Непейвода Николай Николаевич
Доктор физико-математических наук, профессор Удмуртского государственного университета. Выпускник мехмата МГУ. Автор более 150 печатных трудов. Один из основателей теории неформализуемых понятий и теории логического синтеза программ на базе конструктивных логик. 15 лет ведёт интенсивные эксперименты по перестройке преподавания информатики, рассматривая это как комплексную проблему.

Комбинаторная логика

Мы продолжаем цикл статей, посвящённых функциональному программированию. Комбинаторная логика, о которой идёт речь, представляет собой правила преобразования и соединения загадочных объектов, называемых *комбинаторами*. Порой кажется, что комбинаторы являются некоторыми мистическими объектами, которые таят в себе ответы на множество вопросов и дают ключ к решению многих задач, но понять заключённую в них мистическую силу дано не каждому. В эту игру любят играть информатики-теоретики и профессиональные математики, они получают через это великое понимание... Так оно и есть.

Уже не раз приходилось слышать вопрос: «Отчего в журнале «Потенциал» печатается так много статей, посвящённых не традиционному (императивному) программированию, а функциональному?» Что ж, ответ на этот вопрос есть. Современная информатика переживает сегодня новый подъём, связанный с осмыслением и применением на деле множества открытий, сделанных в прошлом веке, но прошедших мимо информатиков-практиков. Осмысление это, в первую очередь, заключается в выделении нескольких парадигм программирования, которые непосредственно соответствуют различным логикам или, если хотите, стилям мышления. И сегодня уже ясно, что новые задачи, которые ставит перед нами жизнь, могут решаться лишь межпарадигменными специалистами, то есть информатиками-математиками, способными посмотреть сверху как на популярное в промышленности объектно-ориентированное программирование, так и на структурное, функциональное, логическое и другие «программирования».

Установка связи между типами в функциональном языке, комбинаторами и аксиомами логики является одним из таких важных открытий, которое поможет Вам при решении практических задач.

Ворожцов Артём

О фундаментальных понятиях

Некоторые понятия математики естественно называть *фундаментальными*, поскольку на них основывается всё здание математики. Более того, Г. Кантор[1] ещё в 1870 г. заметил, что, судя по всему, основные математические понятия выражаются через единственное понятие — множества и единственное отношение — принадлежность элемента множеству $a \in X$. Уже в 1881 г. немецкий математик Г. Фреге опубликовал свой труд, где практически проделал построение основных математических понятий в рамках теории множеств. В работе Фреге были найдены неустранимые недостатки, но в 1910-1914 гг. Г. Уайтхед и Б. Рассел написали трёхтомник «Principia Mathematica», где проделали практически исчерпывающее на тот момент и поныне признаваемое точным построение практически всех известных на тот момент математических теорий, исходя из теории множеств.

Например, натуральные числа можно выразить как множества следующим образом. **0** — это пустое множество $\varnothing$, **1** — $\{0\}$ или, более скрупулёзно, $\{\varnothing\}$ (множество, состоящее из одного элемента — пустого множества), **2** — $\{0,1\}$ (двухэлементное множество, элементы которого есть множества, соответствующие числам 0 и 1), и так далее.

[1] Г. Кантор — один из создателей современной формы математики. Он происходил из крещёной еврейской семьи, родился в Петербурге, а учился и работал в Германии, тем самым уже по происхождению соединив три наиболее различные и наиболее плодотворные формы теоретического мышления: русскую, немецкую и еврейскую.

Поскольку в математике (в отличие от программирования) проблемы совместимости версий практически нет, корректные математические книги не стареют, и построение Уайтхеда и Рассела остаётся действительным до сих пор[1].

Заметим, что даже практически реализуемая фундаментальность понятия означает лишь теоретическую возможность. При теоретических построениях проблема ресурсов (за исключением самых важных на самом деле, но отнюдь не видимых в продуктах, потребляемых потребителем: интеллектуальных сил, ясности и перестраиваемости) полностью игнорируется, и поэтому теоретически реализованное и теоретически полностью приемлемое решение может быть на практике безнадёжно неэффективным. Именно так происходит с построением математики на основе теории множеств.

