ISBN 978-5-901795-16-3 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ. Переславль-Залесский, 2009

УДК 681.32

А. П. Лисица, А. П. Немытых

Об одном приложении вычислений с оракулом

Аннотация. Пусть дана программа $P(d)$, реализующая частично рекурсивную функцию φ. Рассмотрим на области определения последней функцию $\mathcal{O}_P$, значением которой является путь вычисления программы P на фиксированном данном d_0. Пусть программа $Q(p,d)$ определена тогда и только тогда, когда $p = \mathcal{O}_P(d)$; причем $Q(\mathcal{O}_P(d),d) = P(d)$. Программа $Q(p,d)$, абсурдная с точки зрения ее практического вычисления на конкретных входных данных, может быть практически полезной при ее автоматическом мета-анализе. В статье показывается, как программа $Q(p,d)$ может быть использована для верификации программы $P(d)$ по постусловию. Предлагаемый метод опробован на задачах верификации протоколов когерентности кэша и других распределенных вычислительных систем.

1. Моделирование недетерминированных вычислений посредством детерминированных

Уточним понятия, уже введенные нами в аннотации. Прежде всего, будем считать, что программа P может быть недетерминированной. То есть, в процессе вычисления $P(d_0)$ на конкретных входных данных выбор очередной ветви вычисления может быть неоднозначен. Исполнитель программы P управляется оракулом, который и принимает решение о выборе конкретной ветви вычисления на данном шаге этого вычисления. Повторная попытка вычисления $P(d_0)$ (на тех же самых входных данных) может привести к результату вычисления, отличному от первого запуска этой программы. Логика оракула предсказуема лишь в одном: он всегда выбирает конечный путь вычислений, приводящий к конкретному значению $P(d_0)$ (к одному из возможных), если d_0 принадлежит области определения программы P. Таким образом, путь, выбранный оракулом, не может привести Исполнителя ни к аварийной остановке, ни к незавершенности процесса вычислений в конце этого пути. Договоримся

Работа выполнена при частичной поддержке Программы фундаментальных исследований Президиума РАН №3 «Фундаментальные проблемы системного программирования» и Российского фонда фундаментальных исследований (номера проектов: 07-07-92100-GFEN-a, 08-07-00280-a).

также, что в том случае, когда на каком-то шаге вычисления реального выбора нет (выбор ветви однозначен), оракул делает выбор из единственно возможной ветви. Далее под $\vec{d}$ будем понимать конечную последовательность $d_1, \ldots, d_n$. Пусть даны некоторые множества D и $\mathtt{Im}$, мы будем считать, что недетерминированная программа реализует недетерминированную вычислимую[1] функцию из D в $\mathtt{Im}$.[2]

Понятия абстрактной программы и пути эволюции абстрактной программы зависят от конкретной модели вычислений. Понятие конкретной программы P и конкретного пути эволюции программы P (пути вычисления на конкретных входных данных) зависят от выбора языка программирования и операционной семантики этого языка. Ниже мы даем определение Исполнителя недетерминированной программы P и оракула данной программы P в «абстрактных» терминах. В следующем разделе мы уточняем эти понятия для конкретного языка.

Пусть дана абстрактная программа $P(\vec{d})$, реализующая *недетерминированную* частично рекурсивную функцию φ. Пусть D — область определения φ, $\mathtt{Im}$ — множество значений функции φ, Π — множество возможных конечных путей эволюции программы P.

Определение: *Исполнителем абстрактной программы P назовем программу Q, определяющую* детерминированную *частично рекурсивную функцию из $\Pi \times D$ в $\mathtt{Im}$ такую, что: для всех $\vec{d_0} \in D$, если p_0 есть путь вычисления $P(\vec{d_0})$, приводящий к одному из возможных значений $P_{p_0}(\vec{d_0})$, тогда $Q(p_0, \vec{d_0}) = P_{p_0}(\vec{d_0})$. Иначе программа Q не определена и завершает свою работу в состоянии аварийной остановки.*

Определение: *Пусть дана абстрактная программа $P(\vec{d})$ и ее Исполнитель Q. Оракулом программы $P(\vec{d})$ назовем* недетерминированную *общерекурсивную функцию $\mathcal{O}_P : D \mapsto \Pi$ такую, что для всех $d \in D$ выполняется равенство: $Q(\mathcal{O}_P(\vec{d}), \vec{d}) = P_{\mathcal{O}_P(\vec{d})}(\vec{d})$.*

Замечание: *Часто необходимо различать два вида неопределенности программы P: аварийную остановку и бесконечное время эволюции. В этом случае неопределенность аварийной остановки удобно обозначить специальным символом $\perp$, не входящим в оригинальный*

[1] Частично–рекурсивную или общерекурсивную.

[2] То есть, формально, функцию из D в множество всех непустых подмножеств множества $\mathtt{Im}$. См. определение понятия недетерминированной функции в [1].

образ $\mathtt{Im}(P)$ *этой программы, и расширить этот образ до множества* $\mathtt{Im}(P) \cup \{\underline{\perp}\}$.

2. Язык Рефал-Н

Для уточнения понятий программы и пути вычисления, в качестве объектного языка мы определим язык Рефал–Н. Ниже дана грамматика, описывающая множество Рефал-Н программ. Неопределенные терминалы совпадают с соответствующими терминами функционального языка программирования Рефал [2]:

```
prog ::= entry func^*
entry ::= $ENTRY Go { sent^+ }
func ::= fn { sent^+ }

sent ::= sent_name left = right;
left ::= patt call^*
call ::= , <fn arg>: patt
patt ::= expr
right ::= expr
arg ::= expr
expr ::= texpr expr | empty
texpr ::= (expr) | symbol | var | par

var ::= s.name | t.name | e.name
par ::= #s.name | #t.name | #e.name
evar ::= e.name

symbol :: = SYMBOL    /* в смысле Рефала */
empty ::=

sent_name ::= S_i   /* где i  - порядковый номер предложения
                              в конкретной функции func  */
name ::= имя переменной в смысле Рефала
```

Обозначим $\mathtt{Vars}(\mathtt{expr})$ — множество переменных, входящих в выражение `expr`; $\mathtt{Pars}(\mathtt{expr})$ — множество параметров, входящих в выражение `expr`.

