\mathcal{E}_2|+\ldots+|\mathcal{E}_\mathrm{n}|$ и скопировать в него термы верхнего уровня конкатенируемых выражений. Тем самым время конкатенации прямо пропорционально сумме длин конкатенируемых выражений.

Мы говорим о конкатенации не двух, а сразу нескольких выражений, потому что конкатенация нескольких выражений значительно выгоднее. Действительно, если бы операция конкатенации имела только два операнда, то для соединения выражений $\mathcal{E}_1$, $\mathcal{E}_2 \ldots \mathcal{E}_\mathrm{n}$ нам потребовалось бы $(n-1)$ конкатенаций. В результате первой конкатенации получилось бы временное выражение $\mathcal{E}'$ длиной $|\mathcal{E}_1| + |\mathcal{E}_2|$ термов, в результате второй конкатенации – выражение $\mathcal{E}''$ длиной $|\mathcal{E}'|+|\mathcal{E}_3|$ термов, в результате третьей – выражение $\mathcal{E}^{(3)}$ длиной $|\mathcal{E}''|+|\mathcal{E}_4|$ термов и т.д. Нетрудно посчитать, что суммарная длина временных выражений составляет

$$(n-2)\,(|\mathcal{E}_1| + |\mathcal{E}_2|) + (n-3)\,|\mathcal{E}_3| + \ldots + 2\,|\mathcal{E}_{\mathrm{n}-2}| + |\mathcal{E}_{\mathrm{n}-1}|.$$

Когда не удаётся избежать создания временных выражений, говорят о проявлении эффекта *преждевременной конкатенации*. Мы продемонстрируем этот эффект на примере функции Alpha, заменяющей каждый терм выражения на символ α:

```
Alpha {
    t_X e_1 = α ⟨Alpha e_1⟩;
    = ;
    }
```

Передадим в функцию Alpha выражение, состоящее из пяти термов:

```
⟨Alpha β γ δ ε ζ⟩
```

Это приведёт к выполнению каскада из пяти конкатенаций с образованием четырёх временных выражений $\mathcal{E}'$, $\mathcal{E}''$, $\mathcal{E}^{(3)}$ и $\mathcal{E}^{(4)}$, суммарная длина которых составляет десять термов, и результирующего выражения $\mathcal{E}$:

$$\begin{aligned}
\mathcal{E}' &\leftarrow \alpha \diamondsuit \\
\mathcal{E}'' &\leftarrow \alpha\, \mathcal{E}' \\
\mathcal{E}^{(3)} &\leftarrow \alpha\, \mathcal{E}'' \\
\mathcal{E}^{(4)} &\leftarrow \alpha\, \mathcal{E}^{(3)} \\
\mathcal{E} &\leftarrow \alpha\, \mathcal{E}^{(4)}
\end{aligned}$$

Нетрудно заметить, что начиная со второй конкатенации происходит копирование временного выражения, полученного в результате предыдущей конкатенации, и приписывание к нему слева символа α. То есть с каждым разом очередное копируемое выражение становится больше на один символ, что делает сложность линейного по сути алгоритма, реализованного в функции Alpha, квадратичной.

Для амортизации эффекта преждевременной конкатенации можно выделять память для вектора с запасом, чтобы слева и справа вектор был обрамлён неиспользованными элементами – «полями». В этом случае появляется шанс, что приписываемое к вектору слева или справа выражение целиком поместится в «поле», избавив нас от необходимости копировать вектор на новое место.

Ещё одно досадное проявление проблемы конкатенации связано с передачей параметров. Как известно, в Рефале любая функция принимает ровно один параметр. Если по логике программы необходимо передать в некоторую функцию несколько объектных выражений $\mathcal{E}_1$, $\mathcal{E}_2 \ldots \mathcal{E}_\mathrm{n}$, то рекомендуется упаковать их в одно выражение $(\mathcal{E}_1)\,(\mathcal{E}_2)\ldots(\mathcal{E}_n)$ путём оборачивания в скобки и конкатенации. Однако в реальных программах эта рекомендация соблюдается далеко не всегда, и для упаковки параметров в одно выражение используются другие схемы. Например, два параметра $\mathcal{S}_1$ и $\mathcal{E}_2$ можно не оборачивать в скобки, а просто соединить в выражение $\mathcal{S}_1\,\mathcal{E}_2$, и в классическом списковом представлении с дешёвой конкатенацией эта схема отлично работает, в то время как для векторного представления она оказывается неприемлемо затратной.

Известно два пути решения проблемы передачи параметров. Во-первых, можно полностью переложить её на пользователя, заставив его правильно оборачивать параметры в скобки, а то и указывать «формат» вызова функции (путь, по которому пошли разработчики диалекта Рефал+). Во-вторых, можно встроить в компилятор так называемый *повышатель арности* – алгоритм, выполняющий анализ программы и увеличивающий количество параметров [4].

Заметим, что всё сказанное про параметры одинаково справедливо и для возвращаемых функциями значений, только немного менее актуально.

4 Новое представление

Попробуем сформулировать список требований к способу представления объектных выражений, которое нам предстоит разработать (назовём этот способ *новым* представлением). Расположим требования в том порядке, в котором нам будет удобно в дальнейшем искать пути их удовлетворения:

1. Новое представление должно решать проблему копирования хотя бы не хуже, чем векторное представление.

2. Новое представление должно уметь работать с замыканиями, которые нужны для поддержки функций высшего порядка [6].
   Замыкание является совокупностью адреса функции и значений некоторых её локальных переменных.

3. Новое представление должно решать проблему преждевременной конкатенации.

4. Новое представление должно подходить для хранения поля зрения, а также поддерживать конкатенацию с неготовым значением.

5. Новое представление должно поддерживать функции с несколькими параметрами.

6. Новое представление должно допускать параллельную обработку.

Мы договоримся, что структурами данных, из которых будет формироваться новое представление, станут записи, состоящие из полей, и векторы, состоящие из элементов. При этом в состав записей и векторов смогут входить указатели на другие записи и векторы, что позволит нам формировать из них произвольные списки и деревья.

4.1 Векторная основа

Согласно требованию №1, проблема копирования в новом представлении должна решаться не хуже, чем в векторном представлении. Проще всего этого можно достичь, если использовать векторное представление в качестве отправной точки для разработки нового представления.

Итак, пусть объектное выражение в новом представлении будет храниться в виде неизменяемого вектора. Мы будем называть элементы такого вектора V-*термами*. V-термы соответствуют объектным термам, из

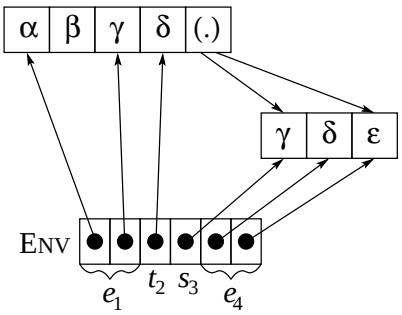

Рис. 7: Хранение значений переменных в окружении.

