Введение в метавычисления и суперкомпиляцию

Андрей В. Климов

> Мы должны научиться так обрабатывать программы на ЭВМ как сейчас обрабатываем числа на Фортране.
> *Валентин Турчин, 1971 (реплика на семинаре)*

1 Метавычисления

Основной вид деятельности в программировании — как следует из самого этого слова — разработка и написание программ человеком. Программы — это тексты, которые предписывают компьютеру что-то делать самостоятельно. Программы — это основной объект изучения науки информатики (computer science). Цель исследований — научиться порождать, анализировать, преобразовывать программы, осуществлять самую разнообразную *деятельность над программами.*

На ранних стадиях развития компьютерной техники, в 1940-е и 50-е гг., математики составляли программы непосредственно в машинных командах. С появлением в 1957 г. языка программирования Фортран, деятельность по программированию стала перекладываться на компьютеры. В течение следующих десятилетий на компьютеры возлагалось все больше и больше того, что раньше делал только человек или вообще никто не делал.

1.1 Метасистемный переход

Для характеристики подобных процессов в эволюции науки (и не только ее), В.Ф. Турчин в книге «Феномен науки» [16] предложил понятие *метасистемного перехода*: перехода от системы, состоящей из некоторых объектов и управляющей системы, осуществляющей какую-то деятельность над этими объектами, управление ими, к *метасистеме*, в которой возникает новый уровень управления, для которого предшествующий уровень становится объектом деятельности.

В нашем случае, деятельность низшего уровня — выполнение программ компьютером. Для следующего уровня она предстает в виде объектов — текстов программ, предписывающих компьютеру его поведение. Поначалу метасистемой верхнего уровня являлся человек, разрабатывающий программы. В дальнейшем появились *метапрограммы*, преобразующие программы, например, транслирующие с языка Фортран в команды машины. Поскольку метапрограммы — это обычные программы, то их можно применять к себе и себе подобным, например, написать транслятор с Фортрана на самом Фортране. Происходит метасистемный переход (рис. 1).

Такие цепочки применения метапрограмм друг к другу могут быть произвольной длины: метапрограмма M_n что-то делает с метапрограммой $M_{n\text{-}1}$, которая в свою очередь преобразует, анализирует и т. п. метапрограмму $M_{n\text{-}2}$ и так далее до программы M_0, которая не заслуживает приставки «мета», поскольку может просто проводить, скажем, численный счет. Каждый следующий уровень метапрограмм M_n совсем не обязан быть качественно новым. Один раз создав достаточно универсальные инструменты, их можно применять к самим себе в различных комбинациях, получая новые эффекты. Самоприменение метапрограмм — это уже легкий процесс по сравнению с актом их создания. Один раз научившись это делать, мы можем разворачивать иерархии метасистем, воспроизводя однотипную деятельность на каждом уровне. Возникает *эффект лестницы*, как назвал его В.Ф. Турчин, порождающий крупномасштабный метасистемный переход, охватывающий серии однотипных метасистемных переходов (рис. 1).

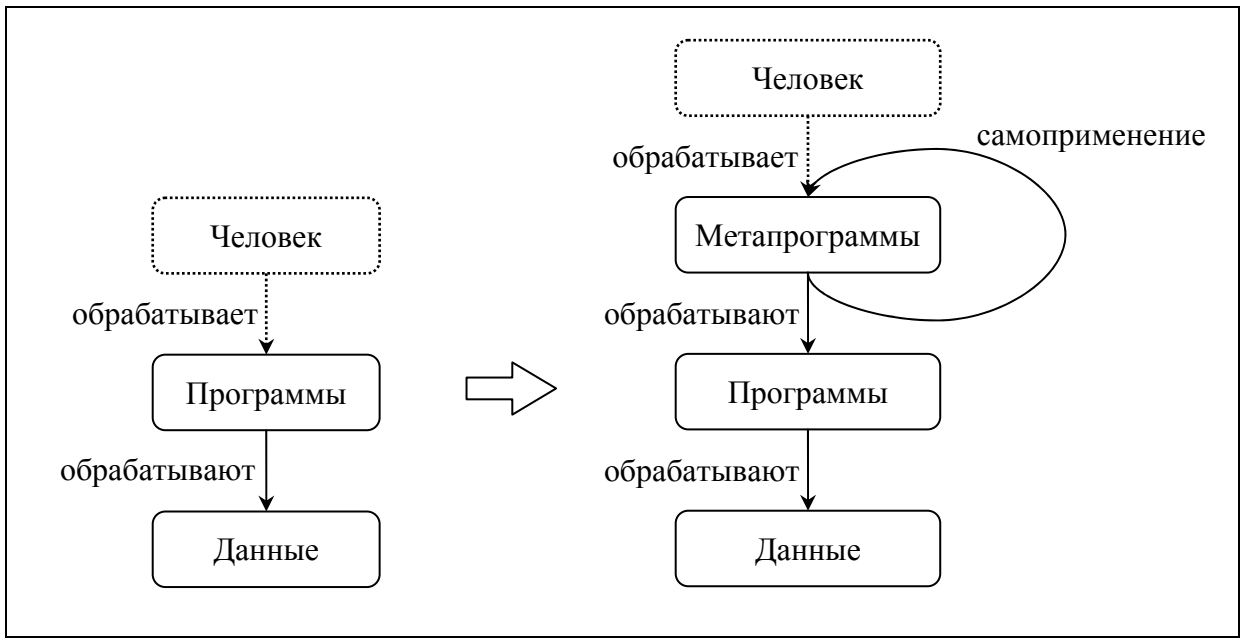


Рис. 1. Метасистемный переход от программирования как деятельности человека над программами к автоматизированной и автоматической обработке программ на компьютере и метапрограммированию

Компьютер — уникальный инструмент в истории человечества: он позволяет человеку создавать внутри себя иерархии метасистем, которые активны сами по себе. Деятельность человека по разработке метапрограмм естественно назвать *метапрограммированием*, а деятельность как компьютера, так и человека по преобразованию программ и метапрограмм *метавычислениями.*

За полувековую историю программирования и информатики появилось много программ, вроде бы заслуживающих приставки «мета»:

- интерпретаторы[1] и компиляторы[2] языков программирования;
- инструменты от текстовых редакторов для набора программ до визуальных студий и интегрированных сред разработки программ, поддерживающих «рефакторизацию» программ — нетривиальные операции по изменению программы, выполняемые по указанию человека;
- средства порождения программ от макросистем (популярных в 70-е гг., а теперь почти забытых) до механизмов типа шаблонов в языке C++ и далее еще более нетривиальных;
- анализаторы и средства проверки правильности и других свойств программ;

[1] *Интерпретатором* некоторого языка L называется программа $intL_M$ на некотором языке M, которая берет текст p_L программы на языке L и начальные данные x для нее и выполняет действия, предписанные программой p, над данными x, получая результат y, т. е. $y = intL_M(p_L, x)$. (Здесь индекс обозначает язык на котором написана программа.) В этом контексте язык M считается *метаязыком*, поскольку на нем определяется другой язык L путем описания его *операционной семантики*. Если метаязык M также задан интерпретатором $intM_N$ на языке N, то можно выполнить двойную интерпретацию: $y = intM_N(intL_M, (p_L, x))$. Такую цепочку интерпретации языков можно продолжать неограниченно.

[2] *Компилятором* некоторого языка L в язык N (или *транслятором* с L в N) называется программа $compLN_M$ на некотором языке M, которая берет текст p_L программы на языке L и переводит его на язык N, т. е. выдает текст p_N эквивалентной программы: $p_N = compLN_M(p_L)$. Эквивалентность исходной и скомпилированной программ понимается как эквивалентность функций: для любых исходных данных x программы p_L и p_N либо выдают одинаковый результат, $p_N(x) = p_L(x)$, либо обе не завершают вычисление, «зацикливаются». Здесь язык M также считается *метаязыком*, поскольку на нем определяется язык L путем его отображения в язык N. В свою очередь, метаязык M может быть также определен на другом метаязыке M' интерпретатором или компилятором. Возникающая иерархия определений языков может иметь произвольную высоту.

- оптимизаторы программ;
- для этого и много чего другого были разработаны разнообразные языки программирования высокого уровня, часть из которых именовались *метаязыками*;
- и т. д., и т. п.

Достаточно ли этого, чтобы сказать, что (хотя бы первый) метасистемный переход над программами завершился? Нет, пожалуй, так считать рано. Слишком много еще человек делает с программами сам, своими руками и головой. Слишком еще редки случаи самоприменения, чтобы получить новое качество благодаря эффекту лестницы. Слишком еще простые задачи мы решаем с помощью преобразований программ.

1.2 Основные задачи метавычислений

Чего же мы хотим от метавычислений? Можно ли назвать примеры задач, которые нам *нужно* решать и которые мы *могли* бы решать с помощью метавычислений; задач, нынешние методы решений которых либо нас плохо устраивают, либо мы вообще не знаем, как к ним подойти.

