

Санкт-Петербургский национальный исследовательский университет  
информационных технологий, механики и оптики

Факультет информационных технологий и программирования  
Кафедра компьютерных технологий

Рыбак Андрей Викторович

**Представление структур данных индуктивными  
семействами и доказательства их свойств**

Научный руководитель: Я. М. Малаховски

Санкт-Петербург  
2014

# Содержание

<b>Введение</b>	<b>4</b>
<b>Глава 1. Обзор</b>	<b>5</b>
1.1 Лямбда-исчисление	5
1.2 Функциональное программирование	6
1.3 Алгебраические типы данных и сопоставление с образцом	6
1.4 Теория типов	6
1.4.1 Отношение конвертабельности	7
1.4.2 Интуиционистская теория типов	7
1.5 Унификация	8
1.6 Индуктивные семейства	8
1.7 Agda	9
1.8 Выводы по главе 1	10
<b>Глава 2. Описание реализованной структуры данных</b>	<b>11</b>
2.1 Постановка задачи	11
2.2 Структура данных «двоичная куча»	11
2.3 Реализация	11
2.3.1 Вспомогательные определения	11
2.3.2 Определение отношений и их свойств	13
2.3.3 КучКучаа	18
2.4 Выводы по главе 2	27
<b>Список литературы</b>	<b>28</b>

# Введение

Структуры данных используются в программировании повсеместно для упрощения хранения и обработки данных. Свойства структур данных происходят из инвариантов, которые эта структура данных соблюдает.

Практика показывает, что тривиальные структуры и их инварианты данных хорошо выражаются в форме индуктивных семейств. Мы хотим узнать насколько хорошо эта практика работает и для более сложных структур.

В данной работе рассматривается представление в форме индуктивных семейств структуры данных приоритетная очередь типа «двоичная куча».

# Глава 1. Обзор

В данной главе производится обзор предметной области и даются определения используемых терминов.

## 1.1. ЛЯМБДА-ИСЧИСЛЕНИЕ

*Лямбда-исчисление* ( $\lambda$ -calculus) — вычислительный формализм с тремя синтаксическими конструкциями, называемыми *пре-лямбда-термами*:

- *вхождение переменной*:  $v$ . При этом  $v \in V$ , где  $V$  — некоторое множество имён переменных;
- *лямбда-абстракция*:  $\lambda x.A$ , где  $x$  — имя переменной, а  $A$  — пре-лямбда-терм. При этом терм  $A$  называют *телом абстракции*, а  $x$  перед точкой — *связыванием*.
- *лямбда-аппликация*:  $BC$ ;

и одной операцией *бета-редукции*. При этом говорят, что вхождение переменной является *свободным*, если оно не связано какой-либо абстракцией. *Лямбда-термы* — это пре-лямбда-термы, факторизованные по отношению *альфа-эквивалентности*.

*Альфа-эквивалентность* ( $\alpha$ -equality) отождествляет два пре-лямбда-терма, если один из них может быть получен из другого путём некоторого *корректного* переименовывания переменных — переименования не нарушающего отношение связности.

*Бета-редукция* ( $\beta$ -reduction) для лямбда-терма  $A$  выбирает в нём некоторую лямбда-аппликацию  $BC$ , содержащую лямбда-абстракцию в левой части  $A$ , и заменяет свободные вхождения переменной, связанной  $A$ , в теле самой  $A$  на терм  $C$ .<sup>1</sup>

---

<sup>1</sup>В терминах пре-лямбда-термов это означает замену свободных вхождений в теле  $A$  на пре-терм  $C$  так, чтобы ни для каких переменных не нарушилось отношение связности. То есть, в пре-терме  $A$  следует корректно переименовать все связанные переменные, имена которых совпадают с именами свободных переменных в  $C$ .

Два лямбда-терма  $A$  и  $B$  называются *конвертабельными*, когда существует две последовательности бета-редукций, приводящих их к общему терму  $C$ . Или, эквивалентно, когда термы  $A$  и  $B$  состоят с друг с другом в рефлексивно-симметрично-транзитивном замыкании отношения бета-редукции, также называемом отношением *бета-эквивалентности*.

За более подробной информацией об этом формализме следует обращаться к [1] и [2].

## 1.2. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

*Функциональное программирование* — парадигма программирования, являющаяся разновидностью декларативного программирования, в которой программу представляют в виде функций (математическом смысле этого слова, а не в смысле, используемом в процедурном программировании), а выполнением программы считают вычисление значений применения этих функций к заданным значениям. Большинство функциональных языков программирования используют в своём основании лямбда-исчисление (например, Haskell [3], Curry [4], Agda [5], диалекты LISP [6–8], SML [9], OCaml [10]), но существуют и функциональные языки явно не основанные на этом формализме (например, препроцессор языка C и шаблоны в C++).