Помимо множеств, за двадцатый век выделены ещё три фундаментальных математических понятия, каждое из которых позволяет полностью развить математику: именование и отношение «быть именем», система и отношение «быть частью» и *функция* вместе с отношением вычисления (одно выражение преобразуется в другое).

Функции как фундаментальный объект рассмотрели в первой четверти XX века два гениальных математика: Янош фон Нейман (Венгрия, затем США), судьба которого сложилась настолько счастливо, что ему приписывают создание принципов работы современных компьютеров, и М. И. Шейнфинкель (Россия) с диаметрально противоположной судьбой. Созданные ими системы были абсолютно различны: фон Нейман перенёс на функции конструкции теории множеств, а Шейнфинкель пошёл своим путём. Столь же различна и их судьба: работа Неймана получила немедленное признание, а затем оказалась абсолютно бесполезной и ныне упоминается лишь в книгах по истории логики; работа Шейнфинкеля первоначально была проигнорирована, но когда сам автор уже был неработоспособен, её нашли и развили два американских математика (Хаскелл Карри и Алонзо Чёрч); несмотря на продолжающееся непонимание и игнорирование, через сорок лет она нашла приложение, а ныне она является основой функционального программирования[2].

---

[1] На форумах автор замечал грубую ошибку, в которую вовлекает людей привычка к изданиям типа «Программирование в .Net версии 2», которые через два года приходится выбрасывать за устарелостью: что хорошего может быть в книге, изданной несколько десятков лет назад? Если книга была хорошей по-настоящему (чего даже в принципе невозможно добиться от только что упомянутых изданий), то она останется хорошей и через несколько тысяч лет. Если книга была просто качественной и очень хорошей, то она не устареет несколько десятков лет, даже если написана в области, где понятия ещё не устоялись (см., например, книгу Гриса «Наука программирования», вышедшую в 1981 г.). А ваши предки были отнюдь не глупее вас, и знали много такого, что вы забыли.

[2] В те времена американцы ещё не считали себя самыми умными в мире, и поэтому ещё не потеряли научную этику и корректно ссылались на основоположников.

Преобразование Карри-Шейнфинкеля

Когда М.И. Шейнфинкель и Х.Б. Карри начали разработку их систематической теории функций, применяемых к функциям, они натолкнулись на важное преобразование, позволяющее уменьшить количество сущностей.

Если функция может выдавать в качестве значения функцию и сама применяться к функциям, то функции многих аргументов не нужны (хотя бы в принципе, но в данном случае модальность «*в принципе*» неточна: показанное ниже преобразование весьма практично, хотя и непривычно). В самом деле, преобразуем вызов **f(x,y,z)** следующим образом. Применим **f** к **x**. Получится *остаточная функция* $\mathbf{f_1(y,z)}$ уже от двух аргументов, которая, будучи применена к **y**, даёт опять остаточную функцию $\mathbf{f_2(z)}$, которая наконец-то выдаст результат. Таким образом, можно было бы писать **f(x)(y)(z)**.

В теории данное преобразование называется *преобразованием Карри-Шейнфинкеля* (в англоязычной и переводной литературе преобразованием Карри (*currying*)), а в практике программирования — *частичной параметризацией*.

Видно, что выражение **f(x)(y)(z)** несколько неуклюже. Чтобы писать покрасивее, Шейнфинкель предложил соглашение, сохранившееся в комбинаторной логике и её последователях до сегодняшнего дня и перешедшее во многие языки программирования действительно высокого уровня, например, **Lisp**, **Рефал**, **Haskell**. Применение функции к аргументу записывается **(f x)**, а выражение типа **(((f x)y)z)** как **(f x y z)**. Таким образом, если скобки в функциональном выражении не расставлены, то они группируются влево, а первый член списка считается функцией, применяемой к остальным членам этого списка. Заметим, что **(f g x)** и **(f (g x))** несут совершенно разный смысл: первое выражение является применением функции к двум аргументам, а второе — композицией двух одноместных функций (композиция в школьных обозначениях соответствует **f(g(x))**). Шейнфинкель не остановился даже перед внешне совершенно парадоксальным в те времена применением функции к самой себе: выражение **(f f)** является совершенно законным.