которых состоит выражение, и расположены в векторе в том же порядке, в каком объектные термы входят в выражение.

Связь переменных и их значений, необходимая для поддержки замыканий (в силу требования №2), будет обеспечиваться так называемым *окружением*. Пусть

Env – структура данных, задающая окружение и представляющая собой вектор указателей на v-термы.

Для хранения в структуре Env значений переменных, имеющих тип s или t, достаточно одного указателя на v-терм. Что касается значений e-переменных и v-переменных, то их можно хранить в виде пары указателей, задающих фрагмент вектора v-термов. На рис. 7 показано, как организовано хранение в окружении Env значений переменных, полученных в результате сопоставления выражения $\alpha\,\beta\,\gamma\,\delta\ (\gamma\,\delta\,\varepsilon)$ с образцом $e_1\,t_2\ (s_3\,e_4)$.

Для удобства дальнейшего изложения имеет смысл определить вспомогательную структуру данных, задающую фрагмент вектора. Пусть

Range – запись, содержащая пару указателей на v-термы.

Модель данных Рефала предполагает существование объектных термов разных типов. В связи с этим нам придётся различать три типа v-термов:

Vs – v-терм, содержащий информацию о символе.
В начале статьи мы договорились не обсуждать виды символов, допускаемые существующими диалектами языка Рефал. Тем самым детали хранения информации внутри v-терма Vs для нас несущественны.

Vc – v-терм, соответствующий замыканию.
Этот v-терм содержит адрес инструкции языка сборки, являющейся точкой входа в функцию, а также указатель на окружение Env, содержащее значения локальных переменных. Кроме того, в случае, если функция – не анонимная, в v-терме Vc хранится указатель на v-терм Vs, содержащий имя функции.

Vb – v-терм, обозначающий подвыражение в скобках.
Внутри этого v-терма хранится структура Range, задающая подвыражение.

Подчеркнём, что векторы v-термов обладают всем комплексом достоинств и недостатков, присущих векторному представлению. Кроме того, особо отметим, что как бы мы не достраивали в дальнейшем новое представление, мы постараемся обеспечить, чтобы значения переменных оставались векторами.

4.2 Списковая надстройка

Требование №3 можно выполнить, если ввести в представление механизм *отложенной конкатенации*. Смысл отложенной конкатенации заключается в том, что объединение выражений откладывается до тех пор, пока результат объединения не понадобится для сопоставления с образцом. Поэтому если выражение формируется в результате объединения нескольких подвыражений, то физически все подвыражения объединяются в один вектор за одну операцию конкатенации, даже если «заявки» на объединение подвыражений поступали на разных шагах работы Рефал-машины.

Для реализации отложенной конкатенации достаточно формировать двусвязный список конкатенируемых выражений. Звенья такого списка мы будем называть l-*термами*. Тем самым новое представление приобретает черты спискового представления. Однако, если в списковом представлении звенья содержат символы, то в новом представлении l-термы должны содержать фрагменты векторов v-термов.

Пусть

Lr – l-терм, содержащий структуру Range, которая задаёт фрагмент вектора.

Определим вспомогательную структуру данных, отражающую результат отложенной конкатенации. Пусть

Chain – запись, содержащая указатели на первый и последний L-термы списка, полученного в результате конкатенации нескольких выражений.

Так как конкатенация может сопровождаться оборачиванием выражений в структурные скобки, одних только L-термов Lr оказывается недостаточно. Поэтому пусть

Lb – L-терм, содержащий структуру Chain, которая задаёт выражение в структурных скобках.

Процесс формирования вектора V-термов из списка L-термов мы будем называть *сборкой* выражения. Сборка должна осуществляться непосредственно перед сопоставлением с образцом. Это гарантирует, что значения, которые получат переменные образца в результате сопоставления, будут фрагментами векторов V-термов.

Интересно, что благодаря одновременному использованию V-термов и L-термов новое представление должно отлично справляться как с копированием, так и с конкатенацией выражений. Хотя неизбежной расплатой за такую универсальность является необходимость периодической сборки выражений, на многопроцессорной ЭВМ сборка выражений может осуществляться параллельно с выполнением программы, что позволяет рассматривать сборку не как неизбежное зло, а как очень ценный ресурс для распараллеливания программы.

4.3 Состояния функций

Итак, объектные выражения в новом представлении могут храниться частично в векторной, частично в списковой форме. Если мы, в соответствии с требованием №4, захотим хранить в новом представлении поле зрения Рефал-программы, нам, вроде бы, достаточно будет добавить ещё один тип L-термов для изображения скобок конкретизации подобно тому, как это сделано в классическом списковом представлении.

Но не всё так просто! Дело в том, что классическое списковое представление разрабатывалось для ранних диалектов Рефала, в которых не было таких элементов синтаксиса как *условия* (в терминологии Рефала-5 [3]) или *действия* (в терминологии Рефала-7 [6]). Условия и действия входят в предложения, из которых состоят функции. Они могут содержать вызовы других функций, и семантика языка такова, что выполнение текущей функции должно быть приостановлено до вычисления этих функций, а затем возобновлено в прежнем окружении. В реализации Рефала-5, использующей классическое представление объектных

выражений, для выполнения условия приходится порождать новое поле зрения. Так как для этого осуществляется рекурсивный вызов интерпретатора языка сборки, использование рекурсии в условиях может привести к аварийному завершению программы в связи с переполнением стека вызовов. Чтобы избежать этой неприятности, нам придётся спроектировать новое представление таким образом, чтобы состояние вычисления функции хранилось не в стеке вызовов, а прямо в поле зрения. Поэтому пусть

Lk – L-терм, содержащий *состояние* выполняемой функции.

Подчеркнём основное отличие L-терма Lk от скобок конкретизации в классическом представлении: если скобки конкретизации выражают лишь потенциальную возможность выполнения функции (т.е. обозначают «заказ» на выполнение), то L-терм Lk соответствует любой фазе выполнения функции.

Давайте разберёмся, какие данные определяют состояние функции. Прежде всего приходит на ум адрес инструкции языка сборки, достигнутой в процессе выполнения функции. При этом, если функция ещё не была запущена, такой инструкцией может считаться точка входа в функцию.

Вторым по значимости компонентом состояния функции, безусловно, являются значения локальных переменных. При этом целесообразно хранить эти значения в виде двух окружений: одно окружение берётся из замыкания и содержит значения заранее определённых переменных, а другое окружение формируется в процессе выполнения функции.

Теперь пришло время ответить на вопрос, как в новом представлении предполагается хранить параметры функций. Напомним, что в классическом представлении параметр функции содержится между скобок конкретизации. Нам такой способ не подходит, потому что, во-первых, параметров может быть несколько (согласно требованию №5), а во-вторых, L-терм Lk в общем случае обозначает уже выполняющуюся функцию, которая уже давным-давно потребила свои параметры.