Определим дополнительные ограничения на синтаксис предложений `sent`. Для произвольного предложения

$\mathtt{S}_j\ \mathtt{patt}_0\mathtt{,}\ \mathtt{<fn}_1\ \mathtt{arg}_1\mathtt{>:}\ \mathtt{patt}_1\mathtt{,}\ \ldots\mathtt{,}\ \mathtt{<fn}_k\ \mathtt{arg}_k\mathtt{>:}\ \mathtt{patt}_k\ \mathtt{= right;}$

должны выполняться условия: для всех $0 < j \leq k.\mathtt{Vars}(\mathtt{arg}_j) \subset \bigcup_0^{(j-1)}\mathtt{Vars}(\mathtt{patt}_m)$, $\mathtt{Vars}(\mathtt{right}) \subset \bigcup_0^k \mathtt{Vars}(\mathrm{patt}_m)$; для всех $0 \leq j \leq k$ множество $\mathtt{Pars}(\mathtt{patt}_j)$ пусто.

Семантика языка Рефал–Н будет модификацией семантики Рефала (см. [2]) — в сторону недетерминизма. А именно:

- под «переменными» `var` понимаются связанные переменные в данном предложении sent; переменным можно присваивать значения, соответствующие их типу;
- под «параметрами» `par` понимаются свободные переменные в данном предложении `sent`; параметры в каждый момент работы программы имеют конкретные значения, соответствующие их типу, — это значения, которые выбираются Рефал–Н машиной недетерминированным образом;
- последовательность вызовов функций в левой части предложения представляет собой композицию этих вызовов с указанным порядком вычисления. При необходимости вычисления конкретного вызова (из стека вызовов функций):
  - если это стартовый вызов `<Go arg>` (входная точка) функции `Go`, тогда Рефал–Н машина недетерминированным образом фиксирует все параметры, входящие в аргумент `arg` и все параметры, входящие в определение функции `Go`;
  - если вызов `<F arg>` не стартовый, тогда (по предыдущему пункту) его аргумент является объектным выражением (не содержит параметров и переменных). Рефал–Н машина недетерминированным образом фиксирует все параметры, входящие в определение функции `F`. Каждому вызову в стеке соответствует своя недетерминированная выборка указанных в данном пункте параметров;
- выбор предложения функции `F` для отождествления конкретных входных данных `d` (то есть, без параметров) вызова `<F d>` с конкретным образцом $\texttt{patt}_0$ (из левой части предложения) будет происходить недетерминированным образом;
- если выбрано предложение $\texttt{S}_i$ $\texttt{patt}_0$ `call`* `= right;` для попытки отождествления и уравнение $\texttt{patt}_0$ `= d` имеет непустое множество значений, тогда результатом отождествления будет считаться одно из этих решений, которое выбирается недетерминированным образом;
- вычисление конструкции `<fn d>:`$\texttt{patt}_i$ состоит в недетерминированном выборе одного из результатов $\texttt{d}_1$ вызова `<fn d>` и решения уравнения $\texttt{patt}_i$ `=` $\texttt{d}_1$. Если это уравнение имеет

непустое множество значений, тогда результатом отождествления будет считаться одно из решений, выбранное недетерминированным образом;

- имена предложений при отождествлении игнорируются, то есть, являются комментариями и введены нами по техническим причинам.

Все другие понятия семантики языка Рефал–Н совпадают с понятиями семантики языка Рефал [2]. В частности, входной точкой программы, по определению, является вызов функции `<Go expr>`, где `expr` — входное параметризованное выражение (входные данные недетерминированной программы). Заметим, что, в отличие от Рефала, во время отождествления конкретного предложения функции `F` Рефал-Н машина обрабатывает каждый конкретный конструктор–запятую, по определению, только один раз. То есть, она никогда[3] не возвращается к этому конструктору; откаты за конструктор–запятую допускаются семантикой Рефала (см. [2]).

Пример 1. *Пусть дана входная точка* `<Go` n_0`>`, *где* n_0 — *натуральное число, представленное в унарной системе счисления*[4], *тогда нижеследующая Рефал–Н программа вычисляет* n_0*–ое число Фибоначчи.*

```
$ENTRY Go {  S1 e.n, <FibN e.n>: e.fib = e.fib; }
FibN {
S1    = I;
S2  I = I;
S3  I I e.m, <FibN  e.m>: e.x, <FibN I e.m>: e.y = e.x e.y;
}
```

Если не обращать внимание на порядковые номера предложений S_i, *то синтаксис этой программы не выходит за рамки синтаксиса языка Рефал [2], поэтому ее можно рассматривать (с точностью до* S_i*) одновременно и как Рефал программу, и как Рефал-Н программу. В данном случае Рефал–Н семантика программы полностью совпадает с Рефал семантикой. Причины этого в том, что (1) с точки зрения Рефал семантики, предложения в определении* `FibN` *можно переставлять местами, не меняя семантики* `FibN`*; (2) все отождествления однозначны — образцы программы не содержат открытых переменных (см. [2]).*

---

[3] При данном конкретном вызове функции `F`.

[4] Например, `2 = I I`. Ноль представляется пустой строкой.

Пример 2. *Если в предыдущем примере входную точку Рефал–Н программы заменить на* `<Go #e.n>`*, тогда результатом вычисления программы будет либо число Фибоначчи, номер которого выбирается недетерминированным образом; либо аварийная остановка* $\underline{\perp}$*, если недетерминированный выбор конкретного значения параметра* `#e.n` *не представляет натурального числа в унарном счислении.*

Пример 3. *Пусть дана Рефал-Н программа:*

```
$ENTRY Go {  S1 , <Fab A>: e.res = e.res; }
Fab {
S1    = ;
S2  A  e.x, <Fab e.x>: e.z = B e.z;
S3  s.y e.x, <Fab e.x>: e.z = s.y e.z;
}
```

Тогда возможны следующие результаты вычисления ее входной точки `<Go >: A` *или* `B`*.*

Пример 4. *Пусть дана Рефал–Н программа:*

```
$ENTRY Go {  S1 e.x e.y = e.x; }
```

Тогда результатом вычисления ее входной точки `<Go A B C D>` *может быть:*

- *пустое Рефал выражение;*
- `A`*;*
- `A B`*;*
- `A B C`*;*
- `A B C D`*.*

Каждый из указанных вариантов выбирается недетерминированным образом.

