# Зависимые типы

Типизация – одна из тех вещей, которые настолько неотделимы от нашего мышления, что мы едва ли даже задумываемся о концепции типов как таковой? Почему 1 – это int, но, стоит только поместить это значение в кавычки – и оно превращается в string? Что же такое «тип», в сущности? Как это часто бывает в программировании, ответ зависит от формулировки вопроса.

Типы многообразны. В некоторых системах типов существуют очень четкие границы между типами и значениями. Так, 3, 2 и 1 – это значения типа integer, но integer – это не значение. Этот конструкт «вмурован» в язык и принципиально отличается от значения. Но, на самом деле, такое различие необязательно и может лишь ограничивать нас.  
 Если освободить типы и превратить их в еще одну категорий значений, то открывается ряд потрясающих возможностей. Значения можно хранить, преобразовывать и передавать функциям. Таким образом, можно было бы сделать функцию, принимающую тип в качестве параметра, создавая обобщенные функции: такие, которые могут работать со множеством типов без перегрузок. Можно иметь массив значений заданного типа, а не заниматься странной арифметикой указателей и приведением типов, как приходится делать в C. Также можно собирать новые типы по ходу выполнения программы и обеспечивать такие возможности, как, например, автоматическая десериализация JSON. Но, даже если трактовать типы как значения, все равно с типами не сделаешь всего того, что можно делать со значениями. Так, оперируя экземплярами user, можно, например, сравнивать их имена, проверять их возраст или идентификатор и т.д.  
 if user.name == "Marin" && user.age

Однако, при попытке сделать то же самое с типом User, вы могли бы сравнить только имена типов и, возможно, имена свойств. Поскольку это тип, а не экземпляр, нельзя проверить значения его свойств.

if typeof(user) == User { print("Well, it's a user. That's all I know") }

Как было бы круто, если бы у нас была функция, способная получать лишь непустой список пользователей? Либо функция, которая принимала бы адрес электронной почты, лишь если он записан в правильном формате? Для этих целей вам понадобились бы типы «непустой массив» или «адрес электронной почты». В данном случае речь идет о типе, зависящем от значения, т.е. о зависимом типе. В мейнстримовых языках подобное невозможно.

Чтобы типами можно было пользоваться, компилятор должен проверять их. Если вы утверждаете, что переменная содержит integer, то лучше бы в ней не было string, а то компилятор станет ругаться. В принципе, это хорошо, поскольку не дает нам набажить. Проверять типы совсем просто: если функция возвращает integer, а мы пытаемся вернуть в ней "Marin", то это ошибка.

Однако, с зависимыми типами все сложнее. Проблема заключается в том, когда именно компилятор проверяет типы. Как ему убедиться, что в массиве именно три значения, если программа еще даже не выполняется? Как убедиться, что целое число больше 3, если оно еще даже не присвоено? Для этого есть магия… или, иными словами, математика. Если можно математически доказать, что множество чисел всегда больше 3, то компилятор может это проверить.

Вернемся к программированию

Итак, мы установили, что некоторые вещи вполне можно сначала доказать, а потом переходить к конкретным значениям. Чтобы сделать это на языке программирования, нужен способ выразить эти утверждения в коде, который будет записан в саму систему типов, то есть, систему типов требуется усовершенствовать.

Рассмотрим пример. Здесь у нас есть функция append, принимающая два массива и комбинирующая их. Как правило, сигнатура такой функции будет выглядеть примерно так:

append: (arr1: Array, arr2: Array) -> Array

Однако, просто взглянув на сигнатуру, мы не можем быть уверены в правильности реализации. Сам тот факт, что функция возвращает массив, еще не означает, что она что-то сделала. Один из способов проверить результат – убедиться, что длина результирующего массива равна сумме длин массивов параметров.

newArray = append([1], [2, 3]) assert(length(newArray) == 3)

Но зачем же проверять это во время выполнения, если можно создать ограничение, которое будет проверяться во время компиляции программы:

append: (arr1: Array, arr2: Array) -> newArray: Array where length(newArray) == length(arr1) + length(arr2)

Мы объявляем, что append – это функция, принимающая два аргумента Array и возвращающая новый аргумент Array, который мы назвали newArray. Лишь на сей раз мы добавляем оговорку о том, что длина нового массива должна равняться сумме длин всех аргументов функции. Утверждение, имевшееся у нас выше во время выполнения, во время компиляции преобразуется в тип.

