СМЕШАННЫЕ ВЫЧИСЛЕНИЯ: ПОТЕНЦИАЛЬНЫЕ ПРИМЕНЕНИЯ И ПРОБЛЕМЫ ИССЛЕДОВАНИЯ

Резюме. *Смешанные вычисления – это неполная обработка информации. Их продуктом является частично обработанная информация и так называемая остаточная программа, которой надлежит впоследствии завершить обработку остальной информации. Многие виды практической работы с программами оказываются не чем иным, как получением остаточной программы. В качестве примера мы показываем применение смешанных вычислений в трансляции. Затем мы изучаем способы описания смешанных вычислений. При вычислительном подходе смешанные вычисления обобщают операционную семантику языка включением шагов генерации команд остаточной программы. При трансформационном подходе остаточная программа получается как результат серии так называемых базовых трансформаций программного текста. Мы аргументируем точку зрения, согласно которой трансформационный подход является более фундаментальным, поскольку позволяет описать разнообразие смешанных вычислений и, главное, связать смешанные вычисления с другими видами систематической работы с программами: исполнением, оптимизацией, макрообработкой и синтезом. Такой интегрированный подход приводит к концепции трансформационной машины.*

1. ВВЕДЕНИЕ

Основная идея. Характерным для этой конференции является стремление объединить теорию и практику. С этой точки зрения предмет изложения является убедительной иллюстрацией к тому актуальному тезису, что программирование весьма нуждается в хорошей теории, в то время как теория уже сейчас может очень хорошо помочь практике. Автор непосредственно работает над смешанными вычислениями порядка трех лет и не перестает восхищаться возможностями этого емкого понятия. Смешанные вычисления дают нам идеальный пример того, как абстрактная концепция оказывается ключом к раскрытию сущности многих вещей в программировании. Это особенно относится к трансляции, в которой смешанные вычисления поистине являются стержневым понятием.

Есть два подхода к трактовке смешанных вычислений: один можно назвать функциональным, другой операционным.

Функциональный подход. Пусть у нас есть программа p, вычисляющая функцию $f(X, Y)$ (если X – переменная, то x – ее значение). Найти программу p_x, вычисляющую функцию $g(Y) = f(x, Y)$. Программу p_x назовем остаточной программой или проекцией p на x, а процесс ее получения – смешанным вычислением, поскольку в нем, по-видимому, чисто вычислительные шаги, связанные с обработкой информации x, перемежаются с шагами конструирования (генерации) остаточной программы.

Операционный подход. Пусть программа p задает некоторое множество C элементарных вычислительных актов и пусть задано разбиение μ этого множества на два составляющих $\mu : C' = C' \cup C''$. Одно из них, C', назовем множеством допустимых вычислений, а другое, C'' , – множеством задержанных вычислений. Смешанные вычисления над программой p при разбиении – это процесс выполнения допустимых вычислений и формирования остаточной программы p_μ, имеющей своим множеством вычислений множество C'' задержанных вычислений.

Операционный подход, вообще говоря, является более общим, нежели функциональный, во всяком случае последний легко выражается через первый. Действительно, пусть в программе $p(X, Y)$ задано значение x первого аргумента, а второй оставлен свободным. Тогда множество допустимых вычислений C' – это те, которые зависят только от X, а дополнение к ним образует множество задержанных вычислений C''. Можно наоборот: определить C'' как множество задержанных вычислений, которые каким бы то ни было способом зависят от Y, и взять C' как дополнение к нему. В любом случае соответствующая остаточная программа будет вычислять функцию $f(x, Y)$.

Примеры. Продолжая нашу неформальную трактовку, мы приведем два примера конкретного подхода к организации смешанных вычислений. В обоих примерах исходной программой будут программы, вычисляющие x^n (n целое) с использованием двоичного разложения показателя, но записанные в разных формализмах. Также в обоих случаях цель смешанных вычислений – получить программу для x^5, но организация смешанных вычислений будет различна: в первом примере мы будем применять вычислительный подход, а во втором – трансформационный.

Вычислительный подход будет продемонстрирован в формализме программ с памятью. Для нашей функции x^n естественной будет следующая программа:

y := 1;
<u>while</u> n > 0 <u>do while even</u> n <u>do</u> n : = n/2; x : = x² <u>od</u>; n : = n - 1;
y := y × x od

Контроль, выполняя программу, извлекает из нее последовательность выполняемых элементарных вычислительных актов. Для детерминированных вычислений эта последовательность однозначно определяется значениями исходных данных; в данном случае она не зависит от *x*, но зависит от *n* и для $n = 5$ и любого *x* будет иметь вид

$y := 1;\ n > 0^+;\ \underline{even}\ n^-;\ n := n - 1;\ y := y \times x;\qquad (n = 4);$
$n > 0^+;\ \underline{even}\ n^+;\ n := n / 2;\ x := x^2;\qquad (n = 2);$
$n > 0^+;\ \underline{even}\ n^+;\ n := n / 2;\ x := x^2;\qquad (n = 1);$
$n > 0^+;\ \underline{even}\ n^-;\ n := n - 1;\ y := y \times x;\qquad (n = 0);$
$n > 0^-.$

При обычных вычислениях каждая команда из этой последовательности просто выполняется, приводя тем самым к нужному результату. При смешанных вычислениях контроль сортирует потенциально выполнимые команды на две категории: те, которые относятся к множеству C', – выполняются, а команды из C'' формируют в нужной последовательности остаточную программу. Очевидно, что разбиение *C* на C' и C'' не может быть произвольным: задержанная команда *i* неизбежно задержит любую другую команду i', которая хоть как-то "зависит" от *i*. Содержательно зависимость i' от *i* может быть "информационной" – аргументы i' зависят от результатов *i* – или "логической" – выполнение или невыполнение i' зависит от результатов команды *i*. Для нашей программы задержанными окажутся все команды, содержащие вхождение *x*, и косвенно – все команды, содержащие *y*, кроме самой первой $y := 1$. Кроме того, хотя первое вхождение команды $y := y \times x$ уже будет задержано из-за *x*, терм *y* в источнике $y \times x$ может быть вычислен (его значение равно 1). Итак, мы получим следующее частичное вычисление:

$y := 1;\ n > 0^+;\ \underline{even}\ n^-;\ n := n - 1;$
$n > 0^+;\ \underline{even}\ n^+;\ n := n / 2;$
$\underline{even}\ n^+;\ n := n / 2;$
$\underline{even}\ n^-;\ n := n - 1;$
$n > 0^-;$

и следующую остаточную программу:

$$y := 1 \times x;\ x := x^2;\ x := x^2;\ y := y \times x.$$

В нашем простейшем случае все логические условия в программе оказались выполнимыми, и поэтому остаточная программа получилась линейной. Задержанная передача управления *j* тянет за собой в остаточную программу целый "шлейф" команд, логически от *j* зависимых. Более точно, шлейфом оказываются все потенциально достижимые от *j* команды, предшествующие первому обязательному преемнику команды *j*. Так, если бы мы хотели из программы для x^n получить остаточную программу для, скажем, 5^n, то тогда в частичные вычисления вошла бы только команда у := 1, а остаточная программа практически не отличалась бы от исходной:

x = 5; y = 1;
<u>while</u> n >0 <u>do while</u> <u>even</u> n <u>do</u> n : =n / 2; x : =x² <u>od</u>; n : =n -1; y : =y × x <u>od</u>.

Трансформационный подход уместен тогда, когда общее вычисление может быть описано как набор преобразований программы. В дальнейшем мы увидим, что такой подход к описанию обработки информации является вполне универсальным, а пока сошлемся на известный формализм рекурсивных программ, где процесс вычислений описывается как свободное применение следующих правил преобразований (более точно они описаны в разделе 4 статьи):

1. Конкретизация (подстановка числа в обе части определения функции).
2. Упрощение (замена терма его значением и устранение альтернативы у условного терма с известным условием).
3. Разъединение (замена обращения к функции ее правой частью).
4. Устранение ненужного (неиспользуемого) определения функции.

В формализме рекурсивных программ вычисление x в степени n может иметь следующий вид:

p(x, n) = <u>if</u> n = 0 <u>then</u> 1 <u>else</u> <u>if</u> <u>even</u> n <u>then</u> p² (x, n / 2)
<u>else</u> x × p (x, n - 1) <u>fi</u> <u>fi</u>.