3. Понятие пути вычисления в языке Рефал-Н

Будем пользоваться терминологией языка Рефал [2]. Пусть даны Рефал–Н программа `P` и ее входная точка `<Go d>`. Пара (`P,<Go d>`) определяет корневое ориентированное дерево вычислений $\mathcal{T}$ — согласно правилам семантики. $\mathcal{T}$, вообще говоря, может быть бесконечным. Каждая вершина помечена состоянием Рефал–Н машины в данный момент вычисления. Корень `<Go d>` имеет только выходные ребра; все другие вершины имеют по одному входному ребру. Листья дерева

$\mathcal{T}$ помечены либо данным — результатом вычисления программы, либо символом $\perp\!\!\!\perp$ аварийной остановки (тупика). Вершины, не являющиеся листьями, также помечены либо меткой `call`, либо меткой `return`. Вершины типа `call` соответствуют точкам входа в конкретный вызов функции и являются точками ветвления. Вершины типа `return` соответствуют точкам выхода из конкретного вызова функции и также являются точками ветвления. Скажем, что вершина, не являющаяся листом, имеет имя `f`, если она соответствует вызову функции с именем `f`.

Опишем множество ребер, выходящих из вершины b типа `call`, соответствующей вызову `<f d>`, где `d` — Рефал выражение, возможно, содержащее параметры (только тогда, когда рассматриваемый вызов является корнем дерева). Рассмотрим произвольное предложение функции `f`:

$$\texttt{S}_i\ \texttt{patt}_0\ \ldots$$

и соответствующее ему уравнение с параметрами $\texttt{patt}_0$ `= d`. Обозначим σ — отображение–подстановку, сопоставляющее каждому параметру из `Pars`(d) конкретное данное, соответствующее типу этого параметра. Множество решений уравнения $\sigma(\texttt{patt}_0) = \sigma(\texttt{d})$ обозначим через $\mathcal{X}_i^\sigma$. Множество $\mathcal{X}_i^\sigma$ может быть либо пустым, либо конечным.

- Если объединение (по всем возможным подстановкам σ и предложениям функции `f`) множеств $\mathcal{X}_i^\sigma$ пусто, тогда из вершины b выходит единственное ребро, которое является входящим в лист типа $\perp\!\!\!\perp$.
- Иначе, для каждой σ, каждого i и каждого $x_0 \in \mathcal{X}_i^\sigma$ существует единственное ребро, выходящее из b. Пометим это ребро термом `((in` σ`) (`$\texttt{S}_i$ x_0`))`.

Теперь опишем множество ребер, выходящих из вершины b типа `return`, соответствующей конструкции `<f d>:` $\texttt{patt}_j$, где `d` — объектное выражение (оно, по определению семантики (см. выше), не может содержать параметров). Множество всех возможных значений вызова `<f d>` обозначим через $\mathcal{R}$. Оно может быть пустым, конечным или бесконечным. Пусть $\texttt{r}_0 \in \mathcal{R}$. Множество решений уравнения $\texttt{patt}_j = \texttt{r}_0$ обозначим через $\mathcal{X}_{\texttt{r}_0}$. Оно может быть либо пустым, либо конечным.

- Если объединение (по всем возможным $\texttt{r}_0$ из $\mathcal{R}$) множеств $\mathcal{X}_{\texttt{r}_0}$ пусто, тогда из вершины b выходит единственное ребро, которое является входящим в лист типа $\perp\!\!\!\perp$.

- Иначе, для каждого $\mathtt{r}_0 \in \mathcal{R}$ и каждого $x_0 \in \mathcal{X}_{\mathtt{r}_0}$ существует единственное ребро, выходящее из b. Пометим это ребро термом $\mathtt{(out\ (r}_0\ x_0\mathtt{))}$.

Определение: *Пусть даны Рефал–H программа* `P` *и ее входная точка* `<Go d>`*, где* `d` *— параметризованное данное Рефала. Рассмотрим* $p = (v_0, v_1), (v_1, v_2), \ldots (v_{(n-1)}, v_n)$ *— путь в дереве вычислений пары* (`P, <Go d>`)*, выходящий из корня* v_0 *и заканчивающийся в некотором листе* v_n*. Каждому ребру* $(v_i, v_{(i+1)})$ *сопоставим терм* a_i *по следующему правилу:*

- *если* v_i *— вершина типа* `call` *— есть точка ветвления вызова* `<f d>`*, тогда* $a_i =$ `(f ((in` σ`) (`S_i x_0`)))`*, где* `((in` σ`) (`S_i x_0`))` *есть метка ребра* $(v_i, v_{(i+1)})$*;*
- *если* v_i *— вершина типа* `return` *— есть точка выхода из вызова* `<f d>: patt`*, тогда* $a_i =$ `(f (out` x_0`))`*, где* `(out (r`$_0$ x_0`))` *есть метка ребра* $(v_i, v_{(i+1)})$*.*

Пусть $v_n \neq \perp$*. Путем вычисления пары* (`P, <Go d>`) *назовем последовательность термов* $a_0, \ldots, a_{(n-1)}$*.*

Замечание: *Множество путей вычисления пары* (`P, <Go d>`) *может быть конечным, бесконечным или пустым. Каждый путь вычисления является конечной последовательностью и представляет собой историю конкретного вычисления.*

Замечание: *Конкретный путь вычисления однозначно определяет результат каждого вызова функции. Следовательно, данное нами определение пути вычисления корректно.*[5]

Пример 5. *Пусть входная точка программы* `FibN` *(см. параграф 1) есть* `<Go >`*, где*

```
$ENTRY Go {  S1 , <FibN I I>: e.res = e.res; }
```

тогда путь вычисления, соответствующий указанным программе и входной её точке, есть:

$a_0 =$ `(Go ((in ) (`S_1`)))`,
$a_1 =$ `(FibN ((in ) (`S_3 `(e.m := ))))`,
$a_3 =$ `(FibN ((in ) (`S_1 `())))`,
$a_4 =$ `(FibN (out) (e.x := I)))`,

[5]Мы не включили в определение метки $\mathtt{r}_0$, помечающие исходящие ребра вершин типа `return`.

