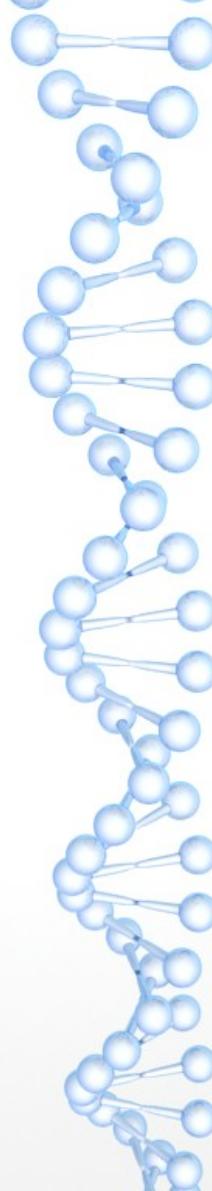


Функциональное и логическое программирование

Лекция 4. Типы.

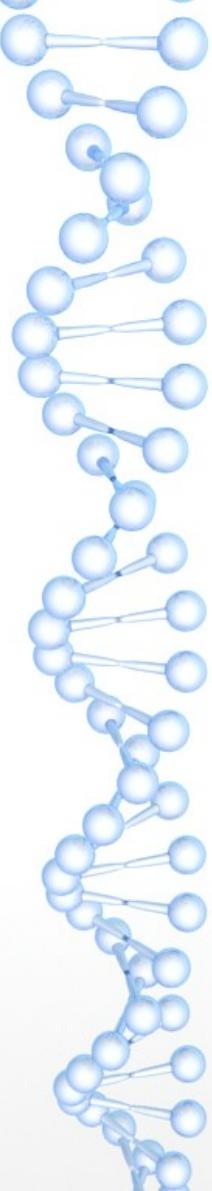
- 4.1 Списки Чёрча
- 4.2 Типизированное лямбда исчисление
- 4.3 Полиморфизм



Списки Чёрча

Числа Чёрча могут быть обобщены для представления списков. Список $[x_1, x_2, \dots, x_n]$ в таком случае можно было бы представить как функцию от f и y с телом $fx_1(fx_2 \dots (fx_ny) \dots)$. Такие списки содержали бы собственную структуру управления.

В качестве альтернативы представим списки через операцию образования пар. Этот способ кодирования легче понять, потому что он ближе к прикладным реализациям. Список $[x_1, x_2, \dots, x_n]$ будет представлен как $x_1 :: x_2 :: \dots :: x_n :: nil$. Чтобы операции были максимально простыми, мы будем использовать два уровня пар. Каждая “ячейка списка” $x :: y$ будет кодироваться как **(false, (x, y))**, где **false** является отличительным полем (тегом). По правилам, **nil** тоже следовало бы представить парой, в которой первый компонент был бы равен **true**, но работает и более простое определение.

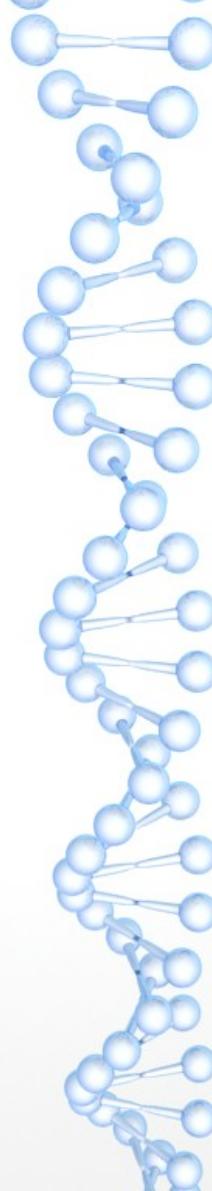


Списки Чёрча

```
LET  nil  =  $\lambda z. z$ 
LET  cons =  $\lambda xy. \text{pair } \text{false} (\text{pair } xy)$ 
      LET null = fst
      LET hd =  $\lambda z. \text{fst } (\text{snd } z)$ 
      LET tl =  $\lambda z. \text{snd } (\text{snd } z)$ 
```

Следующие свойства легко проверить; они выполняются для любых выражений M и N :

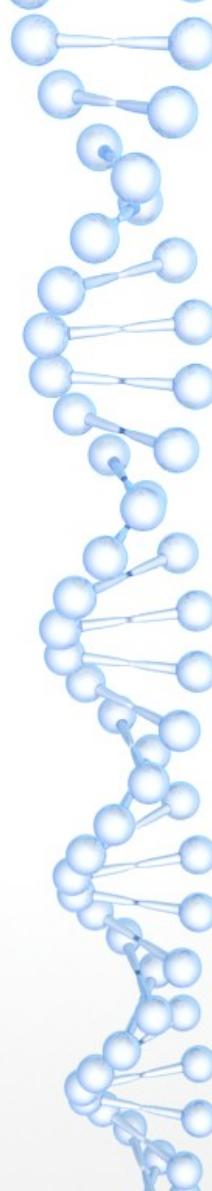
```
null nil  = true ,
null (cons  $M N$ ) = false ,
hd (cons  $M N$ ) =  $M$  ,
tl (cons  $M N$ ) =  $N$  .
```



Списки Чёрча

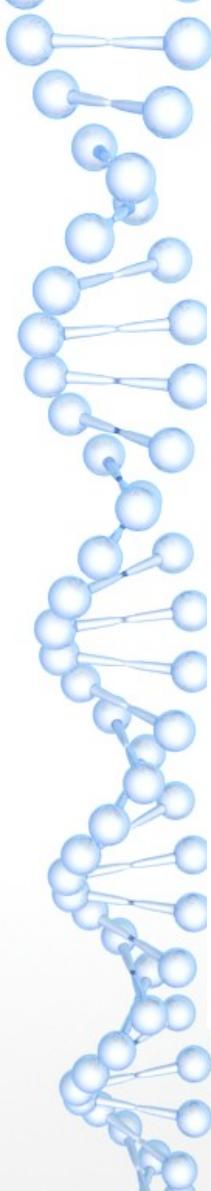
Обратите внимание, что **null nil = true** вышло действительно случайно, в то время как другие соотношения являются следствием использования наших операций над парами.