Обозначим для краткости правую часть определения $p(x, n)$ через $T(x, n)$ и покажем, как последовательным применением преобразований 1–4 можно получить из $p(x, n)$ остаточную программу для x^5, т.е. $p(x, 5)$. Для того чтобы подчеркнуть формальный характер процесса, мы приведем его во всех деталях.

1) $p(x, n) = T(x, n)$.

Конкретизация 1) при $n = 5$:

1) $p(x, n) = T(x, n)$,

2) $p(x, 5) = T(x, 5)$.

Упрощение 2):

1) $p(x, n) = T(x, n)$,

2) $p(x, 5) = x\ Ф\ p(x, 4)$.

Конкретизация 1) при $n = 4$ и последующее упрощение:

1) $p(x, n) = T(x, n)$,

2) $p(x, 5) = x\ Ф\ p(x, 4)$,

3) $p(x, 4) = p^2 (x, 2)$.

Конкретизация 1) при $n = 2$ и последующее упрощение:

1) $p(x, n) = T(x, n)$,

2) $p(x, 5) = x\ Ф\ p(x, 4)$,

3) $p(x, 4) = p^2 (x, 2)$,

4) $p(x, 2) = p^2 (x, 1)$.

Конкретизация 1) при $n =1$ и последующее упрощение:

1) $p (x, n) = T(x, n)$,

2) $p (x, 5) = x \times p (x, 4)$,

3) $p (x, 4) = p^2 (x, 2)$,

4) $p (x, 2) = p^2 (x, 1)$,

5) $p (x, 1) = x \times p (x, 0)$.

Конкретизация 1) при $n =0$ и последующее упрощение:

1) $p (x, n) = T(x, n)$,

2) $p (x, 5) = x \times p (x, 4)$,

3) $p (x, 4) = p^2 (x, 2)$,

4) $p (x, 2) = p2 (x, 1)$,

5) $p (x, 1) = x \times p (x, 0)$,

6) $p (x, 0) = 1$.

Устранение ненужного 1) и разъединение 5) и 6):

1) $p (x, 5) = x \times p (x, 4)$,

2) $p (x, 4) = p^2 (x, 2)$,

3) $p (x, 2) = p^2 (x, 1)$,

4) $p (x, 1) = x \times 1$

5) $p (x, 0) = 1$.

Устранение ненужного 5) и разъединение 3) и 4):

1) $p (x, 5) = x \times p (x, 4)$,

2) $p (x, 4) = p^2 (x, 2)$,

3) $p (x, 2) = (x \times 1) 2$,

4) $p (x, 1) = x \times 1$.

Устранение ненужного 4) и разъединение 2) и 3):

1) $p (x, 5) = x \times p (x, 4)$,

2) $p (x, 4) = ((x \times 1)^2)^2$,

3) $p (x, 2) = (x \times 1)^2$.

Устранение ненужного 3) и разъединение 1) и 2):

$1)\ p\ (x, 5) = x\ ((x \times 1)^2)^2,$
$2)\ p\ (x, 4) = ((x \times 1)^2)^2.$

Устранение ненужного 2):

$1)\ p\ (x, 5) = x\ ((x \times 1)^2)^2.$

Особенностью трансформационного подхода является то, что в нем обычные вычисления формально не отличаются от смешанных: процесс, выполняющий преобразования, "не знает", какого типа вычисления выполняются. Можно сказать, что на каждом шагу получается некоторая остаточная программа, которая при полной обработке информации просто редуцируется к окончательному результату.

Общая схема. Дадим теперь общую схему определения смешанных вычислений. Пусть дан алгоритмический язык $L = (D, P, V)$, где $D = \{d\}$ – предметная область, $P\{p\}$ – множество программ и $V : P \times D \to D$ – (обычное) вычисление в языке L. Смешанным вычислением в L называется отображение $M : P \times D \times \underline{M} \to \times D$,
удовлетворяющее следующим свойствам:

$M = \{ \mu \}$ – множество параметров, содержащее минимальный и максимальный объекты μ_0 и μ_1 соответственно,

$M\ (p, d, \mu_0) = (\varnothing, V\ (p, d))$ *(∅– "пустая" программа),*
$M\ (p, d, \mu_1) = (p\ ,d),$
$V\ (p, d) = V \circ M\ (p, d, \mu)$ *(∘– композиция отображений).*

Первая и вторая компоненты M называются остаточными программой и данными соответственно. Смысл параметра может быть различен в зависимости от конкретного подхода к организации смешанных вычислений. Так, если D представимо в виде прямого произведения, то тогда μ является некоторым расчленением D в прямое произведение $D_1 \times D_2$, т.е. $\mu : D \to D_1 \times D_2$. В этом случае $\mu_0 : D \to D \times \varnothing$, а $\mu : D \to \varnothing \times D$ (здесь мы несколько отступаем от общепринятого, допуская "пары" с отсутствующей компонентой). Назовем D_1 связанными (доступными) данными, а D_2 свободными (недоступными) данными. Пусть X и Y – переменные, принимающие значения из D_1 и D_2 соответственно. Программы $p \in P$ становятся тогда функциями переменных $p(X, Y)$, а смешанные вычисления при значении X, равном x, и свободном Y – это получение остаточной программы p_x как функции от $Y : p_x(Y)$. Остаточная программа в этом случае называется проекцией программы p на связанную часть x своих аргументов. Минимальное μ_0 означает, что все данные доступны (нет свободных аргументов); максимальное μ_1 означает, наоборот, что все данные недоступны (все аргументы свободны).

В соответствии с операционным подходом можно разбивать множество C вычислительных актов программы на выполнимые C' и задерживаемые C'' акты $\mu : C \to C' \cup C''$. В этом случае также при μ_0 имеем $C' = C$, а при μ_1 имеем $C'' = C$. Разбиение данных естественно индуцирует разбиение вычислительных актов: любые недоступные данные задерживают все относящиеся к ним операции выборки и (иногда) засылки.

2. ПРИМЕНЕНИЕ СМЕШАННЫХ ВЫЧИСЛЕНИЙ

Получение главных программных процессоров. Наиболее замечательным применением смешанных вычислений является их использование в трансляции. Пусть $L = (D, P, V)$ – язык реализации, который мы также будем считать и объектным языком. Последнее ограничение не является принципиальным. Мы будем далее считать, что D расчленимо на прямые сомножители $D = (X, Y)$, где X – связанная, а Y – свободная части, а смешанное вычисление $M\ (P, X, Y)$ определяется как нахождение проекции программы p на значение x связанной части как функции $p_x(Y)$ свободной части Y.

Пусть дан некоторый входной язык $l\ (\Xi, \Pi, s)$, где $\Xi = \{ \xi \}$ – его предметная область, $\Pi = \{ \pi \}$ – множество программ, а S – семантика языка, т.е. функция $S\ (\Pi, \Xi)$, определяющая результат $\pi\ (\xi)$ применения программы к ее входным данным $\xi : \pi\ (\xi) = S\ (\pi, \xi)$. Нас интересует задача трансляции языка l как представителя некоторого класса входных языков в объектный язык L. В связи с этим нам нужно уметь строить в языке реализации L следующие программы:

1) для любой программы p языка $l = (\Xi, \Pi, s)$ найти ее объектный код, т.е. такую $ob \in P$, что для любого $\xi \in \Xi\ ob(\xi) = \pi(\xi);$

2) для любого входного языка $l = (\Xi, \Pi, s)$ найти его транслятор, т.е. такую $comp \in P$, что для любой $\pi \in \Pi$ $comp(\pi, \Xi) = ob(\Xi)$;

3) в языке L построить транслятор трансляторов, т.е. такую $cocom \in P$, что для любого языка $l = (\Xi, \Pi, s)$ $cocom(S, \Pi, \Xi) = comp(\Pi, \Xi)$.

Мы начнем наши построения со следующих программ в языке L:

интерпретатор языка l, т.е. программа $int(\Pi, X)$, такая что $int(\pi, \xi) = S(\pi, \xi) = \pi(\xi)$;

автопроектор языка L, т.е. программа $mix(P, X, Y)$, которая моделирует смешанные вычисления в этом языке, т.е.

$$mix(p, x, Y) = V(mix, (p, x, Y)) = M(p, x, Y) = p_x(Y).$$

Программа mix называется автопроектором, потому что вычисляет проекции программ в том языке, в каком она сама написана.

Соберем теперь вместе все требуемые равенства.

