Вычисление образа функции как инструмент метавычислений. Опыт использования

А.А. Емельянов, А.О. Боровилов

ФГБОУ ВО «Волжский государственный университет водного транспорта»

Аннотация. В данной работе представлены результаты вычислительных экспериментов с авторским алгоритмом вычисления образа функции. Данный алгоритм позволяет решать некоторые задачи метавычислений, и в данной статье рассматриваются подходы к решению задач инверсных вычислений, на примере решения алгебраических уравнений в целых положительных числах. Также в работе представлен опыт использования разных представлений чисел и их влияние на ход и производительность вычисления образа функции.

Ключевые слова: метавычисления, образ функции, инверсное вычисление

Function image computation as a metacomputation tool. Experience

A.A. Emelyanov, A.O. Borovilov

Volga state university of water transport

Abstract. In this paper, the results of computational experiments with the author's algorithm for computing the image of a function, are presented. This algorithm allows solving some metacomputation problems, and in this paper approaches to solving inverse computation problems are considered, for example, the solution of algebraic equations in positive integers. Also, the work presents the experience of using different representations of numbers and their effect on the sequence and performance of calculating the function image.

Keywords: metacomputation, function image, inverse computation

В данной работе под функцией понимается структурная единица компьютерной программы, которая имеет аргументы и возвращает некоторое значение. Другими словами, под функцией мы будем понимать те самые функции и/или методы, с которыми мы встречаемся в таких распространенных языках программирования как C, Java, Haskell, Javascript, PHP, C# и так далее. Образ функции – это множество значений, которые могут быть возвращены в качестве результата вызова данной функции над некоторыми аргументами. Так,

для любого элемента e из образа функции существует как минимум один вариант значений аргументов, вызов данной функции над которыми возвращает e в качестве результата вычислений. Легко убедиться, что даже если функция может вернуть только значение некоторого конкретного типа данных (это всегда справедливо для статически типизированных языков), то это не означает, что образ данной функции совпадает с множеством всех значений указанного типа. Продемонстрируем это на примере функции возведения в квадрат, которая определена на множестве целых беззнаковых чисел. Очевидно, что не существует такого аргумента, при котором данная функция вернет, например, значение 5. Образом этой функции будет множество, которое можно записать следующим образом: $\{x^2 \mid x \in N\}$, где N – множество натуральных чисел. Эта запись является хорошим примером описания бесконечного множества значений некоторого типа с помощью относительно короткого (а что еще важнее – конечного) выражения. В данном примере мы также видим, что указанное выражение отчасти состоит из реализации функции: x^2, и в случае с более сложной функцией данная запись может оказаться много сложнее. Кроме того, содержание реализации функции или части ее реализации в выражении множества значений может требовать вычисления данной функции в процессе работы с данным выражением, что может стать проблемой для эффективной работы с выражением. Эта особенность будет более предметно затронута далее. Пока важно, что данный пример хорошо демонстрирует суть понятия образа функции для языка программирования.

В данной работе мы рассматриваем разработанный нами способ вычисления образа функции для чистого ленивого функционального языка программирования. Очевидно, что образ функции в общем случае может быть бесконечным множеством, следовательно, для вычисления образа придется оперировать бесконечными множествами в памяти компьютера, для чего необходимо определить подходящий способ конечного представления множества значений. Далее под образом функции мы будем понимать также и его представление если то, о чем мы говорим, понятно из контекста. Перед тем как определить необходимые свойства представления образа функции следует упомянуть то, как появилась данная задача, и в чем заключается ее актуальность. Исследование, которое стало основой для данной работы предполагало решение следующей задачи. Пусть есть программа-трансформатор, которая в качестве исходных данных берет текст некоторой программы на определенном языке программирования, а в качестве результата возвращает текст программы на том же или на другом языке программирования. Такая программа-трансформатор может, например, решать задачи компиляции, трансляции или оптимизации исходной программы. При этом может возникнуть проблема, связанная с доказательством некоторых свойств программы-трансформатора, такие как качество компиляции, корректность трансляции или гарантированность оптимизации по тем или

иным критериям. В указанном выше исследовании рассматривалось, пожалуй, одно из самых базовых свойств программы-трансформатора – гарантию синтаксической и семантической корректности результирующей программы. Действительно, перед тем как говорить о том, что результирующая программа является более эффективной по некоторым критериям чем исходная, следует убедиться, что результат работы программы-трансформатора вообще является корректной программой. Под корректностью программы здесь можно понимать, например, отсутствие ошибок компиляции данной программы. Чтобы гарантировать корректность результата программы-трансформатора необходимо доказать корректность всех ее возможных результатов. В этом месте появляется задача вычисления множества всех возможных результатов вычисления алгоритма или образ функции, представление которого также необходимо строго определить.