Вышеприведенный код относится к миру типов, а не значений, то есть, знак == свидетельствует о сравнении возвращаемого типа length, а не его значения. Чтобы такой механизм работал, возвращаемый тип length должен давать нам какую-либо информацию о фактическом числе.

Чтобы обеспечить работу такого механизма, нужно убедиться, что каждое число относится к отдельному типу. Тип One может содержать всего одно значение: 1. То же самое касается Two, Three и всех других чисел. Естественно, такая работа очень утомительна, но именно для такой работы у нас есть программирование. Можно написать компилятор, который будет делать это за нас.

Сделав это, можно создать отдельные типы для массивов, содержащих 1, 2, 3 и другое количество элементов. ArrayOfOne, ArrayOfTwo, т.д.

Таким образом, можно определить функцию length, которая станет принимать один из вышеприведенных типов массивов и иметь зависимый возвращаемый тип One для ArrayOfOne, Two для ArrayOfTwo и т.д. для каждого числа.

Теперь, когда у нас есть отдельный тип на любую конкретную длину массива, можно убедиться (во время компиляции), что оба массива имеют равную длину. Для этого сравним их типы. А поскольку типы – это такие же значения, как и любые другие, можно назначать операции над ними. Можно определить сложение двух конкретных типов, задав, что сумма ArrayOfOne и ArrayOfTwo равна ArrayOfThree.

Зависимая типизация – это очень круто В таком случае огромное количество багов допустить попросту невозможно. При зависимой типизации можно избежать ошибок на единицу, обращений к несуществующим индексам массивов, исключений нулевого указателя, бесконечных циклов и неработоспособного кода.

При помощи зависимых типов можно выразить практически что угодно. Функция факториала будет принимать только натуральные числа, функция login не будет принимать пустых строк, функция removeLast будет принимать только непустые массивы. Причем, все это проверяется до того, как вы запустите программу.

Проблема с проверками во время выполнения заключается в том, что они не удаются, если программа уже работает. Это нормально, если программа запущена только у вас, но не у пользователя. Зависимые типы позволяют вынести такие проверки на уровень типов, поэтому отказ такого рода в ходе выполнения программы становится невозможен.

Обзор языка Idris

Idris — [чистый функциональный язык](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). Но также поддерживается опциональное требование свойства totality функций, под которым подразумевается два свойства:

1. Определённость всюду: функция должна быть определена для любых входных данных. В чистых языках отсутствие этого свойства — едва ли не единственная возможность для программы «упасть» (другие — IO, в частности убийство процесса системой вследствие каких-то его действий, и возможные баги компилятора).
2. Строгая нормализация: при рекурсивных вызовах функции, хотя бы один из её аргументов должен строго уменьшаться (структурно), либо функция должна быть продуктивна на каждой итерации — т.е. возвращать какую-то часть результата, и обещание просчитать следующую итерацию. В первом случае это гарантирует завершаемость функции, обходя проблему остановки, а во втором — возможность считать конечный префикс любой длины за конечное время.

Деление

Знакомая всем операция, которая есть практически во всех языках программирования (или в их стандартных библиотеках) — деление. Но деление на 0 запрещено, что ведёт к различным спорным решениям:

* Вернуть что-нибудь специальное, вроде Infinity: тогда мы должны определить и правила для работы всех остальных функций/операций с Infinity, что усложняет арифметику в целом, но, например, простое свойство a / b = c → b \* c = a будет утеряно, и далее попытка его использования запросто приведёт к результатам, отличным от ожидаемых.
* Породить исключение: это не только приводит к нарушению нормального потока выполнения программы, заодно опять-таки создавая почву для багов при непойманных исключениях, но и значительно затрудняет рассуждение (в частности, формальное) о функциях, если не делает его невозможным.
* Обернуть результат в [Maybe](https://en.wikipedia.org/wiki/Option_type), возвращая Nothing при попытке деления на 0: чем-то лучше предыдущих двух, разве что сама функция не будет стимулировать программиста писать корректные программы, а формулы могут стать менее аккуратными и/или излишне насыщенными теорией категорий, попутно затрудняя рассуждения о функциях, использующих такую реализацию деления.

