Программирование с зависимыми типами в Agda

1. Введение

В функциональных языках программирования, использующих в качестве системы типов некоторый вариант системы Хиндли-Милнера (Hindley-Milner), существует ярко выраженная граница между *типами* и *значениями*. В языках с *зависимыми* системами типов *(далее просто «зависимо типизированными», по аналогии с «dependently typed» — прим. пер.)* такого чёткого деления нет — типы могут содержать в себе (зависеть от) произвольные значения и фигурировать в качестве аргументов или результатов обычных функций.

Классическим примером зависимого типа является список фиксированной длинны: |Vec A n|, где |A| — тип элементов, а |n| — длина.[1] Многие языки программирования позволяют определять списки (массивы) известного размера, но «зависимость» типа |Vec| заключается в том, что его длина может быть произвольным термом. При этом вовсе не обязательно, чтобы значение этого терма было известно на этапе компиляции.

Зависимые типы дают возможность типам рассуждать о значениях, или, иначе — кодировать свойства значений в виде типов. При этом элементами таких типов являются доказательства того, что закодированное типом свойство выполняется. Другими словами, язык с зависимой системой типов может быть использован в качестве формальной логики.[2] Для того, чтобы эта формальная логика была непротиворечивой, необходимо, чтобы программы на этом языке были *полными*, другими словами, они не имеют права аварийно завершаться или зацикливаться.

Структура настоящей работы такова: в разделе 2 описывается зависимо типизированный язык программирования Agda и его основные особенности, в разделе **??** рассматриваются несколько приёмов программирования, доступных благодаря использованию зависимых типов.

2. Основы Agda

Agda — зависимо типизированный язык программирования, основанный на интуиционистской теории типов, разрабатываемый в университете Chalmers в Гётеборг.[3] В данном разделе описываются основные особенности языка Agda и их использование в построении зависимо типизированных программ. Информация о загрузке и установке системы Agda, а также подробности рассматриваемых в этом разделе вопросов доступны на сайте Agda wiki [**?**].

Текущий раздел представляет собой «грамотную» («литературную», «literate») программу, которую можно скомпилировать непосредственно при помощи системы Agda.

Начнём с самого начала. Каждый файл с исходным кодом на языке Agda начинается с объявления модуля верхнего уровня, имя которого совпадает с именем файла. В данном случае, файл называется `AgdaBasics.lagda`[4].

```
module AgdaBasics where
```

Основная часть программы находится внутри этого модуля верхнего уровня. Для начала определим несколько простых типов данных и функций.

[1] |Vec| — тип, |A| — тип, |n| — натуральное число, то есть значение. Следует отметить, что в современной западной математической традиции *натуральными* числами принято называть те объекты, которые в русской математической традиции принято называть *неотрицательными целыми* числами. Иначе говоря, натуральные числа начинаются с нуля. — прим. пер.

[2] И системы проверки доказательств (proof checker). — прим. пер.

[3] Юго-запад Швеции. — прим. пер.

[4] Файлы с грамотными программами на Agda имеют расширение `lagda`, а файлы с обычными программами – `agda`.

2.1. Типы данных и сопоставление с образцом

Аналогично языкам Haskell и ML, основной концепцией Agda является повсеместное использование механизма сопоставления с образцом по алгебраическим типам данных. При этом, присутствие в языке зависимых типов делает этот механизм ещё более мощным инструментом, как показано в разделах 2.4 и **??**. Однако, перед тем, как обратиться к этим вопросам, рассмотрим способы представления просто типизированных функций и типов данных, доступные в языке.

Алгебраические типы данных определяются при помощи синтаксической конструкции `data`, задающей имя и тип определяемого типа данных, а также имена и типы всего его конструкторов. Например, определение типа булевских значений имеет вид:

```
data Bool : Set where
  true : Bool
  false : Bool
```

При этом тип Bool имеет тип Set — тип малых типов[5]. Функции над типом Bool могут быть определены при помощи сопоставления с образцом при помощи очень знакомого программистам на Haskell синтаксиса:

```
¬ : Bool → Bool
¬ true = false
¬ false = true
```

Функциям языка Agda запрещено аварийно завершаться, поэтому определение должно рассматривать все возможные *(и при этом корректные в отношении типизации (см. раздел 2.4) — прим. пер.)* варианты комбинаций входных параметров. Соответствие функции данному требованию проверяется компилятором, а в случае, если один или более вариантов определением функции не рассматривается, порождается ошибка компиляции.