В процессе работы над задачей вычисления образа функции было обнаружено следующее свойство: алгоритм вычисления образа функции в некоторых случаях можно применять для решения задачи реверсивного вычисления. Данное свойство будет подробнее рассмотрено далее. Задача реверсивного вычисления является эквивалентом обратной задачи, то есть для некоторой существующей функции ***f(x)*** задача реверсивного вычисления может быть сформулирована как задача вычисления множества таких ***x***, для которых справедливо: ***f(x) = y***, где ***y*** – известное значение. Задача реверсивного вычисления является некоторым обобщением для целого ряда задач, а из указанного выше свойства следует, что задача вычисления образа функции является обобщением следующего уровня, так как реверсивное вычисление есть лишь частное ее применение. Это дало основание полагать, что практические методы вычисления образа функции являются актуальной проблемой для целого множества теоретических и прикладных задач.

Задачи реверсивного вычисления уже рассматривались другими авторами, например, в [1-4, 15]. В частности, можно отметить алгоритм УРА [15], который позволяет решать такие задачи. Алгоритм УРА основан на так называемом вычислении табличной формы функции, где табличная форма есть список пар: представление множества входных значений функции и соответствующее ему представление множества выходных значений. Для алгоритма УРА оговаривается, что табличная форма может быть бесконечным списком. На практике проблема оперирования бесконечным списком может быть решена, например, с помощью использования ленивых вычислений, когда вся табличная форма не требуется для решения конкретной задачи. В силу того, что задача вычисления образа функции была сформулирована в общем виде, где результатом вычисления является конечное представление образа функции, бесконечный список не подходит в качестве решения. Особенность ленивых вычислений заключается в том, что бесконечные списки могут быть представлены в виде термов, которые содержат вызовы рекурсивных функций.

Такие представления являются промежуточными состояниями вычисления программы. В этом смысле для алгоритма вычисления образа функции они ничем не отличаются от исходной программы, которая: 1) содержит информацию о своем образе, 2) предполагает некоторую последовательность вычислений (возможно бесконечную). Для решения задачи вычисления образа функции было решено использовать такие конечные представления множества значений, которые не содержат термов, подлежащих вычислению, то есть термов, в которых нет вызова функции. Это свойство представления образа функции является ключевым в данной работе.

Цель данной работы – демонстрация базовых принципов вычисления представления образа функции удовлетворяющего указанному выше условию, подходов к такой формулировке ряда задач, при которой они могут быть решены через вычисление образа некоторой функции и проблем, которые возникают при решении некоторых задач указанным методом.

p	$::=$	$s_1 \dots s_n$	(a program)
s	$::=$	$d = c_1(d_1^1, d_2^1, \dots)$ "|" $c_2(d_1^2, d_2^2, \dots)$ "|" $\dots$	(a data type definition or
	$\mid$	f_{spec}	a function specification)
f_{spec}	$::=$	$f_{def}\ f_{impl}$	
f_{def}	$::=$	$f :: d_1 \to d_2 \to \dots \to d_m$	(a function definition)
f_{impl}	$::=$	$f(v_1, v_2 \dots) = f_{body}$	(a function implementation)
f_{body}	$::=$	t	(a term)
	$\mid$	**case** v_i **of** $z_1 \to t_1\ \dots\ z_u \to t_u$	(a case-expression)
z	$::=$	$c(v_1, v_2, \dots)$	(a case-pattern)
	$\mid$	$\_$	(the «otherwise» case-pattern)
t	$::=$	v	(a variable name or
	$\mid$	$f(t_1, t_2, \dots)$	a function call or
	$\mid$	$c(t_1, t_2, \dots)$	a constructor call)

where d is a data type identifier, v is a variable identifier

Рис. 1. Синтаксис рабочего языка

1. Рабочий язык

В данном разделе мы определим функциональный язык программирования для его дальнейшего использования в качестве рабочего языка. Программа на рабочем языке является последовательностью определения типов данных и спецификаций функций. Каждая спецификация состоит из двух частей: определения функции и её реализации (рис. 1). Данный язык является чистым ленивым функциональным языком программирования

первого порядка со статической типизацией. Синтаксис языка является подмножеством синтаксиса языка Haskell.

2. Предикат инверсного вычисления

Пусть нам дана некоторая функция ***f*** с типом ***A→B***, где ***A*** и ***B*** – некоторые типы данных. Пусть также определена функция эквивалентности ***eq :: B→B→Bool*** для значений типа ***B***. Рассмотрим следующую программу на рабочем языке:

```
data Bool = True | False
data MaybeA = Nothing | Just (A)

invcalc :: A→MaybeA
invcalc(a) = invcalc_aux(eq(f(a),b_value), a)

invcalc_aux :: Bool→A→MaybeA
invcalc_aux(p, a) = case p of
        True → Just(a)
        False → Nothing
```

Рис. 2.