Также заметьте, что ключевые характеристические соотношения **hd (cons M N) = M** и **tl (cons M N) = N** выполняются для любых M и N , в том числе даже для таких выражений, которые не имеют нормальных форм! Таким образом, **pair** и **cons** — “ленивые” операторы, то есть они не вычисляют свои аргументы. Следовательно, после введения рекурсивных определений мы сможем с их помощью работать даже с бесконечными списками.



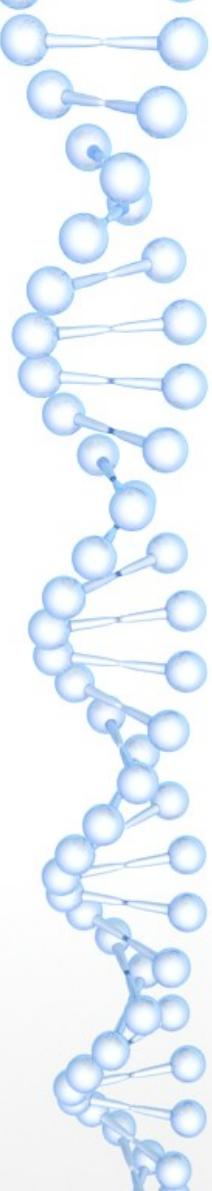
Списки Чёрча

nil	$= \lambda cn. n$
cons	$= \lambda xlcn. c x (l c n)$
head	$= \lambda l. l (\lambda xy. x) \text{nil}$
tail	$= \lambda l. \text{snd} (l (\lambda xp. (\text{cons} x (\text{fst} p), \text{fst} p)) (\text{nil}, \text{nil}))$
append	$= \lambda l_1 l_2. l_1 \text{ cons } l_2$
applists	$= \lambda L. L \text{ append nil}$
map	$= \lambda fl. l (\lambda x. \text{cons} (f x)) \text{nil}$
length	$= \lambda l. l (\lambda x. \text{suc}) 0$
tack	$= \lambda xl. l \text{ cons} (\text{cons} x \text{nil})$
reverse	$= \lambda l. l \text{ tack nil}$
filter	$= \lambda lp. l (\lambda x. (p x) (\text{cons} x) (\lambda y. y)) \text{nil}.$



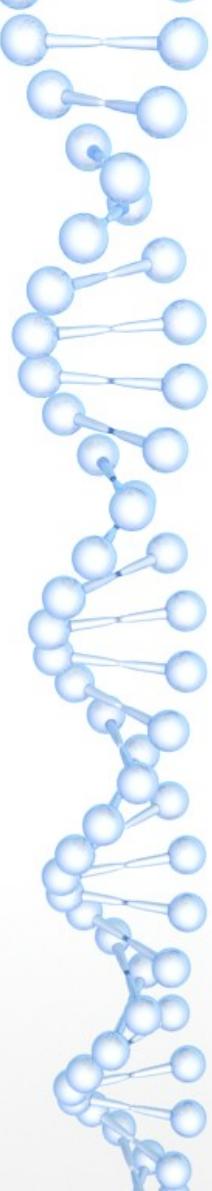
Типизированное лямбда исчисление

В первом приближении расширение лямбда-исчисления понятием типа не представляет особого труда, но в итоге, как будет показано, потребуется куда больше усилий. Основная идея состоит в том, что каждому терму назначается *тип*, после чего выражение $s\ t$, т. е. применение терма s к терму t , допустимо исключительно для совместимых типов, то есть в случае, когда типы s и t имеют вид $\sigma \rightarrow \tau$ и σ соответственно. Результирующий терм будет иметь при этом тип τ . Такую типизацию принято называть *сильной*.⁴ Терм t обязан иметь тип σ , подтипы и преобразования не допускаются. Такой подход составляет резкий контраст с некоторыми языками программирования, например, с языком C, в котором функция, ожидающая аргумент типа `float` либо `double`, принимает также значения типа `int`, выполняя автоматическое преобразование. Аналогичные понятия подтипов и преобразований возможно задать и в рамках лямбда-исчисления, но их освещение завело бы нас слишком далеко.



Типизированное лямбда исчисление

Введём для отношения « t имеет тип σ » обозначение $t : \sigma$. Подобная запись традиционно используется математиками при работе с функциональными пространствами, поскольку $f : \sigma \rightarrow \tau$ обозначает функцию f , отображающую множество σ во множество τ . Будем считать типы множествами, которые содержат соответствующие объекты, и трактовать $t : \sigma$ как $t \in \sigma$. Однако, несмотря на то, что мы предлагаем читателям также воспользоваться этой удобной аналогией, типизированное лямбда-исчисление будет в дальнейшем рассматриваться исключительно как формальная система, свободная от каких-либо интерпретаций.