$a_5 =$ `(FibN ((in ) (`S_2 `())))`,
$a_6 =$ `(FibN (out) (e.y := I)))`
Здесь для всех a_i *подстановки* σ *тривиальны и обозначены пустыми выражениями, в* a_3 *и* a_5 *пустыми скобками* `()` *обозначены тривиальные решения.*

4. Верификация с предусловием по постусловию

Пусть даны Рефал–Н программа $P(d)$, ее Исполнитель Q (см. 1) и оракул $\mathcal{O}_P$. Пусть D — область определения программы P, `Data` — множество данных Рефала. Пусть G есть множество `Data` $\cup \{\underline{\perp}\}$.

Определение: *Пусть в множестве* D *выделена некоторая константа* a. *a–тождественной характеристической функцией множества* $B \subset$ `Data` $\setminus \{a\}$ *назовем* общерекурсивную *функцию* $\mathcal{I}_a$: `Data` $\setminus \{a\} \mapsto G$ *такую, что если* $x \in B$, *тогда* $\mathcal{I}_a(x) = x$, *иначе* $\mathcal{I}_a(x) = a$.

Ниже в данном параграфе, функции, имена которых помечены нижним индексом a, будут обозначать a–тождественные характеристические функции. Рассмотрим $\psi_{\underline{\perp}}$, ξ_a — *имярек*-тождественно характеристические функции множеств $A \subset D, Data \setminus B$ соответственно.

Пусть константа `false` не принадлежит образу программы `P`. Под задачей верификации программы `P` с предусловием $\psi_{\underline{\perp}}$ по постусловию $\xi_{\texttt{false}}$ мы будем понимать необходимость доказательства следующего свойства:

$$\forall d \in D.\ \xi_{\texttt{false}}(\texttt{P}(\psi_{\underline{\perp}}(d))) = d' \neq \texttt{false}.$$

Если сформулированное выше утверждение верно, тогда скажем, что программа `P` корректна по предусловию $\psi_{\underline{\perp}}$ и постусловию $\xi_{\texttt{false}}$.

Если $\psi_{\underline{\perp}}$ и $\xi_{\texttt{false}}$ заданы в виде *детерминированных* программ, то можно попытаться поручить решить эту задачу *детерминированной* мета-программе M, анализирующей композицию $\xi_{\texttt{false}}(\texttt{P}(\psi_{\underline{\perp}}(\#d)))$ в контексте определений `P` и $\psi_{\underline{\perp}}, \xi_{\texttt{false}}$, где $\#d$ — параметр, пробегающий множество Рефал данных.

Мы хотим поставить эту задачу для мета–программы M, которая способна анализировать только *детерминированные* программы.

Для всех Рефал данных $d \in D$ имеем равенство

$$Q(\mathcal{O}_P(d), d) = P_{\mathcal{O}_P(d)}(d).$$

Следовательно, для всех $d \in D$

$$\xi_{\texttt{false}}(Q(\mathcal{O}_P(d), \psi_{\underline{\perp}}(d))) = \xi_{\texttt{false}}(P_{\mathcal{O}_P(d)}(\psi_{\underline{\perp}}(d))).$$

Рассмотрим композицию $\xi_{\texttt{false}}(Q(p, \psi_{\underline{\perp}}(d)))$. По определению программ Q и $\xi_{\texttt{false}}$, $\psi_{\underline{\perp}}$ вычисление этой композиции всегда завершается, но, возможно, в аварийном состоянии Рефал машины (напомним, что все эти программы *детерминированные*). Таким образом, если M сможет доказать равенство $\xi_{\texttt{false}}(Q(p, \psi_{\underline{\perp}}(d))) = d' \neq \texttt{false}$ для любого $p \in \Pi$ и любого $d \in D$, тогда для любого $d \in D$ имеет место равенство $\xi_{\texttt{false}}(P(\psi_{\underline{\perp}}(d))) \neq \texttt{false}$.

И поставленная в начале данного параграфа задача будет решена.

Ниже, в параграфе 7, мы покажем, что описанная нами схема успешно может быть применена для верификации реальных распределенных недетерминированных вычислительных систем, *работающих бесконечное время*.

5. Двойственные вычислительные системы

В этом параграфе мы определим понятия двойственной недетерминированной вычислительной системы S' по отношению к данной недетерминированной вычислительной системе S и двойственной задачи верификации системы S' по отношению к данной задаче верификации системы S.

Пусть дана некоторая абстрактная недетерминированная вычислительная система S. Под двойственной к S системой S' мы будем понимать вычислительную систему, все шаги эволюции которой противоположны шагам эволюции системы S. То есть, если система S из состояния q переходит в состояние q', тогда система S' из состояния q' переходит в состояние q.

Определение: *Пусть* $D \subset \texttt{Data}$. *Генератором множества* D *назовем недетерминированную программу* $\gamma_D : \epsilon \mapsto D$, *образ которой совпадает с* D.[6] *Параметризованное выражение* ***expr*** *определяет некоторое подмножество в множестве данных Рефала* Data. *Обозначим генератор этого подмножества через* $\gamma_{\texttt{expr}}$.

Уточним это понятие в конкретных терминах языка Рефал–Н. Во–первых, мы построим граф вычислений данной программы P,

[6] Здесь ϵ — пустое Рефал выражение.

который представляет дерево вычислений программы P в виде, симметричном относительно входа и выхода P.

Пусть даны Рефал–Н программа P и ее входная точка `<Go expr>`. В параграфе 3 мы описали дерево вычислений $\mathcal{T}$, определяемое этой парой. Корень дерева $\mathcal{T}$ определяет входные данные вычислительной системы S, соответствующей программе P; листья дерева $\mathcal{T}$ — выходные данные системы S. Легко видеть, что посредством переопределения P можно ограничиться входными точками вида `<Go #e.n>`, не меняя основную часть структуры дерева вычислений $\mathcal{T}$. Для этого достаточно переопределить функцию `Go` к виду:

```
$ENTRY Go {
  s1 patt, <Out arg>: patt1 = right;
}
```

Где `patt` — образец, построенный из `expr` заменой всех параметров на одноименные им переменные; `Out` — вспомогательная функция, передающая управление другим функциям из P посредством одного шага вычисления программы (одного ребра в дереве вычислений программы), $\texttt{patt}_1$ — образец, принимающий результат вычисления вызова функции `Out`. Ниже, если не оговорено противное, мы ограничимся входными точками вида `<Go #e.n>` и определением функции `Go` описанного вида. Назовем программы, определение `Go` которых удовлетворяет описанным синтаксическим свойствам, нормализованными.