Часто бывает, что некоторый вопрос является частным случаем более общей проблемы. В нашем случае вопрос передачи параметров представляет собой частный случай проблемы хранения объектных выражений, формируемых в процессе выполнения функции. Решив эту проблему, мы не только разберёмся с передачей параметров, но и получим механизм построения новых выражений через отложенную конкатенацию уже существующих выражений, а также научимся присоединять к полю зрения выражения, вычисление которых требуется для выполнения действий

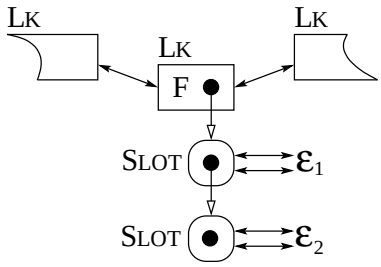

Рис. 8: Передача параметров через стек объектных выражений.

(или для вычисления условий). Самым простым решением проблемы будет добавление в состояние функции стека объектных выражений. Пусть

Slot – запись, соответствующая одному элементу стека объектных выражений.
Она содержит структуру Chain, задающую объектное выражение, а также указатель на следующий элемент стека.

Давайте обсудим, каким образом использование стека объектных выражений позволяет решить перечисленные выше задачи.

Итак, пусть нам требуется вызвать некоторую функцию F, принимающую два параметра. Для этого мы создаём новый L-терм Lk и привязываем его к нужной части поля зрения (к какой именно части, зависит от контекста). Выражения $\mathcal{E}_1$ и $\mathcal{E}_2$, которые нам нужно передать в функцию в качестве параметров, мы заворачиваем в структуры Slot и присоединяем к созданному L-терму Lk в качестве стека объектных выражений (рис. 8). Когда функция F получит управление, она сможет забрать выражения $\mathcal{E}_1$ и $\mathcal{E}_2$ со своего стека. Тем самым будет осуществлена передача параметров.

Для отложенной конкатенации двух выражений в языке сборки будет предусмотрена инструкция, потребляющая два элемента с вершины стека и оставляющая на стеке результат их конкатенации. Ещё одна инструкция будет оборачивать выражение на вершине стека в структурные скобки.

Вычисление выражения, возникающего при выполнении действия, потребует добавления этого выражения в стек. Тем самым получается, что выражения в действиях выступают в роли дополнительных параметров, которые функция получает уже в процессе выполнения.

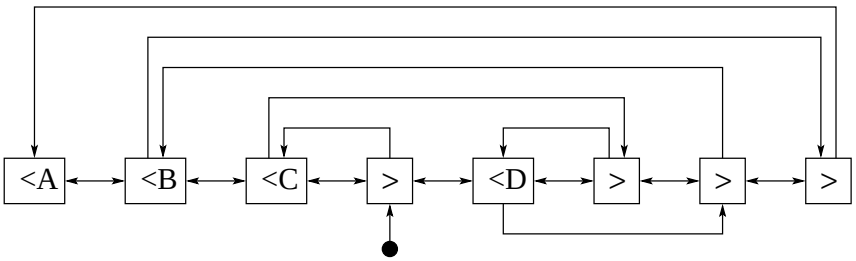

Рис. 9: Организация поля зрения в классическом представлении.

4.4 Сеть конкретизаций

Одной из основных операций, производимых с полем зрения, является выбор ведущей пары скобок конкретизации. Напомним, что *ведущая пара*, во-первых, не содержит внутри себя других скобок конкретизации, а во-вторых, она расположена в поле зрения левее всех других пар, не содержащих скобок конкретизации. На каждом шаге своей работы Рефал-машина выбирает очередную ведущую пару и вызывает соответствующую функцию.

Эффективная реализация Рефал-машины не может позволить себе поиск ведущей пары, так как время поиска прямо пропорционально размеру поля зрения. Поэтому приходится усовершенствовать представление поля зрения таким образом, чтобы выбор ведущей пары осуществлялся за константное время. Чтобы понять основную идею этого усовершенствования, обратимся к [1] и посмотрим, как с этой проблемой справились разработчики Рефала-2.

В Рефале-2 используется классическое списковое представление выражений. В этом представлении звено списка, соответствующее правой скобке конкретизации, содержит в поле info указатель на парную левую скобку конкретизации. Это даёт возможность, зная правую скобку, моментально локализовать параметр функции. Однако указатель, хранящийся в поле info левой скобки, вопреки соображениям симметрии, указывает не на парную ей правую скобку конкретизации, а на правую скобку другой пары. Так обозначается, что эта другая пара станет ведущей после того, как данная пара будет вычислена. На рис. 9 продемонстрирован принцип организации поля зрения в Рефале-2 на примере выражения

```
⟨A ⟨B ⟨C⟩ ⟨D⟩⟩⟩.
```

Можно заметить, что все скобки конкретизации в поле зрения объединены в список, задающий порядок вычисления функций. Первым звеном

этого списка является правая скобка ведущей пары. Указатель на первое звено хранится в специальной переменной, которая обозначена на рисунке кружком. Значение этой переменной в процессе изменения поля зрения постоянно корректируется таким образом, чтобы она всегда указывала на правую скобку ведущей пары.

В новом представлении вместо скобок конкретизации мы решили использовать состояния функций. Поэтому аналогом ведущей пары в нём будет *ведущее состояние*, то есть состояние, готовое к выполнению. Причём в силу требования №6 таких состояний может быть несколько.

Дадим несколько вспомогательных определений.

Мы будем называть функцию *чистой*, если её выполнение не зависит от внешнего окружения и не изменяет внешнее окружение. Чистая функция обладает полезным свойством: возвращаемое ею значение полностью определяется значениями переданных ей параметров.

Если функция не является чистой, мы будем говорить, что она имеет *побочные эффекты*. Функции, имеющие побочные эффекты, могут быть выявлены путём анализа Рефал-программы на этапе компиляции.

Чистую функцию мы будем называть *абсолютно чистой*, если может быть доказана её терминируемость. Абсолютно чистая функция не может привести к зависанию программы.

[В Рефале-7 предусмотрены так называемые *откатные* функции. Они отличаются от остальных функций тем, что не могут вызывать аварийный останов программы, возвращая в аварийных ситуациях специальное значение, обозначающее *неудачу*.]

Порядок передачи управления между состояниями функций в поле зрения, определяемый семантикой Рефала, мы будем называть *аппликативным* и обозначать знаком $\prec$. То есть запись $x \prec y$ будет означать, что состояние x по семантике Рефала должно получить управление раньше, чем состояние y. Отношение $\prec$ иррефлексивно, антисимметрично и транзитивно, то есть является отношением строгого порядка.

Определим отношение строгого линейного порядка $\prec$ на множестве состояний поля зрения. Для этого приведём алгоритм обхода поля зрения в глубину, формирующий последовательность состояний $l_1 \prec l_2 \prec \ldots \prec l_n$.

