УДК 681.322

БИБЛИОТЕЧНАЯ ПОДДЕРЖКА ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ МОДЕЛИ ЯЗЫКА РЕФАЛ

© 2009 г. И. Е. Бронштейн, А. В. Столяров

igor.bronstein@intelib.org, avst@cs.msu.ru

Кафедра алгоритмических языков

1 Особенности языка Рефал

Язык Рефал был предложен В. Ф. Турчиным в 1966 году [10]. В. Ш. Кауфман отмечает сходство изобразительных средств Рефала с нормальными алгоритмами Маркова [3]. Стилевую модель языка Рефал В. Ш. Кауфман называет *ситуационным программированием*, выделяя две ключевые абстракции — *анализ* исходной структуры данных и *синтез* результирующей структуры.

Рефал чрезвычайно удобен при решении задач, связанных с преобразованиями символьных данных. В. Ф. Турчин изначально позиционировал Рефал как *метаалгоритмический* язык [9] и позднее использовал его, в числе прочего, для изучения эквивалентных преобразований программ [11]. В. И. Сердобольский отметил удобство языка для работы с алгебраическими формулами [2]. С середины восьмидесятых годов и до настоящего времени Рефал успешно применяется в исследованиях, связанных с суперкомпиляцией и частичными вычислениями [12].

К настоящему времени существует несколько диалектов языка Рефал, среди которых наиболее динамично развиваются Рефал-5 [13] и Рефал Плюс. Разработаны методы эффективной реализации языка; созданы системы программирования для различных программно-аппаратных платформ.

Несмотря на это, Рефал крайне редко применяется в индустриальном программировании. При этом Рефал нельзя назвать неизвестным языком; многие профессиональные программисты с этим языком знакомы. Основной причиной слабой востребованности языка Рефал следует, видимо, считать ограниченность круга задач, в которых этот язык оказывается удобнее других; при этом для задач, не входящих в этот круг, применение Рефала хотя и возможно, но оказывается нецелесообразным.

С учётом этого становится актуальной задача *интеграции* вычислительной модели языка Рефал с системами программирования, основанными на других языках. В частности, можно рассмотреть один из самых популярных на сегодняшний момент индустриальных языков программирования — язык Си++ [8]; идея предоставления программистам в рамках проектов на Си++ выразительных возможностей, характерных для языка Рефал, выглядит достаточно привлекательной.

2 Библиотека InteLib

Библиотека InteLib (см., напр., [4, 6]), являясь библиотекой классов Си++, реализует *метод непосредственной интеграции*, впервые предложенный в работе [1]. Идея метода состоит в моделировании структур данных альтернативной вычислительной модели и введении для создаваемых классов такого набора операций, с помощью которого в тексте на базовом языке программирования (в данном случае Си++) можно формировать арифметические выражения, визуально напоминающие конструкции альтернативного языка. Реализовав соответствующий вычислитель, мы можем сделать такие выражения семантически эквивалентными исходным конструкциям.

В рамках библиотеки InteLib были реализованы модели языков Лисп и Scheme [7], делались также попытки создания моделей языков Плэнер [14] и Пролог. Созданная в 2001 году и описанная в статье [5] модель языка Рефал оказалась непригодна к практической работе в силу своей крайне низкой эффективности и не совсем удачно выбранного набора операций для формирования Рефал-выражений.

В настоящей статье описывается реализация вычислительной модели языка Рефал, имеющая более удобные средства задания Рефал-выражений и приемлемую (хотя и не очень высокую) эффективность по времени выполнения.

3 Представление конструкций Рефала выражениями Си++

Рефал является языком для работы со слабо структурированными данными. Единицами поля видимости здесь могут являться как символы (в традиционном смысле — как, например, символы в Си++), числа и «символы-метки», так и выражения в угловых или круглых скобках. Таким образом, уже на этапе выбора представления для этих конструкций в Си++ возникает вопрос, что должен хранить объект класса, представляющего единицу поля видимости, и как лучше организовать работу со скобочными структурами. Эти проблемы могут решаться двумя способами:

1. нелинейной структурой — скобочному выражению соответствует объект, содержащий указатель на структуру данных, представляющую выражение в скобках;

2. линейной структурой — как каждая из скобок, так и оставшиеся элементы выражения в скобках представляются отдельными объектами.