## 1.3. АЛГЕБРАИЧЕСКИЕ ТИПЫ ДАННЫХ И СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ

### 1.4. ТЕОРИЯ ТИПОВ

*Теория типов* — какая-либо формальная система, являющаяся альтернативой наивной теории множеств, сопровождаемая классификацией элементов такой системы с помощью типов, образующих некоторую иерархию. Элементы теории типов — выражения, также называемые *термами*. Если терм  $M$  имеет тип  $A$ , то это записывают так:  $M : A$ . Например,  $2 : \mathbb{N}$ .

Теории типов также содержат правила для переписывания термов — замены подтермов формулы другими термами. Такие правила также называют правилами *редукции* или *конверсии* термов. Например, термы  $2 + 1$  и  $3$  — разные термы, но первый редуцируется во второй:  $2 + 1 \rightarrow 3$ . Про терм, который не может быть редуцирован, говорят, что терм — в *нормальной форме*.

#### 1.4.1. Отношение конвертабельности

Два терма называются *конвертабельными*, если существует терм, к которому они оба редуцируются. Например,  $1 + 2$  и  $2 + 1$  — конвертабельны, как и термы  $x + (1 + 1)$  и  $x + 2$ . Однако,  $x + 1$  и  $1 + x$  (где  $x$  — свободная переменная) — не конвертабельны, так как оба представлены в нормальной форме.

#### 1.4.2. Интуиционистская теория типов

Интуиционистская теория типов основана на математическом конструктивизме [11].

Операторы для типов в ИТТ:

- П-тип (пи-тип) — зависимое произведение. Например, если  $\text{Vec}(\mathbb{R}, n)$  — тип кортежей из  $n$  вещественных чисел,  $\mathbb{N}$  — тип натуральных чисел, то  $\prod_{n:\mathbb{N}} \text{Vec}(\mathbb{R}, n)$  — тип функции, которая по натуральному числу  $n$  возвращает кортеж из  $n$  вещественных чисел.
- $\Sigma$ -тип — зависимая сумма (пара). Например, тип  $\sum_{n:\mathbb{N}} \text{Vec}(\mathbb{R}, n)$  — тип пары из числа  $n$  и кортежа из  $n$  вещественных чисел.

Конечные типы в ИТТ:  $\perp$  или  $0$  — пустой тип, не содержащий ни одного элемента;  $\top$  или  $1$  — единичный тип, содержащий единственный элемент. *Тип равенства*: для  $x$  и  $y$  выражение  $x \equiv y$  обозначает тип доказательства равенства  $x$  и  $y$ . То есть, если тип  $x \equiv y$  населен, то  $x$  и  $y$  называются равными. Есть только один каноничный элемент типа  $x \equiv x$  — доказательство рефлексивности:  $refl : \prod_{a:A} a \equiv a$ .

## 1.5. УНИФИКАЦИЯ

Унификация — процесс и алгоритм решения уравнений над выражениями в теории типов. Алгоритм унификации находит подстановку, которая назначает значение каждой переменной в выражении, после применения которой, части уравнения становятся конвертабельными. Пример: равенство двух списков  $cons(x, cons(x, nil)) \equiv cons(2, y)$  — уравнение с двумя переменными  $x$  и  $y$ . Решение: подстановка  $x \mapsto 2, y \mapsto cons(2, nil)$ .

## 1.6. ИНДУКТИВНЫЕ СЕМЕЙСТВА

**Определение 1.1.** *Индуктивное семейство* [12] — это семейство типов данных, которые могут зависеть от других типов и значений.

Тип или значение, от которого зависит зависимый тип, называют *индексом*.

Одной из областей применения индуктивных семейств являются системы интерактивного доказательства теорем.

Индуктивные семейства позволяют формализовать математические структуры, кодируя утверждения о структурах в них самих, тем самым перенося сложность из доказательств в определения.

В работах [13, 14] приведены различные подходы к построению функциональных структур данных.

Пример задания структуры данных и инвариантов — тип данных AVL-дерева и для хранения баланса в AVL-дереве [15].

Если  $m \sim n$ , то разница между  $m$  и  $n$  не больше чем один:

```
data _~_ : ℕ → ℕ → Set where
  ~+ : ∀ {n} → n ~ 1 + n
  ~0 : ∀ {n} → n ~ n
  ~- : ∀ {n} → 1 + n ~ n
```

## 1.7. AGDA

*Agda* [5] — чистый функциональный язык программирования с зависимыми типами. В *Agda* есть поддержка модулей:

```
module AgdaDescription where
```

В коде на *Agda* широко используются символы Unicode. Тип натуральных чисел —  $\mathbb{N}$ .

```
data  $\mathbb{N}$  : Set where
  zero :  $\mathbb{N}$ 
  succ  :  $\mathbb{N} \rightarrow \mathbb{N}$ 
```

В *Agda* функции можно определять как *mixfix* операторы. Пример — сложение натуральных чисел:

```
_+_ :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ 
zero + b = b
succ a + b = succ (a + b)
```

Символы подчеркивания обозначают места для аргументов.