Таким образом, в теории можно без ограничения общности рассматривать лишь одноместные функции.

Операции, применяемые к функциям, осознанно использовались в математике и раньше. Они назывались *операторами* (если результат — функция) либо *функционалами* (если результат — объект). Сейчас они чаще всего называются функционалами (высших типов).

В данной статье все рассматриваемые нами объекты (если явно не оговорено противное) являются заодно и функциями, а функции — заодно и объектами.

Краеугольные камни

Шейнфинкель выделил три краеугольных камня, *комбинаторы*, как он назвал, определяемые следующим образом:

(I x) = x
(K x y) = x
(S x y z) = ((x z)(y z))

Функция может применяться, когда она стоит первым членом в списке и когда число членов в списке достаточно для её применения. Таким образом, если она не первый член или аргументов недостаточно, то она не применяется. И нужно всегда помнить, что **(x y z u)** то же самое, что **(((x y) z) u)** или **((x y z) u)**, поэтому, когда элементов больше, чем нужно, остаток списка просто игнорируется.

Рассмотрим пример.

$$((S\ I\ I)\ x) = (S\ I\ I\ x) \Rightarrow ((I\ x)\ (I\ x)) \Rightarrow$$
$$\Rightarrow (x\ (I\ x)) \Rightarrow (x\ x)$$

Знаком ⇒ обозначается преобразование (применение функции). Таким образом, список комбинаторов **(S I I)** является одноместной функцией, «удваивающей» свой аргумент (точнее, применяющей его к самому себе). Поэтому её можно обозначить как новый комбинатор **D** и в дальнейшем использовать как целое.

Заметим, что из преобразования Карри-Шейнфинкеля следует, что «лишние» данные просто передаются дальше до тех пор, пока они не понадобятся. Рассмотрим пример.

(K K x y z) ⇒ ((K K x) y z)⇒(K y z) ⇒ y

Таким образом, комбинация **(K K)** из трёх аргументов оставляет лишь второй. Комбинацию комбинаторов, производящую некое осмысленное и законченное действие, естественно также называть комбинатором и, единожды определив, использовать в дальнейшем. Поэтому определим комбинатор $\mathbf{K_2^3}$ **= (K K)** с действием

$$(\mathbf{K_2^3}\ x\ y\ z) \Rightarrow y.$$

А следующий комбинатор вечно преобразуется сам в себя и служит выражением зацикливающейся программы:

$$\Omega = (D\ D) = S\ I\ I(S\ I\ I) \Rightarrow$$
$$\Rightarrow (S\ I\ I\ (I\ (S\ I\ I)) \Rightarrow$$
$$((I\ (S\ I\ I)(I(S\ I\ I))) \qquad (S\ I\ I\ (S\ I\ I))=$$
$$\Rightarrow (I\ (S\ I\ I)(S\ I\ I)) \Rightarrow$$
$$(D\ D) = \Omega.$$

В данном случае мы видим ещё одну особенность преобразований в комбинаторной логике: *если вычисления могут быть произведены в нескольких местах выражения, то их можно производить в каком угодно порядке.* Естественно, что предоставленной свободой надо пользоваться с умом. Например, рассмотрим комбинатор **(K (I I) Ω)**. Если мы начнём с **K**, то быстро его вычислим; если с **I**, то упростим выражение, но опасность останется; а если с **Ω** и будем в этом упорствовать, то зациклимся.