Существуют реализации языка Рефал, использующие каждый из этих способов. В рассматриваемой реализации отдано предпочтение линейным структурам, представленным в виде двусвязных списков. В этих списках вместе с объектами, соответствующими структурным и операторным скобкам, хранятся указатели на соответствующие им парные объекты; это позволяет ускорить навигацию по представлению выражения.

Список составляется из элементов структурного типа, названного `RfListItem`. В список, состоящий из таких элементов, могут входить объекты трёх основных типов: обычный (алфавитный) символ; элемент разметки (структурная или операторная скобка); все остальные объекты, а именно «символы-метки», числа, переменные и т. д., представляемые в виде произвольного S-выражения из ассортимента библиотеки InteLib.

Рефал-выражение как целое инкапсулируется в класс `RfExpression`. Объект этого класса хранит начало и конец двусвязного списка, представляющего выражение, и вводит операции для работы ним, такие как добавление новых элементов в конец списка, навигацию, извлечение элементов и т. п. Класс `RfExpression` входит в основную полиморфную иерархию библиотеки InteLib, его объекты заводятся в динамической памяти; в качестве основного интерфейса для работы с ними, как и с другими объектами этой иерархии, используются «умные указатели» [15], в данном случае — объекты класса `RfReference`.

Существования в InteLib класса `RfReference` уже достаточно, чтобы сформировать структуру данных, представляющую выражение Рефала, и затем обрабатывать её. Однако, как уже говорилось, целью метода непосредственной интеграции является реализация конструкций, которые синтаксически близки к исходным конструкциям интегрируемого языка. Для этого был введён статический класс `RfFormConstructor`. Конструкция «`1 2 3`», например, создаётся с помощью этого класса следующим образом:

```
RfFormConstructor R;
RfReference ref = (R|1, 2, 3);
```

Возможость использовать такую конструкцию обеспечивается переопределением операций. Например, для приведённого примера существенны две операции:

```
RfReference operator|(const SReference &r) const;
RfReference operator,(const SReference& ref);
```

Таблица 1: Представление конструкций Рефала выражениями Си++

Рефал-5	**Си++**
`1`	`1`
`'-'2`	`-2`
`'abcd'`	`"abcd"`
`symbol`	`symbol`
`"a symbol"`	`a_32_symbol`
`e.1`	`e_1`
`1 2 3 4`	`(R|1, 2, 3, 4)`
`(1 2 3 4)`	`(R|(R|1, 2, 3, 4))`
`<f 1 2 3 4>`	`(R<f, 1, 2, 3, 4>R)`
	`(~R)`
`()`	`(R|(~R))`

`f {`
`  1 = T;`
`  0 = F;`
`}`
	`f`
	`  || 1 ^ T`
	`  || 0 ^ F`

`f {`
`  e.1, <g e.1> :`
`  True = False`
`}`
	`f`
	`  || e_1 & (R|R<g, e_1>R) |`
	`  True ^ False`

Сначала операцией `operator|` (метод класса `RfFormConstructor`), применённой к `1`, создаётся объект класса `RfReference`, указывающий на список из одного элемента, который соответствует символу-числу `1`. Затем с помощью операции `operator,` (метод класса `RfReference`) к этому списку последовательно добавляются справа новые элементы.