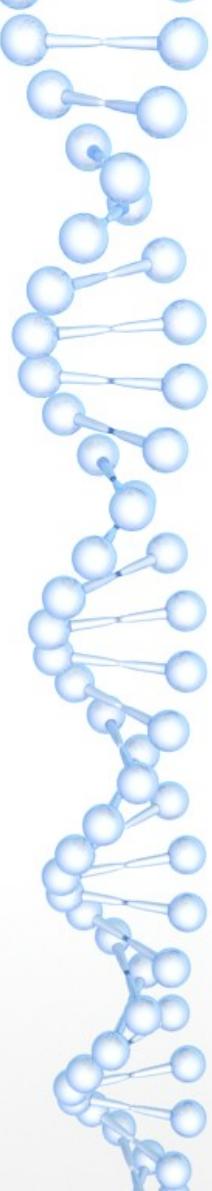
Типизированное лямбда исчисление

Начнём формализацию строгим определением понятия типа. Предположим, что у нас имеется некоторое множество *примитивных типов*, в которое входят, например, типы `bool` и `int`. Составные типы могут быть определены при помощи *конструктора типа функции*. Формально, индуктивное определение множества типов Ty_C , основанного на множестве примитивных типов C , выглядит так:

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \rightarrow \tau \in Ty_C}$$

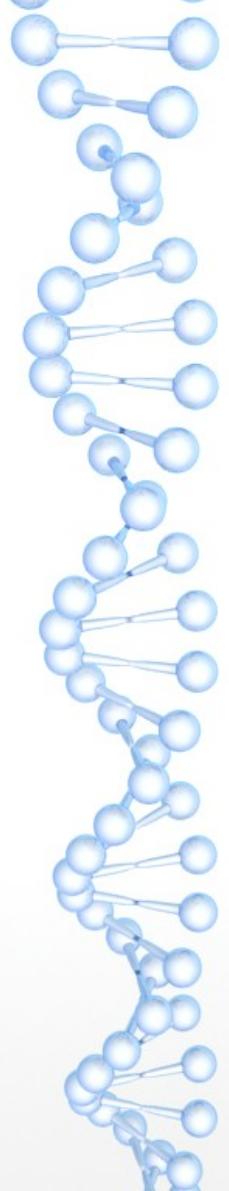
Например, в рамках данного определения допустимы типы int , $bool \rightarrow bool$ либо $(int \rightarrow bool) \rightarrow int \rightarrow bool$. Будем считать операцию « \rightarrow » правоассоциативной, т. е. полагать выражение $\sigma \rightarrow \tau \rightarrow v$ равным $\sigma \rightarrow (\tau \rightarrow v)$. Такая трактовка естественно согласуется с другими синтаксическими правилами, касающимися каррирования.



Типизированное лямбда исчисление

Следующим нашим шагом будет расширение системы типов в двух направлениях. Во-первых, введём наравне с примитивными типами (которые выполняют роль констант) так называемые *переменные типа*, которые впоследствии лягут в основу полиморфизма. Во-вторых, разрешим использование множества конструкторов других типов, помимо типа функции. Например, в дальнейшем нам понадобится конструктор \times для типа декартова произведения. Как следствие, наше индуктивное определение должно быть дополнено ещё одним выражением:

$$\underline{\sigma \in \text{Ty}_C \quad \tau \in \text{Ty}_C}$$



Типизированное лямбда исчисление

Известны два основных подхода к определению типизированного лямбда-исчисления. Один из них, разработанный Чёрчем, подразумевает явное указание типов. Каждому терму при этом назначается единственный тип. Другими словами, в ходе построения термов каждому нетипизированному терму, которые были рассмотрены ранее, в дополнение указывается тип. Типы констант являются предопределёнными, но типы переменных могут быть произвольными. Точные правила построения типизированных термов приведены ниже:

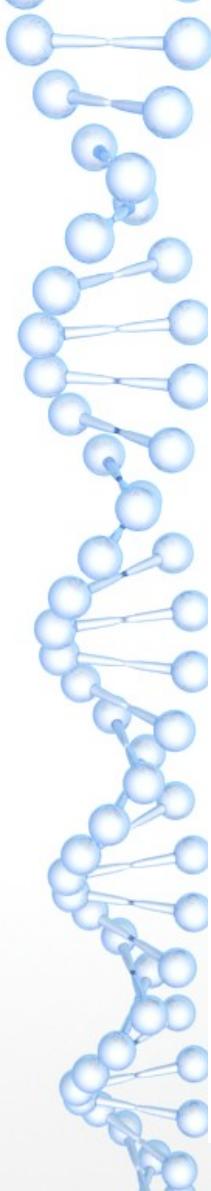
$$\overline{v : \sigma}$$

Константа c имеет тип σ

$$c : \sigma$$

$$\frac{s : \sigma \rightarrow \tau \quad t : \sigma}{s\ t : \tau}$$

$$\frac{v : \sigma \quad t : \tau}{\lambda v. t : \sigma \rightarrow \tau}$$

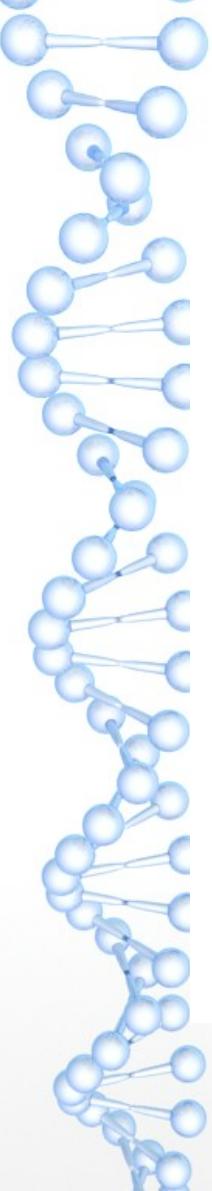


Типизированное лямбда исчисление

В то же время, некоторые формальные аспекты назначения типов по Карри оказываются достаточно сложными. Отношение типизируемости не может быть задано в отрыве от некоторого *контекста*, представляющего собой конечное множество утверждений относительно типов переменных. Обозначим через

$$\Gamma \vdash t : \sigma$$

утверждение «в контексте Γ терму t может быть назначен тип σ ». (Если это утверждение справедливо при пустом контексте, выражение сокращается до $\vdash t : \sigma$ или даже до $t : \sigma$.) Элементы множества Γ имеют вид $v : \sigma$ т. е. сами по себе являются утверждениями относительно типов отдельных переменных, обычно тех, которые входят в терм t . Будем полагать, что контекст Γ не содержит противоречивых утверждений о типе некоторой переменной; при желании, мы можем рассуждать о нём как о частичной функции, отображающей индексное множество переменных во множество типов. Использование нами символа \vdash соответствует его роли в традиционной логике, где $\Gamma \vdash \phi$ принято трактовать как «утверждение ϕ следует из множества посылок Γ ».