Рассмотрим ещё один пример.

$$(S\,S\,x\,y\,z) \Rightarrow ((S\,S\,x\,y)\,z) \Rightarrow ((S\,y)\,(x\,y)\,z) \Rightarrow$$
$$\Rightarrow (S\,y\,(x\,y)\,z) \Rightarrow ((y\,z)\,((x\,y)\,z)) \Rightarrow (y\,z\,(x\,y\,z)).$$

Таким образом, комбинатор (**SS**) производит хитроумные перестановки трёх аргументов. А теперь удлиним его ещё на одно **S**:

$$(S\,S\,S\,x\,y\,z) \Rightarrow (S\,x\,(S\,x)\,y\,z) = (x\,y\,(S\,x\,y)\,z).$$

Здесь не получилось законченного преобразования — выражение **(S x y)** «ждёт ещё одного аргумента». Элемент **z**, стоящий после него, не является его аргументом, так как скобки по умолчанию расставляются так:

$$(((x\,y)\,((S\,x)\,y))\,z).$$

Поэтому данную последовательность символов **(S S S)** комбинатором называть не стоит.

Заметим, что могут быть комбинаторы, требующие как угодно много аргументов. Например, рассмотрим следующее выражение

$$(K\,(K\,(KK))\,xyzuv) \Rightarrow (K\,(KK)\,yzuv) \Rightarrow$$
$$\Rightarrow (\,KKzuv) \Rightarrow (Kuv) \Rightarrow u.$$

Таким образом, данный комбинатор действует так:

$$(K\,(K\,(K\,K)\,)\;x\,y\,z\,u\,v) \Rightarrow u.$$

Комбинатор **(K (K (K K)))** можно обозначить как $\mathbf{K}_4^5$. Заметим, что, следуя сложившейся у нас системе обозначений, можно исходные комбинаторы переназвать:

$$K = K_1^2,\; I = K_1^1.$$

В дальнейшем Чёрч выяснил, что **I** можно выразить через **S** и **K**:

$$(S\,K\,K\,x) \Rightarrow\; (K\,x\,(K\,x)) \;\Rightarrow\; x.$$

А теперь несколько задач на определение, комбинатор ли некоторое выражение и что оно делает.

1. **(S (K K))**
2. **(K K K)**
3. **(K (K K))**
4. **(S (S S))**
5. **(S (S K))**
6. **(S I S)**

Построение комбинаторов

Построим важный комбинатор

$$(\mathbf{C\ x\ y\ z}) = (\mathbf{x\ (y\ z)})$$

Этот пример заодно иллюстрирует общераспространённый метод решения творческих задач: начинайте с конца!

Выражение, которое должно получиться в конце, напоминает результат применения комбинатора **S**, но на втором месте нет **z**. Чтобы удалить член списка, у нас есть комбинатор **K**, и получается промежуточное выражение

$$(\mathbf{S\ (K\ x)\ y\ z})$$

C = (S (K x)) ещё не является решением, поскольку внутрь выражения попала переменная **x**, но, опять применяя аналогию с **S**, получаем

$$\mathbf{C = (S\ (K\ S)\ K)},$$

что и является решением. Проверьте!

Рассмотрим ещё один пример, иллюстрирующий важнейший способ рекурсивного построения решения. Построим комбинатор

$$\mathbf{K_n^n (x_1 \ldots x_n) \Rightarrow x_n}.$$

Очевидно, что для **n=1** можно взять $\mathbf{K_1^1 = I}$.

Теперь построим ещё два примера.

$$\mathbf{K_2^2 = (K\ I) = (K\ K_1^1)};$$

$$\mathbf{K_3^3 = (K\ (K\ I)) = (K\ K_2^2)}.$$

И теперь приходим к общему решению:

$$\mathbf{K_n^n = (K\ K_{n-1}^{n-1})};$$

Таким образом, в рекурсивном построении обязательно должен быть начальный случай, когда мы прямо строим решение, а все другие случаи постепенно сводятся к этому начальному: следующий — через предыдущий.