Зависимые типы позволяют определять типы, зависящие (индексированные) от значений других типов. Пример — список, индексированный своей длиной:

```
data Vec (A : Set) :  $\mathbb{N} \rightarrow$  Set where
  nil  : Vec A zero
  cons :  $\forall \{n\} \rightarrow A \rightarrow$  Vec A n  $\rightarrow$  Vec A (succ n)
```

В фигурные скобки заключаются неявные аргументы.

Такое определение позволяет нам описать функцию `head` для такого списка, которая не может бросить исключение:



$\text{head} : \forall \{A\} \{n\} \rightarrow \text{Vec } A (\text{succ } n) \rightarrow A$

У аргумента функции `head` тип  $\text{Vec } A (\text{succ } n)$ , то есть вектор, в котором есть хотя бы один элемент. Это позволяет произвести сопоставление с образцом только по конструктору `cons`:

$\text{head } (\text{cons } a \text{ as}) = a$

## 1.8. ВЫВОДЫ ПО ГЛАВЕ 1

Рассмотрены некоторые существующие подходы к построению структур данных с использованием индуктивных семейств. Кратко описаны особенности языка программирования *Agda*.

# Глава 2. Описание реализованной структуры данных

В данной главе описывается разработанная функциональная структура данных приоритетная очередь типа «двоичная куча».

## 2.1. ПОСТАНОВКА ЗАДАЧИ

Целью данной работы является разработка типов данных для представления структуры данных и инвариантов.

Требования к данной работе:

- Разработать типы данных для представления структуры данных
- Реализовать функции по работе со структурой данных
- Используя разработанные типы данных доказать выполнение инвариантов.

## 2.2. СТРУКТУРА ДАННЫХ «ДВОИЧНАЯ КУЧА»

**Определение 2.1.** Двоичная куча или пирамида [16] — такое двоичное подвешенное дерево, для которого выполнены следующие три условия:

- Значение в любой вершине не больше (если куча для минимума), чем значения её потомков.
- На  $i$ -ом слое  $2^i$  вершин, кроме последнего. Слои нумеруются с нуля.
- Последний слой заполнен слева направо

На рисунке 2.1 изображен пример кучи.

## 2.3. РЕАЛИЗАЦИЯ

### 2.3.1. Вспомогательные определения

Часть общеизвестных определений заимствована из стандартной библиотеки Agda [17].

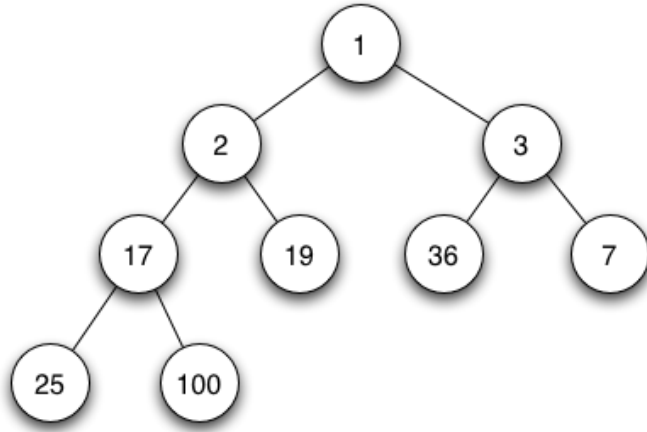


Рис. 2.1. Пример заполненной кучи для минимума

Тип данных для пустого типа. У этого типа нет конструкторов, и, как следствие, нет термов, населяющих этот тип.

`data  $\perp$  : Set where`

Из элемента пустого типа следует что-угодно.

`$\perp$ -elim :  $\forall \{a\} \{Whatever : Set\} a \rightarrow \perp \rightarrow Whatever$`   
 `$\perp$ -elim ()`

Логическое отрицание.

`$\neg$  :  $\forall \{a\} \rightarrow Set\ a \rightarrow Set\ a$`   
 `$\neg P = P \rightarrow \perp$`

Контрадикция, противоречие: из  $A$  и  $\neg A$  можно получить любое  $B$ .