Типизированное лямбда исчисление

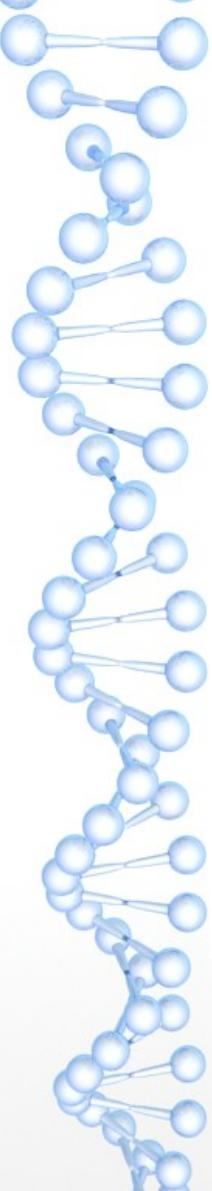
Формулировка правил назначения типов выражениям достаточно естественна. Прежде, чем мы приведём эти правила, напомним ещё раз, что $t : \sigma$ следует интерпретировать как « t может иметь тип σ ».

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma}$$

$$\frac{\text{Константа } c \text{ имеет тип } \sigma}{c : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \rightarrow \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s t : \tau}$$

$$\frac{\Gamma \cup \{v : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda v. t : \sigma \rightarrow \tau}$$



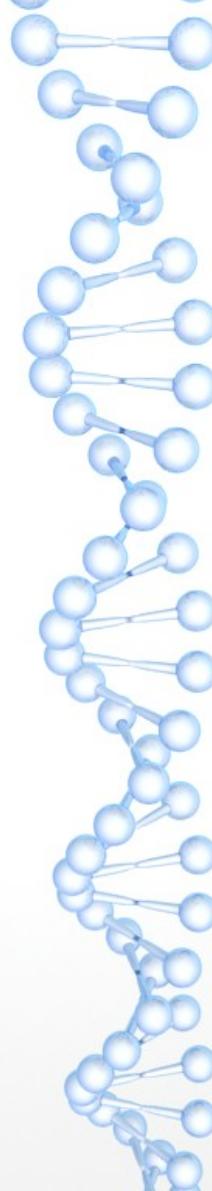
Типизированное лямбда исчисление

Ещё раз повторим, что эти выражения следует понимать как индуктивное определение отношения типизируемости, так что терм может иметь тип лишь тогда, когда последний выводим при помощи упомянутых выше правил. В качестве примера рассмотрим процедуру вывода типа тождественной функции. Согласно правилу типизации переменных, мы имеем:

$$\{x : \sigma\} \vdash x : \sigma$$

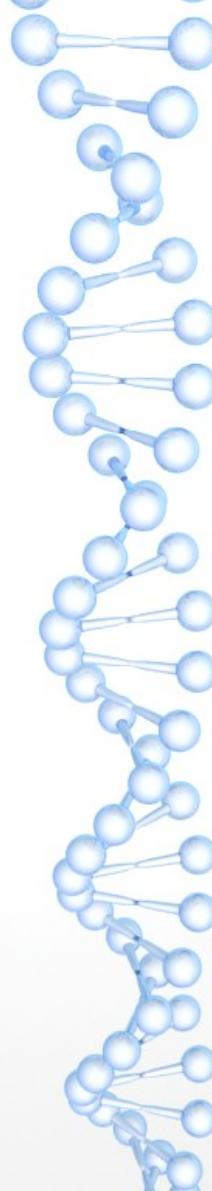
откуда, применив последнее правило, получаем:

$$\emptyset \vdash \lambda x. x : \sigma \rightarrow \sigma$$



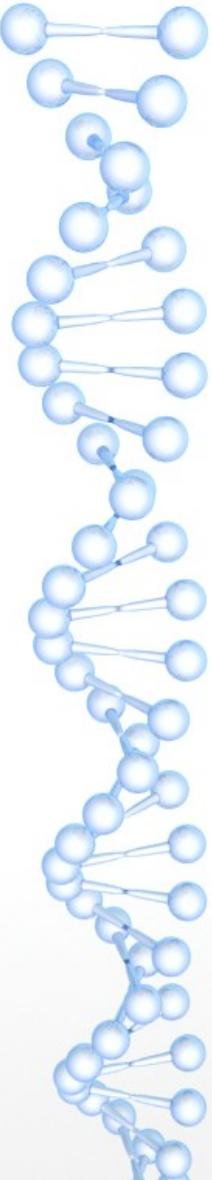
Типизированное лямбда исчисление

Применив установленное ранее соглашение о пустых контекстах, мы можем сократить это выражение до $\lambda x. x : \sigma \rightarrow \sigma$. На данном примере также хорошо видна как важная роль контекстов в типизации по Карри, так и их необязательность в рамках типизации по Чёрчу. Опуская контекст, мы можем вывести $x : \tau$ для произвольного типа τ , после чего, согласно последнему правилу, получаем $\lambda x. x : \sigma \rightarrow \tau$ — налицо различие с интуитивной трактовкой тождественной функции! Эта проблема не возникает в ходе типизации по Чёрчу, поскольку в её рамках либо обе переменные имеют тип σ , откуда получаем $\lambda x. x : \sigma \rightarrow \sigma$, либо эти переменные на самом деле различны (поскольку различны их типы, которые считаются неотъемлемой частью терма). В последнем случае тип выражения в действительности будет равен $\lambda x : \sigma. (x : \tau) : \sigma \rightarrow \tau$, но это обосновано тем, что данное выражение альфа-эквивалентно $\lambda x : \sigma. (y : \tau) : \sigma \rightarrow \tau$. Поскольку в ходе типизации по Карри термы не содержат в себе типов явно, нам требуется некоторый механизм связывания между собой одинаковых переменных.