Все выходные ребра из `return`–вершин $\texttt{<Out arg>: patt}_1$ дерева вычислений $\mathcal{T}$ являются входными ребрами листов. Причем, для каждого листа b дерева $\mathcal{T}$, метка которого не равна $\perp$, существует `return`–вершина $\texttt{<Out arg>: patt}_1$ такая, что входное ребро листа b является выходным ребром вершины $\texttt{<Out arg>: patt}_1$. Обозначим множество возможных значений вызова `<Go #e.n>` через $\mathtt{Im}_P$. Склеим все листы дерева, метки которых не равны $\perp$, и полученную таким образом вершину пометим как $\texttt{<Out } \gamma_{\mathtt{Im}_P}()\texttt{>}$. Построенный граф назовем графом вычислений пары `(P,<Go #e.n>)`. Для полной симметричности входа и выхода далее входную точку программы будем представлять следующим образом: $\texttt{<Go } \gamma_{\#\mathtt{e.n}}()\texttt{>}$.

Рассмотрим следующее формальное преобразование $\mathcal{W}$ нормализованной Рефал–Н программы P.

$$\mathcal{W}(\texttt{entry func}^*) = \mathcal{W}(\texttt{entry})\ \ \mathcal{W}(\texttt{func})^*$$

$$\mathcal{W}(\texttt{\$ENTRY Go \{ sent}^+\ \texttt{\}}) = \texttt{\$ENTRY Go \{ }\mathcal{W}(\texttt{sent})^+\ \texttt{\}}$$
$$\mathcal{W}(\texttt{fn \{ sent}^+\ \texttt{\}}) = \texttt{fn}'\ \texttt{\{ }\mathcal{W}(\texttt{sent})^+\ \texttt{\}}$$

$\mathcal{W}(\mathtt{S_j\ patt_0}, \mathtt{<fn_1\ arg_1>: patt_1}, \ldots, \mathtt{<fn_k\ arg_k>: patt_k} = \mathtt{right};)$
$= \mathtt{S_j}\ \mathcal{U}(\mathtt{right}, \mathtt{<fn'_k\ patt_k>: arg_k}, \ldots, \mathtt{<fn'_1\ patt_1>: arg_1} = \mathtt{patt_0});$

где отображение $\mathcal{U}$ преобразует некоторые переменные и параметры по правилу:

- Для всех j таких, что $0 \leq j < k$, для всех переменных
$$v \in \mathtt{Vars}(\mathtt{patt}_j) \setminus (\mathtt{Vars}(\mathtt{right}) \cup \bigcup_{m=j+1}^{k} \mathtt{Vars}(\mathtt{arg}_m))$$
преобразовать v в параметр со «свежим» именем и типом, совпадающим с типом переменной v.
- Для всех параметров $p \in \mathtt{Pars}(\mathtt{right})$ преобразовать p в переменную со «свежим» именем и типом, совпадающим с типом параметра p.
- Для всех j таких, что $0 < j \leq k$, для всех параметров $p \in \mathtt{Vars}(\mathtt{arg}_j)$ преобразовать p в переменную со «свежим» именем и типом, совпадающим с типом параметра p.

Здесь каждое преобразование конкретной переменной/параметра происходит: (1) последовательно по убыванию индекса j[7] и (2) одновременно по всем вхождениям этой переменной/параметра в рассматриваемое предложение `sent`.

Программу $P' = \mathcal{W}(P)$ назовем программой обратной/двойственной к программе P.

Пример 6. *Нижеследующая Рефал–Н программа P с входной точкой* `<Go #n>` *является нормализованной версией программы из примера 2 (параграф 2).*

```
$ENTRY Go {  S1 e.n, <Out e.n>: e.out = e.out; }
Out {  S1 e.n, <FibN e.n>: e.fib = e.fib; }

FibN {
S1    = I;
S2  I = I;
S3  I I e.m, <FibN  e.m>: e.x, <FibN I e.m>: e.y = e.x e.y;
}
```

Двойственная ей программа P' выглядит следующим образом:

[7] То есть, слева направо.

```
$ENTRY Go {  S1 e.out, <Out' e.out>: e.n = e.n; }
Out' {  S1 e.fib, <FibN' e.fib>: e.n = e.n; }

FibN' {
S1  I = ;
S2  I = I;
S3  e.x e.y, <FibN'  e.y>: I e.m, <FibN' e.x>: e.m = I I e.m;
}
```

Возможные результаты вычисления входной точки `<Go I>` *программы* P'*:*

- *Путь вычисления* `(Go ((in I) (S1 (e.out := I)))`
  `(Out' ((in I) (S1 (e.fib := I)))) (FibN' ((in I) (S1 ())))`
  `(FibN' (out) (e.n := )) (Out' (out) (e.n := ))` *соответствует результату* `<Go I> =` [8].
- *Путь вычисления* `(Go ((in I) (S1 (e.out := I)))`
  `(Out' ((in I) (S1 (e.fib := I)))) (FibN' ((in I) (S2 ())))`
  `(FibN' (out) (e.n := I)) (Out' (out) (e.n := I))` *соответствует результату* `<Go I> = I`.
- *Все пути вычисления, проходящие через третье предложение* S_3 *функции* `FibN'`*, приводят к результату отождествления со значением пустого выражения либо переменной* `e.x`*, либо переменной* `e.y`*. И, следовательно, к бесконечному пути вычисления; ибо выхода по пустому выражению определение функции* `FibN'` *не содержит.*

Для любой нормализованной Рефал–Н программы P верны следующие утверждения:

Утверждение 1: *С точностью до переименования функций, переменных и параметров выполняется равенство* $P'' = P$.

Утверждение 2: *Пусть* $p = (v_0, v_1), (v_1, v_2), \ldots (v_{(n-1)}, v_n)$ *— путь в дереве вычислений пары* $(\mathtt{P}, <\mathtt{Go\ d}>)$*, выходящий из корня* v_0 *и заканчивающийся в некотором листе* v_n*, не отмеченном как* $\underline{\perp}$*. Тогда* $p' = (v_n, v_{(n-1)}) \ldots (v_2, v_1), (v_1, v_0)$ *есть путь в дереве вычислений пары* $(\mathtt{P}', <\mathtt{Go}\ \gamma_{\mathtt{Im_P}}()>)$*, выходящий из корня* v_n *и заканчивающийся в листе* v_0*, не отмеченном как* $\underline{\perp}$*. (В случае графа вычислений,* p *есть*

[8] Пустое выражение.