Здесь ***b_value*** – это некоторое интересующее нас конкретное значение типа ***B***. Если в качестве аргумента для функции ***invcalc*** указать такое значение ***a :: A***, для которого ***f(a)=b_value***, то выражение ***eq(f(a),b_value)*** будет эквивалентно ***True***, а результат вызова ***invcalc(a)***, будет равен ***Just(a)***, в противном случае – ***Nothing***, в соответствии с определением функции ***invcalc_aux***.

Таким образом, нетрудно убедиться, что образ функции ***invcalc*** есть ничто иное, как результат инверсного вычисления функции ***f*** дополненный значением ***Nothing***. Предполагая, что у нас есть инструмент, позволяющий вычислять образ функции и исключать из образа отдельные элементы (такие как, например, ***Nothing***), то решение задачи обратного вычисления некоторой функции ***f*** сводится к реализации соответствующих функций ***invcalc*** и ***invcalc_aux***.

В представленном выше примере можно обнаружить, что решение задачи инверсного вычисления является лишь частным случаем применения некоторого общего шаблона. Действительно, рассмотрим вместо функции ***invcalc*** следующую абстрактную функцию с типом **A → MaybeA**:

g(a) = invcalc_aux(p_expression, a)

где ***p_expression*** – некоторое выражение типа ***Bool***, которое, например, зависит от аргумента ***a :: A*** абстрактной функции ***g***. Очевидно, что функция ***invcalc*** является частным случаем абстрактной функции ***g***. Функция ***g*** параметризована

выражением типа ***Bool***, которое может рассматриваться как предикат над аргументом функции ***g***. Следовательно, если ***A*** – это числовой тип, а в качестве предиката указано, например, неравенство ***a > 5***, то результатом вычисления образа будет соответствующее этому неравенству множество (строгий интервал). При этом никаких ограничений на данный предикат не накладывается, кроме, конечно, возможности выразить его в виде выражения на рабочем языке программирования. Одной из простейших форм таких предикатов являются уравнения, то есть уже знакомое по предыдущему примеру выражение ***eq(f(a), b_value)***. Здесь можно подчеркнуть, что «правая часть» уравнения не обязательно должна быть значением. Так, например, допустимым является и следующее выражение-предикат: ***eq(f(a), h(a))***, как, впрочем, и любые другие варианты и комбинации функций рабочего языка.

В данной работе мы рассмотрим алгебраические уравнения над целыми положительными числами, два варианта определения целых положительных чисел и их влияние на ход вычисления образа функции для решения рассматриваемых уравнений.

3. Пример с унарной системой счисления

Рассмотрим пример расчета образа функции, который также является решением алгебраического уравнения. Здесь для примера мы взяли простейший вид алгебраического уравнения – линейное уравнение. Результаты расчета образа функции для более сложных уравнений будут затронуты ниже. В данном разделе мы сосредоточимся на самом процессе вычисления образа функции.

```
data N = Z | S(N)
data Bool = T | F

isZ :: N→Bool
isZ(a) = case a of
         Z → T
         S(a1) → F
```

```
eq :: N→N→Bool
eq(a, b) = case a of
           Z → isZ(b)
           S(a1) → eqb(a1,b)

eqb :: N→N→Bool
eqb(a1, b) = case b of
             Z → F
             S(b1) → eq(a1, b1)
```

Рис. 3.

Унарную систему счисления в виде типов данных рабочего языка запишем в виде ***data N = Z | S(N)***. Этот тип данных позволяет выразить любое натуральное число включая нуль. Конструктор ***Z*** символизирует нулевое значение. Конструктор ***S*** (от англ. successor) – функцию следования: ***S(n) = n+1***. Таким образом число 1 запишется в виде ***S(Z)***, число 2 – ***S(S(Z))***, число 3 – ***S(S(S(Z)))*** и так далее.

Для того чтобы выразить необходимый нам предикат потребуется несколько дополнительных функций. Во-первых, нам потребуется функция проверки на равенство ***eq***. Рассмотрим участок программы на рисунке 3. Функция проверки на равенство двух чисел имеет тип ***N→N→Bool***, где ***N*** – тип проверяемых чисел, ***Bool*** – логический тип. В первую очередь мы определяем тип ***Bool***, который представлен двумя значениями-конструкторами: ***T*** (истина) и ***F*** (ложь). Затем мы реализуем функцию проверки числа на равенство нулю ***isZ***, которая потребуется нам далее. Ее реализация элементарна: если аргумент ***a*** построен конструктором ***Z***, то есть равен нулю, функция возвращает ***T*** (истина), в противном случае – ***F*** (ложь).

Рабочий язык не содержит продвинутой системы синтаксического сахара и имеет ряд ограничений по сравнению с другими функциональными языками. Так, например, одним из ограничений является возможность использовать case-выражение только один раз в реализации функции – в качестве верхнего уровня ее определения. Это заставляет нас разбивать функцию на несколько, но упрощает структуру алгоритма вычисления образа функции и, как следствие, упрощает анализ хода работы этого алгоритма.