Самые популярные задачи метавычислений такие:

- Специализация программ:

$$f(x,y) \rightarrow f_A(y) = f(A, y).$$

- Композиция программ:

$$f(x), g(x) \rightarrow f_g(x) = f(g(x)).$$

- Инверсия программ:

$$f(x) \rightarrow f^{-1}(y) = x, \text{ когда } y = f(x).$$

Специализация программ — это порождение по универсальной программе с множеством параметров специализированной программы, когда значения части параметров известны и фиксированы. При специализации известная информация распространяется по тексту программы и используется для ее оптимизации и поднятия эффективности во много раз. Практическую пользу от хороших средств специализации невозможно переоценить. Программисты многократно создают версии программ на разные случаи жизни из-за того, что универсальные программы не устраивают их по эффективности. Более того, оказывается, самоприменение специализатора позволяет решать необычные задачи, например, превращение интерпретатора в компилятор.

Композиция программ — это порождение по двум (или нескольким) подпрограммам, передающим результаты одной в качестве аргументов другой, более эффективной программы, которая либо выполняет ту же работу вообще без формирования промежуточных данных («за один проход»), либо содержит прооптимизированные версии исходных подпрограмм с учетом того, что они работают в контексте друг друга. Композиция программ является обобщением специализации, поскольку $f_A(y) = f(g(y), y))$, где $g(y) = A$. Эффекты от ее применения были бы еще более разнообразны, чем от специализации. Упомянем лишь снятие неэффективности модного ныне *сборочного* или *компонентного программирования*. Обзор задач, сводимым к специализации и композиции можно найти в статье [2]. [3]

Инверсия программ — по программе, реализующей некоторую функцию, порождение программы, реализующей обратную функцию, т. е. по значению первой про-

[3] Одна из таких задач — сжатие иерархий определений языков, упомянутых в предыдущих сносках, т. е. автоматическое превращение последовательности определений языков, заданных интерпретаторами и компиляторами, в более эффективные прямые определения «высших» языков иерархии через «низшие».

граммы выдающей соответствующий аргумент. Ценность инверсии программ в том, что очень часто обратную функцию задать намного легче, чем ту, которую нужно запрограммировать. Полноценная инверсия программ — очень сложная задача; сейчас она успешно решается сейчас лишь для некоторых классов программ. Инверсия фактически лежит в основании логического программирования.

Кроме этих трех задач можно предложить еще много операций над программами, которые хотелось бы выполнять. Но исследования показывают, что многие из них неожиданно сводятся к этим трем, их комбинациям, взаимному применению и самоприменению. Поэтому они являются главными тестовыми задачами метавычислений. Прогресс по методам их решения тут же даст результаты и для других задач.

2 Суперкомпиляция

Автор концепции метасистемного перехода В.Ф. Турчин предложил самый, пожалуй, интересный и перспективный метод метавычислений, названный им *суперкомпиляцией*.

Суперкомпиляция строится по схеме метасистемного перехода от обычных вычислений (это базовый уровень) к следующему (управляющему) уровню. Сама по себе эта схема не нова, а входит в плоть и кровь математики. Ново ее применение ее к программам как к новому математическому объекту. Этот метасистемный переход аналогичен переходу от арифметики к алгебре: от вычислений к формулам. Формулы описывают семейства однотипных вычислений, используя для этого запись с переменными, пробегающими по множествам значений. Формулы затем становятся объектом манипуляций, «метавычислений». Этот путь прошло человечество в процессе зарождения математики [16]; этот путь проходит и каждый школьник, продвигаясь от начальной школы к средней. (Отметим, что в этой одинаковости действует фундаментальный принцип эволюции: филогенез совпадает с онтогенезом.)

Давайте повторим этот путь для программ. Итак, мы имеем обычные вычисления: дана *программа*; даны *начальные значения* (исходные данные); имеется вычислительное устройство, *компьютер*, в которое загружаются программа и данные; нажимаем кнопку «пуск» и устройство начинает работать по шагам. Скажем, если программа описывает функцию *add* двух аргументов, то начальным состоянием может быть терм *add* (2, 3). Считывая на каждом шаге состояние вычислительного устройства, мы наблюдаем *процесс вычислений* и можем записать его трассу, путь. Если в какой-то момент компьютер переходит в состояние «стоп», процесс останавливается. После этого из определенного места конечного состояния берем *результат* вычисления, в данном случае, 5.

Теперь давайте применим к программам схему метасистемного перехода от арифметики к алгебре. Первый шаг: вводим переменные и разрешаем заменять конкретные значениями символами переменных, желая рассмотреть ситуацию в общем виде для всевозможных значений переменных. При обычных вычислениях мы формируем начальные состояния компьютера, загружая в него его конкретные значениями. Теперь вместо некоторых значений мы «подсовываем» ему символы переменных, например, *add* (x_1, 3).

Вторым шагом надо как-то обобщить наше умение проводить вычисления с известными значениями на термы с переменными. Попробуем выполнять шаги вычислений несмотря на то, что начальный терм содержит переменные. Нам может повезти: компьютер может выполнить несколько шагов, «не замечая» переменных. Если крупно повезет, он может дойти до состояния «стоп» и закончить работу. Тогда переменные могут остаться в результате. Но чаще какая-нибудь операция «наткнется» на переменную и не сможет выполниться. Чаще всего такой операцией бывает проверка условия в операторе if-then-else или ему подобном. В этом случае в зависимости от значения

«мешающей» переменной процесс вычисления должен перейти на одну или другую ветвь.

Ниже мы рассмотрим примеры, на которых увидим в деталях, как это происходит и до чего можно довести такие «вычисления с переменными». Но чтобы не томить читателя, грубо обозначим дальнейшие шаги предстоящего пути:

- Первая идея: *будем параллельно проводить вычисления на всех ветвях*. Для этого размножим состояние компьютера и установим по копии на начало каждой из ветвей для продолжения вычислений с этих точек независимо друг от друга. Многократно натыкаясь на переменные и строя таким же образом развилки, мы будем порождать уже не линейный путь вычислений, а *потенциально бесконечное дерево*.
- Внимательно всмотревшись в структуру этого дерева, мы обнаружим вторую идею: это дерево можно использовать как новую программу, эквивалентную исходной при всевозможных значениях переменных, стоящих в начальном состоянии. У этой программы будет только один недостаток: она (потенциально) бесконечна.
- Дальнейшие идеи будут касаться того, как сворачивать потенциально бесконечное дерево программы в конечный граф, который после перевода в подходящий язык программированная и будет результатом преобразования.

Это и будет суперкомпиляция. Чтобы теперь перейти от общей идеи к конкретным алгоритмам суперкомпиляции, нужно выбрать некоторый язык программирования.

2.1 Введение в язык Haskell и примеры функций сложения **add** *и равенства* **eq** *натуральных чисел*

Будем программировать на самом, пожалуй, красивом из современных функциональных языков программирования — языке Haskell [6], по-русски называемом Хаскель. На Хаскеле обращение к функции принято изображать, просто приписывая (через пробел) имя функции к аргументам: *add* 3 5. Скобки используются, только если аргументом является другой вызов функции: *add* 3 (*add* 4 1). Обращения к функциям, их аргументы, числа — это *термы* языка.

Запрограммируем одну из самых простых из осмысленных функций: сложение натуральных чисел *add*. Как и во всех языках программирования, целые числа встроены в Хаскель. Однако не будет ими пользоваться, а определим натуральные числа средствами языка «с нуля».

В математике натуральные числа (включающие ноль) определяются следующими индуктивными правилами:

1. Ноль есть натуральное число. Будем обозначать его термом Z.
2. Для каждого натурального числа n, существует следующее за ним натуральное число, задаваемое «функцией следования» S n.
3. Нет других натуральных чисел, кроме порождаемых этими правилами; причем $\mathrm{Z} \neq \mathrm{S}\ n$ для любого натурального числа n.

Механизм определения типов данных языка Хаскель позволяет компактно кодировать такие правила. Вводя обозначение N для типа «натуральные числа», записываем это определение на Хаскеле:

data N = Z | S N

Здесь **data** — ключевое слово языка Хаскель, предваряющее каждое определение типа данных; N — идентификатор определяемого типа. После знака равенства записываются альтернативы, разделяемые знаками «|». Каждая альтернатива вводит но-

вое имя терма, называемого *конструктором*. Здесь конструкторы — Z и S. После имени конструктора перечисляются типы его аргументов, если они есть. Z — конструктор без аргументов; S — конструктор с одним аргументом типа *N*. Таким образом, целые числа кодируются значениями определенного нами типа *N* следующим образом:

«0» → Z,
«1» → S Z,
«2» → S (S Z),
«3» → S (S (S Z)) и т. д.

В математике определение операции сложения «+» через ноль 0 и функцию следования S $n = n + 1$ задается следующими аксиомами:

$x + 0 = x$,
$x + (y + 1) = (x + y) + 1$.