Определение остаточной программы:

$$M(p, x, Y) = p_x(Y). \quad (1)$$

Свойство остаточной программы:

$$p(x, y) = p\,x(y). \quad (2)$$

Функция автопроектора $mix(P, X, Y)$:

$$Mix(p, x, Y) = p_x(Y). \quad (3)$$

Функция интерпретатора $int(\Pi, \Xi)$:

$$Int(\pi, \xi) = \pi(\xi). \quad (4)$$

Свойство объектного кода $ob(\Xi)$:

$$ob(\xi) = \pi(\xi). \quad (5)$$

Свойство транслятора $comp(\Pi, \Xi)$:

$$comp(\pi, \Xi) = ob(\Xi). \quad (6)$$

Свойство транслятора трансляторов cocom (S, P , X):

$$cocom(S, \Pi, \Xi) = comp(\Pi, \Xi). \quad (7)$$

Для построения объектного кода применим автопроектор к интерпретатору с программным аргументом, связанным программой π, а предметной переменной Ξ свободной. Получаем (между знаками равенств записываем номер используемого соотношения)

$$mix(int, \pi, \Xi) = (3) = int\,\pi(\Xi). \quad (8)$$

Для любого ξ имеем

$$int\,\pi(\xi) = (2) = int(\pi, \xi) = (4) = \pi(\xi).$$

Так как $int\,\pi$ – программа в L, получаем в силу (5)

$$Int\ \pi\,(\,\Xi) = ob\,(\,\Xi). \qquad (I)$$

Итак, объектный код – это проекция интерпретатора на входную программу.

Для построения транслятора сделаем 2-й аргумент программы *mix* в равенстве (8) свободным и подвергнем ее смешанному вычислению или – в силу (1) и (3) – выполним *mix* со следующими аргументами:

$$Mix\ (mix,\ int,\ (\,\Pi,\ \ \Xi)) = (3) = mix_{\ int}(\,\Pi,\ \ \Xi). \qquad (9)$$

Для любой π имеем

$$mix_{\ int}(\,\pi,\ \ \Xi) = (3) = mix\ (int,\ \ \pi,\ \ \Xi) = (8) = int\ \pi\,(\,\Xi) = (I) = ob\,(\,\Xi).$$

Так как $mix_{\ int}$ – программа в L, получаем в силу (6)

$$mix_{\ int}(\,\Pi,\ \ \Xi) = comp(\,\Pi,\ \ \Xi). \qquad (II)$$

Итак, транслятор – это проекция автопроектора на интерпретатор входного языка.

Сделаем теперь в (9) еще один шаг обобщения: сделав 2-й аргумент S программы mix свободным, подвергнем ее смешанному вычислению, т.е. получим

$$mix(mix,\ mix,\ (S,\ \ \Pi,\ \ \Xi)) = (3) = mix_{\ mix}(S,\ \ \Pi,\ \ \Xi). \qquad (10)$$

Для любого интерпретатора *int* имеем

$$mix_{\ mix}(int,\ \ \Pi,\ \ \Xi) = (3) = mix(mix,\ int,\ (\,\Pi,\ \ \Xi)) = (9) = mix_{\ int}(\,\Pi,\ \ \Xi)) = \\ = (II) = comp\ (\,\Pi,\ \ \Xi).$$

Так как $mix_{\ mix}$ – программа в L, получаем в силу (7)

$$mix_{\ mix}(S,\ \ \Pi,\ \ \Xi) = cocom\ (S,\ \ \Pi,\ \ \Xi). \qquad (III)$$

Итак, транслятор трансляторов – это проекция автопроектора языка реализации на самого себя.

Рассмотрев I, II, III, мы можем заключить, что смешанные вычисления являются фундаментальным процессом, лежащим в основе трансляции.

Заметим, что если бы мы в (10) сделали еще один шаг обобщения, попробовав выполнить

$$mix(mix,\ mix,\ (P,\ S,\ \ \Pi,\ \ \Xi)),$$

мы бы уже не получили чего-то интересного, т.к. остаточная программа будет той же самой – *mixmix*, а возможности ее первого аргумента *P* будут ограничены требованием "понимания" роли последующих аргументов.

Эффективность смешанных вычислений. Уже предварительные исследования показывают, что смешанные вычисления служат не только концептуальной базой трансляции, но могут и играть роль эффективного технологического инструмента в построении трансляторов и их работе. Хорошо определенное смешанное вычисление, которое в некотором смысле максимально использует информацию, заданную связанной частью аргументов, выдает весьма эффективную остаточную программу. В качестве примера (Ершов [1977b]) приведем объектный трехадресный код входной программы, записанной в Милане – простом языке структурного программирования (Пейган [1976]). Объектный код получен как остаточная программа в некоторой естественно определенной схеме смешанных вычислений для подмножества Алгола 68, взятого в качестве языка реализации.

Входная программа:	Объектный код:
<u>read</u> *n;*	*in(n);*
i := 0;	*i := 0;*
<u>while</u> *i < n* <u>do</u>	*M1 : r1 : = i < n ;*
i := i + 1;	*if c r1* <u>then</u> <u>goto</u> *M2* <u>fi</u>*;*

```
        read x, y ;
        if x < 0 then
                z := x + (5 × y );
                write z
        fi od;
write z.
```

```
                r2 := i + 1;
                i := r2 ;
        in (x );
        in (y);
                r3 := x < 0;
        if ¬ r3 then goto MM1 fi;
                r4 := 5  × y ;
                r5 := x + r4 ;
                z := r5 ;
                out(z);
 MM1:   goto MM2 ;
MM2 :   goto M1 ;
  m2 :  out(z).
```

В качестве примера получения транслятора как проекции автопроектора на интерпретатор покажем семантическую процедуру интерпретатора, исполняющую оператор
while-do-od (mode while = = struct (relation heading, ref series body)):

```
    proc WHILE=(ref while L)void;
            begin M1 : RELATION (heading of L);
                    if ¬ value of heading of L then goto M2 fi:
                    SERIES (body of L); goto M1; M2 : end.
```

и ее образ в трансляторе:

```
    proc WHILE = (ref while L)void ;
            begin nM1 := copy(M1); nM2 := copy(M2);
                    M1 : T(nM1 :); RELATION(heading of L );  n1 := value of
                    heading of L; T(if  c n1 then goto nM2 fi;);
                    SERIES (body of L); T(goto nM1;);
                    M2 : T(nM2 :) end.
```

Здесь *copy* – операция копирования локального объекта программы, *n1*, *n2* и т.д. – литеральные переменные, *T* – оператор генерации остаточной программы, "печатающий" значения литеральных переменных в конкатенации с указанным текстом.

Любое вычисление обладает важным свойством избирательности, оно извлекает из универсальной программы ту единственную последовательность команд, которая нужна, чтобы "привести к цели" конкретные данные программы. Такой же избирательностью обладают смешанные вычисления, оставляя в остаточной программе только операции, которые нужны для обработки информации, подразумеваемой свободной переменной, и полностью используя информацию, заключенную в связанной переменной. В результате при построении объектного кода остаточной программы автоматически достигается эффект, обеспечиваемый в обычной технике трансляции такими непростыми и дорогими средствами, как смешанная стратегия программирования (Ершов [1966]) и кейсинг (Елсон и Рейк[1970]). В качестве примера возьмем семантическую процедуру конкатенации двух строк в языке ПЛ/1 (Картер [1975]). Эта процедура содержит 55 инструкций внутреннего языка и требует для следующего фрагмента ПЛ-овской программы

```
P:      PROC OPTIONS (MAIN);
        DCL (B,C) CHAR (10), A CHAR (50);
        A = B || C; END.
```

выполнения 73 операций. В то же время соответствующий фрагмент объектного кода, автоматически полученный из интерпретатора с помощью смешанных вычислений, будет иметь следующий вид:

```
T1 = B (9 + 1 BYTES)
T2 = C (9 + 1 BYTES)
A = T1 (9 + 1 BYTES)
(:A:+ 10) = T2 (9 + 1 BYTES)
(:A:+ 20) = blank (0 + 1 BYTES)
(:A:+ 21) = (:A: + 20) (28 + 1 BYTES).
```

Оптимизирующие свойства смешанных вычислений допускают автоматическую реализацию таких методов улучшения программы, как экономия констант, раскрытие циклов, открытая подстановка процедур, условная компиляция и ряд других.