В Haskell и ML тип функции ¬ может быть выведен из её тела, а потому использование явных типовых сигнатур не обязательно. В языке с зависимыми типами вывод типа терма (в общем случае) является неразрешимой задачей, а потому указание сигнатуры ¬ обязательно. Это не самое страшное ограничение компенсируется тем, что явное выписывание сигнатуры позволяет компилятору не только рано сообщать об ошибках, но быть проводником при интерактивном конструировании программы. С ростом выразительной мощности системы типов диалог между программистом и компилятором становится всё более и более интересным.

Следующим полезным типом данных является тип (унарных) натуральных чисел:

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
```

Сложение на таких числах определяется рекурсивно:

```
_+_ : Nat → Nat → Nat
zero + m = m
suc n + m = suc (n + m)
```

Кроме ограничения на рассмотрение всех вариантов входных данных к функциям языка Agda также предъявляется требование о порождении результата за конечное время. Для этого все рекурсивные вызовы должны производится со структурно убывающими аргументами. В примере выше

[5] В языке Agda используется возрастающая иерархия типов. Типом Set является Set1, чьим типом является Set2 и так далее.

_+_ проходит проверку на завершаемость поскольку первый аргумент рекурсивного вызова убывает (n < suc n). Теперь определим умножение через сложение.

```
_*_ : Nat → Nat → Nat
zero * m = zero
suc n * m = m + n * m
```

Agda поддерживает механизм так называемых *mixfix* операторов. При использовании в имени функции нижних подчёркиваний (_) при её вызове можно использовать операторный синтаксис. При этом аргументы должны располагаться там, где в имени функции располагаются подчёркивания. Иначе говоря, например, терм n + m, является альтернативной синтаксической формой обычного применения функции к аргументам _+_ n m.

Не существует (практически) никаких ограничений на символы, которые разрешено использовать в именах операторов. Например, можно определить

```
_and_ : Bool → Bool → Bool
true and x = x
false and _ = false
if_then_else_ : {A : Set} → Bool → A → A → A
if true then x else y = x
if false then x else y = y
```

Нижнее подчёркивание во втором *предложении* определения функции _and_[6] является символом-джокером и сообщает компилятору, что функцию абсолютно не интересует значение этого аргумента, а потому не стоит заставлять пользователя придумывать ему имя. Конечно, это также означает, что к этому аргументу нельзя будет обратиться из правой части предложения.

Приоритет и ассоциативность операторов могут быть заданы при помощи конструкций семейства **infix**:

```
infixl 60 _*_
infixl 40 _+_
infixr 20 _and_
infix 5 if_then_else_
```

В типе if_then_else_ присутствует ещё одна интересная не рассмотренная конструкция: {A : Set} →. На данном этапе достаточно считать, что она объявляет функцию, полиморфную по A. Более подробно эти вопросы рассматриваются в разделах 2.2 и 2.4.

Аналогично Haskell и ML типы данных в Agda могут быть параметризованы другими типами. Например, тип списков элементов произвольного типа определяется следующим образом:

```
infixr 40 _::_
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

Следует снова отметить, что Agda достаточно либерально относится тому, что считается допустимым именем. [] и _::_ — обычные имена. Формально, в Agda допустимым именем является любая строка, состоящая из произвольных непробельных символов таблицы UNICODE за исключением круглых и фигурных скобок. То есть, если захотеть (хотя лучше не стоит), тип списков можно было бы определить как

[6] В оригинале эта функция называется _or_, что, очевидно, опечатка. — прим. пер.

```
data _⋆ (α : Set) : Set where
  ϵ : α ⋆
  _◁_ : α → α ⋆ → α ⋆
```

Эта либеральность по отношению к именам означает, что очень важно щедро расставлять пробелы в исходном коде. Например, not:Bool->Bool не будет являться корректно типовой сигнатурой функции ¬, поскольку это вполне допустимое имя.