Чтобы научиться быстро выбирать ведущие состояния в поле зрения, необходимо понять, чем ведущие состояния как состояния, готовые к выполнению, отличаются от всех прочих состояний, к выполнению не готовых. Нам будет удобно разобраться в этом вопросе на примере. Поэтому пусть в поле зрения лежит выражение

```
⟨G ⟨F ⟨I⟩ ⟨J⟩⟩⟩ ⟨A ⟨B⟩ ⟨C⟩⟩.
```

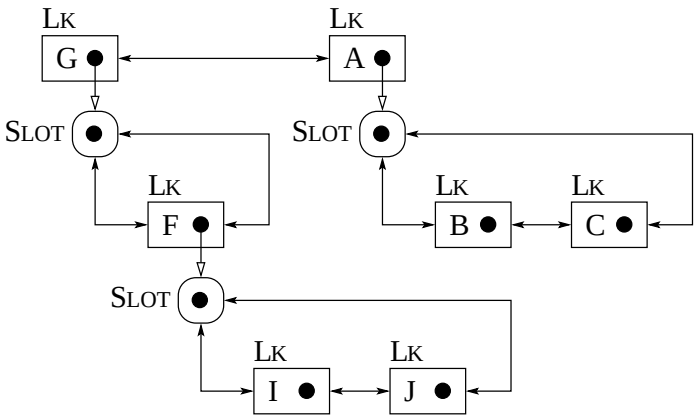

Рис. 10: Схема поля зрения.

Это выражение специально составлено таким образом, чтобы функции во всех состояниях поля зрения различались. Это даст нам возможность упростить изложение, называя состояния по именам соответствующих им функций. Схема поля зрения для выражения показана на рис. 10 .

Пусть, дополнительно, функции B, C, I и F имеют побочные эффекты, а остальные функции – чистые.

Давайте сначала выберем те состояния, которые точно не готовы к выполнению. Состояние F не готово, потому что ожидает, чтобы функции I и J вернули какие-нибудь значения. Аналогично, не готовы к выполнению состояния G и A. Остаются состояния B, C, I и J.

Мы знаем, что функции B, C и I имеют побочные эффекты. Это означает, что их нельзя выполнять в произвольном порядке. То есть разработчик Рефал-программы подразумевает, что эти функции будут вызваны в аппликативном порядке, а именно: сначала I, а потом уже B и C. Тем самым получается, что состояния B и C к выполнению не готовы, и поле зрения содержит только два ведущих состояния: I и J, которые можно выполнять параллельно.

Из рассмотренного примера можно сделать вывод, что выбор ведущих состояний определяется системой зависимостей между состояниями. То есть отличие ведущих состояний от состояний, не готовых к выполнению, заключается в том, что ведущие состояния – независимы.

Мы будем говорить, что состояние y *зависит по данным* от состояния x (обозначается $x \overset{dat}{\rightarrowtail} y$), если состояние x находится в составе стека объектных выражений состояния y. В таблице 1 приведена система зависимостей по данным для нашего примера.

Очевидно, что из $x \overset{dat}{\rightarrowtail} y$ следует $x \prec y$.

Таблица 1: Система зависимостей по данным.

Состояние	Множество зависимостей
A	$\{ \text{B} \overset{dat}{\rightarrowtail} \text{A}, \text{C} \overset{dat}{\rightarrowtail} \text{A} \}$
B	$\varnothing$
C	$\varnothing$
I	$\varnothing$
J	$\varnothing$
F	$\{ \text{I} \overset{dat}{\rightarrowtail} \text{F}, \text{J} \overset{dat}{\rightarrowtail} \text{F}\}$
G	$\{ \text{I} \overset{dat}{\rightarrowtail} \text{G}, \text{J} \overset{dat}{\rightarrowtail} \text{G}, \text{F} \overset{dat}{\rightarrowtail} \text{G} \}$

Таблица 2: Система зависимостей по побочным эффектам.

Состояние	Множество зависимостей
B	$\{ \text{I} \overset{se}{\rightarrowtail} \text{B}, \text{J} \overset{se}{\rightarrowtail} \text{B}, \text{F} \overset{se}{\rightarrowtail} \text{B}, \text{G} \overset{se}{\rightarrowtail} \text{B} \}$
C	$\{ \text{I} \overset{se}{\rightarrowtail} \text{C}, \text{J} \overset{se}{\rightarrowtail} \text{C}, \text{F} \overset{se}{\rightarrowtail} \text{C}, \text{G} \overset{se}{\rightarrowtail} \text{C}, \text{B} \overset{se}{\rightarrowtail} \text{C} \}$
I	$\varnothing$
F	$\{ \text{I} \overset{se}{\rightarrowtail} \text{F}, \text{J} \overset{se}{\rightarrowtail} \text{F} \}$

Мы будем говорить, что состояние y *зависит по побочным эффектам* от состояния x (обозначается $x \overset{se}{\rightarrowtail} y$), если функция y имеет побочные эффекты, функция x не является абсолютно чистой, и $x \prec y$. В таблице 2 приведена система зависимостей по побочным эффектам для нашего примера.

Поясним два момента, связанные с определением зависимости по побочным эффектам. Во-первых, в определении специально оговаривается, что функция y имеет побочные эффекты, потому что выполнение чистой функции не зависит от работы предшествующих функций и может быть инициировано в любой момент времени, коль скоро её параметры полностью вычислены.

Во-вторых, из определения следует, что состояние y зависит не только от состояний, функции которых имеют побочные эффекты, но и от состояний, функции которых не абсолютно чистые. Причину этого можно понять на примере программы:

```
$ENTRY Go { = ⟨Infinite⟩ ⟨Prout α⟩; }
Infinite { = ⟨Infinite⟩; }
```

Согласно семантике Рефала, функция Go вызывает функцию Infinite,

Таблица 3: Полная система зависимостей.

Состояние	Множество зависимостей
A	$\{ B \rightarrowtail A, C \rightarrowtail A \}$
B	$\{ I \rightarrowtail B, J \rightarrowtail B, F \rightarrowtail B, G \rightarrowtail B \}$
C	$\{ I \rightarrowtail C, J \rightarrowtail C, F \rightarrowtail C, G \rightarrowtail C, B \rightarrowtail C \}$
I	$\varnothing$
J	$\varnothing$
F	$\{ I \rightarrowtail F, J \rightarrowtail F \}$
G	$\{ I \rightarrowtail G, J \rightarrowtail G, F \rightarrowtail G \}$

которая является чистой, но, тем не менее, впадает в бесконечную хвостовую рекурсию. С одной стороны, бесконечная рекурсия не есть побочный эффект, но с другой стороны, оказывается, что функция Prout никогда не будет вызвана. Именно поэтому мы считаем, что состояние Prout зависит по побочным эффектам от состояния Infinite.

Построим полную систему зависимостей между состояниями функций в поле зрения, объединив множество зависимостей по данным и множество зависимостей по побочным эффектам. Пусть состояние y зависит от состояния x (записывается как $x \rightarrowtail y$) тогда и только тогда, когда $x \overset{dat}{\rightarrowtail} y$ или $x \overset{se}{\rightarrowtail} y$. Для нашего примера полная система зависимостей приведена в таблице 3.