Смешанные вычисления имеют параметр μ, который характеризует, в частности, степень доступности входной информации. Входная информация для интерпретатора – это дерево разбора входной программы π, содержащей ее входные данные ξ среди терминальных вершин дерева разбора. Значение μ_0 соответствует полностью доступному дереву разбора, μ_1 – полностью недоступному дереву разбора. Есть также некоторое значение μ^*, соответствующее недоступности только входных данных. Для этих значений параметра получим следующие результаты смешанных вычислений:

$$M(int, (\pi, \xi), \mu_0) = [\varnothing, V(int, (\pi, \xi)) = int(\pi, \xi) = \pi(\xi)],$$
$$M(int, (\pi, \xi), \mu^*) = [ob(\Xi), \xi],$$
$$M(int, (\pi, \xi), \mu_1) = [int, (\pi, \xi)].$$

Будем называть смешанные вычисления периодом компиляции, выполнение остаточной программы – периодом исполнения. Построим график (рис.1), на оси абсцисс которого будет отложен параметр μ (условно – от μ_0 до μ_1), а по оси ординат – объем вычислений периода компиляции и периода исполнения. Тогда указанные выше соотношения дадут на графике шесть точек. Можно представить себе "плавное" изменение параметра смешанных вычислений, при котором объемы вычислений будут представлены указанными кривыми. Можно сказать, что этот график представляет собой все мыслимые схемы трансляции, различающиеся глубиной интерпретации конструкций входной программы. Например, если считать недоступными в дереве разбора все порождения базисных операторов, мы получим так называемую полуинтерпретирующую схему трансляции. Можно ставить вопрос о схеме трансляции, дающей минимум суммарному объему вычислений.

Уже связи смешанных вычислений с трансляцией достаточно, чтобы стала ясной важность этого понятия и необходимость его изучения. Это тем более так, поскольку трансляция в широком смысле – это не только методы построения компиляторов, но и многие другие виды работы в программировании, связанные с генерацией программ. В частности, очень широкой и практически важной является автоматизированная адаптация универсальных пакетов прикладных программ к конкретным значениям их параметров. Более подробный анализ практической полезности смешанных вычислений можно найти у Ершова [1977] и Бекмана и др. [1976].

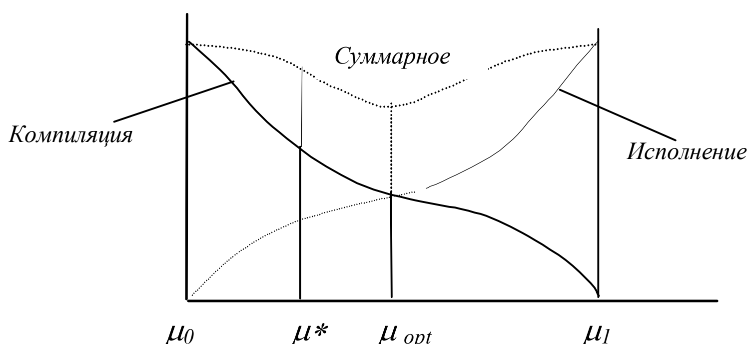

Рис. 1. «Непрерывный» спектр схем трансляции

В заключение мы приведем еще один интересный пример применения смешанных вычислений в трансляции, связанный на этот раз с ее синтаксической фазой.

Как известно, универсальный анализатор для класса грамматик $\{G\}$ – это некоторая функция двух переменных $P(G, s)$, а именно грамматика языка G и его свободной входной строки. В результате у нас получится проекция $P_G(s)$ анализатора на данную грамматику, что естественно считать анализатором, специально ориентированным на данную грамматику. Таким образом, смешанные вычисления дают нам систематическую процедуру построения языково-ориентированных анализаторов, характерных для первых поколений трансляторов.

Построение языково-ориентированных анализаторов с помощью смешанных вычислений изучалось Островским [1980]. При этом обнаружился один интересный феномен. Универсальный анализатор – это обычно весьма тщательно написанная программа, в которой очень трудно что-нибудь улучшить. В то же время по отношению к любой конкретной грамматике этот анализатор неизбежно избыточен. Смешанные вычисления эту избыточность частично ликвидируют сами, а частично вскрывают, делая остаточную программу доступной для дальнейшей оптимизации, которая была бы невозможной в универсальном анализаторе.

3. ОБЩАЯ ТЕОРИЯ СМЕШАННЫХ ВЫЧИСЛЕНИЙ

Смешанные вычисления в теории вычислимых функций. Получение остаточной программы как проекции $f_x(Y)$ программы вычисления функции $f(X,Y)$ на связанную часть аргументов – процесс, хорошо известный в теории вычислимых функций. Он был изучен еще Клини [1952] (§ 65, теорема XXIII) и известен в современной литературе под названием *s-m-n*-теоремы (см., например, Роджерс [1967]).

s-m-n – т е о р е м а. Пусть $U^n(x_0, x_1, \ldots, x_n)$ – универсальная функция для всех частично-рекурсивных функций n переменных. Тогда для любых n и m $(n, m > 0)$ существует общерекурсивная функция $s_m^n(x_0, x_1, \ldots, x_m)$, такая, что для любой функции $f(x_1, \ldots, x_{m+n})$ с номером N_f имеет место тождество

$$f(x_1, \ldots, x_{m+n}) = U^{m+n}(N_f, x_1, \ldots, x_{m+n}) = U^n(s_n^m(N_f, x_1, \ldots, x_m), x_{m+1}, \ldots, x_{m+n}).$$

В нашей терминологии $s_n^m(x_0, x_1, \ldots, x_m)$ – это смешанное вычисление, выдающее номер остаточной программы как функцию номера исходной функции и первых (связанных) аргументов. Фундаментальная роль *s-m-n*-теоремы хорошо известна и неоднократно подчеркивалась авторами учебников. Для программирования, однако, *s-m-n*-теорема может играть лишь принципиальную, но не конструктивную роль, т.к. подразумеваемое ею смешанное вычисление весьма тривиально, т.к. состоит всего лишь в подстановке заданных констант $x_1, \ldots, x_m$ в места вхождения соответствующих переменных в программу N_f.

Смешанные вычисления и транслируемость. *s-m-n*-теорема, опираясь на специальный вид смешанных вычислений, играет заметную роль в изучении сводимости одних видов нумерации вычислимых функций к другим. В более общей и явной форме связь смешанных вычислений с транслируемостью одного языка на другой (что, по существу, то же самое, что сводимость одной нумерации к другой) была установлена Успенским [1960] (для *s-m-n*-теоремы) и Сусловым [1979] (для смешанных вычислений).

Ниже и до конца параграфа все функции будут считаться одноместными функциями кортежей разной длины, а также будет использована алгебраическая символика отображений и их композиций, C – класс вычислимых функций.

Язык $L = (D, P, V)$, где D – предметная область, P – программы и $V : P \times D \to D$ – вычислитель, называется универсальным, если

$$\forall f \in C \; \exists p \in P : f = \lambda x\, V(p,x).$$

Пусть дан также язык $L_1 = (D_1, P_1, V_1)$. Язык L_1 транслируем в L, если существуют тотальные функции $t : P_1 \to P$, $\tau : D_1 \to D$ и $æ : D \to D_1$ такие, что диаграмма отображений

$$\begin{array}{ccc} P \times D & \xrightarrow{V} & D \\ {\scriptstyle t \times \tau}\uparrow & & \downarrow{\scriptstyle æ} \\ P_1 \times D_1 & \xrightarrow{V_1} & D_1 \end{array}$$

коммутативна.

Любое инъективное отображение $\rho : D \to D \times D$ называется расчленением, а отображение $G : P \times D \to P$ – генератором. Генератор G осуществляет в языке L корректное смешанное вычисление, задаваемое расчленением , если имеет место следующее включение:

$$V \supseteq V \circ (G \times e_D) \circ (e_P \times \rho)$$

(здесь e_D – тождественное отображение $D \to D$). На всякий случай покажем читателю, как раскрывается правая композиция:

$$(e_P \times \rho)(p, d) = (p, \rho_1(d), \rho_2(d)),$$
$$(G \times e_D)(p, \rho_1(d), \rho_2(d)) = G(p, \rho_1(d)), \rho_2(d)).$$

Универсальный язык L называется главным, если в него транслируется любой другой язык.

Теорема (С. Суслов). Для того чтобы язык L был главным, необходимо и достаточно, чтобы в нем для любого расчленения существовало бы корректное смешанное вычисление.

Доказательство необходимости строит по данному расчленению некоторый язык L_1, который в силу транслируемости в L обеспечивает существование генератора, подходящего для построения корректного смешанного вычисления.