В функции проверки на равенство двух чисел в унарной системе ***eq*** первый аргумент проверяется case-выражением. Если первый аргумент (первое проверяемое число) равно нулю (сопоставляется с конструктором ***Z***), то равенство двух чисел определяется равенством второго аргумента нулю. Следовательно, в сопоставлении первого аргумента конструктору ***Z*** вызывается функция ***isZ(b)***, которая проверяет равенство ***b = 0***. Если первый аргумент не равен нулю, следовательно, он построен конструктором ***S*** и имеет вид ***S(a1)***, что эквивалентно выражению ***a1+1***. В этом случае вызывается дополнительная функция ***eqb*** над ***a1*** и вторым аргументом ***b*** исходной функции ***eq***. Задача функции ***eqb*** – подвергнуть аргумент ***b*** анализу case-выражением. Если ***b*** построен конструктором ***Z***, то результат ложный, так как к этому моменту уже определено, что первый аргумент функции ***eq*** не равен нулю. Если ***b*** построен конструктором ***S*** и имеет вид ***S(b1)***, то мы приходим к необходимости проверки двух чисел ***a1+1*** и ***b1+1*** на равенство. Очевидно, что результат сравнения этих двух числен эквивалентен результату сравнения чисел ***a1*** и ***b1***. Основываясь на этом факте в сопоставлении аргумента ***b*** конструктору ***S*** указан вызов функции ***eq*** над числами ***a1*** и ***b1*** в качестве аргументов, то есть рекурсивный вызов.

Для выражения необходимого предиката нам также потребуется функция сложения двух чисел в унарной системе счисления. Функцию сложения мы реализуем следующим образом:

```
sum :: N→N→N
sum(a, b) = case a of
        Z → b
        S(a1) → S(sum(a1, b))
```

Рис. 4. Функция сложения чисел в унарной системе счисления

Здесь функция ***sum*** реализована с использованием следующих двух свойств операции сложения: свойства нулевого элемента ***0+b = b*** и свойства сочетания ***(1 + a1) + b = 1 + (a1 + b)***. Нетрудно убедиться, что такая рекурсивная форма реализации функции сложения корректна и завершиться за конечное время (не зациклится) для любых конечных аргументов.

Теперь у нас есть определения всех необходимых типов и реализации всех необходимых функций для того чтобы сформулировать предикат и основанную на нем функцию, образ которой содержит решение алгебраического уравнения. Начнем с функции ***f***, образ которой мы будем искать:

```
data M = U | J(N)

f :: N→M
f(x) = g( eq( sum(S(S(Z)),x),S(S(S(S(Z)))) ), x )

g :: Bool→N→M
g(p, x) = case p of
        F → U
        T → J(x)
```

Рис.5.

Здесь тип данных ***M*** – аналог типу ***MaybeA*** из примера предыдущего раздела, а конструктор ***U*** – представляет собой значение, которое может появиться в качестве элемента образа функции, и которое мы в дальнейшем будем исключать для получения искомого множества. Первый аргумент функции ***g*** рассматривается как результат вычисления выражения-предиката и имеет логический тип. Если предикат истинный, то возвращается проверяемое значение (второй аргумент функции ***g***) обернутое в конструктор ***J***, в противном случае возвращается значение ***U***. В качестве первого аргумента функции ***g*** записано выражение-предикат, которое в более привычной форме можно записать следующим образом: ***2 + x = 4***.

Представление образа функции будем искать в виде системы уравнений следующих типов:

- **W = W1 ⋃ W2**
- **W3 = { V1 }**

, где **W1, W2, W3** – переменные, символизирующие множества значений, **V1** – конкретное значение, **W** – искомый образ функции. При этом в процессе

вычисления образа функции в виде системы указанных уравнений будем также использовать системы с уравнением вида: **expr => W4**, где **expr** – терм который может содержать вызовы функций. Таким образом результирующее представление удовлетворяет сформулированному ранее условию – не содержит термов с вызовом функций в отличии от промежуточных представлений. Начальная задача поиска образа функции **f** может быть сформулирована в виде системы из одного уравнения: **f => W**. Тогда процесс вычисления образа функции можно определить, как цепочку эквивалентных преобразований исходной системы уравнений, которая завершается системой уравнений без термов с вызовом функций.