На Хаскеле мы можем дать рекурсивное определение, один в один отражающее эти аксиомы:

```
add x Z = x
add x (S y) = S (add x y)
```

Эта программа на языке Хаскель определяет функцию *add* с двумя аргументами, каждый из которых имеет тип *N*. (Компилятор Хаскеля сам выводит типы аргументов из определений функций и проверяет, что при обращении к ним все аргументы имеют правильные типы.) Каждое предложение функции задает шаг вычисления при некотором множестве аргументов, описываемом левой частью предложения — *образцом* с переменными. Первое предложение выполняется, когда второй аргумент имеет значение Z, а второе предложение — когда второй аргумент имеет вид S *y*, где *y* — любой терм подходящего типа. На первые аргументы *x* левые части этих предложений ограничений не накладывают.

Для демонстрации семантики языка Хаскель, а затем и суперкомпиляции, определим через функцию *add* еще одну функцию *add2*, прибавляющую двойку S (S Z) к аргументу. Получившаяся программа изображена на рис. 2 в левой колонке.

Исходная программа	Путь вычислений
data *N* = Z | S *N* *add x* Z = *x* *add x* (S *y*) = S (*add x y*) *add2 x* = *add x* (S (S Z))	<u>*add2* (S Z)</u> <u>*add* (S Z) (S (S Z))</u> S (<u>*add* (S Z) (S Z)</u>) S (S (<u>*add* (S Z) Z</u>) S (S (S Z))

Рис. 2. Программа на языке Хаскель, определяющая тип данных *N*, две функции *add* и *add2*, а также путь вычисления *add2* (S Z)

Процесс вычисления программы на любом языке — это пошаговые преобразования *состояний* абстрактной машины-исполнительницы данного языка. Правила этих преобразований определяют *семантику* языка. Последовательность состояний, которые проходит процесс, образуют *путь* вычислений. Например, на рис. 2 в правой колонке показан путь вычисления *add2* (S Z), т. е. прибавления 2 к 1.

Состояние Хаскель-машины — это терм, состоящих из вызовов функций и конструкторов. Начальное состояние — вызов функции, который нужно вычислить; в нашем примере, *add2* (S Z). Конечное состояние не содержит вызовов функций; оно представляет собой ответ — значение функции; в нашем примере, это S (S (S Z)), т. е. «три». Преобразуемые на каждом шаге термы подчеркнуты.

В качестве еще одного примера программы на Хаскеле определим функцию равенства чисел *eq*. Оттолкнемся от одного из вариантов аксиоматического определения операции равенства:

$0 = 0$,
$0 \neq y + 1$,
$x + 1 \neq 0$,
$x = y \Leftrightarrow x + 1 = y + 1$.

Определение рекурсивной функции *eq* на Хаскеле один в один следует этим аксиомам:

eq Z Z = True
eq Z (S *y*) = False
eq (S *x*) Z = False
eq (S *x*) (S *y*) = *eq x y*

2.2 Транзитная прогонка

Теперь совершим обещанный метасистемный переход: введем переменные в обращение к функции и посмотрим, что с этим можно делать.

Рассмотрим вызов функции *add2* x_1. В отличие от предыдущего примера, в нем вместо терма (S Z) проставлена переменная x_1.

Терм *add2* x_1 вместо конкретного состояния машины изображает *обобщенное состояние*, т. е. *множество* состояний. Переменная x_1 является *параметром*, который, пробегая всевозможные значения (допустимые типом *N*), порождает это множество. На жаргоне теории суперкомпиляции термы с переменными, а также описываемые ими множества, называются *конфигурациями*. Чтобы отличать переменные — параметры конфигураций от других видов переменных, будем применять к ним эпитет *конфигурационные*.[4]

Путь вычислений (прогонка)	Остаточная программа
add2 x_1 *add* x_1 (S (S Z)) S (*add* x_1 (S Z)) S (S (*add* x_1 Z) S (S x_1)	**data** *N* = Z | S *N* *add2* x_1 = S (S x_1)

Рис. 3. Суперкомпиляция функции *add2* с начальной конфигурации *add2 x*

Итак, *add2* x_1 — это *начальная конфигурация*. В нее входит *конфигурационная переменная* x_1. На первом шаге не требуется знания значения x_1, чтобы применить определение функции *add2* и заменить вызов *add2* x_1 на *add* x_1 (S (S Z)). (При выполнении шага программная переменная *x* получает значение конфигурационной переменной x_1.) На рис. 3 изображен путь дальнейших вычислений. Нам повезло: переменная x_1 не мешала нам до конца вычислений и вошла в результат S (S x_1). В данном случае мы построили линейный путь — такой же, как при обычном вычислении, но в конфигурациях встречается переменная x_1.

Что это дает? Мы делаем вывод, что обращение к функции *add2* вида *add2* x_1 приводит к результату вида S (S x_1) при любом x_1. Этот факт можно записать новым определением функции *add2*, у которого левая часть — начальная конфигурация, а правая часть — конечная конфигурация:

add2 x_1 = S (S x_1)

Такое определение функции *add2* эквивалентно исходному. Это и есть результат суперкомпиляции функнии *add2*. На жаргоне специалистов по специализации программ результат такого преобразования называется *остаточной программой* (*residual program*). Она содержит «остаток» от *частичных вычислений*: все, что смогли, вычислили, а что не смогли — «выпало в остаток», вошло в «остаточную программу». Наш

[4] Чтобы отличать конфигурационные переменные от имен функций и программных переменных (и те, и другие мы записываем курсивом) конфигурационные переменные будем писать с индексами.

случай оказался простым: все шаги выполнились, а «в остатке» только начальная и конечная конфигурации.

Перед тем, как перейти к общему случаю, введем еще несколько терминов. Процесс вычислений над конфигурациями называется *прогонкой.* Базовый случай, рассмотренный в этом разделе, — выполнение шагов, как при обычном вычислении, когда переменные не мешают, — называется *транзитной прогонкой*, выполнением *транзитных шагов. Транзитными конфигурациями* назовем те из них, у которых следующий шаг транзитный, т. е. все множество состояний, описанных конфигурацией, делает следующий шаг по одному и тому же предложению программы. Теперь перейдем к *нетранзитным* конфигурациям, шагам и прогонке.

2.3 Построение дерева процессов

Картина усложнится, если в нашем примере функции *add* мы поменяем местами известный и неизвестный аргументы. Определим функцию «прибавить *к* 2» — *addTo2*, поместив константу S (S Z) в первый аргумент функции *add*, а переменную *x* во второй:

$$addTo2\ x = add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x$$

Далее идем по проторенной дорожке: для суперкомпиляции этой функции формируем начальную конфигурацию с вызовом максимально общего вида — с конфигурационными переменными вместо всех аргументов. Поскольку у функции *addTo2* лишь один аргумент, начальная конфигурация *addTo2* x_1 содержит одну конфигурационную переменную x_1.

Как и в предыдущий раз, переменная x_1 не мешает выполнить первый шаг транзитно, перейдя к конфигурации *add* (*S* (S Z)) x_1. Но на этом наше везение заканчивается: теперь процесс вычисления должен выбрать либо первое, либо второе предложение функции *add* в зависимости от значения второго аргумента, а он является конфигурационной переменной x_1.

Таким образом, множество состояний, изображенное конфигурацией *add* (*S* (S Z)) x_1, распадается на два непересекающихся подмножества: элементы одного из них (когда x_1 равен Z) преобразуются по первому предложению функции *add*, а элементы другого (когда x_1 имеет вид S x) — по второму предложению. Эти подмножества изобразимы конфигурациями *add* (*S* (S Z)) Z и *add* (*S* (S Z)) (S x_2) соответственно, где x_2 — новая конфигурационная переменная. Они строятся из исходной конфигурации *add* (*S* (S Z)) x_1 подстановкой вместо переменной x_1 значений Z и S x_2 соответственно. Эти подстановки будем называть *сужениями* и записывать так:

$$x_1 \to \mathrm{Z},$$
$$x_1 \to \mathrm{S}\ x_2.$$

Перед тем, как сделать следующим логический шаг, подытожим, что мы прошли. Сначала мы перешли от конкретных состояний к множествам состояний (конфигурациям), внедрив в описания состояний переменные. Конкретные пути вычислений стали множествами путей, но для начала мы ограничились путями, проходящими через одни и те же предложения программы и изобразимые одной общей линейной последовательностью конфигураций. Теперь наступило время обобщить до множества путей общего вида.

Множество путей с развилками можно изобразить *потенциально бесконечным деревом* (рис. 4). Вершины дерева соответствуют конфигурациям. У дерева есть одна *начальная* конфигурация (*вход*), помеченная входящей стрелкой с кружком, и много *конечных* конфигураций (*выходов*), помеченных исходящими стрелками с кружками. Дуги дерева соответствуют либо транзитным шагам, либо сужениям. Из вершины выходит либо одна транзитная дуга, либо две (или больше) дуги, помеченных сужениями.