2.2. Зависимые функции

Теперь обратимся к зависимым типам. Самым простым зависимым типом является тип функции, результат которой зависит от значения её аргумента: `(x : A) → B`, где `x` — аргумент типа `A`, а `B` — тип результата, при этом `x` может встречаться в терме `B`. В частности, само `x` может быть типом. Например, можно определить следующие термы:

```
identity : (A : Set) → A → A
identity A x = x
zero' : Nat
zero' = identity Nat zero
```

Первый из них — зависимая функция, принимающая тип в качестве первого аргумента `A`, элемент типа `A` в качестве второго аргумента и возвращающая этот элемент. Полиморфные функции в Agda определяются при помощи таких конструкций. Приведём более замысловатый пример зависимой функции — функцию, принимающую зависимую функцию и применяющую её к аргументу:

```
apply : (A : Set) (B : A → Set) →
  ((x : A) → B x) → (a : A) → B a
apply A B f a = f a
```

При наличии скобок вокруг аргументов компилятор может самостоятельно однозначно расставить некоторые опущенные стрелки. Поддерживаются следующие сокращения:

- `(x : A) (y : B) → C` вместо `(x : A) → (y : B) → C`;
- `(x y : A) → B` вместо `(x : A) (y : A) → B`.

Как и в Haskell, функции в Agda можно определять при помощи лямбда-нотации. При этом ти́повые аннотации переменных можно указывать явно. Приведём несколько альтернативных методов определения функции identity:

```
identity₂ : (A : Set) → A → A
identity₂ = \A x → x
identity₃ : (A : Set) → A → A
identity₃ = λ (A : Set) (x : A) → x
identity₄ : (A : Set) → A → A
identity₄ = λ (A : Set) x → x
```

Вместо символа λ можно также использовать символ λ таблицы UNICODE.

2.3. Неявные аргументы

В предыдущем разделе было показано как при помощи зависимых функций, принимающих типы в качестве аргументов, моделировать полиморфные типы. Вся прелесть полиморфных функций заключается в том, что им не нужно сообщать тип аргументов к которым их хотят применить, поскольку этот тип может быть автоматически выведен компилятором. Тем не менее, в примерах определения функции identity и её использования zero приходилось явно указывать тип аргумента. В Agda эта проблема решается при помощи механизма *неявных аргументов*. Для того, чтобы объявить аргумент функции неявным достаточно вместо круглых скобок окружить его фигурными: тип $\{x : A\} \to B$ означает тоже самое, что и $(x : A) \to B$, за тем исключением, что компилятор попытается самостоятельно вывести значение аргумента x при каждом использовании такой функции.

Определим новую версию функции identity, которая не требует явного указания типа своего аргумента, с использование рассмотренного синтаксиса:

```
id : {A : Set} → A → A
id x = x

true' : Bool
true' = id true
```

Заметим, что типовый аргумент неявен и при объявлении *(отсутствует в левой части предложения — прим. пер.)*, и при использовании функции id. Не существует каких-либо ограничений на то, какие аргументы можно делать неявными, и не существует каких-либо гарантий того, что неявный аргумент будет выведен компилятором. Например, глупо было бы сделать неявным второй аргумент функции identity:

```
silly : {A : Set} {x : A} → A
silly {_} {x} = x

false' : Bool
false' = silly {x = false}
```

Компилятор не способен самостоятельно угадать значение второго аргумента функции silly. Для того, чтобы явно указать значение неявного аргумента следует использовать синтаксические конструкции f {v} (при этом v будет передано в качестве первого из неявных аргументов функции f) и f {x = v} (при этом v будет передано в качестве значения неявного аргумента с именем x). При использовании синтаксиса f {x = v} имя неявного аргумента (x) определяется сигнатурой функции, а не её телом:

```
silly : {A : Set} {x : A} → A
silly {_} {y} = y

false : Bool
false = silly {x = false}  -- OK
  -- false``bad : Bool
  -- false``bad = silly` y = false – Ошибка
```

Кроме явного указания неявности аргумента существует также возможность попросить компилятор попытаться вывести терм, который следовало бы указать явно. Для этой цели в сигнатуре функции или справа от знака равенства в её теле следует поставить знак подчёркивания[7].

```
one : Nat
one = identity _ (suc zero)
```

[7]Напомним, что подчёркивание в левой части тела означает игнорирование аргумента

Важно отметить, что компилятор не будет производить какого-либо поиска, пытаясь угадать неявные аргументы и термы, поставляемые вместо подчёркиваний. Им просто будут выписаны все типовые ограничения и выполнена унификация[8].