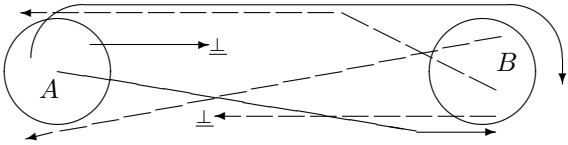

Рис. 1. Пути прямой и двойственной систем, соответствующие прямой и двойственной задачам верификации

путь из вершины <Go d> *в вершину* <Out $\gamma_{\mathtt{Im_P}}()$>, *а* p' *есть путь из вершины* <Out $\gamma_{\mathtt{Im_P}}()$> *в вершину* <Go d>.)

Утверждение 3:
Пара (P′,<Go $\gamma_{\mathtt{Im_P}}()$>) *определяет отношение на* Data × Data *обратное к отношению, определяемому парой* (P,<Go d>).

6. Задача верификации, двойственная к данной

Пусть P — программа, для которой существует двойственная программа P'. Пусть $\mathtt{D}_P$ — область определения программы P, а $\mathtt{Im}_P$ — ее образ. Пусть дано некоторое множество $B \subset \mathtt{Im}_P$. Рассмотрим $\psi_{\underline{\perp}}, \xi_a$ — *имярек*-тождественно характеристические функции множеств $A \subset \mathtt{D}_P$, $\mathtt{Im}_P \setminus B$ соответственно.

Пусть константа $\mathtt{false} \notin \mathtt{D}_P \cup \mathtt{Im}_P$. Пусть дана задача $\mathcal{Z}$ верификации программы P с предусловием $\psi_{\underline{\perp}}$ по постусловию $\xi_{\mathtt{false}}$. Задачей $\mathcal{Z}'$ — двойственной к задаче $\mathcal{Z}$ — назовем задачу верификации программы P' с предусловием $\xi'_{\underline{\perp}}$ по постусловию $\psi'_{\mathtt{false}}$, где $\xi'_{\underline{\perp}}$ есть тождественно характеристическая функция множества B, а $\psi'_{\mathtt{false}}$ — тождественно характеристическая функция множества $\mathtt{D}_P \setminus A$.

Утверждение: *Программа* P *корректна по предусловию* $\psi_{\underline{\perp}}$ *и постусловию* $\xi_{\mathtt{false}}$ *тогда и только тогда, когда программа* P' *корректна по предусловию* $\xi'_{\underline{\perp}}$ *и постусловию* $\psi'_{\mathtt{false}}$.
Доказательство: Предположим обратное — программа P' корректна, а программа P не корректна по указанным в утверждении условиям. Тогда в графе вычислений пары $(P, <\mathtt{Go}\ \gamma_A()>)$ существует путь p из вершины <Go $\gamma_A()$> в вершину <Out $\gamma_B()$>. Но тогда в графе

вычислений пары $(P', <\texttt{Go}\ \gamma_B()>)$ путь p' (двойственный к p) начинается в вершине $<\texttt{Go}\ \gamma_B()>$ и заканчивается в вершине $<\texttt{Out}\ \gamma_A()>$. Что противоречит сделанному предположению. □

Таким образом, решив задачу верификации $\mathcal{Z}$, мы имеем решение двойственной ей задачи $\mathcal{Z}'$ и наоборот. Если задача $\mathcal{Z}$ поставлена перед мета–программой M, анализирующей программу P и ее параметризованную входную точку, тогда цель программы M — доказать, что не существует путей эволюции этой пары из множества A в множество B. Попытка решения двойственной задачи $\mathcal{Z}'$ должна доказать отсутствие путей эволюции соответствующей ей пары из множества B в множество A. Таким образом, в задаче $\mathcal{Z}$, по существу, исследуются существующие пути из A в $\mathtt{Im}_P \setminus B$, а в задаче $\mathcal{Z}'$ — пути из B в $D \setminus A$. Два указанных здесь множества путей, рассматриваемых с точностью до ориентации, не пересекаются. Тем самым для программы M одна из задач $\mathcal{Z}$, $\mathcal{Z}'$ может оказаться значительно проще другой задачи. То есть, построенная нами конструкция может иметь практическое значение.

7. Результаты экспериментов

В работах [3–7] мы показали, что описанная нами в параграфе 4 задача верификации может быть успешно решена для класса параметризованных недетерминированных протоколов кэш когерентности мультипроцессорных систем [8–10] автоматическим специализатором программ [11, 12]. В наших экспериментах[9] мы использовали в качестве такого специализатора суперкомпилятор SCP4 (см. [13–18]) программ, написанных на функциональном языке программирования Рефал–5 [2, 19].

Недетерминированные протоколы кэш когерентности моделировались нами по схеме, данной нами в параграфе 1. Описание этих протоколов и деталей их кодировки в виде программ на языке Рефал–5 можно найти в работах [6, 9, 10]. Отметим, что протоколы кэш когерентности представляют собой вычислительные системы, работающие бесконечно долго. Под параметризацией, в данном случае, понимается параметризация по числу процессоров. Условие корректности формулируется в терминах недостижимости определенного

[9]Результаты этих и других успешных экспериментов представлены на Интернет странице [6].

вида глобальных состояний соответствующей вычислительной системы, работа которой может начинаться из определенного множества ее стартовых состояний. Условие принадлежности к этому множеству и является предусловием задачи верификации. Возможность автоматического решения этой задачи, конечно, зависит от свойств применяемого специализатора, который специализирует рассматриваемую систему по множеству допустимых стартовых конфигураций.

Говоря точнее, для кодировки этих протоколов использовался ограниченный Рефал (см. [19]) — некоторый фрагмент языка Рефал–5. В ограниченном Рефале не допускается использование конструктора *запятой* (см. параграф 2 и [19]) и наложены дополнительные синтаксические ограничения на использование образцов, которые обеспечивают единственность или отсутствие решений уравнений, возникающих при отождествлении с образцами. Кроме того, ограниченный Рефал, как подмножество Рефала–5, конечно, является детерминированным языком программирования. Данные свойства ограниченного Рефала не позволяют использовать напрямую нашу синтаксическую конструкцию построения двойственной программы к данной программе (параграф 5).