Доказательство достаточности фактически повторяет конструкцию построения объектного кода как проекции интерпретатора (здесь используется универсальность) на входную программу в подходящем оформлении.

Таким образом, мы видим, что связь с трансляцией является внутренним свойством смешанных вычислений.

Генерирующее расширение. Для программистов может оказаться полезной еще одна трактовка трансляционных свойств смешанных вычислений. Пусть в некотором языке L (D, P, V) задано смешанное вычисление M $(P, D, \underline{M})$. Пусть далее существует некоторое естественное представление программных текстов в предметной области. Оно необходимо для составления в L программы *mix* и вообще позволит нам не выделять специально генераторы из общего класса программ. Определим формально генерирующее расширение $G : P \times M \rightarrow P$ как такое преобразование, при котором

$$\forall p \in P \quad \forall \mu \in M \ \forall d \in D \ V(G(p, \ \mu), d) = M(p, d, \ \mu).$$

Другими словами, генерирующее расширение G (p, μ) программы p в режиме обычного вычисления делает то же самое, что p в режиме смешанного вычисления с тем же μ. Из этого определения сразу становится ясным, что транслятор – это генерирующее расширение интерпретатора, а соотношение $mix_{\ int} = comp$ обобщается на любое генерирующее расширение:

$$mix_{\ p}(\mu) = G(p, \ \mu). \qquad \text{(IV)}$$

Глубина и надежность смешанных вычислений. Даже при заданном μ смешанное вычисление может быть определено далеко не единственным способом. Поясним это простым примером. Пусть

$$p(x, y, z) = \frac{(x + y + z)}{(y - z) \times x + z^2 / y \times x^2} \quad ,$$

Пусть, далее, (y, z) – связанная часть при $y = 4$, $z = 2$ и x – свободная часть. Для выражений, которые являются термами, смешанные вычисления состоят просто в редукции термов, т.е. замене терма его вычисленным значением. Самое простое смешанное вычисление – это подстановка констант в связанные переменные:

$$P_{4,2}^{\max} = \frac{(x + 4 + 2) \times 4}{(4 - 2) \times x + 2^2 / 4 \times x^2},$$

Примерно так, как уже говорилось, получается остаточная программа в *s-m-n*-теореме. Можно, например, ограничиться однократной редукцией термов, получив некоторую промежуточную форму остаточной программы:

$$P_{4,2}^{\text{int}}(x) = \frac{(x + 6) \times 4}{2x + 4 \times x^2 / 4},$$

Можно, наконец, получить в определенном смысле минимальную остаточную программу, в которой никакие дальнейшие редукции невозможны:

$$P_{4,2}^{\min}(x) = \frac{(x + 6) \times 4}{2 \times x + 1 \times x^2},$$

Эта программа может быть оптимизирована с использованием некоторых правил преобразований:
Оставив пока в стороне вопрос оптимизирующих преобразований, остановимся на вопросе прямой

$$P_{4,2}^{opt}(x) = \frac{2(2x) + 24}{2x + x^2},$$

редуцируемости. Очевидно, что мы максимально разгрузим остаточную программу, если потребуем, чтобы любая команда, которой доступны ее операнды, выполнялась бы. Такая постановка, по крайней мере, придает определенность вопросу о глубине смешанных вычислений. Так и поступают во многих случаях. Однако на этом пути возникает хорошо известное препятствие, которое делает смешанные вычисления ненадежными. Рассмотрим на рис. 2 традиционный фрагмент операционной системы, который проверяет очередь заданий.

Рис. 2. Программа с бесконечным циклом

Если бы мы подвергли эту программу смешанному вычислению с задержанной очередью заданий, но с требованием выполнять все, что выполнимо, то смешанные вычисления немедленно вошли бы в непрерывный цикл ждущего теста памяти. Таким образом, в общем случае его корректность выражается соотношением

$$V \subseteq V \circ M,$$

и лишь только специальная забота о надежности вычислений обеспечивает равенство $V = V \circ M$.

Противоречие между глубиной и надежностью смешанных вычислений носит реальный характер, требует тщательного балансирования в построении смешанных вычислений и в целом еще слабо изучено.

Смешанные вычисления в языке структурных программ с памятью и присваиваниями. Язык Милан строится над заданной предметной областью, конечной совокупностью базисных тотальных функций заданных арностей, счетным множеством переменных и констант. Структура языка весьма проста:

<программа> ::= <ряд операторов>
<ряд операторов> ::= <оператор>|<оператор>; <ряд операторов>
<присваивание> ::= <переменная-получатель> ::= <терм-источник>
<альтернатива> ::= *if* <терм-условие> *then* <ряд операторов> *else*
<ряд операторов> fi
<цикл> ::= *while* <терм-условие> *do* <ряд операторов> od.

Программа в Милане работает над фиксированным множеством переменных, образующих память программы, и преобразует состояние памяти в состояние памяти. Параметр смешанного вычисления разбивает память на свободную и связанную части, которые нам будет удобнее называть замороженной и доступной (незамороженной) частями. Программа является источником выполняемых операторов, которые поступают в устройство контроля. Оператор, находящийся в устройстве контроля, может оказаться задержанным. В начальный момент ни один оператор не задержан. Выполнение оператора может быть нормальным и литеральным, при котором происходит запись (последовательно) оператора в остаточную программу, вначале пустую. Главной вычислительной операцией является редукция терма, т.е. замена терма константой, равной его значению.

Выполнение присваивания. Редуцируется терм-источник. Допустимыми аргументами являются константы и незамороженные переменные. Если источник редуцируется к константе, получатель не заморожен и оператор не задержан, то присваивание выполняется нормально. В противном случае оператор выполняется литерально, т.е. записывается с редуцированным источником в остаточную программу, получатель замораживается и управление переходит к следующему оператору.

Выполнение альтернативы. Редуцируется терм-условие. Если оно редуцируется к константе, то управление переходит к первому оператору соответствующего ряда. В противном случае все его составляющие операторы объявляются задержанными, а сама альтернатива выполняется литерально, т.е. в остаточную программу записывается *if* и редуцированное условие, а управление передается на его *then*.

Выполнение then. При обычном выполнении пропускается, при литеральном – записывается в остаточную программу. Управление передается следующему оператору.

Выполнение else. При обычном выполнении передает управление на оператор, стоящий вслед за *fi* соответствующей альтернативы. При литеральном выполнении записывается в остаточную программу и передает управление следующему оператору.

Выполнение fi – так же, как и *then*.

Выполнение цикла. Редуцируется терм-условие. Если оно истинно и цикл не задержан, то передается управление на *do*. Если оно ложно, то передается управление на оператор, следующий за циклом. В противном случае все составляющие операторы цикла задерживаются, а цикл выполняется литерально, т.е. в остаточную программу пишется *while* и редуцированное условие в управление передается на его *do*.

Выполнение do. При обычном выполнении пропускается, при литеральном – записывается в остаточную программу. Управление передается следующему оператору.

Выполнение od. При нормальном выполнении передает управление на свой *while*, при литеральном – пишется в остаточную программу и передает управление следующему оператору.

Это довольно жесткая, но надежная схема смешанных вычислений. Корректность обеспечивается "с запасом" монотонным ростом замороженных переменных. Надежность достигается запретом нормального выполнения всех циклов, попадающих в шлейф задержанного разветвления. В то же время термы редуцируются всюду, где возможно. Более подробный анализ информационных связей и размножение переменных позволили бы нормально выполнять многие задержанные присваивания.

Несколько алгоритмов смешанных вычислений для неструктурированных алголоподобных программ описаны Ершовым и Иткиным [1977].

4. ТРАНСФОРМАЦИОННЫЙ ПОДХОД

Обработка программ. До сих пор мы изучали смешанные вычисления, так сказать, сами по себе. В то же время связь смешанных вычислений с другими видами обработки программ просматривалась с самого начала. Для автора, например, весьма впечатляющим было наблюдение, как смешанные вычисления "провоцируют" оптимизацию остаточной программы, невозможную в исходной. Сейчас мы максимально расширим контекст нашего рассмотрения.