Транзитная дуга описывает шаг вычислений по одному из правил программы, дуга-сужение — выделение подмножества состояний, которые будут преобразованы на следующем шаге по одному предложению программы (рис. 4).

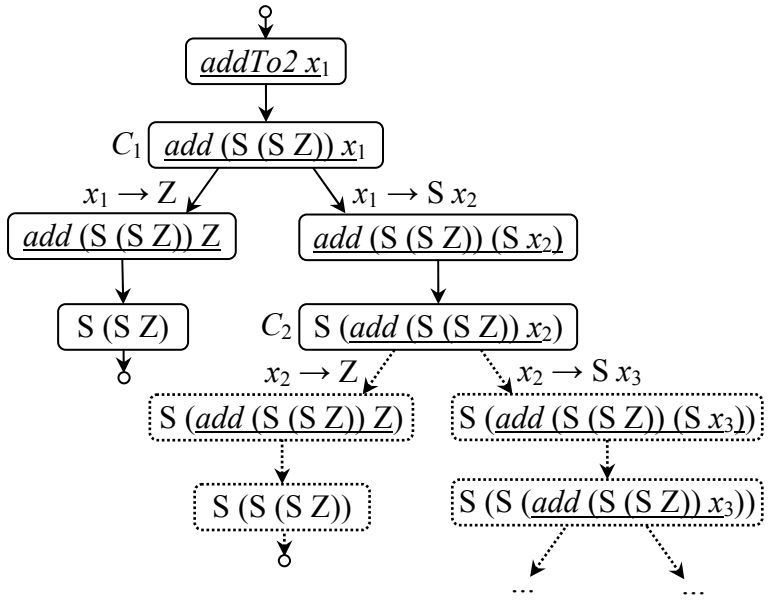
Исходная программа	Потенциально бесконечное дерево процессов
data N = Z | S N *add* x Z = x *add* x (S y) = S (*add* x y) *addTo2* x = *add* (S (S Z)) x	

Рис. 4. Исходная программа и дерево процессов для функции *addTo2*

После разбиения конфигурации *add* (S (S Z)) x_1 на две подконфигурации *add* (S (S Z)) Z и *add* (S (S Z)) (S x_2) сужениями $x_1 \to$ Z и $x_1 \to$ S x_2 продолжаем прогонку каждой из них по отдельности (рис. 4). Первая за 2 транзитных шага дойдет до конечной конфигурации S (S Z). Вторая через один шаг породит конфигурацию S (*add* (S (S Z)) x_2), которая требует сужений переменной x_2 аналогично пройденной конфигурации *add* (S (S Z)) x_1. Нетрудно видеть, сравнивая конфигурации C_1 = *add* (*S* (S Z)) x_1 и C_2 = S (*add* (S (S Z)) x_2), что процесс прогонки будет повторно разворачивать похожие фрагменты дерева до бесконечности. Фрагмент, показанный на рис. 4 пунктиром, отличается от своего предшественника дополнительным конструктором S (…) в конфигурациях и переменной x_2 вместо x_1. При дальнейшей развертке будут порождаться новые переменные сужениями вида $x_i \to$ S x_{i+1}, а число конструкторов S (…) в конфигурациях будет нарастать.

Будем называть построенное дерево *деревом процессов*, подчеркивая тот факт, что процессам соответствуют пути в дереве, или *деревом конфигураций*, подчеркивая тот факт, что вершины дерева — это конфигурации.

Следующий наш шаг — свертка потенциально бесконечного дерева процессов в конечный *граф конфигураций*. Этим займемся через один раздел. Сначала обсудим семантику (смысл) дерева процессов. При свертке в граф мы должны сохранить ее.

2.4 Дерево процессов как остаточная программа

Какова семантика дерева процессов? Какими свойствами оно обладает?

Мы строили дерево так, чтобы конфигурации включали в себя все возможные состояния, а дуги учитывали все возможные разветвления. Благодаря этому можно сказать, что *дерево процессов «содержит в себе» все возможные процессы вычислений*.

Состояние процесса вычислений задается указанием на текущую вершину дерева (конфигурацию) и набором текущих значений конфигурационных переменных.
Начальное состояние: в начальный момент переменным начальной конфигурации присваиваются исходные значения.
Процесс вычислений состоит из итеративного выполнения следующих шагов:

1. Если текущая конфигурация *конечная*, оканчиваем вычисление и выдаем в качестве результата конфигурацию, записанную в вершине, подставив в нее текущие значения конфигурационных переменных.
2. Если текущая конфигурация *транзитная,* т. е. из нее исходит лишь одна *транзитная* дуга, то:
   2.1. переходим по этой дуге к следующей вершине;
   2.2. набор переменных и их значений не меняется.
3. Если текущая конфигурация — *ветвление*, т. е. из нее выходят несколько дуг с сужениями $x_i \to \ldots x_j \ldots x_{j+1} \ldots$, где $x_j, x_{j+1}, \ldots$ — новые переменные, то:
   3.1. переходим по той дуге, у которой значение сужаемой переменной x_i равно правой части при некоторых значениях входящих в нее переменных $x_j, x_{j+1}, \ldots$;
   3.2. из набора конфигурационных переменных удаляем сужаемую переменную x_i и добавляем переменные из правой части сужения $x_j, x_{j+1}, \ldots$, присвоив им значения, найденные в предыдущем пункте.

Во всех правилах, при переходе к следующей вершине, оставляем в наборе переменных только те, от которых зависит следующая конфигурация. Остальные переменные удаляем из текущего состояния.

Рис. 5. Алгоритм интерпретация дерева процессов как программы

Более точно это формулируется так:

- каждому набору *значений* конфигурационных переменных в начальной конфигурации соответствует *процесс*, использующий эти значения как исходные данные;
- каждому процессу вычислений соответствует *путь* в дереве.

Обратное утверждение — любому пути в дереве соответствует некоторый процесс вычислений — в общем случае неверно, хотя в нашем примере это так. При построении дерева (а потом и графа) нам важно не потерять возможные вычисления, а лишние нереализуемые пути, которым не соответствует никакие процессы, не мешают.

Что же можно делать с таким представлением множества процессов?

Оказывается, дерево процессов содержит в себе всю необходимую информацию для вычислений, и, чтобы выполнить процесс, уже не нужно обращаться к исходной программе. Это значит, что *дерево процессов можно интерпретировать как программу, эквивалентную исходной.* Алгоритм интерпретации приведен на рис. 5.

Основные идеи алгоритма интерпретации дерева процессов состоят в следующем:

- конфигурационные переменные становятся локальными переменными остаточной программы;
- транзитные конфигурации ничего не значат;
- сужения — это проверки, с помощью которых реализуются ветвления в программе;
- конечные вершины описывают выдачу результата после подстановки значений конфигурационных переменных в конфигурацию.

Возможность интерпретации дерева процессов как программы — это принципиальный момент на пути к суперкомпиляции. Дерево процессов — это не просто другая эквивалентная форма программы, но и более эффективная по числу исполняемых операций: в дереве остались операции только над значениями конфигурационных переменных, а все операции над известной частью данных выполнились в процессе прогонки и развертки дерева. Это и есть основной источник оптимизаций, выполняемых суперкомпилятором.

2.5 Упрощение дерева процессов

Построенное дерево процессов можно упростить, сохраняя возможность его интерпретации по приведенному выше алгоритму. Транзитные вершины ничего не значат для интерпретации, поэтому их можно удалить. При интерпретации всех вершин, кроме входной и выходных, собственно конфигурации не используются, а нужны лишь списки переменных, от которых они зависят. Поэтому во внутренних вершинах вместо конфигураций можно оставить лишь списки конфигурационных переменных.

Упрощенное таким образом дерево процессов примера *addTo2* изображено на рис. 6. Во входной и выходных вершинах выписаны начальная и конечная конфигурации, а во внутренних записаны лишь множества конфигурационных переменных $\{x_2\}$, $\{x_3\}$, $\{x_4\}$.

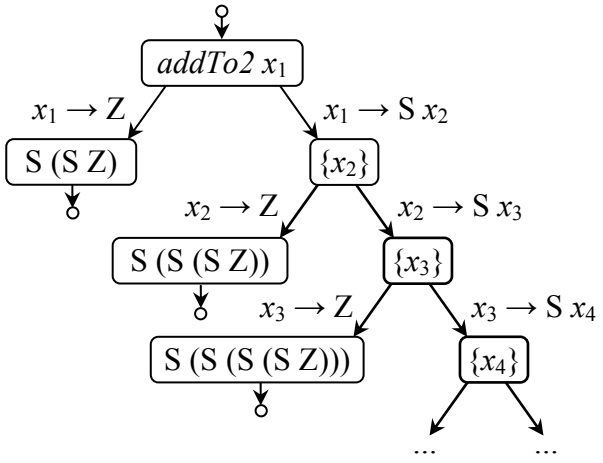
Исходная программа	Упрощенное дерево процессов
data N = Z | S N *add* x Z = x *add* x (S y) = S (*add* x y) *addTo2* x = *add* (S (S Z)) x	

Рис. 6. Исходная программа и упрощенное дерево процессов для функции *addTo2*: без транзитных вершин и со списками переменных вместо конфигураций во внутренних вершинах

2.6 Свертка дерева процессов в граф конфигураций

У дерева процессов лишь один недостаток — в общем случае оно бесконечно. Следующая задача — научиться свертывать потенциально бесконечное дерево в конечный граф так, чтобы у него была такая же семантика, как у дерева, т. е., чтобы существовал алгоритм интерпретации графа как остаточной программы.