Мы закодировали двойственные вычислительные системы к ряду протоколов кэш когерентности вручную, оставаясь в рамках семантической идеи (см. параграф 5). Позволим себе повторить ее еще раз.

Пусть дана некоторая абстрактная недетерминированная вычислительная система S. Под двойственной к S системой S' мы будем понимать вычислительную систему, все шаги эволюции которой противоположны шагам эволюции системы S. То есть, если система S из состояния q переходит в состояние q', тогда система S' из состояния q' переходит в состояние q.

Суперкомпилятор SCP4 смог удачно решить двойственные задачи верификации для следующих параметризованных протоколов кэш когерентности — Synapse N+1 [9, 20], MSI [21], MOSI [22], MESI [9, 20], MOESI [9, 20], The University of Illinois [9, 20], Berkley [9, 20], DEC Firefly [9, 20], IEEE Futurebus+ [9, 20], Xerox PARC Dragon [10]; а также и других параметризованных вычислительных систем — Java Meta–Locking Algorithm [23], Reader–Writer protocol [24], Steve German's directory–based consistency protocol [25], Load Balancing Monitor protocol, Data Race Free Synchronization Model [10], Control Server

Monitor protocol [8].[10] Фактически, эксперименты по автоматическому решению двойственных задач верификации по отношению к данным задачам оказались успешными *для всех закодированных нами двойственных вычислительных систем*. Но ручная кодировка этих систем является трудоемким процессом.

8. Заключение

Насколько нам известно, интерес исследователей к обращению эволюции вычислительных систем проявлялся, в основном, с двух точек зрения. Физики рассматривали задачу построения обращаемых по времени вычислительных машин как наиболее энергосберегаемых (см., например [26]). Программистов интересовал вопрос автоматического построения эффективной программы, вычисляющей обратный алгоритм по отношению к данному алгоритму. Если отбросить требование эффективности, то классическим примером языка программирования, поддерживающего «вычислительно неориентированный» стиль мышления, является логический язык Пролог. Одна и та же программа на языке Пролог описывает одновременно прямое и обратное отношение на декартовом произведении множеств определения ее аргументов. Важно подчеркнуть, что семантика программы совершенно симметрична по отношению к прямому и обратному отношению[11]. Так как Пролог позволяет описывать только предикаты, то оба этих отношения только *выделяются* их характеристическими функциями, а *не вычисляются* — даже тогда, когда отношение на паре множеств (X, Y) представляет собой график взаимно–однозначной функции f.

Интересной задачей является построение по данной программе, реализующей функцию f в терминах функционального языка программирования, программы, вычисляющей обратную к f функцию, если f^{-1} существует, или, иначе, обратное отношение к f — в терминах того же языка программирования. Этой задачей занимались достаточно много авторов. Мы отметим здесь обращаемый по времени[12] язык программирования Janus, разработанный в 1982 году для

[10] Описание всех этих протоколов можно найти также на нашей электронной странице [6].

[11] Мы вынуждены оговориться: если эта программа не использует конструкцию `CUT`.

[12] time–reversible

студенческих экспериментов в Калифорнийском институте технологии (Caltech) [27]. Любые попытки решения этой задачи приводят к симбиозу понятий логических и функциональных языков программирования (см., например, [28]). С нашей точки зрения, большой интерес представляют работы С. Абрамского по обращению вычислений (например, [29]). В контексте языка Рефал в этом направлении активно работал А.Ю. Романенко [30, 31]. По сути дела, он предпринимал попытки построения функционально–логического языка, основанного на понятиях Рефала, включающего в свою трансформационную семантику элементы суперкомпиляции. К сожалению, А. Романенко не имел в своем распоряжении какого–либо приемлемого преобразователя–оптимизатора программ, который бы позволил ему проводить интересные эксперименты. В последнее время интерес к обращению вычислений возник в связи с теорией квантовых вычислений [32].

Нам неизвестны какие–либо работы, связанные с обращением вычислений в контексте верификации недетерминированных вычислительных систем. Недетерминированность вычислений снимает некоторые проблемы обращения вычислений. Конструкция, описанная нами в данной статье, как мы показали на экспериментах, иногда позволяет автоматически верифицировать программные кодировки недетерминированных вычислительных систем; формально обращать их на уровне синтаксиса и ставить задачу верификации обратной вычислительной системы, решение которой приводит к автоматическому решению прямой задачи верификации. При этом задачи верификации решались нами автоматически посредством суперкомпилятора SCP4. Кодировка вычислительной системы, обратной к данной, которую мы назвали двойственной, строилась нами вручную. Задача состоит в автоматизации этой кодировки на основе конструкции, которую мы описали в данной работе.

Список литературы

[1] Лялин И. В. *О решении автоматных уравнений*: Дискрет. матем. Т. **16:2**, 2004, с. 104–116. ↑2

[2] Turchin V. F., Turchin D. V., Konyshev A. P., Nemytykh A. P. Refal–5: sources, executable modules, 2000, http://www.botik.ru/pub/local/scp/refal5/. ↑2, 1, 3, 7

[3] Лисица А. П., Немытых А. П. *Верификация как параметризованное тестирование (эксперименты с суперкомпилятором SCP4)*: «Программирование». — Т. **33(1)**. — Москва, 2007, с. 22–43. ↑7

[4] Lisitsa A. P., Nemytykh A. P. *Verification of parameterized systems using supercompilation. A case study* // Of APPSEM05 ред. Hofmann M., Loidl H. W. — Germany, 2005, Accessible via: ftp://www.botik.ru/pub/local/scp/refal5/appsem_verification2005.ps. ↑

[5] Lisitsa A. P., Nemytykh A. P. *Verification via Supercompilation*: International Journal of Foundations of Computer Science. — Т. **19(4)**, 2008, с. 953–970. ↑

[6] Lisitsa A. P., Nemytykh A. P. Experiments on verification via supercompilation, 2007, http://refal.botik.ru/protocols/. ↑7, 9, 10

[7] Lisitsa A. P., Nemytykh A. P. *Verification as Specialization of Interpreters with Respect to Data* // Of First International Workshop on Metacomputation in Russia. — Pereslavl-Zalessky, 2008, с. 94–112. ↑7