Можно без колебаний сказать, что систематическое программирование как вид деятельности почти исключительно состоит из следующих видов обработки программ:
– выполнения;
– трансляции;
– построения. Его можно рассматривать как систематический процесс преобразования теоремы существования решения задачи $\forall x\ \exists y\ S(x, y)$ в программу $p\ (X)$, доставляющую в качестве требуемый y:

$$\forall x\ \exists y\ S\ (x, y)\ \to p\ (X) :\ \forall x\ S\ (x, p\ (x));$$

– оптимизации;
– перевода. Мы его отличаем от трансляции тем, что при переводе модели вычислений входного и объектного языков существенно различны (рекурсивные – итеративные, апликативные – с памятью, последовательные – параллельные);

– комплексирования. Широкий круг манипуляций, включающий ассемблирование, макрообработку, редактирование и работу с библиотеками и в целом обеспечивающий сборку программы из частей, имеющих самостоятельное содержание;

– генерации программных комплексов. Также разнообразный набор манипуляций, обеспечивающих адаптацию универсального программного комплекса к специфическим условиям его применения.

На практике все эти процессы, не совпадая друг с другом, обладая как сходствами, так и различиями, образуют каждый свой круг понятий и приемов со своей литературой и методами. В то же время, даже если согласиться с их различием, мы увидим, что на практике работа с программой требует применения названных процессов в сочетании, с учетом их совокупного влияния на программу.

В последние годы постепенно складывается убеждение, что все эти процессы обработки программ в значительной степени опираются на сравнительно малое число элементарных преобразований программного текста. Эти преобразования – мы назовем их базовыми трансформациями – носят фундаментальный характер. Каждая из них отражает некоторый существенный факт обработки информации, а взятые вместе они позволяют выразить любой программный процессор.

Конечно, сказанное только что – это больше постановка, нежели констатация достигнутого. До общей теории обработки программ еще далеко, и в литературе мы можем найти только фрагменты такой теории.

Наша цель – продемонстрировать систему базовых трансформаций для двух простых, но содержательных моделей программ. Эти системы описывают достаточно широкий класс обработки программ, включающий выполнение программ, смешанное вычисление и оптимизацию. Теоретически системы исследованы еще далеко не полностью, хотя и опираются на ряд установленных фактов.

Базовые трансформации для рекурсивных программ. Напомним коротко формализм рекурсивных программ. Базисный алфавит термов содержит счетное множество констант, формальных параметров и определяемых функций и конечные наборы функциональных символов и предикатных символов. Общий класс термов – это обычные функциональные и предикатные термы, а также условные термы $(\ \pi |\ \tau 1 |\ \tau 2)$, где π– предикатный терм, а τ_1 и τ_2 – общие термы. Рекурсивная программа – это терм (главная программа) и

конечное множество уравнений вида $f(A) = \tau$, где f – символ определяемой функции, $A = (a_1, \ldots, a_n)$ – аргументные термы, а τ – терм тела функции.

Базовые трансформации для рекурсивных программ (система IR):

1. Редукция терма (раскрытие константы)

$$\frac{\vdash \tau = c}{P(\tau) \sim P(c)}$$

(здесь τ - терм, c - константа, равная его значению, $P(\tau)$ – программа P, содержащая терм τ).

2. Редукция истинного (обусловливание истинным)

$$(\underline{\text{true}} \mid \tau_1 \mid \tau_2) \sim \tau_1.$$

3. Редукция ложного (обусловливание ложным)

$$(\underline{\text{false}} \mid \tau_1 \mid \tau t_2) \sim \tau t_2.$$

4. Устранение двойной проверки по истинному (введение двойной зависимости от истинного)

$$(\pi \mid (\pi \mid \tau_1 \mid \tau_2) \mid \tau_3) \sim (\pi \mid \tau_1 \mid \tau_3).$$

5. Устранение двойной проверки по ложному (введение двойной зависимости от ложного)

$$(\pi \mid \tau_1 \mid (\pi \mid \tau_2 \mid \tau_3) \sim (\pi \mid \tau_1 \mid \tau_3).$$

6. Задание (устранение ненужного)

$$\frac{\neg (f \text{ входит в } P)}{P \sim \cup \{f(A) = \tau\}}$$

7. Конкретизация (поглощение)

$$\frac{P(f(X) = \tau(X))}{P \sim P \cup \{f(T) = \tau(T)\}}$$

(здесь $X = x_1, \ldots, x_n)$ – формальные параметры, входящие в аргументы терма, а $T = (t_1,\ldots,t_n)$ – конкретизирующие их термы).

8. Разъединение (соединение)

$$\frac{P(f(T) = \tau(T))}{P(f(T)) \sim P(\tau(T))}$$

Заметим, что все преобразования обратимы и в скобках даны их названия при реверсном употреблении, которое обозначается также звездочкой у номера преобразования.

Для описания любых вариантов обычных вычислений в рекурсивных процедурах достаточно трех преобразований: редукции, конкретизации и разъединения, употребляемых для функций; устранения ненужного и поглощения для сборки мусора. Известно, что определенная дисциплина их применения обеспечивает вычисление наименьшей неподвижной точки рекурсивной программы.

Покажем в качестве менее тривиального примера смешанное вычисление функции Аккермана $A(x, y)$ для $x = 3$. Введем обозначения: $A(3, y) = exp(y)$, $A(2, y) = mult(y)$, $A(1, y) = add(y)$. Шаги вычисления функции Аккермана для полностью связанных аргументов опустим. Применения трансформаций будут указываться их номерами. Как известно,

$$A(x, y) = (x = 0 \mid y - 1 \mid (y = 0 \mid A(x - 1, 1) \mid A(x, y - 1))).$$

Обозначим правую часть уравнений через (x, y). Ниже следует цепочка трансформаций:

a) A(3, y) → *7.* b) для *x = 3* дает *c), 1.* в *c) 3 = 0 =* false, *3 1 = 2, 3.* в *c), 6.* дает *a):*
b) A(x, y) = a (x, y)
a) A(3, y)
b) A(x, y) = a (x, y) → *7.* в *d)* для *y = y 1* дает *e):*
c) A(3, y) = (y = 0 | A(2, 1) | A(2, A(3, y - 1)))
d) exp (y) = A(3, y)
a) A(3, y)
b) A(x, y) = a (x, y) → *8** в *c)* с *e), 7** в *e)* с *d),*
c) A(3, y) = (y = 0 | A(2, 1) | A(2, A(3, y - 1))) *8** в *a)* с *d):*

d) exp(y) = A(3, y)
e) exp(y 1) = A(3, y - 1)
a) exp(y)
b) A(x, y) = a (x, y) → *8* в *d)* с *c), 6** в *c),d) c):*
c) A(3, y) = (y = 0 | A(2, 1) | A(2, exp(y - 1)))
d) exp(y) = A(3, y)
a) exp(y) →
b) A(x, y) = a (x, y)
c) exp(y) = (y = 0 | A(2, 1) | A(2, exp(y - 1))).

После выполнения аналогичных манипуляций с *A(2, y)* программа получит вид:

a) exp(y)
b) A(x, y) = a (x, y)
c) exp(y) = (y = 0 | A(2, 1) | mult(exp(y - 1)))
d) mult(y) = (y = 0 | A(1, 1) | A(1, mult(y - 1))).

Наконец, после обработки *A(1, y)* получим
a) exp(y)
b) exp(y) = (y = 0 | A(2, 1) | mult(exp(y - 1)))
c) mult(y) = (y = 0 | A(1, 1) | add(mult(y - 1)))
d) add(y) = (y = 0 | 0 | add(y - 1) + 1).

Ершов [1978] показал, что существует дисциплина применения базовых трансформаций, обеспечивающая корректные и надежные смешанные вычисления в близком формализме, однако не использующем явно правило конкретизации.

Система IR, в принципе, хорошо известна, кроме того, выполнение рекурсивных программ легче поддается формализации благодаря аппликативному характеру языка. Тем не менее и здесь теория имеет много пробелов. В частности, слабо изучались оптимизация и вообще схемные преобразования рекурсивных программ. Для рекурсивных схем программ еще не построено удобного инварианта, на основе которого можно было бы ввести понятие разрешимой эквивалентности и доказать полноту системы по отношению к этой эквивалентности.

Базовые трансформации для программ с памятью. Если трансформационный подход к описанию выполнения рекурсивных программ считается практически общепринятым, то для программ с памятью автор не знает ни одной публикации, в которой излагалось бы что-нибудь похожее. В то же время преобразования схем последовательных программ с памятью изучались весьма интенсивно, в частности в интересах оптимизации. Оказалось, что достаточно добавить к этим преобразованиям правило редукции термов для того, чтобы построить "трансформационную семантику" программ с памятью. Не касаясь этого вопроса во всей общности, приведем систему базовых трансформаций, достаточную для описания нормальных и смешанных вычислений в языке Милан (слегка расширенном). Предварительно введем одно понятие. Два вхождения x_r и x_a одной величины x информационно связаны, если оператор *A*, воспринимающий x_a, достижим от оператора *R*, вырабатывающего x_r, причем на пути от *R* до *A* не встречается операторов, вырабатывающих *x*. Построим информационный граф программы для величины *x*, вершинами которого будут вхождения *x* в программу, а дугами – отношение информационной связи. Множество вхождений величины *x*, относящееся к одной компоненте связности информационного графа, называется областью действия D_x.