Имеется теорема Карри-Шейнфинкеля, говорящая, в частности, о том, что для любого выражения $\mathbf{E[x_1, \ldots, x_n]}$, построенного из переменных $\mathbf{x_1, \ldots, x_n}$, имеется комбинатор **E**, такой, что

$$\mathbf{(E\ x_1 \ldots x_n) \Rightarrow E\ [x_1, \ldots, x_n]}.$$

Это означает, что любое осмысленное преобразование функций может быть записано с помощью комбинаторов Шейнфинкеля, и его система по праву получила имя комбинаторной логики. Но это не означает, что любое преобразование, которое взбрело Вам в голову, можно записать таким образом. Например, нет комбинатора со следующим действием:

$$\mathbf{(U\ (x\ y))} \Rightarrow \mathbf{(y\ x)}.$$

Здесь выражение **(x y)** есть результат применения функции **x** к функции **y**. После применения функции уже невозможно восстановить, какая же именно функция была применена для получения результата. Никакой комбинатор **U** не может восстановить историю получения переданного ему аргумента и тем более что-то в этой истории поменять.

Система комбинаторной логики обладает ещё одним исключительно редким и красивым свойством, теряющимся практически при любом её расширении. При преобразованиях комбинаторов невозможно сбиться с пути. Чисто математически это означает следующее.

Теорема Чёрча-Россера. Пусть **t, r** и **s** — комбинаторы. Если $\mathbf{t \Rightarrow r}$, $\mathbf{t \Rightarrow s}$,

то найдётся такой комбинатор **q**, что $\mathbf{s} \Rightarrow \mathbf{q}$, $\mathbf{r} \Rightarrow \mathbf{q}$.

Занимаясь преобразованием комбинатора **t** и выбирая различные пути его преобразования, можно быть уверенным, что в конечном итоге найдётся комбинатор **q**, который можно получить, независимо от сделанного выбора пути.

Внимание! Теперь настоятельно рекомендуется решить ряд задач на построение комбинаторов. Вы можете взмолиться: а где же метод? Думайте! Вам полезно разок побывать в ситуации, где общий метод практически не работает, и приходится напрягать естественный интеллект. Далее используйте уже построенные комбинаторы при построении новых.

Используя базовые комбинаторы **S**, **K** и **I**, построить комбинаторы, работающие так, как указано (сразу отметим, что эти задачи имеют бесконечно много правильных решений):

а) $(\mathbf{K}_1^n\ x_1 \ldots x_n) \Rightarrow x_1$;

б) $(\mathbf{L}_{n-1}^n\ x_1 \ldots x_n) \Rightarrow (x_1 \ldots x_{n-1})$;

в) $(\mathbf{D}^n\ x) \Rightarrow (x \ldots x)$ **(n раз)**;

г) $(\mathbf{K}_i^n\ x_1 \ldots x_n) \Rightarrow x_i$;

д) **(P x y)** $\Rightarrow$ **(y x)** (эта задача сложнее);

е) **(W x y z)** $\Rightarrow$ **(x z y)** (эта задача сложнее всего).

Типы и изоморфизм Карри–Ховарда

Рассмотрим еще одно исключительно красивое свойство комбинаторов. В программировании (особенно в том, которое наиболее распространено сейчас) типы данных являются базисным понятием. В комбинаторной логике типов данных нет. Нет их (в том виде, как это общепринято) и в языке Lisp[1], наиболее применяемом из языков функционального программирования. В чем же здесь дело?

Типы данных появились (как и многие другие понятия информатики) в чистой математике, но так же, как и многие другие созданные математиками понятия, математики сами их практически не использовали (ввиду привычки к неаккуратности и нестрогости, отличающей их от информатиков; привычка же к необоснованности и примитиву отличает информатиков от математиков)[2].

Причиной появления типов была фундаментальная ошибка, найденная Б. Расселом в фундаментальном труде Г. Фреге по основаниям математики. Если взять множество всех множеств, не являющихся собственными элемен-

[1]Примечание для специалистов. Когда мы говорим Lisp, то под этим общим словом объединяются все диалекты данного языка, в частности, Common Lisp, Autolisp, Scheme.