Полную систему зависимостей можно представить в виде графа. Пусть *граф зависимостей* между состояниями функций – это ориентированный граф, вершинами которого являются состояния функций в поле зрения, а дуги представляют зависимости между состояниями. Для дуги графа, проведённой из состояния x в состояние y мы будем использовать обозначение $x \rightarrow y$. При этом достижимость состояния y из состояния x будет обозначаться как $x \Rightarrow^* y$.

Докажем, что граф зависимостей является бесконтурным. Пусть в графе существует замкнутый контур $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \rightarrow v_0$. Рассмотрим произвольную дугу $x \rightarrow y$, принадлежащую контуру. По определению графа зависимостей наличие в нём этой дуги означает, что между состояниями x и y существует зависимость. Если это зависимость по данным, то $x \prec y$. Если это зависимость по побочным эффектам, то, опять же, $x \prec y$. Тем самым получается, что $v_0 \prec v_1 \prec \ldots \prec v_n \prec v_0$. Так как отношение $\prec$ транзитивно, то $v_0 \prec v_0$, что для отношения строгого порядка невозможно. Поэтому замкнутые контуры в графе зависимостей не существуют.

A Структуры данных

В этом приложении мы попытаемся систематизировать размышления о новом представлении, изложенные в п. 4, и подготовить новое представление к непосредственной реализации на ЭВМ. Эта систематизация, ко всему прочему, потребуется нам в приложении B для изложения спецификации языка сборки.

В качестве целевой платформы мы возьмём абстрактную машину, способную напрямую выполнять инструкции языка сборки. Память абстрактной машины будет разделена на две части: одна часть будет предназначена для потока инструкций, другая – для поля зрения. Такое разделение очевидно упростит компилирующую реализацию абстрактной машины.

И поток инструкций, и поле зрения будут допускать «плоскую» адресацию. Это означает, что мы будем рассматривать обе части памяти как векторы, состоящие из одинаковых ячеек. Адрес некоторого объекта в таком случае является номером ячейки, начиная с которой этот объект расположен в памяти. При этом мы будем иметь в виду, что целые числа, числа с плавающей запятой, указатели и другие значения могут занимать сразу по несколько ячеек.

Подчеркнём, что память абстрактной машины является «неуправляемой». Это означает, что она не накладывает никаких ограничений на типы данных, хранящихся в ячейках. Поэтому реализация абстрактной машины на платформах с «управляемой» памятью потребует дополнительных ухищрений, которые мы оставим за рамками этой статьи.

Описания структур данных, составляющих новое представление, распределены по нескольким таблицам. В таблице 4 перечислены базовые структуры, внутреннее строение которых нами не рассматривается.

Для того чтобы различать разные виды V-термов и L-термов мы будем использовать структуры-перечисления (см. таблицу 5), представляющие фиксированный набор целочисленных констант.

Описание векторных структур данных приведены в таблице 6, а структуры-записи перечислены в таблице 7. При этом названия полей, принадлежащих структурам-записям, в дальнейшем будут активно использоваться для определения семантики инструкций языка сборки.

Таблица 4: Базовые структуры данных.

Имя базовой структуры	Описание
Integer	Целое число, разрядность которого достаточна для представления общего количества ячеек памяти.
Bool	Эта структура может принимать только два значения: «истина» или «ложь».
Symbol	Некоторая информация, необходимая для кодирования значения символа.
Ptr	Адрес в той части памяти, что предназначена для хранения поля зрения. Допустимы три операции над адресами: 1. доступ к значению соответствующей ячейки памяти; 2. прибавление целого числа к адресу; 3. вычисление разности двух адресов. Для того чтобы показать, что некоторый указатель предназначен для хранения адреса структуры типа X, мы будем использовать запись Ptr(X). При этом запись Ptr(X|Y) будет означать, что указатель может содержать либо адрес структуры типа X, либо адрес структуры типа Y.
Cmdptr	Адрес в той части памяти, что отведена для хранения инструкций языка сборки. Такие адреса используются для передачи управления на соответствующую инструкцию.
Entryptr	Почти то же самое, что и Cmdptr, но только обозначает точку входа в функцию. Мы будем считать, что зная адрес Entryptr, Рефал-машина может получить дополнительную информацию о функции (например, её имя).

Таблица 5: Структуры-перечисления.

Имя структуры-перечисления	Константа	Описание
TAGV	TAGV_?	Группа констант, соответствующих различным видам символов.
	TAGV_C	Замыкание.
	TAGV_B	Подвыражение в структурных скобках.
TAGL	TAGL_F	Фрагмент вектора.
	TAGL_B	Подвыражение в структурных скобках.
	TAGL_K	Состояние функции.

Таблица 6: Векторные структуры.

Имя векторной структуры	Описание
ARRAY	Вектор V-термов, лежащий в основе нового представления. Допускается как использование одного вектора для хранения всех V-термов, так и создание отдельных векторов для представления различных объектных выражений. Так как элементами вектора ARRAY являются структуры VS, VC и VB, то эти структуры должны иметь одинаковый размер.
ENV	Вектор указателей на V-термы, используемый для хранения значений переменных окружения.
DEP	Динамический вектор указателей на L-термы LK, используемый для хранения зависимостей между вызовами функций в поле зрения. Размер вектора увеличивается по мере добавления указателей.

Таблица 7: Структуры-записи.

Имя структуры-записи	Имя поля	Содержимое поля	Описание поля
Range	first	Ptr(Vs|Vc|Vb)	Указатель на первый V-терм фрагмента.
	last	Ptr(Vs|Vc|Vb)	Указатель на последний V-терм фрагмента (для того чтобы показать, что фрагмент пустой, значение поля last делают равным first − 1).
Vs	*tagv*	Tagv	Константа, задающая вид символа.
	info	Symbol	Значение символа.
Vc	*tagv*	Tagv	Всегда равно tagv_c.
	entry	Entryptr	Точка входа в функцию.
	env	Ptr(Env)	Значения некоторых локальных переменных функции.
	name	Ptr(Vs)	Имя функции (равно Null для анонимной функции).
Vb	*tagv*	Tagv	Всегда равно tagv_b.
	range	Range	Фрагмент вектора.
Chain	left	Ptr(Lr|Lb|Lk)	Левый L-терм.
	right	Ptr(Lr|Lb|Lk)	Правый L-терм.
Base *	tagl	Tagl	Вид L-терма.
	links	Chain	Указатели на соседние L-термы.
Lr	*base*	Base	Базовая часть.
	fragment	Range	Фрагмент вектора.
Lb	*base*	Base	Базовая часть.
	chain	Chain	Список L-термов.

*Base – вспомогательная структура, включающая общие для всех L-термов поля.