Рассмотрим теперь представление функций языка Рефал. Тело функции состоит из набора предложений. Каждое такое предложение (если исключить из рассмотрения with-блоки, которые ранее не поддерживались в InteLib) можно записать в следующем виде:

```
pattern[1],
expression[1] : pattern[2],
...,
expression[N-1] : pattern[N] = expression[N],
```

Стоит, однако, заметить, что, заменив знак равенства на запятую и сгруппировав участвующие пары «образец — выражение», можно получить более единообразный вид, который позволит ввести удобную и простую структуру данных для хранения предложений и тела функций в целом:

```
pattern[1], expression[1] :
pattern[2], expression[2] :
... :
pattern[N], expression[N]
```

Для хранения таких пар в рассматриваемой реализации введён класс `RfClause`. Для представления предложения языка Рефал используется двусвязный список объектов этого класса. Для представления тела функции целиком потребовалось связать первые пары всех предложений также в односвязный список: в `RfClause` был введён атрибут, указывающий для первой пары текущего предложения на первую пару следующего, либо равный `0` в остальных случаях.

В случае с функциями языка Рефал для обеспечения синтаксической схожести моделирующих и исходных конструкций также используется переопределение арифметических операций. Идея состоит в том, чтобы для объекта, представляющего символ Рефала, иметь возможность в начале выполнения программы указать, какая точно конструкция, представляющая тело функции, ему соответствует. Например, пусть дана функция:

Рефал:

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

Си++:

```
Pal
    || (R|s_1, e_2, s_1) ^
       (R|R<Pal, e_2>R)
    || (~R) ^ (R|True)
    || (R|s_1) ^ (R|True)
    || (R|e_1) ^ (R|False)
```

Рис. 1: соответствие конструкций, описывающих тело функции на Рефале-5 и на Си++

```
f1 {
  s.1 s.2, <f2 s.1> : True = True;
  e.1 = False
}
```

Ей будет соответствовать следующий текст программы в конструкциях библиотеки InteLib:

```
RfSymbol f1("f1");
RfSymbol f2("f2");
f1 || (R|s_1, s_2) & (R<f2, s_1>R) | (R|True) ^ (R|True)
   || (R|e_1) ^ (R|False);
```

Для облегчения понимания того, как после применения переопределённых арифметических операций объект класса `RfSymbol` становится связанным со структурой, описанной выше, две последние строчки следует переписать с учётом приоритета операций в Си++:

```
RfSymbol f1("f1");
RfSymbol f2("f2");
f1 || (((R|s_1, s_2) & (R<f2, s_1>R)) | ((R|True) ^ (R|True))))
   || ((R|e_1) ^ (R|False);
```

Существенными для приведённого примера являются следующие переопределённые операции:

```
RfClause operator&(RfReference &pattern, RfReference &result);
RfClause operator^(RfReference &pattern, RfReference &result);
RfClause& operator|(const RfClause& a);
RfClause& operator||(const RfClause &cla);
```

Сначала с помощью операций `operator&` и `operator^` (которые выполняют одни и те же действия) создаётся новый объект класса `RfClause`, в котором «склеены» в пару образец и выражение; затем с помощью оперции `operator|` образуются односвязные списки, представляющие предложение; наконец, на третьем этапе выполняются операции `operator||`. Первая из них связывает символ `f1` с первым предложением-списком, остальные последовательно «присоединяют» первую пару нового предложения к уже существующей конструкции.

Такие операторы инициализации всех символов, соответствующих функциям в Рефале, удобно поместить в одну процедуру, которая будет вызываться в начале программы перед непосредственным вызовом этих функций.

4 Внутренние структуры данных

Опишем основные структуры данных рефал-вычислителя.

Связи между переменными и соответствующими значениями хранятся в переменных структурного типа `RfBinding`, в котором предусмотрены указатели на связываемую переменную и элементы выражения, в котором находится соответствующее значение.

Ключевым понятием при реализации рефал-машины является понятие диапазона. В диапазоне указываются начало и конец разбираемых в каждый конкретный момент участков образца и выражения. При этом там также могут содержаться ссылки на уже просмотренные диапазоны либо на диапазоны, которые только предстоит просмотреть. Отдельной задачей является выбор оптимальной структуры данных для хранения диапазонов. В рассматриваемой реализации в качестве такой структуры выбрано двоичное дерево — «дерево диапазонов».