Тем не менее, это не мешает немалому количеству термов быть выведенным автоматически. Например, можно определить максимально абстрактную зависимую функциональную композицию. (Осторожно: типовая сигнатура не для слабых сердцем!)

```
_∘_ : {A : Set} {B : A → Set} {C : (x : A) → B x → Set}
  (f : {x : A} (y : B x) → C x y) (g : (x : A) → B x)
  (x : A) → C x (g x)
(f ∘ g) x = f (g x)

plus-two = suc ∘ suc
```

На примере plus-two можно видеть, что компилятор способен вывести типовые аргументы A, B и C при использовании _ ∘ _.

Выше были рассмотрены способы определения просто типизированных типов данных и функций, а также использование зависимых типов и неявных аргументов для реализации полиморфных функций. Завершим данный подраздел, определив несколько хорошо знакомых часто используемых функций, работающих со списками.

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x :: xs) = f x :: map f xs

_++_ : {A : Set} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

2.4. Семейства типов данных

В предыдущих подразделах зависимые типы применялись только для моделирования полиморфизма, в данном подразделе рассматривается также несколько новых любопытных примеров их использования.

Тип списков известной длинны, упоминавшийся во введении, может быть определён следующим образом:

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Первая строка данного определения говорит компилятору, что Vec A имеет тип Nat → Set. Это означает, что каждый элемент типа Vec A представляет собой семейство типов, индексированное натуральными числами. Другими словами, для каждого натурального n выражение Vec A n является типом. При этом конструкторам разрешается конструировать элементы, принадлежащие любому из типов данного семейства. В частности, [] конструирует элемент из Vec A zero, а _::_ — из Vec A (suc n) для некоторого n.

У зависимых типов данных выделяют *параметры* и *индексы*. Говорят, что Vec параметризован типом A и индексирован натуральными числами.

[8] Miller pattern unification, если точнее

Тип конструктора _::_ является очередным примером типа зависимой функции. Первый аргумент _::_ представляет собой неявное натуральное число n, показывающее длину хвоста списка, передаваемого в качестве третьего аргумента. При этом неявность n не вызывает проблем, поскольку компилятор может вывести его значение из типа хвоста.

Отметим также, что, в отличие от Haskell, в Agda не требуется, чтобы множества имён конструкторов различных типов данных не пересекались. Например, имена конструкторов типов Vec и List совпадают.

Интересные свойства семейств типов данных проявляются, когда их по элементам производится сопоставление с образцом. Предположим, например, что мы хотим получить голову непустого списка. При помощи Vec мы можем выразить тип непустых списков — Vec A (suc n) для любых A и n. Тогда определение функции head принимает вид:

```
head : {A : Set} {n : Nat} → Vec A (suc n) → A
head (x :: xs) = x
```

При этом, приведённое определение допускается компилятором как полное, не смотря на то, что в определении отсутствует предложение для варианта []. Это происходит потому, что [] не подходит по типу. Единственный способ построить элемент типа Vec A (suc n) — конструктор _::_.

Правило, определяющее нужно ли рассматривать некоторый вариант входных данных, звучит очень просто: *если у варианта корректный тип, то он должен быть рассмотрен.*

2.4.1. Пунктирные образцы

Рассмотрим следующую функцию над типом Vec:

```
vmap : {A B : Set} {n : Nat} → (A → B) → Vec A n → Vec B n
vmap f [] = []
vmap f (x :: xs) = f x :: vmap f xs
```