Для свертки в конечный граф в процессе построения дерева к конфигурациям применяются три операции: *зацикливание, рассечение конфигураций* и *обобщение конфигураций.* Конечность графа достигается за счет зацикливания, а рассечение и обобщение помогают приводить конфигурации к виду, подходящему для зацикливания. В

нашем примере *addTo2* двух первых операций — зацикливания и рассечения — достаточно, что превратить дерево на рис. 4 в граф, изображенный на рис. 7.

Идея свертки дерева в граф проста: вместо развертки точно такого же дерева при повторении конфигурации проводим дугу от повторившийся конфигурации назад к предыдущей. Если конфигурации отличаются именами переменных, то на дуге записываем соответствие переменных, которое будет интерпретироваться как оператор присваивания. Эта операция над графом и называется *зацикливанием.*

Часто, как в нашем примере, старая конфигурация повторяется не целиком, а входит в новую как часть. Чтобы формально изобразить этот факт в графе, разбиваем конфигурацию на две части — объемлющий терм и вложенную повторившуюся конфигурацию. Вложенную конфигурацию зацикливаем, а для объемлющей продолжаем прогонку, поставив вместо вынутой подконфигурации новую конфигурационную переменную, называемую *связной*. Она изображает тот факт, что после рассечения объемлющая конфигурация «ничего не знает» о значениях, которые могут возникнуть в результате вычисления вложенной конфигурации; объемлющая конфигурация должна рассчитывать на любое значение. Эта операция над конфигурацией называется *рассечением.* Интерпретация рассеченной конфигурации аналогична вызову подпрограммы.

Посмотрим, что произойдет с деревом на рис. 4, если мы будем применять эти операции в процессе его построения. Из конфигурации $C_1 = add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x_1$ через один шаг на правой ветви возникла конфигурация $C_2 = \mathrm{S}\ (add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x_2)$. Видим, что в конфигурацию C_2 вложен терм $add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x_2$ совпадающий с C_1 с точностью до переименования переменной. Это значит, что интерпретация дерева, начиная с конфигурации C_2, даст такой же результат, как и с C_1, но заключенный в объемлющий терм S (…). Мы изображаем это графом следующим образом (рис. 7):

- *рассекаем* конфигурацию $C_2 = \mathrm{S}\ (add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x_2)$ на объемлющую конфигурацию $C_{21} = \mathrm{S}\ (x_3)$, где новая *связная* переменная x_3 обозначает место изъятия, и вложенную конфигурацию $C_{22} = add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x_2$, и изображаем связь между ними специальной *связной* дугой (пунктирной), помеченной переменной x_3;
- соединяем конфигурацию $C_{22} = add\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}))\ x_2$ с конфигурацией C_1 дугой, помеченной *присваиванием* $x_1 := x_2$, обозначающим способ получения значения x_1 при перемещении из конфигурации C_{22} в C_1.

Теперь упростим граф на рис. 7, очистив его от транзитных вершин и дуг, как мы это делали для дерева, а в единственную оставшуюся внутреннюю вершину C_{22} запишем список конфигурационных переменных $\{x_2\}$, поскольку, как и для дерева, сама конфигурация не имеет значения при интерпретации графа. Полученный граф (рис. 8.) можно либо непосредственно интерпретировать как остаточную программу, либо перевести на исходный язык Хаскель. На рис. 8 слева внизу приведена эквивалентная программа на Хаскеле, которую можно вывести из графа чисто формально. Алгоритм перевода приводить не будем, упомянем лишь естественное пожелание к нему — выдавать остаточную программу в как можно более компактном и читабельном для человека виде. Отметим, что для компиляции в команды машины перевод на язык высокого уровня не требуется: граф, фактически являющийся блок-схемой программы, для этого подходит гораздо лучше.

Сравнивая остаточную программу *addTo2* с исходной функцией *add*, видим, что у функции был удален первый аргумент *x*, а его значение S (S Z) подставлено в правую часть первого предложения, куда входит *x*. Преобразование вполне логичное. Хотя оптимизация в этом примере оказалась не очень большой, важно, что мы действовали по общему алгоритму, который в других случаях может давать неожиданные результаты. Такой пример рассмотрим ниже в разделе 4.

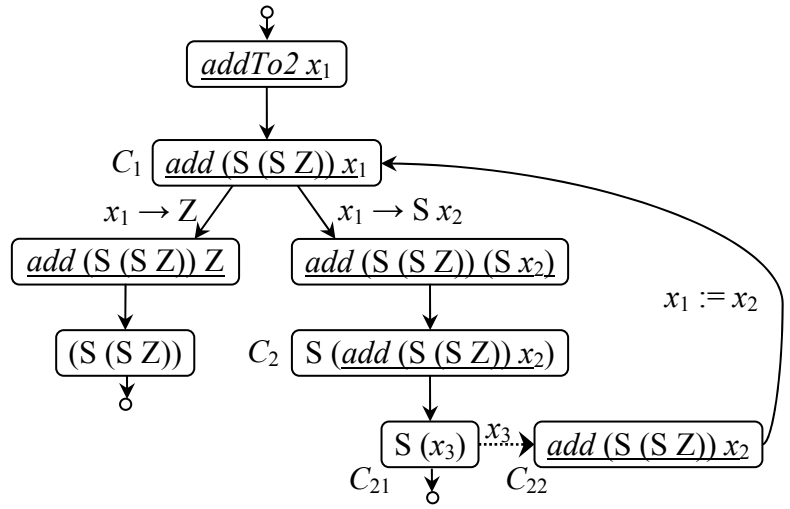
Исходная программа	Граф конфигураций
data N = Z | S N *add* x Z = x *add* x (S y) = S (*add* x y) *addTo2* x = *add* (S (S Z)) x	

Рис. 7. Исходная программа и граф конфигураций для функции *addTo2*

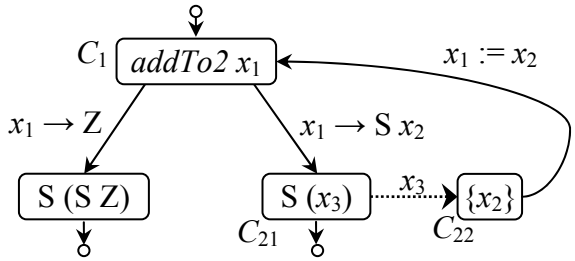
Исходная программа	Упрощенный граф конфигураций
data N = Z | S N *add* x Z = x *add* x (S y) = S (*add* x y) *addTo2* x = *add* (S (S Z)) x	
Остаточная программа	
addTo2 Z = S (S Z) *addTo2* (S x_2) = S (*addTo2* x_2)	

Рис. 8. Исходная программа, упрощенный граф конфигураций и эквивалентная ему остаточная программа на Хаскеле для функции *addTo2*

2.7 Алгоритм интерпретации графа конфигураций

Алгоритм интерпретации графа является естественным расширением алгоритма интерпретации дерева (рис. 5), поскольку дерево — это частный случай графа, а алгоритм интерпретации дерева никак не учитывал, что он обрабатывает именно дерево, а не произвольный граф. Чтобы получить из него алгоритм интерпретации графа, нужно лишь добавить обработку новых разновидностей дуг. Дополнительные правила обработки *дуг с присваиваниями* и *связных дуг* приведены на рис. 9.

4. Если из текущей конфигурации выходит одна дуга с *присваиванием* $x_i := T$, где в общем случае T — некоторый терм из конструкторов и переменных, то:
   4.1. переходим по этой дуге к следующей вершине;
   4.2. добавляем переменную x_i в набор переменных, присвоив ей значение терма T после подстановки в него текущих значений конфигурационных переменных.
5. Если из текущей конфигурации выходит *связная* дуга, помеченная переменной x_i, то:
   5.1. рекурсивно запускаем процесс интерпретации графа, начиная с конфигурации, на которую указывает связная дуга;
   5.2. когда этот процесс интерпретации закончится с некоторым результатом, присваиваем его связной переменной, добавив ее к набору переменных;
   5.3. продолжаем обработку текущей конфигурации по правилам 1–5.

Рис. 9. Дополнительные правила интерпретации вершин графа, после добавления которых к алгоритму интерпретации дерева (рис. 5), он превращается в алгоритм интерпретации графа

3 Применение суперкомпиляции для доказательства теорем

3.1 Подход к доказательству общезначимости вычислимых предикатов

Мы в какой-то степени научились преобразовывать программы. Поэтому из математических утверждений выберем такие, которые можно выразить программами.