Базовые трансформации для программ с памятью (система *M*):

1. Редукция терма (раскрытие константы)

$$\frac{\vdash \tau = c}{P(\tau) \sim P(c)}$$

2. Редукция истинного (обусловливание истинным)

if true then *s1* else *s2* fi *s1*.

3. Редукция ложного (обусловливание ложным)

if false then *s1* else *s2* fi *s2*.

4. Раскрытие цикла (свертка цикла)

while π do *s* od if π then *s*; while π do *s* od fi.

5. Отделение области действия (присоединение области действия)

$$\frac{\neg y \text{ входит в } P}{P(D_x) \sim P(D_y)}$$

6. Подстановка источника (экономия команд)

$$\frac{\exists^1 x := E \text{ в } P}{P(x_a) \sim P(E)}$$

7. Устранение ненужного (добавление работы)

$$\frac{\neg x_a \text{ входит в } P}{P(x := E) \sim P(0)}$$

Вспомним нашу программу возведения в степень; одновременно введем сокращенные обозначения для ее двух циклов:

y : = *1*;while *n* > 0 do while even *n* do *n* : = *n / 2; x:=x* od; *n := n - 1; y:=y × x* od.

dod — while even *n* do *n* : = *n / 2; x:=x* od

DOD — while *n* > 0 do … od

Опишем получение остаточной программы для x^5 с помощью системы *M*:

y := 1; DOD → *4.* в *DOD:*
y := 1; if n > 0 then dod; n := n 1; y := y × x; DOD fi → *1., 2.*
в *n > 0:*
y := 1; dod; n := n - 1; y := y × x; DOD → 4. в dod,
1.,3. в even n :
y := 1; n := n - 1; y := y × x; DOD - 5. с *y* на *s* и *n* на *m*:
z := 1; m := m -1; y := z × x; DOD(m = 5) → *6.* с *z* и *m*:
z := 1; n := 5 1; y := 1 × x; DOD → *1.* в *5 - 1, 7.* с *z:*
n : = 4; y := 1 × x; DOD. →

Мы вычленили первую команду остаточной программы, которая соответствует *n = 5*, и сделали *n* равным 4. Повторив аналогичную серию преобразований, мы получим:

y : = 1 × x; n : = 2; x : = x^2*; dod; y : = y × x; n := n - 1; DOD* →
y : = 1 × x; x : = x^2*; n : = 1; x : =* x^2*; dod; y : = y × x; n : = n - 1; DOD* →
y : = 1 × x; x : = x^2*; x : =* x^2*; y : = y × x; n : = 0; DOD* →
y : = 1 × x; x := x^2*; x :=* x^2*; y := y × x.*

Общий принцип, надеемся, очевиден. Схемные преобразования и редукция условий создают впереди программы экземпляр выполняемого присваивания. Разыменование переменных отщепляет индивидуальную информационную связь, осуществляемую получателем присваивания, и позволяет подставить источник в программу. После этого оператор устраняется. Если оператор не редуцируется полностью, он минуется и становится компонентой остаточной программы.

В отличие от рекурсивных программ в схемах программ с памятью базовые трансформации неплохо изучены с позиций оптимизации. Система М фактически является фрагментом более общей системы преобразований, описанной Сабельфельдом [1978]. Эта система полна по отношению к логико-термальной эквивалентности и разрешает эту эквивалентность в полиномиальное время (Сабельфельд [1980 a]). С помощью этой системы удалось строго описать серию практически полезных оптимизаций: экономию памяти, экономию команд, чистку циклов (Сабельфельд [1980 b]).

Проблемы трансформационного подхода. Трансформационный подход вносит недетерминизм в обработку программ, а с ним – присущие ему проблемы. Это – полнота системы базовых трансформаций либо по отношению к достижимости некоторого внешним способом определяемого результата обработки, либо по отношению к некоторому определению эквивалентности. Возникает проблема однозначности результата применения преобразований, уже изучавшаяся для некоторого фрагмента системы IR (см. Розен [1973]). Трансформационный подход может дать свою трактовку уже упоминавшимся важным понятиям продуктивности и надежности обработки программ. Наконец, эстетические соображения и стремление познать наиболее элементарные, но содержательные концепции обработки программ будут побуждать к изучению независимости и достижению простоты системы базовых трансформаций. Из более прикладных задач появляются проблемы описания преобразований более высокого уровня в терминах базовых трансформаций.

Трансформационная машина. Автору представляется весьма важным создание концепции трансформационной машины как некоторого целостного устройства, системой команд которой являются базовые трансформации, а программами – программные процессоры. Предметной областью трансформационной машины являются программные тексты и их данные. Мы снова, как в машине фон Неймана, операционно не различаем программу и ее данные, но уже, так сказать, на более высоком уровне, когда программа и ее данные в равной степени обрабатываются программным процессором.

Реализация программного процессора как программы для трансформационной машины обладает интересной семантической особенностью. Ее каждая отдельная команда – базовая трансформация – заключает в себе некоторое минимальное осмысленное действие, поскольку оно заведомо сохраняет некоторый инвариант данных – семантику обрабатываемой программы. В результате проблема правильности программного процессора, по существу, сводится к проблеме остановки. Транслятор, запрограммированный как преобразователь интерпретатора входного языка в системе базовых трансформаций, будет, если он доходит до остановки, всегда правилен, поскольку он всегда выдает некоторую версию объектного кода. Это свойство значительно повышает надежность программирования трансформационной машины. Представляет также интерес изучение связи неподвижных точек программ трансформационной машины как ситуаций, когда все преобразования становятся неприменимыми.

Совсем из другой области, но не менее интересная проблема – это создание аппаратной реализации трансформационной машины. Если же не вдаваться в спекуляции, то достаточно будет сказать, что концепция трансформационной машины заведомо будет полезна для более определенной постановки проблемы эффективности обработки программ на основе базовых трансформаций.

5. ССЫЛКИ

В этом обзоре автор хотел бы сделать ссылки на известные ему и повлиявшие на него оригинальные, независимо выполненные работы, имеющие прямое отношение к предмету статьи и не упомянутые по ходу изложения.

Идея смешанных вычислений как комбинации обработки заданной информации и заготовки впрок команд обработки незаданной информации восходит к методологической работе Ломбарди [1967]. Конструктивная реализация этой идеи для Лиспа (Ломбарди, Рафаэл [1964]) привела к понятию частичного вычисления, которое употребляется в литературе и поныне. Функциональный подход к получению остаточной программы в языке Рефал был описан Турчиным [1974]. Термин "смешанные вычисления" и операционный подход к получению остаточной программы был предложен Ершовым [1977a]. Автор предпочитает термин "смешанное" вычисление, потому что, по его мнению, этот термин базируется на более общем понятии

"вычисление" (computation), нежели "получение значения" (evaluation), и лучше подчеркивает комбинированный результат вычисления – состояние памяти и остаточную программу.

Получение объектного кода как проекции интерпретатора на входную программу было независимо найдено Футамурой [1971], Ершовым [1977a] и Турчиным [1979]. Определение транслятора как генерирующего расширения интерпретатора было дано Ершовым [1977a]. Идея определения транслятора с помощью автопроектора принадлежит Турчину (Рефал [1977]). Эта идея, сообщенная автору С. А. Романенко в декабре 1976 г., позволила ему установить эквивалентность генерирующего расширения проецированию автопроектора. В неявном виде эта идея была также предвосхищена Футамурой [1971]).

Другой подход к получению транслятора путем систематической оптимизации интерпретатора с учетом информации, извлекаемой из входной программы, был независимо найден Ершовым [1979b] и Харрисоном [1977]. Фактическая эквивалентность этих методов и смешанных вычислений становится очевидной в свете трансформационного подхода. Схема "непрерывного"
спектра схем трансляции взята у Ершова и Чинина [1978]. Важность изучения шлейфа как структурной единицы анализа программ была установлена Иткиным [1978].