[2]Достаточно привести пару примеров. Речь и даже статьи математиков пестрят фразами: «Очевидно», «Аналогично», «Ясно, как это сделать, так что не будем на этом задерживаться». Аргументация информатиков фразами — «Это и так работает» и мотивами типа «Чего думать, трясти надо».

тами (обозначим его **R**), то получаем противоречие вида $R \in R \Leftrightarrow R \notin R$. Рассел вскрыл, что причиной этой ошибки является смешение элементов разных типов, и предложил строго делить все элементы по типам. Элементом множества первого типа могут быть лишь объекты, элементом множества второго типа — лишь множества первого типа, и так далее. Карри вскрыл и причину, по которой парадокс Рассела не проникает в комбинаторную логику: в ней нет логического следования. В ней символ $\Rightarrow$ обозначает лишь преобразование, а преобразования не могут привести ни к чему худшему, чем зацикливание. Вот если добавить к комбинаторной логике хотя бы элементы обычной...

Необходимость строго следить за соответствием типов «стесняла свободу выражения» математиков, и они предпочли залатать теорию множеств, произвольным образом постулировав те аксиомы, которые, как кажется, не приводят к противоречию и которые достаточны для того, чтобы во многих случаях работать с неаккуратным различением типов. В последнее время появились грозные признаки того, что заплатки вышли неудачные, но концепция типов давно уже живёт и развивается там, где за формальные ошибки немедленно карают: в программировании.

Зачастую и в комбинаторной логике можно расставить типы данных. При этом нужно следить, чтобы аргумент функции был соответствующего типа.

Итак, у нас есть элементарные типы (обозначаемые буквами) и по каждым двум типам τ и π можно построить тип функций из τ и π: $\tau \to \pi$. То есть «$\tau \to \pi$» обозначает тип функции, которые получают аргумент типа τ и «возвращают» результаты типа π.

Тогда шейнфинкелевским комбинаторам удаётся приписать типы (и, более того, бесконечным числом способов). Например, **I** имеет типы формы $\tau \to \tau$ (тип аргумента равен типу результата), **K** — любые типы формы $(\tau \to (\pi \to \tau))$. При определении типа комбинатора следует считать, что комбинатор получает один аргумент (частичная параметризация) и возвращает новый комбинатор, который принимает оставшиеся аргументы.

Определим, для тренировки, тип комбинатора **F**:

$$(F\ x\ y) = (y\ x).$$

Пусть **x** имеет тип τ. Тогда комбинатор **y** должен быть функцией, принимающей аргумент типа τ, то есть иметь тип функции $\tau \to \pi$, где π — некоторый тип. Тип **F** получается равным $\tau \to ((\tau \to \pi) \to \pi)$.

Более нетривиальная задача — найти тип **S**, но и с ней можно справиться, если идти от конца к началу.

$$(S\ x\ y\ z) \Rightarrow ((x\ z)\ (y\ z)).$$

z везде является лишь аргументом, ему можно сопоставить произвольный тип **ρ**. А вот **x** и **y** являются функциям, принимающими **z** в качестве аргумента, и поэтому должны иметь типом аргумента **ρ**. **(y z)** далее ни к чему не применяется, и поэтому тип **y** можно описать как **(ρ→ π)**. А вот **(x z)** далее применяется как функция к аргументу типа **π**, и типом **x** должен быть **(ρ→(π→τ))**. В итоге тип **S** имеет вид

$$(((\rho \to (\pi \to \tau)) \to ((\rho \to \pi) \to (\rho \to \tau))).$$

Оказывается, что возможность правильно расставить типы в комбинаторном выражении является достаточным условием того, чтобы исключить зацикливание. В частности, уже **D** не может иметь типа, поскольку тип его аргумента **τ** должен совпадать с **(τ→τ)**, поскольку аргумент должен быть применён сам к себе.