Корнем дерева является главный диапазон — диапазон, соответствующий целым образцу и выражению. Во всех узлах помимо соответствующих указателей на образец и выражение хранятся указатели на левого и правого потомка, а также на родителя данного узла; соответствующая структура называется `RfRange`.

Все объекты классов, соответствующих абстракциям, используемым при реализации рефал-вычислителя (связи между переменными и их значениями, сохранённые выражения, диапазоны сопоставления и другие), объединены в общей структуре данных — контексте, который реализуется классом `RfContext`.

5 Реализация Рефал-вычислителя

В рамках Рефал-вычислителя нам необходимо реализовать сопоставление образца и выражения, подстановку значений переменных в заданное выражение, обработку отдельного предложения Рефал-программы и, наконец, шаг работы Рефал-машины, представляющий собой замену в поле зрения вызова функции на результат такого вызова.

5.1 Сопоставление

Сигнатура функции сопоставления выражений выглядит следующим образом:

```
bool MatchProcess(RfContext &context, RfRange *main_range);
```

Аргумент `context` представляет собой контекст, который будет изменяться в ходе выполнения функции сопоставления, а `main_range` — диапазон, с которого нужно начинать обход дерева.

Дерево диапазонов обходится в глубину («левый потомок — узел — правый потомок»). При этом обход считается произведённым успешно, если успех достигнут в каждом узле дерева. Если сопоставление в каком-либо узле оказалось неуспешным, неуспех выдается всей функцией сопоставления. Однако это не значит, что всё дерево диапазонов уничтожается. Оно сохраняется в списке деревьев (в контексте), и в дальнейшем к нему может быть осуществлён возврат. Сопоставление в каждом конкретном узле считается успешным, когда текущие участки и образца, и выражения пусты.

В каждый конкретный момент выполнения функции сопоставления специально выделен один из узлов дерева диапазонов, называемый рабочим диапазоном. Однако ни он, ни другие узлы не меняются в процессе выполнения функции сопоставления. Все модификации производятся в текущем диапазоне — объекте, в который копируются все данные из только что выбранного рабочего диапазона.

Если в образце встретился символ, алгоритм проверяет наличие такого же символа в соответствующем месте выражения. Если выражение в круглых скобках, то проверяется, есть ли соответствующая скобочная структура в выражении. Если она отсутствует, то сопоставление считается неуспешным. Иначе вносятся изменения в дерево диапазонов: к рабочему узлу добавляются два потомка. Они соответствуют двум половинам образца и, соответственно, выражения, на которые их «разбивает» скобочная структура. Далее сопоставление в узле дерева считается успешным, и новым рабочим диапазоном становится левый потомок старого.

Если в образце встретилась уже связанная с некоторым значением переменная, алгоритм сопоставления сводится к попытке найти соответствующее значение в выражении, соответствующим образом изменить текущий диапазон (если значение найдено) и продолжить работу с ним. Если же ещё не связанная с некоторым значением переменная типа `s` или `t`, то она связывается с соответствующим символом или термом из выражения, если это возможно.

Соответствущая связь «переменная — значение» добавляется в начало списка связей. Далее функция сопоставления продолжает работу с модифицированным текущим диапазоном.

Переменные типа `e`, ещё не связанные с каким-либо значением, могут быть либо закрытыми, либо открытыми. Если в текущем диапазоне в образце осталась лишь одна не связанная e-переменная, то ей сразу можно сопоставить всё соответствующее диапазону выражение. Такая `e`-переменная называется закрытой. В противном случае она является открытой. Алгоритм сопоставления старается избегать открытых переменных. Если такая переменная встретилась при сопоставлении слева направо, направление сопоставления меняется на противоположное и попытка повторяется. Таким образом, необходимость работы с новой переменной типа `e` возникает лишь тогда, когда в текущем участке образца переменных этого типа больше одной и они находятся как в начале, так и в конце участка.