Начальная форма этого обзора была частично представлена Советско-французскому коллективу по информатике, собиравшемуся в ИРИА в Париже 2–8 октября 1978 г. Трансформационный подход к описанию программных процессоров излагается по недавнему докладу автора (Ершов [1979a]). На формирование этого подхода повлияли также работы группы профессора Бауэра из Мюнхенского университета (Бауэр [1979], Пеппер [1979]), работы Дарлингтона и Берсталла [1976], в особенности Дарлингтона [1978], у которого автор заимствовал систему IR, а также уже упомянутые в тексте работы группы экспериментального компилятора Исследовательского центра ИБМ в Йорктаун-Хейтсе и работы Сабельфельда по преобразованиям схем программ.

6. ЗАКЛЮЧЕНИЕ

Посвятив нашу работу смешанным вычислениям, мы фактически сфокусировали изложение на трансформационном подходе к систематическому программированию. Автор глубоко верит, что выделение в языке реализации системы базовых трансформаций позволяет унифицировать программные процессоры, дать им строгое обоснование, а главное – применять в недоступных ранее сочетаниях.

Трансформационный подход не является конгломератом всевозможных приемов, а создает определенное направление как в теории, так и в технологии программирования. Это направление еще далеко от своего окончательного формирования, однако многие последние работы достаточно конструктивно подкрепляют желание выработать настоящую "алгебру программ", в символической форме выражающую сущности программирования.

Поясним эту мысль в части технологии программирования на примере поиска идеального языка системного программирования (ЯСП) и разворачивания над ним инструментальных комплексов.

Во-первых, хороший ЯСП должен иметь достаточно прозрачную структуру, с тем чтобы подлежащая математическая модель, для которой будет построена система базовых трансформаций, легко извлекалась бы из текста программы и, с точностью до допустимых абстракций, была бы адекватна ей.

Во-вторых, в ЯСП надо писать не столько монолитные программные процессоры, а скорее, иметь инструментальную систему, в основе которой лежит "трансформационная машина" и набор разных орудий: библиотек трансформаций большей глубины, заготовок программных процессоров и, естественно, разных средств сборки и редактирования. В соединении с диалоговой работой и системой представления знаний (для накопления "ненаивной" – см. Дейкстра [1977] – информации о правилах применения преобразований) такая расчлененная инструментальная система будет объединять мощность с гибкостью и обеспечивать то самое совокупное воздействие на программу, о котором мы говорили в статье.

Автор в начале доклада высказал надежду, что в программировании мы сейчас находимся на пороге реального обручения теории с практикой. Место наибольших ожиданий практики и наиболее многообещающего применения теории сейчас – это создание хорошей математической модели реального языка программирования. Построение для него строгой математической семантики и полной системы базовых трансформаций позволит, в частности, подойти вплотную к созданию "идеального" языка реализации и построить солидный фундамент технологии программирования.

Эта работа в своей основе была написана на свежем августовском воздухе Завьяловской базы отдыха. Автор признателен А. И. Лапину и его помощникам, сделавшим пребывание в Завьялове одновременно приятным и плодотворным.

СПИСОК ЛИТЕРАТУРЫ

Бауэр (Bauer F. L.)

[1979] Program development by stepwise transformations – the Project CIP. In F.L.Bauer and M. Broy (Eds.) "Program construction". Lecture Notes in Computer Science, vol. 69, Springer, Berlin – Heidelberg – New York, 237–272.

Бекман и др. (Beckman L., et al.)

[1976] A partial evaluator and its use as a programming tool. Artificial Intelligence, 7, 319.

Дарлингтон (Darlington J.)

[1978] A synthesis of several sorting algorithms. Acta Informatica, 11:1, 1–30.

Дарлингтон и Берсталл (Darlington J., Burstall R. M.)

[1976] A system which automatically improves programs. Acta Informatica, 6:1, 41–60.

Дейкстра (Dijkstra E.W.)

[1977] Why naive program transformation systems are unlikely to work. Manuscript EWD 636, Nuenen.

Елсон и Рейк (Elson M., Rake S. T.)

[1970] Code-generation for large-language compilers. IBM Syst. J., 9:3, 166–188.

Ершов (Ershov A.P.)

[1966] ALPHA – an automatic programming system of high efficiency. J. ACM, 13:1, 17–24.

[1977a] On the partial computation principle. IPL, 6:2, 38–41.

[1977b] On the essence of compilation. In E. J. Neuhold (Ed.) "Formal description of programming concepts", North-Holland, Amsterdam – New York – Oxford, 391–420.

[1978] Mixed computation in the class of recursive program schemata. Acta Cybernetica, 4:1, 19–23.

[1979a] Трансформационный метод в технологии программирования. Технология программирования. Тезисы докладов I Всесоюзной конференции. Пленарные доклады и общие материалы. Институт кибернетики АН УССР, Киев, 12–26.

[1979b] The British lectures. British Computer Society, London.

Ершов и Иткин (Ershov A. P., Itkin V. E.)

[1977] Correctness of mixed computation in Algol-like programs. In J. Gruska (Ed.) "Mathematical foundations of computer science 1977". Lecture Notes in Computer Science, Springer-Verlag, Berlin – Geidelberg – New York, 59–77.

Ершов и Чинин (Ershov A. P., Chinin G. D.)

[1978] Design specifications for a quality compiler factory. In P. G. Hibbard and S. A. Schuman (Eds.) "Constructing Quality Software". North-Holland, Amsterdam – New York – Oxford, 99–116.

Иткин В. Э.

[1978] Эквивалентность свободных схем программ. Кибернетика, Киев, № 1, 1–9.

Картер (Carter J. L.)

[1975] A case study of a new compiling code generation technique. RC 5666, IBM T. J. Watson Research Center, Yorktown Heights.

Клини (Kleene S. C.)

[1952] Introduction to metamathematics. Van Nostrand, New York.

Ломбарди (Lombardi L.A.)

[1967] Incremental computation. Advances in Computers, 8, Academic Press, New York.

Ломбарди и Рафаэл (Lombardi L. A., Raphael B.)

[1964] LISP as the language for an incremental computer. In "The programming language LISP: its operation and applications". Information International, Inc., Cambridge, 204–219.

Островский Б. Н.

[1979] Получение языково-ориентированных анализаторов с помощью смешанных вычислений. В сб. "Языки и системы программирования" под ред. И. В. Поттосина. Вычислительный центр СО АН СССР, Новосибирск.

Пейган (Pagan F. G.)

[1976] On interpreter-oriented definitions of programming languages. The Computer Journal 19:2, 151–155.

Пеппер (Pepper P.)

[1979] A study on trtansformational semantics. In F. L. Bauer and M. Broy (Ed.) "Program construction". Lecture Notes in Computer Science, vol.69, Springer, Berlin – Heidelberg – New York, 322–405.

Рефал

[1977] Базисный Рефал и его реализация на вычислительных машинах (методические рекомендации), TcNIPIASS, Moscow, 92–95.

Роджерс (Rodgers H., Jr.)

[1967] Theory of recursive functions and effective computability. McGraw-Hill, New York.

Розен (Rosen B. K.)

[1973] Tree-manipulating systems and Church-Rosser theorems. J. ACM, 20:1, 160–187.

Сабельфельд (Sabelfeld V. K.)

[1978] Aquivalence Transformationen f①r Flussdiagramme. Acta Informatica, 10:2, 127–155.

[1979a] Полиномиальная оценка сложности распознавания логико-термальной эквивалентности. ДАН СССР, 249, № 4, 793–796.

[1979b] Некоторые оптимизирующие преобразования для стандартных схем. В сб. "Языки и системы программирования", Вычислительный центр СО АН СССР, Новосибирск.

Суслов С. Л.

[1979] Устное сообщение.

Турчин В. Ф. (Turchin V. F.)

[1974] Эквивалентные преобразования программ на Рефале. Автоматизированная система управления строительством. Труды института, вып.6, ЦНИПИАСС, 36–68.

[1979] A supercompiler system based on the language REFAL. SIGPLAN Notices, 14:2, 46–54.

Успенский В. А.

[1960] Лекции о вычислимых функциях. ГИФМЛ, Москва, 11.

Футамура (Futamura Y.)

[1971] Partial evaluation of computation process – an approach to a compiler-compiler. J. Inst. Electrinics and Communication Engineers. Systems, Computers, Control. Tokyo, 2:5, 45–50.

Харрисон (Harrison W.)

[1977] A new strategy for code generation the General Purpose Optimizing ompiler. The 4th ACM Symposium on Principles of Programming Languages, ACM, New York, 29–37.