Типизированное лямбда исчисление

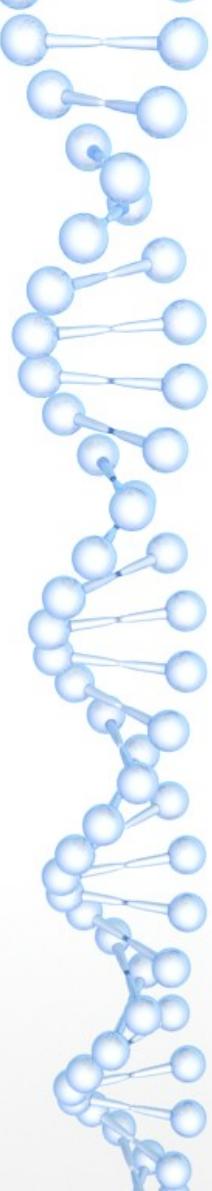
Очевидное сходство структуры термов типизированного и нетипизированного лямбда-исчисления порождает естественное желание применить в типизированном случае аппарат формальных преобразований, разработанный для нетипизированного. Однако, нам потребуется предварительно доказать, что тип выражения в ходе преобразований не изменяется (такое свойство называется *сохранением типа*). Убедиться в этом не представляет труда; мы рассмотрим краткое изложение доказательства для случая η -преобразования, предоставив остальные читателю в качестве упражнения. Прежде всего, докажем две леммы, которые весьма очевидны, но тем не менее, требуют формального обоснования. Во-первых, покажем, что добавление новых элементов в контекст не влияет на типизируемость:



Типизированное лямбда исчисление

Лемма 4.1 (О монотонности) Если $\Gamma \vdash t : \sigma$ и $\Gamma \subseteq \Delta$, то справедливо $\Delta \vdash t : \sigma$.

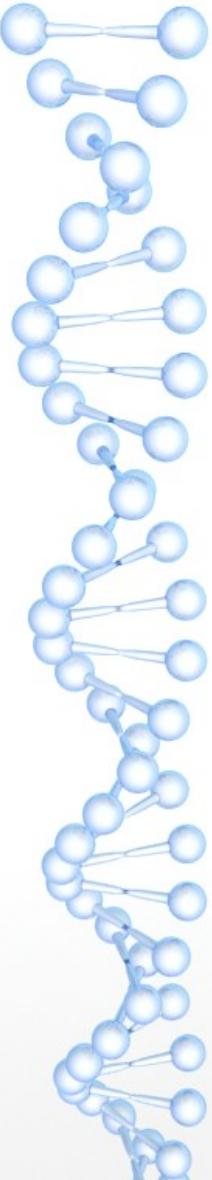
Доказательство: Применим индукцию по структуре t . Зафиксирував t , докажем приведённое выше утверждение для всевозможных Γ и Δ , поскольку в ходе шага индукции для абстракций эти множества изменяются. Если t — переменная, справедливо $t : \sigma \in \Gamma$, из чего следует $t : \sigma \in \Delta$, откуда получаем требуемое. Если t — константа, жслемый вывод очевиден, поскольку множество констант и их типов не зависит от контекста. В случае терма t , имеющего вид комбинации термов s и u , для некоторого типа τ выполняется $\Gamma \vdash s : \tau \rightarrow \sigma$ и $\Gamma \vdash u : \tau$. Согласно индуктивному предположению, $\Delta \vdash s : \tau \rightarrow \sigma$ и $\Delta \vdash u : \tau$, откуда также получаем требуемое. Наконец, если терм t представляет собой абстракцию $\lambda x. s$, то согласно последнему правилу типизации σ имеет вид $\tau \rightarrow \tau'$, а также справедливо, что $\Gamma \cup \{x : \tau\} \vdash s : \tau'$. Так как $\Gamma \subseteq \Delta$, мы получаем $\Gamma \cup \{x : \tau\} \subseteq \Delta \cup \{x : \tau\}$, откуда по индуктивному предположению $\Delta \cup \{x : \tau\} \vdash s : \tau'$. Применяя правило типизации абстракций, получаем требуемое. \square



Типизированное лямбда исчисление

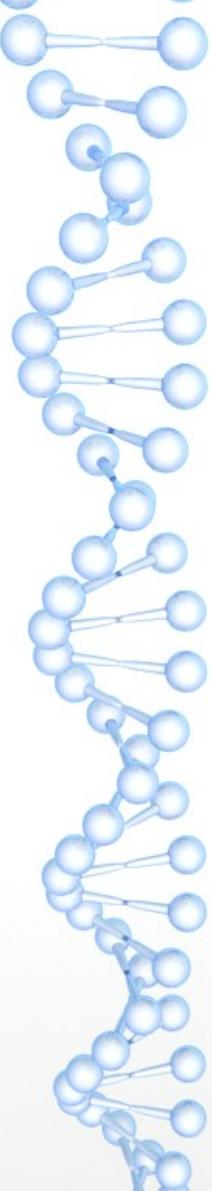
Во-вторых, элементы контекста, представляющие переменные, которые не являются свободными в заданном терме, могут игнорироваться.