Пусть дана программа *p*, описывающая на Хаскеле предикат, т. е. функцию, выдающую значения True и False:

```
p x = …
… … …
```

Область данных *Bool* (от Boolean — Булевская), состоящая из констант True и False, на Хаскеле определяется так:

```
data Bool = True | False
```

Можно попытаться прооптимизировать определение функции *p* с помощью любого оптимизатора программ для языка Хаскель. Если повезет, оптимизатор выдаст в ответ функцию *p* в виде программы, всегда вырабатывающую True за один шаг:

```
p x = True
```

В этом случае, полагаясь на корректность оптимизатора, мы заключаем, что предикат *p* является общезначимым, т. е. истинным при любом *x*:

$$(\forall x)\ p\ x.$$

Нам может повезти и по-другому. Прооптимизированная программа может принять такой вид:

```
p константа = False
p … = …
… … …
```

Тогда мы заключаем, что наша гипотеза об общезначимости предиката *p* опровергнута с предъявлением контрпримера, записанного в левой части первого предложения.

В остальных случаях мы ничего не можем сказать о предикате. Компьютер, как и человек, иногда доказывает гипотезу, иногда опровергает, и часто разводит руками: «не могу».

Этот подход к использованию суперкомпиляции для доказательства теорем был предложен В.Ф. Турчиным в работе [12].

3.2 Постановка задачи суперкомпилятору на доказательство коммутативности прибавления единицы

Выше в разделе 2.1 мы определи натуральные числа и операции сложения и равенства. Теперь можем формулировать их свойства в виде функций-предикатов на Хаскеле. Рассмотрим одну из самых простых теорем в формальной арифметике — коммутативность сложения:

$$(\forall x, y)\ \ x + y = y + x.$$

Для еще большей простоты ограничимся случаем $y = 1$:

$$(\forall x)\ \ x + 1 = 1 + x.$$

Выразим через ранее определенные функции *add* и *eq* предикат *p*, общезначимость которого надо доказать:

p x = *eq* (*add x* (S Z)) (*add* (S Z) *x*)

На рис. 10 целиком приведена программа, определяющая функцию *p* вместе со вспомогательными функциями, и пример пути вычислений для *p* (S (S Z)). Теперь мы можем подать эту функцию суперкомпилятору в надежде получить остаточную программу вида

p x = True

Как увидим в разделе 3.4, так оно и получится.

Исходная программа	Путь вычислений
data *N* = Z | S *N* *add x* Z = *x* *add x* (S *y*) = S (*add x y*) *eq* Z Z = True *eq* Z (S *y*) = False *eq* (S *x*) Z = False *eq* (S *x*) (S *y*) = *eq x y* *p x* = *eq* (*add x* (S Z)) (*add* (S Z) *x*)	*p* (S (S Z)) *eq* (*add* (S (S Z)) (S Z)) (*add* (S Z) (S (S Z))) *eq* (S (*add* (S (S Z)) Z)) (*add* (S Z) (S (S Z))) *eq* (S (S (S Z))) (*add* (S Z) (S (S Z))) *eq* (S (S (S Z))) (S (*add* (S Z) (S Z))) *eq* (S (S (S Z))) (S (S (*add* (S Z) Z))) *eq* (S (S (S Z))) (S (S (S Z))) *eq* (S (S Z)) (S (S Z)) *eq* (S Z) (S Z) *eq* Z Z True

Рис. 10. Программа на Хаскеле, определяющая предикат *p x* = (*x* + 1 = 1 + x), и путь вычисления *p* 2

3.3 О порядке вычислений при выполнении программы и при прогонке

При вычислении обращения к функции *p* на рис. 10 использован порядок вычислений, привычный для программистов, использующих распространенные языки программирования: композиция функций вычисляется *изнутри наружу*: на каждом шаге выбирается самый левый вложенный вызов функции; у него в аргументах стоят готовые значения. Такой порядок вычислений называют *аппликативным*. Однако в Хаскеле принят другой порядок, называемый *ленивым* (*lazy evaluation*) или *снаружи*

внутрь: на каждом шаге сначала выбираем самый *внешний* левый вызов функции; если при выполнении шага окажется, что требуется значение некоторого невычисленного вызова функции, то выбор переходит на него, и так далее, пока не удастся выполнить шаг. Эпитет «ленивый» отражает тот факт, что шаг вычисления вызова функции делается только тогда, когда понадобится его значение, причем выполняется столько шагов, сколько надо для выдачи требуемой части результата, и не более того.

Для суперкомпиляции не важно, какой порядок вычислений используется «на базовом уровне» — при выполнении программы, но оказывается, что использование *ленивого* порядка «на метауровне», т. е. при прогонке и построении графа конфигураций, дает лучшую остаточную программу.

Возможны некоторые комбинации аппликативного и ленивого порядка при прогонке. Например, хороший результат суперкомпиляции дает такой вариант, который мы используем в следующем разделе. А именно, транзитные шаги выполняем в аппликативном порядке, изнутри наружу: начиная с самого левого вложенного вызова функций, выполняем до конца или до того момента, когда помешает конфигурационная переменная или недовычисленный вызов функций в аргументе. А когда ни один вызов функции нельзя вычислить транзитно, выбираем вызов функции и конфигурационную переменную для сужения в ленивом порядке, снаружи внутрь: пытаемся выполнить самый внешний вызов функции; если мешает конфигурационная переменная, сужаем ее; если мешает вложенный вызов функции, переходим на него и делаем аналогичную попытку; в конце концов мы дойдем до переменной, которую надо сужать.

3.4 Суперкомпиляция предиката x+1 = 1+x

Наконец, мы готовы посмотреть, как справится суперкомпилятор с доказательством простой теоремы из формальной арифметики:

$$(\forall x)\ \ x + 1 = 1 + x.$$

В левой колонке на рис. 11 еще раз приведена целиком программа на Хаскеле, определяющая предикат $p\ x$, стоящий под квантором в нашей теореме. Задание суперкомпилятору даем в виде начальной конфигурация $p\ x_1$ с конфигурационной переменной x_1, пробегающей по всему множеству возможных аргументов функции p. (Мы и компилятор с Хаскеля знаем, что это N, но суперкомпилятору все равно: он будет формально следовать программе.)

Первые 3 шага прогонки — транзитные: вычисляем все, что можно, не выполняя сужений. В конфигурации $C_1 = eq\ (\mathrm{S}\ x_1)\ (add\ (\mathrm{S}\ \mathrm{Z})\ x_1)$ вызов функции *eq* требует значения функции *add*, которому, в свою очередь, надо знать значение конфигурационной переменной x_1. Сужаем переменную x_1 по образцам из двух предложений функции *add*:

$$x_1 \to \mathrm{Z},$$
$$x_1 \to \mathrm{S}\ x_2.$$

На левой ветви, после сужения $x_1 \to \mathrm{Z}$, транзитными шагами доходим до конечной конфигурации True. На правой ветви, после сужения $x_1 \to \mathrm{S}\ x_2$, через два транзитных шага получаем конфигурацию $C_2 = eq\ (\mathrm{S}\ x_2)\ (add\ (\mathrm{S}\ \mathrm{Z})\ x_2)$, которая с точностью до переименования переменных совпадает с конфигурацией C_1. Зацикливаем C_2 на C_1 с присваиванием $x_1 := x_2$. Построение графа конфигураций закончено.

Теперь, как и раньше, упрощаем граф, удаляя транзитные конфигурации и заменяя в единственной внутренней вершине C_2 описание конфигурации на список переменных $\{x_2\}$, от которых она зависит (рис. 12). По такому графу конфигураций легко строится остаточная программа на Хаскеле. Она заметно короче исходной и работает в несколько раз быстрее, но нам она особенно интересна тем, что уже видно, что она всегда выдает True.