Рассмотрим процесс вычисления образа функции ***f***. Процесс вычисления представляет собой преобразование исходного терма: ***f(x)***, где ***x*** – свободная переменная соответствующего типа. Раскрыв функцию ***f*** получаем:

g(eq(sum(S(S(Z)),x),S(S(S(S(Z))))),x)

В силу ленивости рабочего языка раскрытие функции ***g*** требует вычисления первого аргумента, следовательно, необходимо раскрыть функцию ***eq***. Раскрытие функции ***eq*** требует раскрыть свой первый аргумент, то есть вызов функции ***sum***. Раскрытие функции ***sum*** и последующие раскрытия для функций ***eq***, ***eqb*** представлены соответственно ниже:

g(eq(S(sum(S(Z),x)),S(S(S(S(Z))))),x)

g(eqb(sum(S(Z),x),S(S(S(S(Z))))),x)

g(eq(sum(S(Z),x),S(S(S(Z)))),x)

Очевидно, что последние четыре редукции привели к схожему выражению, с той лишь разницей, что пропал конструктор ***S*** из первого аргумента функции ***sum*** и конструктор ***S*** из второго аргумента функции ***eq***. Аналогичным путем вычисление образа функции проходит к следующему терму и его последующей редукции:

g(eq(sum(Z,x),S(S(Z))),x)

g(eq(x,S(S(Z))),x)

Теперь для раскрытия функции ***eq*** необходимо раскрыть переменную ***x***. Это действие порождает две ветки с двумя разными термами для вариантов ***x = Z*** и ***x = S(x1)*** соответственно:

g(eq(Z,S(S(Z))),Z)

g(eq(S(x1),S(S(Z))),S(x1))

Образ функции ***f*** является объединением результатов дальнейших вычислений для каждого из этих двух термов. Подмножество значений для первого терма обозначим ***W1***, для второго – ***W2***. Образ функции ***f*** обозначим ***W = W1*** ∪ ***W2***.

Первый терм редуцируется до значения ***U***, следовательно, ***W1 = {U}***. Второй терм редуцируется следующим образом:

g(eqb(x1,S(S(Z))),S(x1))

g(eq(x1,S(Z)),S(x1))

И снова имеем ситуацию, где терм разбивается на два для разных вариантов значений ***x1***:

g(eq(x1,S(Z)),S(x1)) => W2=W3 ∪ W4

g(eq(Z,S(Z)),S(Z)) => W3 = {U}

g(eq(S(x2),S(Z)),S(S(x2))) => W4

Здесь справа от термов указаны множества, элементы которых входят в состав образа функции ***f***. Дальнейшие шаги вычисления образа выглядят так:

g(eqb(x2,S(Z)),S(S(x2))) => W4

g(eq(x2,Z),S(S(x2))) => W4=W5 ∪ W6

g(eq(Z,Z),S(S(Z))) => W5={S(S(Z))}

g(eq(S(x3),Z),S(S(S(x3)))) => W6

g(eqb(x3,Z),S(S(S(x3)))) => W6

g(F,S(S(S(x3)))) => W6={U}

В итоге образ функции ***f*** представляется следующим выражением:

W=W1 ∪ W3 ∪ W5 ∪ W6={U} ∪ {U} ∪ {S(S(Z))} ∪ {U}={U, S(S(Z))}

Исключая значение ***U*** и используя более привычный вид записи получаем:

W\{U} = {2}, что является решением уравнения ***2+x = 4***.

Нетрудно убедиться, что при использовании унарной системы счисления фактически идет прямой перебор натуральных чисел, так как при выборе вариантов представления аргумента ***x*** и вложенных аргументов его конструкторов фактически идет движение в сторону увеличения аргумента ***x***, а разделение представляет собой выбор между $x = x_n$ и $x > x_n$. Очевидно, что при таком подходе решение задач методом вычисления образа функции совершенно неэффективно. Увеличить эффективность можно используя другой вариант представления чисел.

4. Пример с бинарной системой счисления

Рассмотрим другой вариант представления целого положительного числа (рис. 6). Здесь тип данных ***B*** – это бит данных принимающий два возможных значения: ***X*** - символизирующий нуль и ***I*** - символ единицы. Тип данных ***L*** представляет собой список бит, то есть разрядную сетку, где ***E*** – конструктор

символизирующий пустой список, ***C*** – конструктор связки головы и хвоста списка. Таким образом можно построить, например, беззнаковое 8-битное число:

C(B0,C(B1,C(B2,C(B3,C(B4,C(B5,C(B6,C(B7,E))))))))

где B_i - соответствующий бит представления этого числа со следованием младших разрядов к старшим слева-направо.

Рассмотрим программу, реализующую функцию ***f*** по аналогии предыдущего примера в терминах чисел, представленных типом данных ***L*** и с алгебраическим уравнением ***1 + x = 4*** (рис. 6 и рис. 7.):

```
data L = E | C (B, L)
data B = X | I


data M = U | J(L)
data Bool = T | F

L→M
f(x) = g(isEq( sum( C(I,C(X,C(X,E))), x ),
       C(X,C(X,C(I,E))), ), x)

Bool→L→M
g(p, x) = case p of
    T → J(x)
    F → U

L→L→Bool
isEq(a, b) = case a of
    E → isE(b)
    C(ae, as) → isEqb(ae, as, b)
```

```
L→Bool
isE(a) = case a of
    E → T
    C(ae,as) → F

B→L→L→Bool
isEqb(ae, as, b) = case b of
    E → F
    C(be, bs) → isEqe(ae, as, be, bs)

B→L→B→L→Bool
isEqe(ae, as, be, bs) = case ae of
    X → isEqex(as, be, bs)
    I → isEqei(as, be, bs)

L→B→L→Bool
isEqex(as, be, bs) = case be of
    X → isEq(as, bs)
    I → F

L→B→L→Bool
isEqei(as, be, bs) = case be of
    X → F
    I → isEq(as, bs)
```

Рис. 6.