Лемма 4.2 *Если $\Gamma \vdash t : \sigma$, то справедливо также $\Gamma_t \vdash t : \sigma$, где Γ_t содержит исключительно свободные переменные терма t ($\Gamma_t = \{x : \alpha \mid x : \alpha \in \Gamma \text{ и } x \in FV(t)\}$).*



Типизированное лямбда исчисление

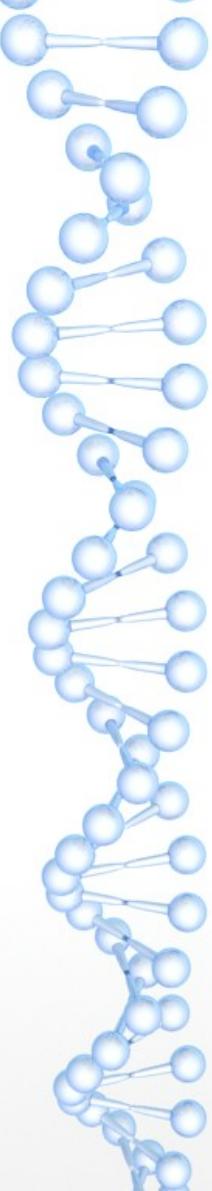
Доказательство: Аналогично предыдущей лемме, докажем наше утверждение для произвольного контекста Γ и соответствующего ему Γ_t путём структурной индукции по t . Если t — переменная, то $\Gamma \vdash t : \sigma$ требует наличия в контексте элемента $x : \sigma$. Согласно первому правилу типизации $\{x : \sigma\} \vdash x : \sigma$, что и требуется. Тип константы не зависит от контекста, так что лемма справедлива и в этом случае. Если терм t представляет собой комбинацию термов вида $s u$, то для некоторого τ справедливо $\Gamma \vdash s : \tau \rightarrow \sigma$ и $\Gamma \vdash u : \tau$. Согласно индуктивному предположению, $\Gamma_s \vdash s : \tau \rightarrow \sigma$ и $\Gamma_u \vdash u : \tau$. Согласно лемме о монотонности, получаем $\Gamma_{su} \vdash s : \tau \rightarrow \sigma$ и $\Gamma_{su} \vdash u : \tau$, поскольку $FV(s u) = FV(s) \cup FV(u)$. Применив правило вывода типа комбинации термов, получаем $\Gamma_{su} \vdash t : \sigma$. Наконец, если t имеет вид $\lambda x. s$, это подразумевает $\Gamma \cup \{x : \tau\} \vdash s : \tau'$, где σ имеет форму $\tau \rightarrow \tau'$. Согласно индуктивному предположению, $(\Gamma \cup \{x : \tau\})_s \vdash s : \tau'$, откуда $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \vdash (\lambda x. s) : \sigma$. Теперь нам требуется лишь отметить, что $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \subseteq \Gamma_t$ и ещё раз применить лемму о монотонности. \square



Типизированное лямбда исчисление

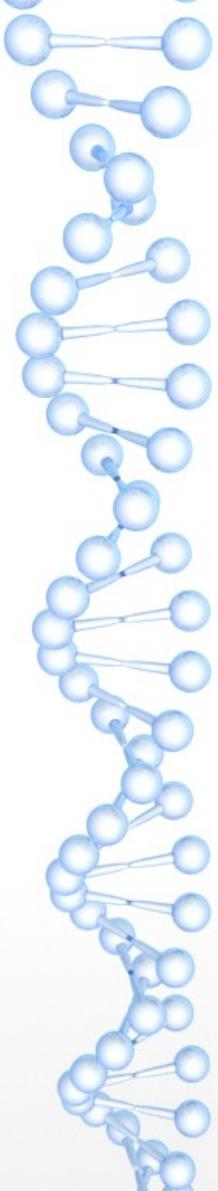
Теорема 4.3 (О сохранении типа) *Если $\Gamma \vdash t : \sigma$ и $t \xrightarrow{\eta} t'$, то из этого следует, что $\Gamma \vdash t' : \sigma$.*

Доказательство: Поскольку по условию теоремы терм t является η -редексом, он должен иметь структуру $(\lambda x. t) x$, причём $x \notin FV(t)$. Следовательно, его тип может быть выведен лишь из последнего правила типизации, при этом σ имеет вид $\tau \rightarrow \tau'$, и справедливо $\{x : \tau\} \vdash (t x) : \tau'$. Дальнейший анализ требует применения правила вывода типа комбинаций. Поскольку контекст может содержать не более одного утверждения о типе каждой переменной, справедливо $\{x : \tau\} \vdash t : \tau \rightarrow \tau'$. Так как по условию $x \notin FV(t)$, то применив лемму 4.2, получаем $\vdash t : \tau \rightarrow \tau'$, что и требовалось. \square



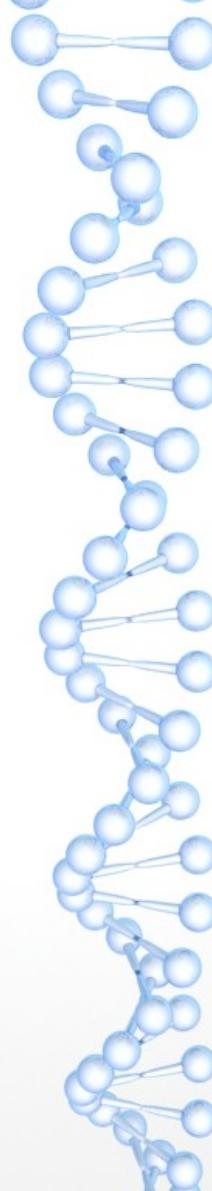
Типизированное лямбда исчисление

Собрав воедино результаты аналогичных доказательств для других преобразований, получаем, что если $\Gamma \vdash t : \sigma$ и $t \longrightarrow t'$, то выполняется также $\Gamma \vdash t' : \sigma$. Важность этого вывода в том, что если бы правила вычислений, применяемые в ходе исполнения программы, могли изменять типы выражений, это подорвало бы основы статической типизации.