Удивительно, но определения map для List и vmap для Vec различаются только в сигнатурах, полностью совпадая телами. Однако это не мешает чему-то любопытному происходить за кулисами vmap. Интересным вопросом является, например, то, как изменяется длина аргумента после сопоставления с образцом. Для того, чтобы это пронаблюдать придётся определить альтернативные версии Vec и vmap с меньшим количеством неявных аргументов:

```
data Vec (A : Set) : Nat → Set where
  nil : Vec A zero
  cons : (n : Nat) → A → Vec A n → Vec A (suc n)
vmap₂ : {A B : Set} (n : Nat) → (A → B) → Vec A n → Vec B n
vmap₂ .zero f nil = nil
vmap₂ . (suc n) f (cons n x xs) = cons n (f x) (vmap₂ n f xs)
```

При сопоставлении с образцом по аргументу со списком становится известна длина его хвоста: если в качестве конструктора там nil, то длина должна быть zero, если же там cons n x xs, то единственное корректное значение длины — suc n для какого-то n. Для того, чтобы сообщить компилятору, что значение было получено выводом типов, а не сопоставлением с образцом, его следует *пунктировать*, предварив его точкой (.).

В принципе, мы могли бы сопоставляться с образцом по длине, а не по списку. В таком случае пунктировать пришлось бы длину хвоста в аргументе конструктора cons[9]:

[9]Вообще говоря, точку можно поставить перед любым из n. Важно то, что для каждой переменной в левой части предложения должно существовать уникальное непунктирное связывание.

```
vmap₃ : {A B : Set} (n : Nat) → (A → B) → Vec A n → Vec B n
vmap₃ zero f nil = nil
vmap₃ (suc n) f (cons .n x xs) = cons n (f x) (vmap₃ n f xs)
```

Правило, определяющее нужно ли пунктировать аргумент, следующее: *если для аргумента существует единственное возможное значение с корректным типом, то аргумент должен быть пунктирным.*

В рассмотренном выше примере, пунктирные выражения были корректными образцами для сопоставления (простыми связываниями переменных или распакованными конструкторами типов данных), однако, в общем случае, они могут быть произвольными термами. Например, определим образ функции следующим образом:

```
data Image_∋_ {A B : Set} (f : A → B) : B → Set where
  im : (x : A) → Image f ∋ f x
```

Данный терм утверждает, что единственным способом получить элемент в образе функции f является применение этой функции к некоторому заранее выбранному x. Теперь, если известно, что некоторое y является элементом образа f, мы можем извлечь x из y, тем самым вычислив значения функции обратной к f на y:

```
inv : {A B : Set} (f : A → B) (y : B) → Image f ∋ y → A
inv f .(f x) (im x) = x
```

Данное выражение кодирует утверждение о том, что единственным способом получить элемент в образе функции f является применение этой функции к некоторому x. При этом f x не является корректным образцом для сопоставления поскольку f не является конструктором типа данных, однако это не мешает использовать пунктирное f x в левой части определения функции.

2.4.2. Абсурдные образцы

Определим семейство типов данных, кодирующее числа меньшие заданного натурального числа.

```
data Fin : Nat → Set where
  fzero : {n : Nat} → Fin (suc n)
  fsuc : {n : Nat} → (i : Fin n) → Fin (suc n)
```

Элементы, сконструированные fzero, принадлежат всем типам Fin n для $n \geqslant 1$ (suc n для любого n). Иначе говоря, конструктор fzero утверждает, что «ноль меньше любого натурального числа кроме самого нуля». Конструктор fsuc утверждает: «если некоторое i меньше n, то i + 1 (fsuc i) меньше n + 1 (suc n)». Отметим, что в семействе отсутствую элементы индексированные zero, иначе говоря: «не существует натуральных чисел меньших нуля».

В случае, когда у аргумента нет доступных конструкторов, производить сопоставление с образцом следует при помощи *абсурдного образца* ():

```
magic : {A : Set} → Fin zero → A
magic ()
```

При использование абсурдного образца правая часть предложения всегда отсутствует, поскольку не существует способа сконструировать элемент, который можно было бы подставить вместо аргумента. На первый взгляд кажется, что можно было бы научить компилятор вообще не требовать от

пользователя явного выписывания таких предложений, однако, как отмечалось ранее, в Agda предложение можно опустить только в том случае, если не существует корректно типизированного способа его записать. В данном же случае, вполне корректной левой частью предложения являлось бы magic x.