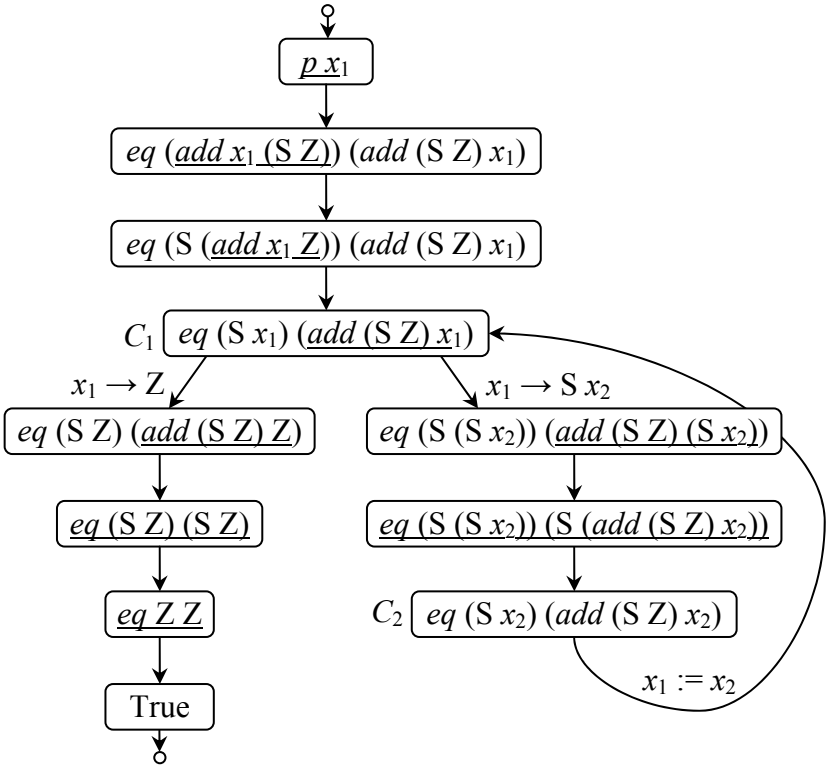
Исходная программа	Граф конфигураций
data N = Z | S N *add* x Z = x *add* x (S y) = S (*add* x y) *eq* Z Z = True *eq* Z (S y) = False *eq* (S x) Z = False *eq* (S x) (S y) = *eq* x y *p* x = *eq* (*add* x (S Z)) (*add* (S Z) x)	

Рис. 11. Суперкомпиляция предиката *p* x = (x + 1 = 1 + x)

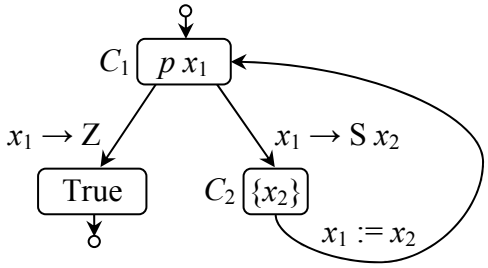
Остаточная программа	Упрощенный граф конфигураций
data N = Z | S N *p* Z = True *p* (S x_1) = *p* x_1	

Рис. 12. Остаточная программа для предиката *p* x = (x + 1 = 1 + x)

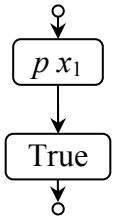
Остаточная программа	Окончательный граф конфигураций
p x_1 = True	

Рис. 13. Остаточная программа для предиката *p* x = (x + 1 = 1 + x) после дополнительного преобразования «упрощение графа с константным выходом»

Все было бы хорошо, только это преобразование графа в общем случае выдаст не совсем эквивалентную программу: если вычисление по исходному графу завершается, то результат действительно такой же, что и при вычислении преобразованного графа. Однако, исходная программа может и «зациклится». Тогда она не выдаст никакого ответа, в то время как вычисление по преобразованному графу всегда выдает ответ за один шаг. Получается, что исходная и остаточная программы неэквивалентны.

Тем не менее, в нашем случае мы уверены (и можем доказать индукцией по длине аргумента), что предикат $p\,x = (x + 1 = 1 + x)$ определен всегда. С учетом этой оговорки, можно считать, что машина с помощью суперкомпилятора доказала теорему $(\forall x)\,p\,x$.

В заключение отметим, что такое преобразование программ соответствует применению *математической индукции* в доказательствах. Правда, это преобразование шире индукции: оно превращает частично определенные функции в полностью определенные, в то время как правило индукции применяется только к всюду определенным предикатам. Эту разницу надо учитывать при использовании преобразований программ для машинного доказательства теорем по описанной схеме.

4 Подводим итоги

4.1 Алгоритм суперкомпиляции

В рассмотренных примерах метод суперкомпиляции работал по достаточно простому алгоритму:

- *прогонка*: разворачиваем дерево процессов, используя ленивый порядок вычисления вызовов функций в конфигурациях (с возможными вариантами, например, использование аппликативного порядка для транзитных шагов);
- если встретилась конфигурация, равная одной из пройденных с точностью до переименования переменных, *зацикливаем*, т. е. проводим дугу от новой конфигурации к старой с оператором *присваивания* на дуге (пример на рис. 11);
- если встретилась конфигурация, *часть* которой равна одной из пройденных с точностью до переименования переменных, *рассекаем* конфигурацию на две части; в объемлющую часть вместо вложенной помещаем новую *связную* переменную и запоминаем ее на специальной *связной* дуге от объемлющей конфигурации к вложенной; продолжаем прогонку каждой конфигурации независимо друг от друга (пример на рис. 7).

Однако, этих действий недостаточно, чтобы суперкомпилятор гарантированно завершал работу, выдавая конечный граф конфигураций для любой программы и для любой начальной конфигурации. Во многих случаях останутся бесконечные ветви дерева процессов. (В качестве примера можно рассмотреть предикат, изображающий общий случай коммутативности сложения: $p\,x\,y = (x + y = y + x)$.)

Чтобы всегда иметь возможность свернуть потенциально бесконечное дерево в конечный граф, надо ввести еще одну операцию — *обобщение конфигурации.*

4.2 Обобщение конфигурации

Обобщение конфигурации — это принятие решения о том, что вместо выполнения следующего шага прогонки некоторой конфигурации C_k по приведенным выше правилам следует заменить некоторую ее часть A, не содержащую вызовов функций, на новую конфигурационную переменную x_i, получая таким образом обобщенную конфигурацию C_{k+1}. Конфигурация C_{k+1} становится следующей за C_k; дуге от C_k к C_{k+1} приписывается оператор присваивания $x_i := A$ (рис. 14). Если одинаковая часть A встречается в конфигурации несколько раз, то все или часть вхождений можно заменить на одну переменную x_i.

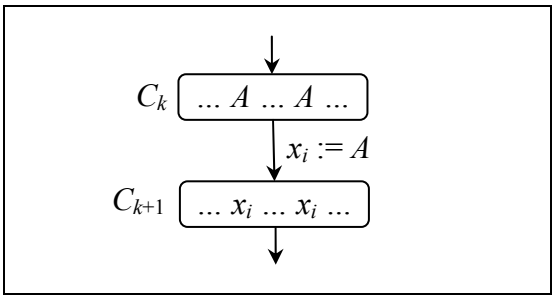


Рис. 14. Обобщение конфигурации C_k до конфигурации C_{k+1} с введением новой конфигурационной переменной x_i и присваивания $x_i := A$

Особенность обобщения в том, что достаточно легко сформулировать, *что такое* обобщение, но не просто решить, *когда и как* его применять, *какую* часть конфигурации заменять на переменную.

Обобщение — это самая сложная часть метода суперкомпиляции и основное направление исследований. В.Ф. Турчиным [8] и другими предложено несколько интересных и практичных методов обобщения, но тема далеко не закрыта и не будет никогда закрыта, поскольку имеет дело с алгоритмически неразрешимой задачей.

4.3 Метасистемные переходы в структуре метода суперкомпиляции

Суперкомпиляция, как и другие мощные методы метавычислений, разрабатываются для осуществления крупномасштабного метасистемного перехода [16] над алгоритмами, программами, и вообще формальными системами: сделать программы объектом легкого манипулирования, глубоких преобразований. Однако и сам метод суперкомпиляции имеет структуру метасистемы, возникшей в результате метасистемного перехода, и будет развиваться дальше как иерархия метасистем, как результат последовательности метасистемных переходов. Это весьма поучительно, поскольку дает образец рецепта по «ручному» совершению метасистемных переходов. Давайте подытожим метод суперкомпиляции с этой точки зрения. Для удобства восприятия разобьем его на два акта.

Первый акт на пути к первому метасистемному переходу над программами резюмируем так:

- состояние как объект деятельности компьютера по программе →
  множество состояний (конфигурация),
- вычисление как процесс во времени, деятельность компьютера →
  путь вычисления как объект →
  множество путей (дерево процессов, граф конфигураций).

Переход от отдельных объектов какого-то вида к множествам этих объектов и от операций над первичными объектами к операциям над объектами-множествами — типичная схема метасистемного перехода, предлагаемого математикой. Переход от процессов, разворачивающихся во времени, от деятельности над какими-то объектами к метаобъектам, статически обозначающих деятельность, — также распространенная схема метасистемных переходов.

Итак, мы имеем *метаобъекты* — конфигурации, деревья процессов, которые потом обобщаются до графов конфигураций. Теперь совершаем второй акт на пути к

метасистемному переходу над программами — определяем операции над метаобъектами, создаем *метадеятельность*:

- *прогонка*: вычисления над конфигурациями (транзитные шаги, сужения) и развертка потенциально бесконечного дерева конфигураций и процессов;
- *конфигурационный анализ:* операции над графом конфигураций с целью свертки бесконечного дерева в конечный граф: зацикливание, рассечение и обобщение конфигураций.

Задачей последующих метасистемных переходов в развитии суперкомпиляции будет построение развитых методов и стратегий обобщения. При существующих методах основным объектом в суперкомпиляторах является конфигурации. Объектами следующего метауровня станут множества путей вычислений, графы конфигураций и тому подобные метаобъекты. В этих направлениях появились интересные идеи (например [10, 11]), но основные достижения еще впереди.