Полиморфизм

Типизация по Карри предоставляет в наше распоряжение разновидность *полиморфизма*, позволяя назначить заданному терму различные типы. Следует различать схожие понятия полиморфизма и *перегрузки*. Оба они подразумевают, что выражение может иметь множество типов. Однако, в случае полиморфизма все эти типы структурно связаны друг с другом, так что допустимы любые из них, удовлетворяющие заданному образцу. Например, тождественной функции можно назначить тип $\sigma \rightarrow \sigma$, или $\tau \rightarrow \tau$, либо даже $(\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$, но все они имеют одинаковую структуру. С другой стороны, суть перегрузки в том, что заданная функция может иметь различные типы, структура которых может различаться, либо же допустимо лишь ограниченное множество типов. Например функции `+` может быть позволено иметь тип $int \rightarrow int \rightarrow int$ либо $float \rightarrow float \rightarrow float$, но не $bool \rightarrow bool \rightarrow bool$.⁶ Ещё одним близким понятием являются подтипы, представляющие собой более жёсткую форму перегрузки. Введение подтипов позволяет трактовать некоторый тип как подмножество другого. Однако, этот подход на практике оказывается куда сложнее, чем кажется на первый взгляд.⁷



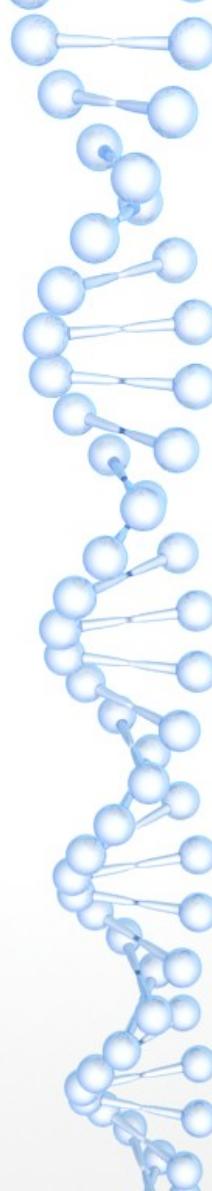
Полиморфизм

К сожалению, определённая выше система типов накладывает некоторые нежелательные ограничения на полиморфизм. Например, следующее выражение абсолютно корректно:

$$\text{if } (\lambda x. x) \text{ true then } (\lambda x. x) 1 \text{ else } 0$$

Докажем, что это выражение может быть типизировано согласно нашим правилам. Предположим, что константам можно назначить типы в пустом контексте, и что мы можем двукратно применить правило типизации комбинации термов для назначения типа `if` (принимая во внимание, что выражение вида `if b then t1 else t2` является всего лишь сокращённой записью для `COND b t1 t2`).

$$\frac{\frac{\frac{\{x : \text{bool}\} \vdash x : \text{bool}}{\vdash (\lambda x. x) : \text{bool} \rightarrow \text{bool}} \quad \frac{\{x : \text{bool}\} \vdash \text{true} : \text{bool}}{\vdash \text{true} : \text{bool}}}{\vdash (\lambda x. x) \text{ true} : \text{bool}} \quad \frac{\frac{\{x : \text{int}\} \vdash x : \text{int}}{\vdash (\lambda x. x) : \text{int} \rightarrow \text{int}} \quad \frac{}{\vdash 1 : \text{int}}}{\vdash (\lambda x. x) 1 : \text{int}}}{\vdash \text{if } (\lambda x. x) \text{ true then } (\lambda x. x) 1 \text{ else } 0 : \text{int}} \quad \frac{}{\vdash 0 : \text{int}}$$



Полиморфизм

Два экземпляра тождественной функции получают типы $\text{bool} \rightarrow \text{bool}$ и $\text{int} \rightarrow \text{int}$ соответственно. Далее рассмотрим другое выражение:

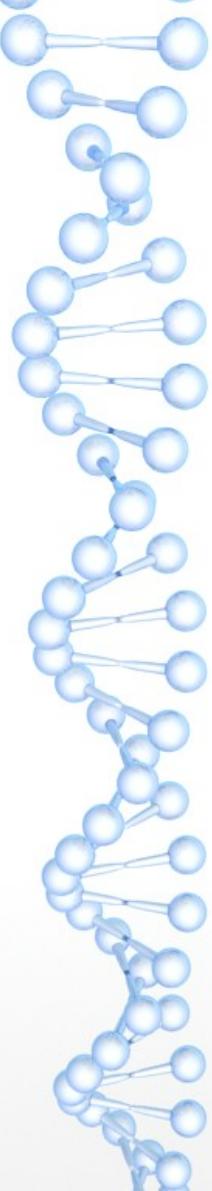
$$\text{let } I = \lambda x. x \text{ in if } I \text{ true then } I \ 1 \text{ else } 0$$

Согласно нашим определениям, это всего лишь удобный способ записи для

$$(\lambda I. \text{if } I \text{ true then } I \ 1 \text{ else } 0) (\lambda x. x)$$

Нетрудно убедиться, что тип этого выражения не может быть выведен в рамках наших правил. Мы имеем *единственный* экземпляр тождественной функции, которому должны назначить единственный тип. Подобное ограничение на практике неприемлемо, поскольку функциональное программирование предполагает частое использование `let`. Если правила типизации не будут изменены, многие выгоды полиморфизма окажутся потерянными. Нашим решением будет отказ от трактовки конструкции `let` как сокращённой записи в пользу реализации её как примитива языка, после чего ко множеству правил типизации следует добавить новое правило:

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$



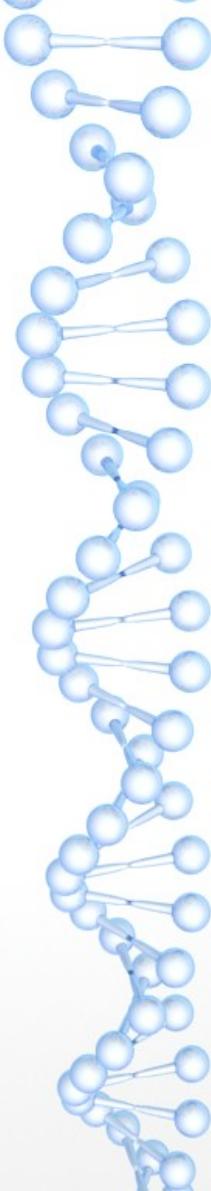
Полиморфизм

Это правило, которым вводится понятие *let-полиморфизма*, демонстрирует, что по крайней мере с точки зрения типизации, *let*-связанные переменные трактуются как простые подстановки соответствующих выражений вместо их имён. Дополнительная посылка $\Gamma \vdash s : \sigma$ требуется исключительно для того, чтобы гарантировать существование корректного типа выражения s , причём точное значение этого типа нас не интересует. Цель данного ограничения в том, чтобы избежать ошибочных выводов о существовании корректных типов для таких термов, как

$$\text{let } x = \lambda f. f f \text{ in } 0$$

Теперь мы в состоянии вывести тип нашего проблемного выражения, пользуясь приведёнными выше правилами:

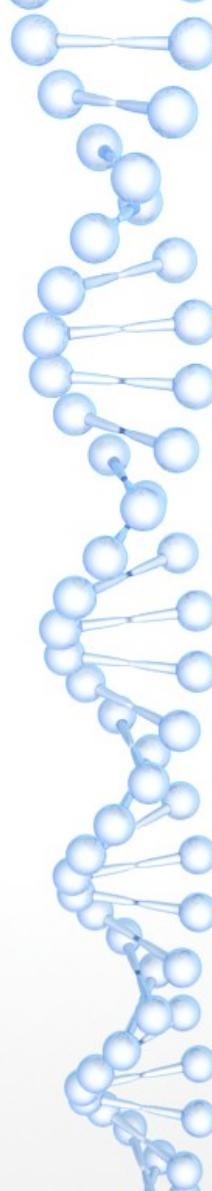
$$\frac{\frac{\frac{\{x : \sigma\} \vdash x : \sigma}{\vdash \lambda x. x : \sigma \rightarrow \sigma} \quad \text{См. выше}}{\vdash \text{if } (\lambda x. x) \text{ true then } (\lambda x. x) 1 \text{ else } 0 : \text{int}}}{\vdash \text{let } I = \lambda x. x \text{ in if } I \text{ true then } I 1 \text{ else } 0 : \text{int}}$$



Нормализация

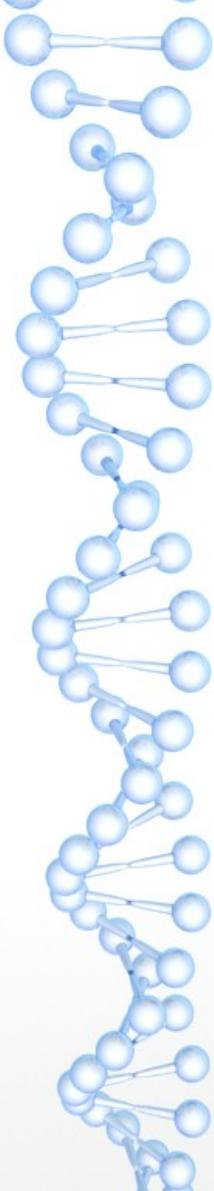
Теорема 4.5 (О сильной нормализации) Любой типизируемый терм имеет нормальную форму, а любая возможная последовательность редукций, которая начинается с типизируемого терма, завершается.¹⁰

На первый взгляд преимущества очевидны — функциональная программа, удовлетворяющая нашей дисциплине типов, может вычисляться в произвольном порядке, при этом процесс редукции всегда конечен и приводит к единственной нормальной форме. (Единственность следует из теоремы Чёрча-Россера, которая остаётся справедливой и в случае типизированного лямбда-исчисления.) Однако, возможность реализации незавершимых функций необходима для обеспечения Тьюринг-полноты,¹¹ в противном случае мы более не в состоянии определить произвольные вычислимые функции, более того — даже не всё множество всюду определённых функций.



Нормализация

Мы бы могли пренебречь этим ограничением, если бы оно позволяло нам использовать все функции, представляющие практический интерес. Однако, это не так — класс всевозможных функций, представимых в рамках типизированного лямбда-исчисления, оказывается весьма узким. Швихтенберг показал, что класс представимых функций на основе нумералов Чёрча ограничен всевозможными полиномами либо кусочными функциями на их основе [58]. Отметим, что этот результат имеет сугубо интенциональную природу, то есть, определяется свойствами заданного представления чисел, а выбор другого представления ведет к другому классу функций. В любом случае, для универсального языка программирования этого недостаточно.



Нормализация

Поскольку все определимые функции являются всюду определёнными, мы, очевидно, не в состоянии давать произвольные рекурсивные определения. В самом деле, оказывается, что обычные комбинаторы неподвижной точки не поддаются типизации; очевидно, что тип $Y = \lambda f. (\lambda x. f(x\ x))(\lambda x. f(x\ x))$ не существует, поскольку x применяется к самому себе, будучи связанным лямбда-абстракцией. Для восстановления Тьюринг-полноты введём альтернативный способ задания произвольных рекурсивных функций, не принося в жертву типизацию. Определим полиморфный оператор рекурсии, всевозможные типы которого имеют вид

$$Rec : ((\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)) \rightarrow \sigma \rightarrow \tau$$

и дополнительное правило редукции, согласно которому для произвольной функции $F : (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$ мы имеем

$$Rec\ F \longrightarrow F\ (Rec\ F)$$

Начиная с этого момента будем полагать, что рекурсивные определения вида `let rec` отображаются на эти операторы рекурсии.