Если в образце встретилась открытая переменная типа `e`, то она исходно считается связанной с пустым значением. При этом в целях обеспечения возможности поиска с возвратом для этой переменной хранятся указатели на конец текущего хранимого значения (`0` в случае пустого значения) и на конец теоретически максимально возможного значения. В дальнейшем в случае поиска с возвратом первый из этих указателей будет при необходимости пробегать все позиции от начала до конца сопоставляемого выражения. Соответствущая связь «переменная — значение» добавляется в начало списка связей.

В случае открытой переменной помимо добавления новой связи так же, как и при сопоставлении скобочных структур, меняется дерево диапазонов. У рабочего узла дерева появляется один потомок, в который копируются данные текущего диапазона. Кроме того, создаются перекрёстные ссылки между только что добавленным диапазоном и связью «переменная — значение». Новый потомок становится рабочим диапазоном, и функция сопоставления продолжает свою работу.

Всё описанное ранее относилось к случаю, когда функция сопоставления начинает создавать дерево диапазонов с самого начала, с главного диапазона. Однако также существует необходимость вызывать эту функцию в случае возникновения поиска с возвратом. В этом случае дерево диапазонов уже построено, но требуется изменение значений некоторых переменных, что может, в свою очередь, привести к модификации дерева.

В первую очередь выбирается переменная, значение которой можно изменить для осуществления поиска с возвратом. Из выбора структур данных для хранения связей «переменная — значение» следует, что такой переменной (если она существует) является первая в списке связей переменная типа `e`, для которой номер фрейма совпадает с текущим номером (т. е. получившая значение в текущем фрейме), а указатель текущего значения не достиг последнего возможного значения. Далее значение это переменной изменяется на следующее возможное, и функции сопоставления в качестве `main_range` передаётся не корень существующего дерева диапазонов, а диапазон, соответствующий выбранной переменной. Связи, находящиеся в списке до выбранной переменной, удаляются, так как значения соответствующих переменных могут измениться. Таким образом, необходимо модифицировать дерево диапазонов с тем, чтобы ещё раз осуществить попытку сопоставления с новым значением выбранной переменной. Из свойств дерева диапазонов следует, что для этого необходимо лишь вновь обойти дерево в глубину, начиная с рабочего диапазона.

Необходимо отметить существование различных возможностей для оптимизации приведённой выше реализации. Например, в ней для переменных типа `e` могут перебираться все варианты значений, в том числе и заведомо невозможные. Так, если справа от этой переменной в образце находится (сразу или после нескольких открывающихся круглых скобок) некоторый символ, называемый подсказкой, перебор значений возможно производить гораздо более оптимально. Достаточно хранить подсказку и соответствующее число скобок до неё и при каждом переходе к новому значению непосредственно проверять, возможно ли оно. Это называется оптимизацией с помощью подсказки.

5.2 Подстановка

Сигнатура функции, осуществляющей преобразование рефальского выражения с учётом подстановки в него значений переменных, выглядит следующим образом:

```
RfReference RefalSubstitute(const RfReference &right_part,
                            RfContext &context,
                            bool is_last);
```

Здесь `right_part` — это выражение, в котором необходимо произвести замену, `context` — текущий контекст, а смысл `is_last` станет понятен несколько позже. Функция возвращает выражение, в котором уже произведены соответствующие замены.

Простейшая реализация этой функции, осуществлённая изначально, выглядит следующим образом. Вначале создаётся новое выражение языка Рефал, в которое затем будут добавляться элементы. Затем осуществляется проход по всем объектам класса `RfListItem`, составляющим выражение. Здесь возможны два случая: если встретился объект, не являющийся переменной, необходимо создать его копию; иначе необходимо скопировать значение встретившейся переменной.