Slot	failure	Bool	Показывает, содержит ли слот значение «неуспех».
	value	Chain	Значение на стеке объектных выражений (имеет смысл, если failure равно «ложь»).
	tail	Ptr(Slot)	Указатель на следующий элемент стека (у последнего элемента равен NULL).
Lk	*base*	Base	Базовая часть.
	addr	Cmdptr	Адрес текущей инструкции.
	params	Ptr(Env)	Значения локальных переменных, определённые в замыкании.
	locals	Ptr(Env)	Значения локальных переменных, вычисленные во время выполнения функции.
	stack	Ptr(Slot)	Указатель на вершину стека объектных выражений (равно NULL, если стек пуст).
	dep	Ptr(Dep)	Массив зависимых состояний.
	count	Integer	Счётчик состояний, от которых данное состояние зависит.
	failcut	Ptr(Slot)	Указатель на слот, в который нужно положить значение «неуспех» в случае неуспешного завершения функции.
	next_leader	Ptr(Lk)	Указатель на следующее состояние в списке ведущих состояний.

Viewfield	roots	Chain	Список L-термов нулевого уровня, образующий поле зрения.
	first_leader	Ptr(Lk)	Указатель на первое состояние в списке ведущих состояний.

B Набор инструкций

Арифметический стек содержит целые числа и указатели на V-термы.

Последовательность вызовов содержит указатели на L-термы Lk.

Регистр замыканий может быть пустым, а может содержать структуру Range, задающую диапазон V-термов, в которых лежат заполняемые взаимнорекурсивные замыкания.

B.1 Инструкции для организации вычислений

B.1.1 const-int

Встроенный операнд: Integer i

Арифметический стек: $\cdots \to \cdots, i$

Описание: кладёт целое число c на арифметический стек.

B.1.2 const-ptr

Встроенный операнд: Ptr(Vs|Vc|Vb) v

Арифметический стек: $\cdots \to \cdots, v$

Описание: кладёт указатель v на арифметический стек.

B.1.3 load

Встроенный операнд: Integer i

Арифметический стек: $\cdots \to \cdots, v$

Ограничение: номер переменной i должен быть меньше размера окружения, указатель на которое хранится в поле locals состояния текущей функции.

Описание: загружает значение локальной переменной на арифметический стек. При этом i – номер ячейки окружения locals, из которой берётся значение переменной.

B.1.4 get

Встроенный операнд: INTEGER i

Ограничение: номер переменной i должен быть меньше размера окружения, указатель на которое хранится в поле params состояния текущей функции.

Арифметический стек: $\cdots \rightarrow \cdots, v$

Описание: загружает значение переменной замыкания на арифметический стек. При этом i – номер ячейки окружения params, из которой берётся значение переменной.

B.1.5 store

Встроенный операнд: INTEGER i

Арифметический стек: $\cdots, v \rightarrow \cdots$

Ограничение: номер переменной i должен быть меньше размера окружения, указатель на которое хранится в поле locals состояния текущей функции.

Описание: сохраняет значение v, лежащее на вершине арифметического стека, в локальную переменную. При этом i – номер ячейки окружения locals, в которую сохраняется значение. Сохранённое значение со стека удаляется.

B.1.6 calc

Суффиксы: +, –, *, /, %, <, >, <=, >=, ==, !=

Арифметический стек: $\cdots, x_1, x_2 \rightarrow \cdots, x_3$

Описание: снимает с вершины арифметического стека значения x_1 и x_2, выполняет над ними указанную арифметическую операцию и помещает результат x_3 на арифметический стек. При этом считается, что x_1 – первый операнд, а x_2 – второй.

B.1.7 duplicate

Арифметический стек: $\cdots, x \rightarrow \cdots, x, x$

Описание: дублирует значение на вершине арифметического стека.

B.1.8 eliminate

Арифметический стек: $\cdots$, $x \to \cdots$

Описание: удаляет значение, лежащее на вершине арифметического стека.

B.2 Инструкции передачи управления

B.2.1 jump

Встроенный операнд: CMDPTR p

Ограничения: регистр замыканий должен быть пустым.

Описание: осуществляет безусловный переход по адресу p.

B.2.2 branch

Суффиксы: .T или .F

Встроенный операнд: CMDPTR p

Арифметический стек: $\cdots$, $b \to \cdots$

Ограничения: регистр замыканий должен быть пустым.

Описание: потребляет с арифметического стека значение b.
Вариант инструкции с суффиксом .T осуществляет переход по адресу p, если b равно «истина».
Вариант инструкции с суффиксом .F осуществляет переход по адресу p, если b равно «ложь».

B.3 Инструкции потребления объектных выражений

B.3.1 failed

Арифметический стек: $\cdots \to \cdots$, b

Стек объектных выражений: $\cdots$, $s \to \cdots$, s

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой.

Описание: Проверяет, содержит ли слот s на вершине стека объектных выражений неуспех.
Результат проверки b помещается на арифметический стек.

B.3.2 pop

Стек объектных выражений: $\cdots$, $s \to \cdots$

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой.

Описание: удаляет значение, лежащее на вершине стека объектных выражений.

B.3.3 dissolve

Арифметический стек: $\cdots \to \cdots$, v_L, v_R

Стек объектных выражений: $\cdots$, $s \to \cdots$

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой.
Слот s на вершине стека объектных выражений должен содержать один единственный L-терм. Причём это должен быть L-терм Lr, полученный функцией в качестве аргумента или возвращённый инструкцией **wait**.

Описание: потребляет значение s, лежащее на вершине стека объектных выражений.
Оставляет на вершине арифметического стека указатели v_L и v_R, полученные из L-терма Lr, содержащегося в s.

B.4 Инструкции сопоставления с образцом

B.4.1 compare

Суффиксы: .L или .R

Арифметический стек: $\cdots$, v_1, v_2, $i \to \cdots$, b

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой, стек объектных выражений должен быть пустым.

Описание: потребляет с арифметического стека указатели v_1, v_2 и целое число i.
Вариант инструкции с суффиксом .L выполняет сравнение двух выражений A и B, где A задано диапазоном V-термов от v_1 до $v_1 + i$, а B задано диапазоном V-термов от v_2 до $v_2 + i$.

Вариант инструкции с суффиксом .R выполняет сравнение двух выражений A и B, где A задано диапазоном V-термов от $v_1 - i$ до v_1, а B задано диапазоном V-термов от $v_2 - i$ до v_2.
Результат сравнения b помещается на арифметический стек.

B.4.2 check

Суффиксы: .B, .S или .C

Арифметический стек: $\cdots$, $v \rightarrow \cdots$, b

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой, стек объектных выражений должен быть пустым.

Описание: потребляет с арифметического стека указатель v и проверяет тип соответствующего V-терма.
Вариант инструкции с суффиксом .B определяет, является ли V-терм, находящийся по адресу v, V-термом VB.
Вариант инструкции с суффиксом .S определяет, является ли V-терм, находящийся по адресу v, V-термом VS.
Вариант инструкции с суффиксом .C определяет, является ли V-терм, находящийся по адресу v, V-термом VC.
Результат проверки b помещается на арифметический стек.