А типы **S** и **K** с точностью до обозначений совпали с двумя аксиомами логики высказываний, рассматривающими импликацию (логическое следование)[1]

$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$$

$$A \Rightarrow (B \Rightarrow A).$$

Поэтому оказалось, что типы данных в комбинаторной логике изоморфны[2] доказуемым формулам *импликативной* логики (подсистемы логики высказываний, содержащей одну лишь импликацию). Таким образом, формулы логики имеют программную интерпретацию. Загадочное гильбертовское доказательство **A⇒ A**

(A⇒ ((A⇒ A)⇒ A))⇒	
((A⇒ (A⇒ A))⇒(A⇒ A))	**S**
(A⇒ ((A⇒ A)⇒ A))	**K**
(A⇒ (A⇒ A))⇒(A⇒ A)	**(S K)**
(A⇒ (A⇒ A)	**K**
A⇒ A	**((S K) K)**

является всего лишь выражением **I** через **K** и **S** в комбинаторной логике с типами. Этот *изоморфизм Карри-Ховарда* был опубликован лишь в 1968 г. во втором томе толстенной и трудной монографии по комбинаторной логике. Среди программистов он стал известен на десять лет позже, а ныне, кажется, опять забыт.

Но для получения всех тождественно истинных импликативных формул логики высказываний при помощи одной лишь импликации нужна ещё одна аксиома, в обычной логике доказывающаяся через посредство закона исключённого третьего: **A∨¬A**. Это формула Пирса:

$$((A \Rightarrow B) \Rightarrow A) \Rightarrow A.$$

[1] Это — одна из формализаций логики высказываний, исключительно удобная в теории и исключительно неудобная на практике: гильбертовская формулировка. Но и для других формулировок ситуация остается подобной.

[2] Изоморфизм — взаимно-однозначное соответствие между двумя математическими структурами, сохраняющее все свойства, выраженные на некотором ограничении математического языка. Знание изоморфизмов является важнейшим элементом математической культуры информатика, поскольку они открывают возможность безопасного перехода между совершенно разными структурами данных.

Программная интерпретация этой формулы состоит в том, что мы можем выпутаться из любой ошибочной ситуации, что не соответствует действительности. Добавление соответствующего комбинатора к комбинаторной логике приводит к разрушению всех её приятных свойств.

Так что мы приходим к выводу, что логика составления программ и та логика, которой пользуются традиционные математики, расходятся. И действительно, математическая теорема отнюдь не всегда даже в принципе даёт соответствующее построение. Но если она его даёт, то она обычно даёт исключительно красивое и широко применимое построение. Так что информатик должен знать математику и, мало того, знать отлично, в некоторых отношениях даже лучше, чем «профессиональные математики», поскольку они лишь решают задачки, а хороший информатик должен их ставить, анализировать и воплощать в работающие системы.

И в заключение статьи заметим, что с алгебраической точки зрения Вы только что познакомились с любопытнейшей системой. Если писать подразумеваемую в скобках операцию, например, обозначая её звёздочкой, то мы получаем простую неассоциативную и некоммутативную алгебраическую систему с двумя константами и двумя аксиомами:

(K*x)*y = x;

((S*x)*y)*z = (x*z)*(y*z).

Это простейшая из известных автору практически важных алгоритмически неразрешимых систем: нет алгоритма, который бы проверял равенство двух выражений в данной системе.

Юмор Юмор Юмор Юмор Юмор Юмор

- После экзамена. Студент – друзьям:

  – Ну и гад наш препод! Я ему билет как орех щёлкнул, а он дополнительными вопросами стал засыпать! Он – вопрос, я – ответ. Он – вопрос, я – ответ. И всё равно только трояк поставил.

  Профессор – преподавателям:

  – Ну и дуб мне сегодня попался. По билету – ни в зуб ногой, чушь какую-то несусветную нёс. Пожалел я его, тянул-тянул наводящими вопросами, еле на троечку вытянул...

- – В чём разница между математиком и физиком?
  – Математик полагает, что достаточно двух точек, чтобы провести через них прямую. Физик обязательно потребует дополнительных данных.