Важно отметить, что использование абсурдного образца допустимо только в тех случаях, когда у рассматриваемого аргумента не существует подходящих по типу конструкторов. Отсутствие у типа замкнутых элементов недостаточно.[10] Например, для определения

```
data Empty : Set where
  empty : Fin zero → Empty
```

аргументы типа Empty не могут быть сопоставлены с абсурдным образцом, поскольку существует корректный конструктор empty x. Для того, чтобы определить аналог функции magic для типа Empty необходимо использовать запись вида:

```
magic' : {A : Set} → Empty → A
magic' (empty ())
  -- magic' () – не допускается компилятором
```

Определим несколько любопытных функций, используя рассмотренный выше инструментарий. Имея список длины n и число i, меньше чем n, можно вычислить i-тый элемент списка (отсчитывая с нуля):

```
_!_ : {n : Nat} {A : Set} → Vec A n → Fin n → A
[] ! ()
(x :: xs) ! fzero = x
(x :: xs) ! (fsuc i) = xs ! i
```

Отсутствие выхода за границы массива в определении данной функции обеспечивается системой типов, поскольку в случае пустого списка не существует возможных конструкторов у аргумента с индексом.

Функция _!_ превращает список в функцию из индексов в элементы. Существует функция, действующая в обратном направлении. Эта функция строит список элементов из функции, возвращающей элемент по его индексу.

```
tabulate : {n : Nat} {A : Set} → (Fin n → A) → Vec A n
tabulate {zero} f = []
tabulate {suc n} f = f fzero :: tabulate (f ∘ fsuc)
```

Следует отметить, что tabulate определена при помощи рекурсии по длиннее результирующего списка не смотря на то, что эта длинна является неявным аргументом. Этот пример демонстрирует тот факт, что, в общем случае, не существует взаимосвязи между неявными данными и данными, не требующимися при вычислении.

2.5. Программы как доказательства

Во введении упоминалось, что система типов Agda достаточно выразительна для представления (почти) любых утверждений в виде типов, чьими элементами являются доказательства этих утверждений. Два самых простых утверждения — это тождественно истинное и ложное высказывания:

[10] Поскольку проблема определения населённости типа неразрешима.

```
data False : Set where
record True : Set where
trivial : True
trivial = _
```

Тождественно ложное высказывание представлено типом данных без конструкторов, а тождественно истинное высказывание — структурным типом без полей (см. раздел ??). Структурный тип без полей содержит ровно один элемент — пустую структуру. Можно было бы определить True в виде обычного типа данных с одним элементом, но компилятор Agda «знает», что в типе пустой структуры содержится ровно один элемент и умеет подставлять его вместо неявных аргументов типа True. Определение trivial использует это свойство — правая часть единственного предложения в теле функции содержит только подчёркивание. Для того, чтобы выписать элемент True явно используется синтаксис **record** { }.

Первых двух определений из листинга выше достаточно для того, чтобы работать с разрешимыми утверждениями. Последние можно смоделировать при помощи функции

```
isTrue : Bool → Set
isTrue true = True
isTrue false = False
```

При этом тип isTrue b является типом доказательств того, что b равняется true. Используя эти определения можно реализовать функцию, возвращающую элемент списка по его индексу и использующую только просто типизированные списки и числа, следующим образом:

```
_<_ : Nat → Nat → Bool
_ < zero = false
zero < suc n = true
suc m < suc n = m < n
length : {A : Set} → List A → Nat
length [] = zero
length (x :: xs) = suc (length xs)
lookup : {A : Set} (xs : List A) (n : Nat) →
  isTrue (n < length xs) → A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (suc n) p = lookup xs n p
```

Здесь вместо использования того факта, что не существуют индекса для пустого списка, используется отсутствие доказательства того, что число n меньше zero.

Отметим, что для функции индексирования по списку использование индексированных типов для представления предусловия на аргументах оказывается удобнее, поскольку при этом не требуется явно передавать объект-доказательство. Однако некоторые свойства не могут быть просто закодированы в виде индексированных типов, в таком случае использование isTrue является хорошей альтернативой.