B.4.3 descend

Арифметический стек: v, $\cdots \rightarrow \cdots$, v_L, v_R

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой, стек объектных выражений должен быть пустым, указатель v на вершине арифметического стека должен содержать адрес V-терма VB.

Описание: потребляет с арифметического стека указатель v, содержащий адрес V-терма VB, и оставляет на вершине арифметического стека указатели v_L и v_R, полученные из этого V-терма VB.

B.5 Инструкции порождения замыканий

B.5.1 allocate

Встроенные операнды: ENTRYPTR p_1, ..., ENTRYPTR p_n

Арифметический стек: $\cdots \to \cdots$, v

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой, стек объектных выражений должен быть пустым.

Описание: инструкция создаёт вектор из n V-термов VC, записывает в них адреса $p_1 \ldots p_n$ и имена соответствующих функций, а также резервирует память для n окружений нужных размеров. Координаты вектора записываются в регистр замыканий. Указатель v на первый элемент вектора помещается на арифметический стек.

B.5.2 put

Встроенные операнды: INTEGER i, INTEGER j

Арифметический стек: $\cdots$, $v \to \cdots$

Ограничения: последовательность вызовов должна быть пустой, стек объектных выражений должен быть пустым. Регистр замыканий должен содержать координаты вектора, содержащего n V-термов VC. При этом номер замыкания i должен быть меньше n. Кроме этого, j должно быть меньше размера окружения, содержащегося в i-том V-терме вектора.

Описание: потребляет с арифметического стека значение v и записывает его в j-тый элемент окружения, содержащегося в i-том V-терме вектора.

B.5.3 done

Ограничения: последовательность вызовов должна быть пустой, стек объектных выражений должен быть пустым. Регистр замыканий должен содержать координаты вектора, содержащего n V-термов VC.

Описание: делает регистр замыканий пустым, тем самым завершая формирование вектора замыканий.

B.6 Инструкции создания объектных выражений

B.6.1 push-range

Арифметический стек: $\cdots$, v_L, $v_R \to \cdots$

Стек объектных выражений: $\cdots \to \cdots, s$

Ограничения: регистр замыканий должен быть пустым.

Описание: создаёт новый L-терм LR, в поле fragment которого записывается структура RANGE, задающая вектор V-термов, начинающийся с адреса v_L и заканчивающийся адресом v_R. Созданный L-терм упаковывается в структуру SLOT и помещается на стек объектных выражений.

B.6.2 push-empty

Стек объектных выражений: $\cdots \to \cdots, s$

Ограничения: регистр замыканий должен быть пустым.

Описание: создаёт новый L-терм LR, в поле fragment которого записывается структура RANGE, задающая вектор V-термов нулевой длины. Созданный L-терм упаковывается в структуру SLOT и помещается на стек объектных выражений.

B.6.3 push-call

Арифметический стек: $\cdots, v \to \cdots$

Стек объектных выражений: $\cdots, s_1, \cdots, s_n \to \cdots, s$

Последовательность вызовов: $\Bbbk_0, \cdots, \Bbbk_r \to \Bbbk_0, \cdots, \Bbbk_r, \Bbbk_{r+1}$

Ограничения: регистр замыканий должен быть пустым. Указатель v на вершине арифметического стека должен содержать адрес V-терма VC, задающего замыкание. Стек объектных выражений должен содержать не менее n слотов, где n – количество аргументов, потребляемых указанной в замыкании функцией.

Описание: потребляет указатель v с арифметического стека, создаёт новый L-терм LK и заполняет в нём поля addr, params и locals в соответствии с информацией, взятой из V-терма VC, который находится по адресу v. Кроме того, в созданном L-терме обнуляются поля count и next_leader, а в поле dep помещается указатель на пустой вектор зависимостей.

Затем со стека объектных выражений снимается n слотов, из которых формируется стек объектных выражений созданного L-терма LK.

Созданный L-терм упаковывается в структуру SLOT и помещается на стек объектных выражений. Кроме того, созданный L-терм добавляется в конец последовательности вызовов функций.

B.6.4 depend

Встроенный операнд: INTEGER i

Последовательность вызовов: $\Bbbk_0, \cdots, \Bbbk_i, \cdots, \Bbbk_r \to \Bbbk_0, \cdots, \Bbbk_i, \cdots, \Bbbk_r$

Ограничения: регистр замыканий должен быть пустым, в последовательности вызовов должно быть хотя бы два элемента. Целое число i должно быть меньше r, где r – это номер последнего элемента в последовательности вызовов. При этом L-терм LK, на который указывает $\Bbbk_r$, не должен находиться в списке ведущих состояний.

Описание: устанавливает зависимость последнего элемента последовательности вызовов от i-того элемента.
Другими словами, в вектор зависимостей dep L-терма LK, на который указывает $\Bbbk_i$, добавляется $\Bbbk_r$.

B.6.5 depend-this

Последовательность вызовов: $\cdots, \Bbbk_r \to \cdots, \Bbbk_r$

Ограничения: регистр замыканий должен быть пустым, в последовательности вызовов должен быть хотя бы один элемент.

Описание: устанавливает зависимость текущего выполняемого L-терма LK от последнего элемента последовательности вызовов.
Другими словами, в вектор зависимостей dep L-терма LK, на который указывает $\Bbbk_r$, добавляется указатель на текущий выполняемый L-терм LK.

B.6.6 copy-dep

Последовательность вызовов: $\cdots, \Bbbk_r \to \cdots, \Bbbk_r$

Ограничения: регистр замыканий должен быть пустым, в последовательности вызовов должен быть хотя бы один элемент. При этом L-терм LK, на который указывает последний элемент последовательности вызовов, должен находиться в списке ведущих состояний (то есть для него должна была отработать инструкция **link**), и

вектор зависимостей, на который указывает его поле dep, должен быть пуст.

Описание: копирует все зависимости текущего выполняемого L-терма Lk в последний элемент последовательности вызовов.
Другими словами, происходит копирование вектора зависимостей текущего выполняемого L-терма Lk в L-терм Lk, на который указывает $\Bbbk_r$. При этом счётчики count всех L-термов Lk, находящихся в копируемом векторе зависимостей, увеличиваются на единицу.

B.6.7 transfer-dep

Последовательность вызовов: $\cdots, \Bbbk_r \to \cdots, \Bbbk_r$

Ограничения: регистр замыканий должен быть пустым, в последовательности вызовов должен быть хотя бы один элемент. При этом L-терм Lk, на который указывает последний элемент последовательности вызовов, должен находиться в списке ведущих состояний (то есть для него должна была отработать инструкция **link**), и вектор зависимостей, на который указывает его поле dep, должен быть пуст.