Рассмотрим несколько первых шагов вычисления образа функции ***f***:

f(x) => W

g(isEq(sum(C(I,C(X,C(X,E))),x),C(X,C(X,C(I,E)))),x) => W

g(isEq(sume(X,C(I,C(X,C(X,E))),x),C(X,C(X,C(I,E)))),x) => W

g(isEq(sumb(X,I,C(X,C(X,E)),x),C(X,C(X,C(I,E)))),x) => W

g(isEq(sumb(X,I,C(X,C(X,E)),C(xe,xs)),C(X,C(X,C(I,E)))),C(xe,xs)) => W

В последнем из представленных выше редукций переменная ***x*** представлена единственным вариантом – конструктором ***C***, что полностью

соответствует реализации функции ***sumb***. Такая реализация связана с неявным условием равенства количества разрядов у обоих аргументов функции ***sum***. Следующие шаги вычисления образа:

g(isEq(sums(X,I,C(X,C(X,E)),xe,xs),C(X,C(X,C(I,E)))),C(xe,xs)) => W

g(isEq(sumsx(I,C(X,C(X,E)),xe,xs),C(X,C(X,C(I,E)))),C(xe,xs)) => W

g(isEq(sumsxi(C(X,C(X,E)),xe,xs),C(X,C(X,C(I,E)))),C(xe,xs)) => W

```
L→L→L
sum(a, b) = sume(X, a, b)

B→L→L→L
sume(e, a, b) = case a of
    E → bisE(b)
    C(ae, as) → sumb(e, ae, as, b)

L→L
bisE(b) = case b of
    E → E

B→B→L→L→L
sumb(e, ae, as, b) = case b of
    C(be, bs) → sums(e, ae, as, be, bs)

B→B→L→B→L→L
sums(e, ae, as, be, bs) = case e of
    X → sumsx(ae, as, be, bs)
    I → sumsi(ae, as, be, bs)
```

```
B→L→B→L→L
sumsx(ae, as, be, bs) = case ae of
    X → sumsxx(as, be, bs)
    I → sumsxi(as, be, bs)

B→L→B→L→L
sumsi(ae, as, be, bs) = case ae of
    X → sumsxi(as, be, bs)
    I → sumsii(as, be, bs)

L→B→L→L
sumsxx(as, be, bs) = case be of
    X → C(X, sume(X, as, bs))
    I → C(I, sume(X, as, bs))

L→B→L→L
sumsxi(as, be, bs) = case be of
    X → C(I, sume(X, as, bs))
    I → C(X, sume(I, as, bs))

L→B→L→L
sumsii(as, be, bs) = case be of
    X → C(X, sume(I, as, bs))
    I → C(I, sume(I, as, bs))
```

Рис. 7. Реализация функции сложения

Здесь ход вычисления расходится на две ветки ($W = W1 \cup W2$) рассмотрим одну из них:

g(isEq(sumsxi(C(X,C(X,E)),X,xs),C(X,C(X,C(I,E)))),C(X,xs)) => W1

g(isEq(C(I,sume(X,C(X,C(X,E)),xs)),C(X,C(X,C(I,E)))),C(X,xs)) => W1

g(isEqb(I,sume(X,C(X,C(X,E)),xs),C(X,C(X,C(I,E)))),C(X,xs)) => W1

g(isEqe(I,sume(X,C(X,C(X,E)),xs),X,C(X,C(I,E))),C(X,xs)) => W1

g(isEqei(sume(X,C(X,C(X,E)),xs),X,C(X,C(I,E))),C(X,xs)) => W1

g(F,C(X,xs)) => W1 = {U}

Суть хода вычисления ветки для ***W1*** заключается в следующем. Алгоритм рассматривает случай, когда младший разряд аргумента ***x*** равен нулю. Другими словами, рассматривается случай четного ***x***. При этом за несколько шагов выясняется, что сумма четного ***x*** и нечетной единицы должна дать нечетное число, что противоречит значению младшего разряда числа 4. Таким образом алгоритм вычисления образа функции отбросил сразу половину кандидатов в элементы результирующего образа функции ***f***. Вычисления ветки ***W2*** повторяют предыдущий ход вычисления и не представляют интереса в рамках данной статьи.