[8] Begin L. The BABYLON Project: A Tool for Specification and Verification of Parameterized Systems to Benchmark Infinite-State Model Checkers, http://www.ulb.ac.be/di/ssd/lvbegin/CST/. ↑7

[9] Delzanno G. *Automatic Verification of Parameterized Cache Coherence Protocols* // Of the 12th Int. Conference on Computer Aided Verification: LNCS. — Т. **1855**: Springer–Verlag, 2000, с. 53–68. ↑7

[10] Delzanno G. Automatic Verification of Cache Coherence Protocols via Infinite-state Constraint–based Model Checking, http://www.disi.unige.it/person/DelzannoG/protocol.html. ↑7

[11] Jones N. D., Gomard C. K., Sestoft P. Partial Evaluation and Automatic Program Generation: Prentice Hall International, 1993. ↑7

[12] Nemytykh A. P. *On the Place of Supercompilation inside Program Specialization* // Of First International Workshop on Metacomputation in Russia. — Pereslavl-Zalessky, 2008, с. 131–144. ↑7

[13] Корлюков А. В. Пособие по суперкомпилятору SCP4, 1999, http://www.refal.net/supercom.htm. ↑7

[14] Немытых А. П. Суперкомпилятор SCP4: общая структура. — Moscow: URSS, 2007. ↑

[15] Nemytykh A. P. *The Supercompiler SCP4: General Structure (extended abstract)* // Of the Perspectives of System Informatics: LNCS. — Т. **2890**: Springer–Verlag, 2003, с. 162–170, Accessible via:ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_PSI03.ps.gz. ↑

[16] Nemytykh A. P. *The Supercompiler SCP4: General Structure*: Программные системы: теория и применение. — Т. **1**. — Москва: Физматлит, 2004, с. 448–485, Available at ftp://ftp.botik.ru/pub/local/scp/refal5/GenStruct.ps.gz. ↑

[17] Nemytykh A. P., Turchin V. F. The Supercompiler SCP4: sources, on–line demonstration, 2000, http://www.botik.ru/pub/local/scp/refal5/. ↑

[18] Turchin V. F. *Theconceptofasupercompiler*: ACM Transactions on Programming Languages and Systems. — Т. **8**: ACM Press, 1986, с. 292–325. ↑7

[19] Turchin V. F. Refal–5, Programming Guide and Reference Manual. — Holyoke, Massachusetts: New England Publishing Co., 1989,
(electronic version: http://www.botik.ru/pub/local/scp/refal5/ ,2000). ↑7

[20] Delzanno G. Automatic Verification of Parameterized Cache Coherence Protocols, 2000, ftp://ftp.disi.unige.it/person/DelzannoG/papers/ccp.ps.gz. ↑7

[21] Emerson E. A., Kahlon V. *Rapid Parameterized Model Checking of Snoopy Cache Coherence Protocols* // Of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: LNCS. — T. **2619**: Springer–Verlag, 2003, с. 114–159. ↑7

[22] Martin M. M. K. Token Coherence: Ph.D. dissertation: University of Wisconsin, 2003. — Page 19 с., http://www.botik.ru/pub/local/scp/refal5/. ↑7

[23] Roychoudhury A., Ramakrishnan I. V. *Inductively Verifying Invariant Properties of Parameterized Systems*: Automated Software Engineering. — T. **11**, 2004, с. 101–139. ↑7

[24] Fribourg L. *Petri Nets, Flat Languages and Linear Arithmetic* // Of the 9th Int. Workshop. on Functional and Logic Programming, 2000, с. 344–365. ↑7

[25] Delzanno G., Bultan T. *Constraint-based Verification of Client-Server Protocols* // Of the 7th International Conference on Principles and Practice of Constraint Programming: LNCS. — T. **2239**: Springer–Verlag, 2001, с. 286–301. ↑7

[26] Bennett C. H. *Logical reversibility of computation*: IBM J. Res. Develop. T. **17**, 1973, с. 525–532. ↑8

[27] Lutz Ch., Derby H. Janus: a time-reversible language: Ph.D. dissertation: California Institute of Technology, 1982, Unpublished report. ↑8

[28] Klimov Yu. A., Orlov A. Y. *XSG: Fair Language with Built-in Equality* // Of First International Workshop on Metacomputation in Russia. — Pereslavl-Zalessky, 2008, с. 85–93. ↑8

[29] Abramsky S. *A Structural Approach To Reversible Computation*: Theoretical Computer Science. — T. **347(3)**, 2005, с. 441–464. ↑8

[30] Romanenko A. Y. *The generation of inverse functions in Refal*: Partial Evaluation and Mixed Computation ред. Bjørner D., Ershov A. P., Jones N. D. — North-Holland, 1988, с. 427–444. ↑8

[31] Romanenko A. Y. *Inversion and metacomputation*: ACM SIGPLAN Notices. — T. **26(9)**: ACM, 1991, с. 12–22. ↑8

[32] Vitányi P. *Time, space, and energy in reversible computing* // Of the 2nd Conference on Computing Frontiers. — Ischia, Italy: ACM, 2005, с. 435–444. ↑8

The University of Liverpool

Исследовательский центр мультипроцессорных систем ИПС РАН

A. P. Lisitsa, A. P. Nemytykh. *On one application of computations with oracle* // Proceedings of Program Systems institute scientific conference "Program systems: Theory and applications". — Pereslavl-Zalesskij, v. **1**, 2009. — p. 245–265. — ISBN 978-5-901795-16-3 (*in Russian*).

Abstract. Let a program $P(d)$ implementing a partial recursive function φ be given. Let us consider a function $\mathcal{O}_P$ defined on the domain of φ. Fixed a data d_0, let the function $\mathcal{O}_P$ map d_0 to the path of computation of the program P on the fixed data d_0. Let a program $Q(p, d)$ be defined iff $p = \mathcal{O}_P(d)$ and let the value of the program be $Q(\mathcal{O}_P(d), d) = P(d)$. The program $Q(p, d)$, which is totally absurd with point of view of its practical computation on concrete input data, might be practically useful when it analyzed by a metaprogram. In the paper we show how the program $Q(p, d)$ can be used with a goal to verify a postcondition imposed on the program $P(d)$. The method suggested in the paper was tested on the verification tasks of cache coherence protocols and other distributed computing systems.