В языках с зависимыми типами, и в Idris в частности, появляется ещё один вариант: запретить деление на 0. Другими словами, деление на 0 запрещено — и мы просто делаем его невозможным; тип функции тогда описывается как «функция, принимающая два числа, второе из которых не равно 0, и возвращающая число». При желании, можно дополнить эту спецификацию свойством, говорящим о связи возвращаемого значения с аргументами функции, т.е. "… такое, что...", но проще это оставить для отдельных теорем, а пока вернёмся к описанию типа функции деления: к обычным двум аргументам, нам нужно добавить третий — доказательство того, что второй аргумент не равен нулю.  
  
Теперь рассмотрим три случая использования деления (напомню, что речь идёт о времени компиляции):

1. Делитель известен: доказательство тривиально, поскольку эквивалентность натуральных чисел разрешима (другими словами, компилятор сам видит, равен делитель нулю или нет), и потому мы можем, при желании, обернуть такую функцию деления в функцию с обычными двумя аргументами, подставляющую один и тот же простой третий аргумент.
2. Делитель неизвестен, но о нём известно что-то, что позволяет убедиться в том, что он не равен нулю: например, что он больше некоторого неотрицательного числа. Тогда нужно или написать доказательство на месте, или воспользоваться функцией, принимающей то, что у нас есть (доказательство того, что число больше некоторого неотрицательного числа), и возвращающей то, что нам нужно (доказательство того, что это число не равно нулю). Такие функции можно называть как просто функциями, так и леммами или теоремами, но, как уже было упомянуто, в данном контексте это одно и то же.
3. Мы сами не уверены в том, что делитель не будет равен нулю (после внимательного рассмотрения ситуации в попытке доказать неравенство, предыдущий случай может переходить в этот): только в этом случае нам действительно нужно ввести проверку перед делением; если делитель равен нулю, то использовать альтернативное решение, а в противном случае — воспользоваться полученной информацией и выполнить деление.

Такой подход, конечно, необязателен и в Idris, но заставляет программиста убедиться в том, что он выполняет расчёты корректно.  
  
И, наконец, пример кода — каркас для функции деления:

*-- (total) функция деления для натуральных чисел*

*-- "==" - функция проверки разрешимой эквивалентности, возвращающая Bool*

*-- "=" - эквивалентность высказываний, тип*

total div : (n, m: Nat) -> ((m == Z) = False) -> Nat

*-- деление на 0 - невозможно; просто показываем это компилятору*

*-- с тем же успехом, впрочем, можно и вернуть любое число*

div n Z f = FalseElim $ trueNotFalse f

*-- деление на положительное число*

div n (S k) f = *{-todo-}*?div\_rhs

Поскольку это только каркас, здесь использована мета-переменная, div\_rhs; людям, знакомым с Agda, уже знакомы «дырки» (holes), более слабой версией которых мета-переменные в Idris являются, а для остальных поясню: Idris позволяет посмотреть её контекст (переменные, которые видны из её позиции) и цель (что должно быть сконструировано/возвращено из данной позиции), что заметно облегчает написание как доказательств, так и программ. Вот так он выглядит в данном случае:

n : Nat

k : Nat

f : S k == 0 = False

--------------------------------------

div\_rhs : Nat

Также есть возможность заполнять их полу-автоматически: имея цель вернуть Nat, можно воспользоваться командой (REPL/IDE) «refine», передав ей, например, имя функции plus, и если функция plus в состоянии вернуть значение требуемого типа, то она будет подставлена вместо этой мета-переменной, а для её аргументов будет подставлено две новых меты. Процесс это тот же, который используется и для другой функции — полностью автоматической замены мета-переменных, т.е. поиском доказательства: в некоторых случаях требуемое доказательство может быть найдено автоматически.  
  