5. Другие алгебраические уравнения

Эксперименты с другими алгебраическими уравнениями показали, что при рассмотрении уравнений вида $x + a = b$, в случае применения унарной системы счисления требуется количество шагов пропорциональное $min(a,b)$, а в случае с бинарной системой счисления количество шагов пропорционально размеру используемой разрядной сетки. Так для 32-битных разрядных чисел решение линейного уравнения требовало около 1200 шагов. При этом было установлено, что для унарной системы счисления решения некоторых уравнений с числами представимыми в виде 32-битных разрядных сеток могут потребовать более 40 миллиардов шагов!

Кроме линейных уравнений были также проведены эксперименты с алгебраическими уравнениями второй, третей и более степеней, также для целых положительных чисел. Алгоритм успешно находит решения этих уравнений через вычисление образа функции. Число ходов необходимое для решения этих видов уравнения пропорционально $p*log(n)$, где p – степень уравнения, а n – размер разрядной сетки представляемых чисел.

6. Связанные работы

Данная работа связана с рядом других исследований, область которых носит название метавычисления. Область метавычислений охватывает целый ряд направлений исследований, в том числе: инверсия программ [1-4], частичные вычисление [5-7], композиция функций [8-9] и другие. Проблемы, которые возникают в сфере метавычислений актуальны уже несколько десятилетий и не снижают своей актуальности. Разработан ряд методов, подходов и инструментов для их решения. Одним таким инструментом является техника суперкомпиляции [10-13] – техника, применяемая для решения целого ряда задач метавычислений. Вычисление образа функции – подход, который рассматривался в данной работе, является еще одним инструментом в арсенале для решения задач метавычислений.

Наиболее близкие к данному материалу статьи представлены в [4,10,15], где авторы исследуют метод построения перфектного дерева, а также его свертки. Одна из реализаций метода с перфектным деревом в его основе –

алгоритм «УРА» [15, 2], который может быть использован в том числе и для решения задач инверсного вычисления. Но в отличии от него в подходе с вычислением образа функции нам не требуется строить таблицу входных и выходных значений. Это в ряде случаев позволяет нам с меньшими затратами получать конечный результат без необходимости учитывать ссылки на входные значения.

Отдельно стоит упомянуть следующие работы по инверсному вычислению и инверсии программ [1-4, 16, 17]. Рассматриваемые там техники используют разные подходы для разных языков программирования. Так, например, для эффективного решения задач инверсии программ, предлагается использовать специальные языки, которые организованы таким образом, что позволяют реализовывать только инъективные функции. Очевидно, что это уже не Тьюринг-полные языки, а, следовательно, не смотря на эффективность предлагаемых решений, такие подходы оставляют большой простор для дальнейших исследований. Рассматриваемый в данной работе метод вычисления образа функций не накладывает подобных ограничений на используемый язык, а рассматриваемый здесь рабочий язык без ущерба может быть расширен необходимым синтаксическим сахаром, чтобы удовлетворять требованиям к современных чистым функциональным языкам.

Заключение

Авторами предложен алгоритм вычисления образа функции в виде системы уравнений специального вида. Вычисление образа функции может быть осуществлен и другими известными алгоритмами, как, например, алгоритм УРА. Особенностью предложенного алгоритма, в частности по сравнению с алгоритмом УРА, является тот факт, что результирующее представление не содержит ни явных вызовов функций ни неявных, как в случае с применением техники ленивых вычислений. Предложенный алгоритм может завершится за конечное время и вернуть конечное представление искомого множества – образ функции.

Несмотря на то, что в данной работе были продемонстрированы лишь примеры решения алгебраических уравнений, из определение образа функции следует, что методы вычисления образа могут решать целый класс вычислительных задач. Пример с бинарной системой счисления показывает возможность эффективного решения задачи вычисления образа функции для упрощенного ленивого чистого функционального языка программирования.

Кроме указанных выше особенностей, в отличии от существующих методов решения задач инверсного вычисления, метод решения построенный на вычислении образа функции обладает как еще одним преимуществом – для решения задачи вычисления образа не требуется накладывать на функцию условие инъективности.