Описание: переносит все зависимости текущего выполняемого L-терма Lk на последний элемент последовательности вызовов.
Другими словами, происходит обмен значений полей dep текущего выполняемого L-терма Lk и L-терма Lk, на который указывает $\Bbbk_r$. (Пояснение: так как L-терм Lk, на который указывает $\Bbbk_r$, до выполнения инструкции **transfer-dep** имел пустой вектор зависимостей, то после обмена значений полей dep этот пустой вектор зависимостей окажется у текущего выполняемого L-терма Lk.)

B.6.8 link

Последовательность вызовов: $\Bbbk_0, \cdots, \Bbbk_r \to \Bbbk_0, \cdots, \Bbbk_r$

Ограничения: регистр замыканий должен быть пустым, в последовательности вызовов должен содержаться хотя бы один элемент. При этом L-терм Lk, на который указывает последний элемент последовательности вызовов, не должен зависеть ни от одного состояния функции (его поле count должно быть равно нулю).

Описание: добавляет L-терм Lk, на который указывает $\Bbbk_r$, в список ведущих состояний.

При этом в поле next_leader этого L-терма записывается значение поля first_leader поля зрения, а затем полю first_leader присваивается значение $\Bbbk_r$.

B.6.9 cut

Встроенный операнд: INTEGER i

Стек объектных выражений: $\cdots, s \to \cdots, s$

Последовательность вызовов: $\Bbbk_0, \cdots, \Bbbk_i, \cdots, \Bbbk_r \to \Bbbk_0, \cdots, \Bbbk_i, \cdots, \Bbbk_r$

Ограничения: регистр замыканий должен быть пустым, в последовательности вызовов должен содержаться хотя бы один элемент. Целое число i должно быть меньше или равно r, где r – это номер последнего элемента в последовательности вызовов. При этом L-терм LK, на который указывает $\Bbbk_i$, должен соответствовать откатной функции.

Описание: указывает, в какой слот стека объектных выражений будет помещено значение «неуспех» в случае, если откатная функция, на состояние которой указывает $\Bbbk_i$, завершится через инструкцию **retf**.

Другими словами, в поле failcut L-терма LK, на который указывает $\Bbbk_i$, записывается указатель на слот s, находящийся на вершине стека объектных выражений.

B.6.10 bracket

Стек объектных выражений: $\cdots, s_1 \to \cdots, s_2$

Ограничения: регистр замыканий должен быть пустым, слот s_1 не должен содержать «неуспех».

Описание: потребляет со стека объектных выражений слот s_1, создаёт новый L-терм LB и записывает в поле chain L-терма значение поля value слота s_1. Тем самым происходит оборачивание выражения, содержащегося в s_1, в структурные скобки. Полученное в результате выражение s_2 помещается на стек объектных выражений.

B.6.11 concat

Стек объектных выражений: $\cdots, s_1, s_2 \to \cdots, s_3$

Ограничения: регистр замыканий должен быть пустым, слоты s_1 и s_2 не должны содержать значение «неуспех».

Описание: потребляет со стека объектных выражений слоты s_1и s_2, осуществляет конкатенацию содержащихся в них выражений и помещает результата конкатенации на стек объектных выражений.

B.7 Инструкции для взаимодействия с системой выполнения

B.7.1 wait

Бла-бла-бла... И ещё бла-бла-бла...

B.7.2 ret

Стек объектных выражений: $s \to \nexists$

Ограничения: регистр замыканий должен быть пустым, арифметический стек должен быть пустым.

Описание: завершает выполнение текущей функции.
При этом L-терм Lk, соответствующий состоянию текущей функции, заменяется на объектное выражение, взятое из слота s на стеке объектных выражений.

B.7.3 retf

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой, арифметический стек должен быть пустым, стек объектных выражений должен быть пустым. Состояние функции должно содержать ненулевое значение в поле failcut.

Описание: завершает выполнение текущей функции, вырабатывая значение «неуспех».
При этом из поля failcut состояния функции берётся указатель на слот стека объектных выражений некоторой другой функции. Объектное выражение, содержащееся в этом слоте, уничтожается, а вместо него в слот записывается значение «неуспех».

B.7.4 crash

Ограничения: регистр замыканий должен быть пустым, последовательность вызовов должна быть пустой, арифметический стек должен быть пустым, стек объектных выражений должен быть пустым.

Описание: осуществляет аварийный останов программы.

Предметный указатель

Аппликативный порядок, 21
Действие (в предложении), 17
Длина (выражения), 4
Функция
  чистая, 21
    абсолютно, 21
  имеющая побочные эффекты, 21
  откатная, 21
Головное звено (списка), 8
Граф зависимостей, 24
Конкатенация
  отложенная, 16
  преждевременная, 12
  с неготовым значением, 9
Неудача, 21
Окружение, 15
Повышатель арности, 13
Проблема
  конкатенации, 12
  копирования, 5
Размер (выражения), 4
Сборка (выражения), 17
Состояние функции, 18
  ведущее, 21
Способ представления выражений
  классический, 5
  компромиссный, 8
  новый, 14
  векторный, 11
Структура данных
  Array, 27
  Base, 28
  Bool, 26
  Chain, 17, 28
  Cmdptr, 26
  Dep, 27
  Entryptr, 26
  Env, 15, 27
  Integer, 26
  L-терм, 16
    Lb, 17, 28
    Lk, 18, 29
    Lr, 16, 28
  Ptr, 26
  Range, 15, 28
  Slot, 19, 29
  Symbol, 26
  Tagl, 27
  Tagv, 27
  V-терм, 14
    Vb, 16, 28
    Vc, 16, 28
    Vs, 15, 28
  Viewfield, 30
Условие (в предложении), 17
Ведущая пара (скобок), 20
Язык сборки, 4
Замыкание, 14
Зависимость состояний
  по данным, 22
  по побочным эффектам, 23

Список литературы

[1] С.А. Романенко. *Реализация Рефала-2.* – М.: ИПМ им. М.В. Келдыша АН СССР, 1987.

[2] С.М. Абрамов, С.А. Романенко. *Представление объектных выражений массивами при реализации языка Рефал.* – М.: ИПМ им. М.В. Келдыша АН СССР, 1988. – Препринт №186.

[3] V.F. Turchin. *REFAL-5 Programming Guide and Reference Manual.* – Holyoke: New England Publishig Co., 1989.

[4] S.A. Romanenko. *Arity Raiser and its Use in Program Specialization* // Proceedings of the ESOP'90. Lecture Notes on Computer Science. – Springer-Verlag, 1990. – Vol. 432.

[5] А.С. Фролов, Л.К. Эйсымонт. *Система динамического распараллеливания Рефал программ PREFRTS* // Научная сессия МИФИ-2005. Сборник научных трудов. – М.: МИФИ, 2005. – Т.2: Технологии разработки программных систем. Информационные технологии.

[6] С.Ю. Скоробогатов, А.М. Чеповский. *Разработка нового диалекта языка Refal* // Информационные технологии. – М.: Новые технологии, 2006. – №9.