Для представления утверждений можно использовать и семейства типов данных. Рассмотрим, например, отношение эквивалентности:

```
data _==_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Для типа A и элемента x типа A, данный тип представляет собой семейство доказательств «быть эквивалентным x». Это семейство населено единственным элементом-доказательством refl при индексе x.

Другим примером является отношение «меньше либо равно» на натуральных числах. Ранее оно было определено при помощи булевой функции, но его можно определить и индуктивно:

```
data _⩽_ : Nat → Nat → Set where
  leq-zero : {n : Nat} → zero ⩽ n
  leq-suc : {m n : Nat} → m ⩽ n → suc m ⩽ suc n
```

Одним из преимуществ такого представления является то, что сопоставление с образцом по объекту-доказательству упрощает доказательства некоторых свойств этого отношения. Например,

```
leq-trans : {l m n : Nat} → l ⩽ m → m ⩽ n → l ⩽ n
leq-trans leq-zero _ = leq-zero
leq-trans (leq-suc p) (leq-suc q) = leq-suc (leq-trans p q)
```

2.6. Ещё о сопоставлении с образцом

В предыдущих разделах сопоставление с образцом производилось только в аргументах функции, однако иногда требуется сопоставляться с образцом по некоторым промежуточным вычислениям. В Haskell и ML такое сопоставление производится при помощи конструкций case и match. Ранее было показано, что в языках с зависимыми типами сопоставление с образцом может не только сообщать о форме сопоставляемого выражения, но и сообщать некоторую информацию о других выражениях. Например, сопоставление с образцом по выражению типа Vec A n сообщает о форме n. Это свойство никак не отражено в обычной конструкции case, поэтому Agda предоставляет более мощный инструмент, позволяющий производить сопоставление с образцом по промежуточным вычислениям в левой части определения функции.

2.6.1. Конструкция with

Основная идея данного инструмента заключается в том, что в случае, когда требуется произвести сопоставление с образцом по некоторому промежуточному вычислению e в определении функции f, производится абстракция функции f по e. Благодаря этому e становится обычным аргументом f, что позволяет производить сопоставление с образцом по e в обычном стиле. Такая абстракция производится при помощи конструкции **with**. Например,

```
min : Nat → Nat → Nat
min x y with x < y
min x y | true = x
min x y | false = y
```

Определение функции min абстрагируясь при помощи **with** получает дополнительный третий аргумент, отделённый от обычных аргументов вертикальной чертой. При этом значение этого аргумента соответствует значению выражения x < y. В случае необходимости абстрагирования сразу по нескольким выражениям их следует записывать друг за другом, разделяя вертикальными чертами, как после ключевого слова **with**, так и при сопоставлении с образцом в левой части предложений. Кроме того, разрешается делать вложенные абстрагирования.

В предыдущем листинге сопоставление с образцом по x < y не сообщает ничего нового об аргументах min, потому повторять их слева от знака равенства слегка утомительно. В таких случаях в левой части предложений обычные аргументы можно заменять на ...:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
... | true = x :: filter p xs
... | false = filter p xs
```

Приведём несколько более содержательный пример, использующий **with**. Два числа можно сравнить на равенство. Однако, вместо того, чтобы возвращать неинтересное булево значение, можно привести доказательство того, что числа действительно совпадают, или пояснение, показывающее чем они различаются:

```
data _ ≠ _ : Nat → Nat → Set where
  z≠s : {n : Nat} → zero ≠ suc n
  s≠z : {n : Nat} → suc n ≠ zero
  s≠s : {m n : Nat} → m ≠ n → suc m ≠ suc n
data Equal? (n m : Nat) : Set where
  eq : n ≡ m → Equal? n m
  neq : n ≠ m → Equal? n m
```

Два натуральный числа различаются, если одно из них zero, а форма второго начинается с suc или если они оба начинаются с suc, но их числа-предшественники (стоящие под suc) различаются. Определим функцию equal?, сравнивающую два числа:

```
equal? : (n m : Nat) → Equal? n m
equal? zero zero = eq refl
equal? zero (suc m) = neq z≠s
equal? (suc n) zero = neq s≠z
equal? (suc n) (suc m) with equal? n m
equal? (suc n) (suc .n) | eq refl = eq refl
equal? (suc n) (suc m) | neq p = neq (s≠s p)
```

Следует обратить внимание на предложение с обоими числами начинающимися с suc. При совпадении предшественников сопоставление с образцом по объекту-доказательству сообщает нам информацию, приводящую к необходимости пунктирования одного из n. В разделе **??** приводится ещё несколько примеров, использующих информативные типы данных такого рода.