Однако, такая реализация является неэффективной. Рассмотрим, например, некоторую функцию, которая разбирает выражение посимвольно и осуществляет некоторую операцию с каждым символом (подобные функции очень типичны для лексического анализа текста и т. д.):

```
SomeFunction {
    s.1 e.2 = <MakeSomethingWith s.1> <SomeFunction e.2>;
    = ;
}
```

Фактически для вышеописанной реализации в приведённой функции необходимо было бы совершить $N - 1 = O(N)$ (где N — длина поля видимости) дорогих операций копирования объектов класса `RfListItem`. Всего же за время выполнения функции потребовалось бы $1 + 2 + \cdots + (N - 1) = O(N^2)$ операций копирования.

В связи с этим возникла необходимость в оптимизации функции замены. Для этого был добавлен новый аргумент `is_last`, который равен `true` в случае, если функция замены гарантированно вызывается последний раз в текущем предложении (т. е. в том случае, если выполняется замена после знака равенства, а не в where-предложении после запятой). Кроме того, для каждой связи «переменная — значение» была добавлена булевская переменная `was_used`, изначально (при добавлении связи) равная `false`.

Далее каждый раз, когда возникает необходимость копировать значение переменной типа `e` либо переменной типа `t` (в том случае, если её значением является не символ, а терм), в первую очередь проверяется, нельзя ли использовать это значение в правой части непосредственно, без копирования. Для каждой переменной такая возможность существует лишь один раз, поэтому осуществляется проверка `was_used` на `false`. Если `was_used` равно `true`, то осуществляется обычное копирование, описанное выше. Иначе `was_used` присваивается `true`, и выполняется следующее: участок, соответствующий значению переменной, «вырезается» из изначального списка и добавляется в конец генерируемого (т. е. происходит использование значения без копирования).

Оптимизированная реализация является гораздо более эффективной. В общем случае, если в правой части предложения нет повторяющихся переменных, число операций копирования теперь не зависит от размера поля видимости и ограничено константой — общим числом переменных в правой части. Итак, для приведённой выше функции на каждом шаге будет выполняться всего одно копирование. Таким образом, в ходе выполнения всей функции потребуется лишь N, а не $O(N^2)$ (где N — длина поля видимости) подобных операций копирования.

5.3 Реализация функциональных вызовов

Сигнатура функции, которая реализует этап работы рефал-вычислителя, связанный с одним конкретным предложением, выглядит следующим образом:

```
bool RfClause::EvaluateClause(RfReference &view,
                              RfReference &res,
                              RfContext &context) const;
```

Данная функция является методом класса `RfClause`. Аргумент `view` представляет собой рефальское выражение, к которому будет применяться соответствующее объекту `RfClause` предложение. Аргумент `context` — это контекст, смысл передачи которого в функцию `EvaluateClause` станет понятен при описании реализации with-блоков (безымянных функций). Результирующее выражение записывается в `res`.

Сопоставление начинается с первой пары, предствляющей предложение, и выбирается направление движения слева направо. В результате работы функции либо достигается успех в последнем узле списка — в этом случае успешным считается всё предложение, либо фиксируется неуспех в первом узле — при отсутствии каких-либо дальнейших возможностей отката назад. Соответствующий результат (успех или неуспех) возвращается функцией `EvaluateClause`.

В одной из локальных переменных хранится текущее сопоставляемое выражение, которое исходно равно выражению, передаваемому в функцию через параметр `view`.

Если в какой-то момент просматривается узел и направлением обхода является направление слева направо, то производится попытка сопоставления текущего сопоставляемого выражения и образца из текущей пары.

Если же направлением обхода является направление справа налево, это значит, что в этот узел был произведён возврат из последующих узлов из-за неудачного сопоставления в них. В этом случае производится описанная в § 5.1 процедура поиска нужной переменной, значение которой возможно изменить.

Если такая переменная найдена, то направление обхода меняется на направление слева направо. Если же нет — необходимо откатиться к предыдущей в списке паре, если таковая существует. В случае, когда пара была первой в списке, фиксируется неуспех всего предложения.