4.4 Состояние дел по разработке суперкомпиляторов

В настоящее время, помимо небольших суперкомпиляторов разных авторов, реализованных для личных исследований, существуют два крупных проекта по экспериментальным суперкомпиляторам:

- суперкомпилятор SCP4 для функционального языка Рефал-5 [7]. Это последний из серии суперкомпиляторов, реализованных В.Ф. Турчиным в процессе разработки методов суперкомпиляции. В последние годы его дальше развивает А.П. Немытых;
- суперкомпилятор JScp для языка Java [4]. Это первая попытка обобщить методы суперкомпиляции, изначально возникшие для функциональных языков, на практические объектно-ориентированные языки.

Информация по методам суперкомпиляции разбросана по статьям, в первую очередь, В.Ф. Турчина [8-12, 17]. Компактное определение прогонки для простого модельного функционального языка можно найти в статье [3]. Систематическое изложение прогонки для похожего модельного языка и исследование того, что можно сделать на ее основе, содержится в книге С.М. Абрамова [13].

4.5 Немного истории и другие методы метавычислений

Хотя суперкомпиляция — самый интересный и перспективный, с нашей точки зрения, метод метавычислений, он далеко не единственный. Многим ученым идеи о важности специализации и других глубоких преобразованиях программ приходили независимо друг от друга.

Официальная история метавычислений отсчитывается от статьи Ёсихико Футамуры 1971 г. [1]. Он был первым, кто понял, что с помощью самоприменения специализатора можно превращать интерпретаторы в компиляторы, автоматизируя процесс построения компиляторов. В 80-е гг. предложенные им формулы получили название *проекций Футамуры.* В статье [1] он сделал набросок своего метода *частичных вычислений* (*partial computation*). Однако, судя по отсутствию публикаций в течение более 10 лет, ему удалось сформировать группу, работающую над этой темой, и продвинуться дальше лишь в 80-е гг. Новая версия его метода получила название *обобщенных частичных вычислений* (*generalized partial computation*). По общей структуре метод похож на суперкомпиляцию, но в процессе прогонки он распространяет дополнительную информацию о значениях конфигурационных переменных в виде логических утверждений и использует систему машинного доказательства теорем для проверки вложений конфигураций друг в друга.

В 1970 г. В.Ф. Турчин закончил рукопись книги «Феномен науки», излагающую концепцию метасистемного перехода. Тогда он сознательно поставил себе задачу совершить метасистемный переход над программами. Я вспоминаю семинар в ИПМ зимой 1971–72 гг., когда он выполнял мелом на доске шаги простого интерпретатора на Рефале, поставив переменные вместо части данных, и, дойдя до результата, воскликнул: «Смотрите, компилирует!». В 1972 г. он опубликовал статью [17] с изложением алгоритма прогонки в виде системы эквивалентных преобразований программ на Рефале. В ней он также показал, что прогонка обладает способностью проводить «инверсные вычисления» — зная результат работы программы, найти аргумент — и что на основе прогонки можно определить «универсальный решающий алгоритм». Ему очень нравилась его аббревиатура УРА, звонко отмечающая первое достижение на пути в тому, что позже получило название метавычислений.

В.Ф. Турчин закончил разработку методов суперкомпиляции «на бумаге» к осени 1974 г., когда он представил их на серии лекций группе студентов. С тех пор идеи метода не претерпели принципиальных изменений за исключением стратегий обобщения конфигураций. Первый изящный и практичный алгоритм обобщения был найден им позднее и доложен на конференции в 1987 г. [8]. Лекции 1974 г. и первый подробный отчет по суперкомпиляции 1980 г. [10] содержали также идеи о методах следующего метасистемного перехода над конфигурационным анализом: так называемый *окрестностный анализ* и преобразования программ через описания множеств путей *грамматиками*. Эти идеи все еще ждут своего дальнейшего развития.

В середине 70-х гг. независимо от В.Ф. Турчина и Ё. Футамуры к близким идеям пришел А.П. Ершов [14]. Свой метод специализации он назвал *смешанными вычислениями.* Кроме того, А.П. Ершов ввел плодотворное понятие *генерирующего расширения* программы. В конце 1976 г. А.П. Ершов проехал с лекциями о своих идеях по ряду институтов. Тогда Ершов и Турчин узнали, что они работают над общей темой, и, встретившись, обменялись идеями и результатами. История дальнейших работ А.П. Ершова и его учеников увлекательно описана в недавно вышедшей книге [15], посвященной его памяти.

В начале 1980-х гг. от Турчина и Ершова этими идеями «заразился» Нил Джоунс (Neil Jones). Вместе с молодыми учениками он «пошел на штурм» задачи превращения интерпретатора в компилятор. Они разработали еще один метод, названный *частичными вычислениями* (*partial evaluation*), и в 1985 г. были первыми, кто выполнил на компьютере проекции Футамуры, включая формулу порождения компилятора компиляторов путем двойного самоприменения специализатора.[5] После этого эта тематика стала популярной в мировой науке, и с тех пор опубликованы сотни работ по специализации программ. Группа Н. Джоунса подвела промежуточный итог своих работ замечательной книгой [5].

Список уже появившихся методов метавычислений на этом не исчерпывается, но мы на этом остановимся.

Еще многое предстоит сделать и в области специализации программ, и в целом в области метавычислений, и главное — выйти на практические приложения. После этого метавычисления получат второе дыхание.

Работа поддержана грантом РФФИ № 06-01-00574.

[5] Формула порождения компилятора компиляторов путем двойного самоприменения специализатора была впервые найдена В.Ф. Турчиным, но так сложилось исторически, что она также причисляется к *проекциям Футамуры*, поскольку является их естественным продолжением.

Литература

1. *Futamura Y.* Partial evaluation of computing process — an approach to a compiler-compiler // *Systems, Computers, Controls* 2, № 5, 1971. P. 45-50.
2. *Glück R., Klimov A.V.* Metasystem transition schemes in computer science and mathematics // *World Futures: the Journal of General Evolution* 45, 1995. P. 213–243. ftp://ftp.vub.ac.be/pub/projects/Principia_Cybernetica/WF-issue/Glueck&Klimov.rtf.
3. *Glück R., Klimov A.V.* Occam's razor in metacomputation: the notion of a perfect process tree // P. Cousot, M. Falaschi, G. File, G. Rauzy (eds.), *Static Analysis Symposium*. Lecture Notes in Computer Science 724, Springer Verlag, 1993. P. 112–123.
4. *Goertzel B., Klimov And.V., Klimov Ark.V.* Supercompiling Java Programs, white paper. Supercompilers, LLC, 2002. http://www.supercompilers.com.
5. *Jones N.D., Gomard C.K., Sestoft P.* Partial Evaluation and Automatic Program Generation. Prentice-Hall, 1993. http://www.dina.kvl.dk/~sestoft/pebook/pebook.html.
6. *Jones S.P.* (ред.) Haskell 98 Language and Libraries: The Revised Report, 2002. http://haskell.org/onlinereport/.
7. *Nemytykh A.P.* Refal-5 and the Supercompiler SCP4 home page. http://www.botik.ru/pub/local/scp/refal5/.
8. *Turchin V.F.* The algorithm of generalization in the supercompiler // Dines Bjørner, Andrei P. Ershov, Neil D. Jones (eds.), Partial Evaluation and Mixed Computation. North-Holland, 1988. P. 531–549.
9. *Turchin V.F.* The concept of a supercompiler // Transactions on Programming Languages and Systems 8, № 3, 1986. P. 292–325.
10. *Turchin V.F.* The language Refal, the theory of compilation and metasystem analysis. Technical Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
11. *Turchin V.F.* Program transformation with metasystem transitions // Journal of Functional Programming 3, № 3, 1993. P. 283–313.
12. *Turchin V.F.* The use of metasystem transition in theorem proving and program optimization // *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science 85, Springer-Verlag, 1980. P. 645–657.
13. *Абрамов С.М.* Метавычисления и их приложения. Москва: Наука. Физматлит, 1995. 128 с.
14. *Ершов А.П.* О сущности трансляции // Программирование, № 5, 1977. С. 21–39.
15. *Марчук А.Г.* (ред.) Андрей Петрович Ершов — ученый и человек. Новосибирск: Изд-во СО РАН, 2006. 504 с.
16. *Турчин В.Ф.* Феномен науки: Кибернетический подход к эволюции. Москва: Наука, 1993. 295 с.
17. *Турчин В.Ф.* Эквивалентные преобразования рекурсивных функций, описанных на языке Рефал // Труды симпозиума "Теория и методы построения систем программирования", Киев-Алушта, 1972. С. 31–42.

Анд.В. Климов, Введение в метавычисления и суперкомпиляцию. В сб.: *Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям*. М.: Изд-во КомКнига, 2008. С. 343-368.