Литература

1. Glück R., Kawabe M. A program inverter for a functional language with equality and constructors // Asian Symposium on Programming Languages and Systems. – Springer, Berlin, Heidelberg, 2003. – С. 246-264.
2. Abramov S., Glück R. Principles of inverse computation and the universal resolving algorithm // The essence of computation. – Springer, Berlin, Heidelberg, 2002. – С. 269-295. .
3. Mu S. C., Bird R. Inverting functions as folds // International Conference on Mathematics of Program Construction. – Springer, Berlin, Heidelberg, 2002. – С. 209-232.
4. Romanenko A. Inversion and metacomputation // ACM SIGPLAN Notices. – ACM, 1991. – Т. 26. – №. 9. – С. 12-22.
5. Glück R., Kawada Y., Hashimoto T. Transforming interpreters into inverse interpreters by partial evaluation // ACM SIGPLAN Notices. – ACM, 2003. – Т. 38. – №. 10. – С. 10-19.
6. Bondorf A., Jørgensen J. Efficient analyses for realistic off-line partial evaluation //Journal of functional Programming. – 1993. – Т. 3. – №. 3. – С. 315-346.
7. Hatcliff J. An introduction to online and offline partial evaluation using a simple flowchart language //Partial Evaluation. – Springer, Berlin, Heidelberg, 1999. – С. 20-82.
8. Glück R., Klimov A. Metacomputation as a tool for formal linguistic modeling //CYBERNETICS AND SYSTEMS'94. – 1994.
9. Horwitz S., Reps T., Binkley D. Interprocedural slicing using dependence graphs // ACM Transactions on Programming Languages and Systems (TOPLAS). – 1990. – Т. 12. – №. 1. – С. 26-60.
10. Turchin V. F. A supercompiler system based on the language Refal // ACM SIGPLAN Notices. – 1979. – Т. 14. – №. 2. – С. 46-54.
11. Turchin V. F., Nirenberg R. M., Turchin D. V. Experiments with a supercompiler // Proceedings of the 1982 ACM symposium on LISP and functional programming. – ACM, 1982. – С. 47-55.
12. Turchin V. F. Program transformation by supercompilation // Programs as Data Objects. – Springer, Berlin, Heidelberg, 1986. – С. 257-281.
13. Glück R., Turchin V. F. Experiments with a Self-applicable Supercompiler // City University New York. Technical Report. – 1989.
14. Bancerek G. The fundamental properties of natural numbers //Formalized Mathematics. – 1990. – Т. 1. – №. 1. – С. 41-46.
15. Абрамов С. М., Пармёнова Л. В. Метавычисления и их применение. – 1995.
16. Korf R. E. Inversion of Applicative Programs // IJCAI. – 1981. – С. 1007-1009.
17. Eppstein D. A Heuristic Approach to Program Inversion //IJCAI. – 1985. – Т. 85. – С. 219-221.

References

1. Glück R., Kawabe M. A program inverter for a functional language with equality and constructors // Asian Symposium on Programming Languages and Systems. – Springer, Berlin, Heidelberg, 2003. – P. 246-264.
2. Abramov S., Glück R. Principles of inverse computation and the universal resolving algorithm // The essence of computation. – Springer, Berlin, Heidelberg, 2002. – P. 269-295. .
3. Mu S. C., Bird R. Inverting functions as folds // International Conference on Mathematics of Program Construction. – Springer, Berlin, Heidelberg, 2002. – P. 209-232.
4. Romanenko A. Inversion and metacomputation // ACM SIGPLAN Notices. – ACM, 1991. – T. 26. – №. 9. – P. 12-22.
5. Glück R., Kawada Y., Hashimoto T. Transforming interpreters into inverse interpreters by partial evaluation // ACM SIGPLAN Notices. – ACM, 2003. – T. 38. – №. 10. – P. 10-19.
6. Bondorf A., Jørgensen J. Efficient analyses for realistic off-line partial evaluation // Journal of functional Programming. – 1993. – T. 3. – №. 3. – P. 315-346.
7. Hatcliff J. An introduction to online and offline partial evaluation using a simple flowchart language //Partial Evaluation. – Springer, Berlin, Heidelberg, 1999. – P. 20-82.
8. Glück R., Klimov A. Metacomputation as a tool for formal linguistic modeling // CYBERNETICS AND SYSTEMS'94. – 1994.
9. Horwitz S., Reps T., Binkley D. Interprocedural slicing using dependence graphs // ACM Transactions on Programming Languages and Systems (TOPLAS). – 1990. – T. 12. – №. 1. – P. 26-60.
10. Turchin V. F. A supercompiler system based on the language Refal // ACM SIGPLAN Notices. – 1979. – T. 14. – №. 2. – P. 46-54.
11. Turchin V. F., Nirenberg R. M., Turchin D. V. Experiments with a supercompiler //Proceedings of the 1982 ACM symposium on LISP and functional programming. – ACM, 1982. – P. 47-55.
12. Turchin V. F. Program transformation by supercompilation // Programs as Data Objects. – Springer, Berlin, Heidelberg, 1986. – P. 257-281.
13. Glück R., Turchin V. F. Experiments with a Self-applicable Supercompiler // City University New York. Technical Report. – 1989.
14. Bancerek G. The fundamental properties of natural numbers // Formalized Mathematics. – 1990. – T. 1. – №. 1. – P. 41-46.
15. Abramov S. M., Parmenova L. V. Metavychisleniia i ikh primenenie. – 1995.
16. Korf R. E. Inversion of Applicative Programs // IJCAI. – 1981. – P. 1007-1009.
17. Eppstein D. A Heuristic Approach to Program Inversion //IJCAI. – 1985. – T. 85. – P. 219-221.