Далее приведена сигнатура функции, реализующей вычисление не одного предложения, а рефальской функции целиком:

```
void RfClause::Apply(RfReference &view, RfContext &context) const;
```

Здесь `view` — это выражение, к которому применяется функция. Результат выполнения функции также записывается в `view`.

В этой функции последовательно применяются `EvaluateClause` для первых пар списков, представляющих рефальские предложения, и результатом работы является результат работы первой завершившейся успехом функции, связанной с конкретным предложением. Если такое предложение не было найдено, возбуждается исключительная ситуация.

Отдельно необходимо сказать о реализации в библиотеке InteLib with-блоков. Фактически они представляют собой безымянные функции, в которые передаётся в качестве аргумента последнее вычисленное в предложении выражение. Следует отметить, что в отличие от обычных функций, перед вызовом которых создаётся новый контекст, при вызове безымянных функций контекст используется существующий. Значения, с которыми были связаны переменные во внешней функции, не могут быть изменены во внутренней.

Для поддержки with-блоков был создан новый статический класс `RfWith`, инкапсулирующий в себе список безымянных символов и соответствующих им функций.

Рассмотрим для примера функцию на языке Рефал, содержащую with-блок:

```
Go {
  s.1 s.2, <f s.1> : { True = s.1; False = s.2 }
}
```

Ниже представлено выбранное для неё представление в виде выражения Си++:

```
RfWith RWITH;

Go
  || (R|s_1, s_2) & (R|R<f, s_1>R) |
  (RWITH
```

```
    || (R|True)        ^ (R|s_1)
    || (R|False)       ^ (R|s_2)
  );
```

В объекте класса `RfWith` хранится список безымянных символов. Операция `operator||` генерирует новый безымянный символ и добавляет его в начало списка.

Для поддержки with-блоков в класс `RfClause` был добавлен булевский атрибут `is_where`, значение которого равно `true`, если пара ссылается на безымянную функцию, и `false` в противном случае.

При вызове операции `operator|` создаётся пара «образец — выражение» следующего вида: значение `is_where` в ней равно `true`, а в `pattern` хранится непосредственно ссылка на соответствующую безымянную функцию.

Когда текущей парой становится пара такого специального вида, вызывается соответствующая безымянная функция от текущего сопоставляемого выражения. В эту функцию передаётся не вновь созданный (как в случае обычных функций), а уже существующий контекст. Перед этим происходит операция обновления контекста. Она заключается в том, что во всех связях между переменными и значениями номеру фрейма связи присваивается значение `0`. Это гарантирует, что значения таких переменных никогда не изменятся во вложенном блоке, что удовлетворяет семантике блоков.

Благодаря реализации with-блоков и функций стандартной библиотеки было достигнуто соответствие поддерживаемого диалекта описанию языка Рефал-5.

6 Транслятор IRINA

Транслятор IRINA (**I**nteLib **R**efal **I**ntelligent, **N**ew and **A**ll-inclusive Translator) принимает на вход один или несколько файлов на Рефале. На выходе генерируется модуль языка Си++ в виде заголовочного файла и файла реализации.

Для настройки параметров IRINA используются директивы трансляции. Синтаксис каждой директивы следующий:

```
НАЗВАНИЕ_ДИРЕКТИВЫ = аргумент
```

либо

```
НАЗВАНИЕ_ДИРЕКТИВЫ аргумент1 = аргумент2
```

Список директив представляет собой последовательность таких псевдопредложений, ограниченную конструкцией `/*%%%...*/`:

```
/*%%%
    ДИРЕКТИВА[1];
    ...
    ДИРЕКТИВА[N];
*/
```

Необходимо отметить важное свойство списков директив такого вида: они являются прозрачными для стандартного транслятора Рефала-5, так как с его точки зрения представляют собой обычный многострочный комментарий. Это позволяет избегать модификации файлов при работе как со стандартной системой программирования на Рефале-5, так и с транслятором IRINA.

Конструкции `/*%%%...*/` могут встречаться в одном файле более одного раза: при этом списки директив, заключённые внутри них, просто считаются объединёнными в общий список.