Типы (немного подредактированы для читабельности):

λΠ> :t div

div : Nat -> (m : Nat) -> (m == 0 = False) -> Nat

λΠ> :t div 1 2

div 1 2 : (2 == 0 = False) -> Nat

λΠ> :t div 1 2 refl

div 1 2 refl : Nat

λΠ> :t div 1 0 refl

(ошибка, программа с подобным вызовом не пройдёт проверку типов)

Теперь отвлечёмся от реализации функции деления и посмотрим, что такой подход нам даёт. Начнём с вышеупомянутой корректности: функция работает именно так, как должна. Мы можем думать о функции так, как обычно думаем о том, что ею реализуется (арифметическое действие, в данном случае: мы не можем делить на ноль, но в процессе деления никакие концепции наподобие исключений не участвуют, а результатом является число). Можем рассуждать о функции так же, как о самой представляемой ею концепции, в том числе и формально, т.е. записывая рассуждения кодом. В функциях, использующих её, рассуждения и выводы теперь могут переплетаться с вычислениями и быть проверены, а не оставаться в голове программиста или текстовых комментариях. И программа становится конструкцией из корректных рассуждений и решений, а не просто рецептом для вычислений.

Списки

Работа со списками (или массивами) — ещё одна частая задача, приводящая примерно к тому же набору спорных решений и последующих ошибок. В частности, рассмотрим функцию, возвращающую энный элемент списка: проблема при её реализации состоит в том, что мы можем запросить элемент с позиции, которой нет в списке. Обычные решения — падать с «access violation» или «segmentation fault», порождать исключение (и, возможно, потом падать с ним), использовать Maybe.  
  
В языках с зависимыми типами появляется ещё пара возможных решений. Первое — аналогично решению проблемы деления на ноль, т.е. последним аргументом функции становится доказательство того, что запрашиваемая позиция меньше длины списка.  
  
Здесь можно заметить, что различные числа, списки, «if-then-else» и т.п. определяются на самом языке. Натуральные числа, реализованные в библиотеке, заменяются на GMP числа при компиляции в C, а такие функции, как сложение и умноженние — на соответствующие функции GMP, что даёт и быстрые расчёты в скомпилированных программах, и возможность рассуждать о них. Теперь взглянем на код:

data Nat = Z | S Nat

data List a = Nil | (::) a (List a)

length : List a -> Nat

length [] = 0

length (x::xs) = 1 + length xs

index : (n : Nat) -> (l : List a) -> (ok : lt n (length l) = True) -> a

index Z (x::xs) p = x

index (S n) (x::xs) p = index n xs ?indexTailProof

index \_ [] refl impossible

indexTailProof = proof {

intros;

rewrite sym p;

trivial;

}

indexTailProof — доказательство с использованием тактик, в стиле Coq, но можно (и мне чем-то больше нравится) писать их и в стиле Agda, т.е. более привычными функциями (говорят, правда, что скоро тактики будут и в Agda).  
  
Второе решение — использовать не просто список, а список фиксированной длины: вектор, как его называют в контексте языков с зависимыми типами. Пример был упомянут выше: это то, что позволяет сказать «список из пяти строк». Также нам может помочь т.н. конечный тип — то есть тип, количество различных значений которого конечно: можем рассматривать его как тот же тип натуральных чисел, но с верхней границей. Удобен этот тип в частности тем, что с ним проще сказать «число, меньшее n», чем с натуральными числами, где это скорее «число, и оно меньше n». Тип функции получения энного элемента вектора тогда читается как «функция, принимающая число, меньшее чем n, и список из n элементов типа a, и возвращающая элемент типа a».  
  