Выражение, абстрагируемое при помощи конструкции **with**, абстрагируется сразу во всём контексте. Иначе говоря, если абстрагируемое выражение встречается в типе аргумента или результата функции, то в каждом предложении, соответствующем этому **with**, каждое вхождение этого выражения будет заменено значением, рассматриваемым слева от знака равенства. Например, рассмотрим доказательство того, что результат функции filter является подсписком её второго аргумента. Для этого сначала определим отношение «быть подсписком»:

```
infix 20 _ ⊆ _
data _ ⊆ _ {A : Set} : List A → List A → Set where
  stop : [] ⊆ []
  drop : forall {xs y ys} → xs ⊆ ys → xs ⊆ y :: ys
  keep : forall {x xs ys} → xs ⊆ ys → x :: xs ⊆ x :: ys
```

Интуиция данного определения заключается в следующем: для того, чтобы получить подсписок некоторого списка каждый элемент первоначального списка нужно оставить на его месте или отбросить.

В тех случаях, когда компилятор сам может вывести тип аргумента в функциональном типе, допустимо использование специального синтаксиса **forall**:

- **forall** {x y} a b → A — сокращение для {x : _} {y : _} (a : _) (b : _) → A.

С использование данного выше определения подсписка лемма «filter вычисляет подсписок своего аргумента» может быть доказана следующим образом:

```
lem-filter : {A : Set} (p : A → Bool) (xs : List A) → filter p xs ⊆ xs
lem-filter p [] = stop
lem-filter p (x :: xs) with p x
... | true = keep (lem-filter p xs)
... | false = drop (lem-filter p xs)
```

Интересным предложением в данном определении является сопоставление с образцом _::_. Рассмотрим этот случай подробно:

```
lem-filter p (x :: xs) = ?
```

Целью доказательства является:

```
(filter p (x :: xs) | p x) ⊆ x :: xs
```

Описание цели (тип рассматриваемого предложения) не может быть средуцировано, поскольку filter применена к нередуцируемому аргументу **with**. Теперь, при абстрагировании по p x, абстрагирование происходит сразу во всём контексте:

```
lem-filter p (x :: xs) with p x
... | px = ?
```

При этом все вхождения p x в описании цели заменяются на px:

```
(filter p (x :: xs) | px) ⊆ x :: xs
```

Теперь, при сопоставлении с образцом по px компилятор получает достаточно информации для того, чтобы средуцировать вызов функции filter:

```
lem-filter p (x :: xs) with p x
... | true = ? {-x :: filter p xs ≡ x :: xs -}
... | false = ? {filter p xs ⊆ x :: xs -}
```

В некоторых случаях бывает полезно использовать **with** для абстрагирования даже тех выражений, по которым не будет производится сопоставления с образцом. В частности, если абстрагируемое выражение будет определено сопоставлением с образцом по некоторым другим выражениям. Например, рассмотрим доказательство n + zero ≡ n:

```
lem-plus-zero : (n : Nat) → n + zero ≡ n
lem-plus-zero zero = refl
lem-plus-zero (suc n) with n + zero | lem-plus-zero n
... | .n | refl = refl
```

На шаге индукции, для того, чтобы доказать suc n + zero = suc n, требуется произвести сопоставление с образцом по гипотезе индукции n + zero ≡ n. Однако это не допускается компилятором, поскольку n + zero не может быть унифицировано с n. С другой стороны, при абстрагировании по n + zero — назовём это выражение m — гипотеза индукции принимает вид m ≡ n, а цель — sum m ≡ suc n. После этого сопоставление с образцом по гипотезе индукции сопоставит m и n.