Директивы трансляции позволяют, в частности: объявлять публичные и/или внешние символы или функции для текущего модуля на Рефале; управлять правилами преобразования имён переменных и символов; задавать имена результирующих файлов; задавать дополнительные настройки, связанные с содержимым результирующего файла реализациии (`.cxx`).

Основная часть транслятора IRINA состоит из нескольких модулей на Рефале, каждый из которых реализует свой этап трансляции: препроцессирование, синтаксический и семантический анализ, редактирование межмодульных связей и, наконец, генерацию выходных файлов.

7 Заключение

Описываемая реализация всё ещё уступает по времени исполнения оригинальным реализациям Рефала, что вполне объяснимо с учётом используемого в ней интерпретируемого характера выполнения рефал-программ, тогда как большинство реализаций Рефала компилируемые. В ходе проводившегося тестирования рассматриваемая реализация работала медленнее, чем реализация Рефала-5, в 8–12 раз. Следует отметить, однако, что предыдущая InteLib-реализация Рефала проигрывала Рефалу-5 в 90–100 раз, так что достигнутые результаты можно рассматривать как обнадёживающие. Оставшийся проигрыш может быть частично компенсирован возможностью реализации части подзадач на базовом языке проекта (на Си++).

Рефал-подсистема в описанном варианте включена в библиотеку InteLib, начиная с версии 0.6.20.

Список литературы

[1] Bolshakova E. Stolyarov A. Building functional techniques into an object-oriented system // Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE, vol. 62 of Frontiers in Artificial Intelligence and Applications, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam, 2000.

[2] Сердобольский В. И. Язык РЕФАЛ и его использование для преобразования алгебраических выражений // Кибернетика. 1969. №3, стр. 45–51.

[3] Кауфман В. Ш. Языки программирования. Концепции и принципы. М., Радио и связь, 1993.

[4] Столяров А. В. Интеграция изобразительных средств альтернативных языков программирования в проекты на C++. Рукопись депонирована в ВИНИТИ РАН 06.11.2001, № 2319-В2001, Москва, 2001.

[5] Столяров А. В. Расширенный функциональный аналог языка Рефал для мультипарадигмального программирования // Л. Н. Королёв, ред., Программные системы и инструменты. Тематический сборник, выпуск 2, стр. 184–195. Издательский отдел факультета ВМиК МГУ, Москва, 2001.

[6] Столяров А. В. Библиотека InteLib — инструмент мультипарадигмального программирования. Тезисы докладов II конференции разработчиков свободных программ «На Протве», Обнинск, 25–27 июля 2005 года, стр. 56–62.

[7] Столяров А. В. Импорт вычислительной модели языка Scheme в объектно-ориентированное окружение // Сборник статей молодых учёных факультета ВМиК МГУ, № 5. М.: Издательский отдел факультета ВМиК МГУ, 2008, стр. 119–130.

[8] Stroustrup B. The Design and Evolution of C++. Addison-Wesley, 1994. 462 pages.

[9] Турчин В. Ф. Метаалгоритмический язык. // Кибернетика. 1968. N 4. Стр. 45-54.

[10] Турчин В. Ф. Программирование на языке Рефал. М.: ИПМ АН СССР, 1971.

[11] Турчин В. Ф. Эквивалентные преобразования программ на Рефале // Автоматизированная система управления строительством. М.: ЦНИПИАСС, 1974, стр. 36–68.

[12] Turchin V. F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. 1986. Vol. 8, N 3. Pg. 292-325.

[13] Turchin V. F. Refal-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, 1989.

[14] Фролова О. Г. Библиотечная реализация вычислительной модели языка Плэнер // Сборник статей молодых учёных факультета ВМиК МГУ, № 5. М.: Издательский отдел факультета ВМиК МГУ, 2008, стр. 24–33.

[15] Элджер Дж. C++: библиотека программиста. СПб.: Питер, 2002.