В таком случае, задачей становится не предоставить доказательство, а сконструировать число нужного типа (т.е. меньшее, чем n). К слову, логика в доказательствах здесь отличается от классической (является интуиционистской), и заключается в конструировании доказательств, то есть доказательством высказывания/типа является любое значение данного типа, так что некоторое сходство с предыдущим подходом проследить можно, но в то же время подход к написанию программы в целом это меняет.  
  
Код:

data Fin : (n : Nat) -> Type where

fZ : Fin (S k)

fS : Fin k -> Fin (S k)

data Vect : Nat -> Type -> Type where

Nil : Vect Z a

(::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a

index : Fin n -> Vect n a -> a

index fZ (x::xs) = x

index (fS k) (x::xs) = index k xs

*-- запросить что-либо из пустого вектора невозможно, в данном случае*

*-- компилятор это "видит" сам*

index fZ [] impossible

Так мы уходим ещё дальше от явных доказательств, но строим, тем не менее, корректные программы.  
  
Ну а раз уж добрались до векторов, то вот и функция их объединения, часто приводимая в пример для демонстрации зависимых типов:

(++) : Vect m a -> Vect n a -> Vect (m + n) a

(++) [] ys = ys

(++) (x::xs) ys = x :: xs ++ ys

Тело то же, что и для обычных списков, но тип поинтереснее: обратите внимание на m + n.

Примеры побольше

Выше приведены примеры отдельных небольших функций, которые Idris помогает сделать корректными, но вот несколько более крупных вещей навскидку:

* Протоколы: есть библиотека [Protocols](https://github.com/edwinb/Protocols), позволяющая описывать протоколы с использованием DSL, а затем реализовывать их; несоответствие реализации описанию протокола приведёт к ошибке.
* [Физика](https://github.com/timjb/quantities) и [игры](https://github.com/edwinb/idris-demos): не только поддерживается корректность вычислений, но и полезно доказывать теоремы уже после написания основных программ.
* Законы функторов, аппликативных функторов и монад [могут быть включены в код](https://github.com/reynir/Verified), в сами тайпклассы.
* [Криптография](https://github.com/idris-hackers/idris-crypto): очевидный пример того, корректность чего важна.
* Разбор и составление текстовых/бинарных форматов: может быть формально доказано, что любые составленные строки разбираются в исходные данные, а разобранные данные — составляются в исходные строки. Примерно это я и выбрал в качестве первого проекта на Idris.
* Системы прав доступа, базы данных, драйверы устройств — реализаций на Idris пока не видел, но тоже вещи, которые хочется видеть надёжными.

Пишется на нём и [веб-фреймворк](https://github.com/idris-hackers/IdrisWeb), и [библиотека для взаимодействия с DOM](https://github.com/idris-hackers/iQuery) (кстати, на момент написания статьи компилятор поддерживает компиляцию в C, LLVM, Java и JavaScript), и всяческие парсеры ([раз](https://github.com/ziman/lightyear), [два](https://github.com/tauli/idris-monadic-parser)), и на системное программирование он нацелен. Переписываются программы с Haskell на Idris достаточно просто.  
  
Не используется Idris пока на производстве — по крайней мере, широко (слышал, что в Potsdam Institute for Climate Impact Research он активно используется, но это, возможно, не совсем то, что обычно понимается под производством). Основная техническая преграда — «необкатанность», которая должна решиться «обкатыванием».

Заключение

Язык это новый, и потому «сырой», но современный, использующий современную теорию и накопленную другими языками практику.  
  
Позволив себе немного помечтать о будущем использовании Idris (и языков, которые появятся после него), можно представить, помимо общего повышения надёжности программ, технические задания, сопровождаемые формальными спецификациями или даже состоящие из них; рост сервисов наподобие [Proof Market](https://proofmarket.org/); библиотеки физических, математических, да и любых других теорий, и языков наподобие Lojban; статьи и книги как списки формальных спецификаций доказанных теорем; общение, выстраиваемое из автоматически проверяемых реплик и пар вопрос-ответ в частности.