`contradiction :  $A \rightarrow \neg A \rightarrow B$`   
`contradiction a  $\neg$ a =  $\perp$ -elim ( $\neg$ a a)`

Контрапозиция

`contraposition :  $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$`

`contraposition = flip _o_`

Пропозициональное равенство из ИТТ.

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

Тип-сумма — зависимая пара.

```
record Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field fst : A ; snd : B fst
```

Декартово произведение — частный случай зависимой пары, Второй индекс игнорирует передаваемое ему значение.

```
_×_ : ∀ {a b} (A : Set a) → (B : Set b) → Set (a ⊔ b)
A × B = Σ A (λ _ → B)
```

Конгруэнтность пропозиционального равенства.

```
cong : ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

### 2.3.2. Определение отношений и их свойств

Для сравнения элементов нужно задать отношения на этих элементах.

```
Rel2 : Set → Set1
Rel2 A = A → A → Set
```

Трихотомичность отношений меньше, равно и больше: одновременно два элемента могут принадлежать только одному отношению из трех.

```

data Tri { A : Set } ( _ < _ _ == _ _ > _ : Rel2 A ) ( a b : A ) : Set where
  tri< : ( a < b ) → ¬ ( a == b ) → ¬ ( a > b ) → Tri _ < _ _ == _ _ > _ a b
  tri= : ¬ ( a < b ) → ( a == b ) → ¬ ( a > b ) → Tri _ < _ _ == _ _ > _ a b
  tri> : ¬ ( a < b ) → ¬ ( a == b ) → ( a > b ) → Tri _ < _ _ == _ _ > _ a b

```

Введем упрощенный предикат, использующий только два отношения — меньше и равенство. Отношение больше заменяется отношением меньше с переставленными аргументами.

```

flip1 : ∀ { A B : Set } { C : Set1 } → ( A → B → C ) → B → A → C

```

```

flip1 f a b = f b a

```

```

Cmp : { A : Set } → Rel2 A → Rel2 A → Set

```

```

Cmp { A } _ < _ _ == _ = ∀ ( x y : A ) → Tri ( _ < _ ) ( _ == _ ) ( flip1 _ < _ ) x y

```

Тип данных для отношения меньше или равно на натуральных числах.

```

data _N≤_ : Rel2 ℕ where

```

```

  z≤n : ∀ { n } → zero N≤ n

```

```

  s≤s : ∀ { n m } → n N≤ m → succ n N≤ succ m

```

Все остальные отношения определяются через `_N≤_`.

```

_N<_ _N≥_ _N>_ : Rel2 ℕ

```

```

n N< m = succ n N≤ m

```

```

n N> m = m N< n

```

```

n N≥ m = m N≤ n

```

В качестве примера компаратора — доказательство трихотомичности для отношения меньше для натуральных чисел.

`lemma-succ-≡ : ∀ {n} {m} → succ n ≡ succ m → n ≡ m`

`lemma-succ-≡ refl = refl`

`lemma-succ-≤ : ∀ {n} {m} → succ (succ n) N≤ succ m → succ n N≤ m`

`lemma-succ-≤ (s≤s r) = r`

`cmpN : Cmp {N} _N<_ _≡_`

`cmpN zero (zero) = tri= (λ ()) refl (λ ())`

`cmpN zero (succ y) = tri< (s≤s z≤n) (λ ()) (λ ())`

`cmpN (succ x) zero = tri> (λ ()) (λ ()) (s≤s z≤n)`

`cmpN (succ x) (succ y) with cmpN x y`

`... | tri< a ¬b ¬c = tri< (s≤s a) (contraposition lemma-succ-≡ ¬b)`

`(contraposition lemma-succ-≤ ¬c)`

`... | tri> ¬a ¬b c = tri> (contraposition lemma-succ-≤ ¬a)`

`(contraposition lemma-succ-≡ ¬b) (s≤s c)`

`... | tri= ¬a b ¬c = tri= (contraposition lemma-succ-≤ ¬a)`

`(cong succ b) (contraposition lemma-succ-≤ ¬c)`

Транзитивность отношения

`Trans : {A : Set} → Rel2 A → Set`

`Trans {A} _rel_ = {a b c : A} → (a rel b) → (b rel c) → (a rel c)`

Симметричность отношения.

`Symmetric : ∀ {A : Set} → Rel2 A → Set`

`Symmetric _rel_ = ∀ {a b} → a rel b → b rel a`

Предикат  $P$  учитывает (соблюдает) отношение  $\_rel\_$  .

$\_Respects\_ : \forall \{\ell\} \{A : Set\} \rightarrow (A \rightarrow Set \ell) \rightarrow Rel_2 A \rightarrow Set \_$   
 $P \text{ Respects } \_rel\_ = \forall \{x\} y \rightarrow x \text{ rel } y \rightarrow P x \rightarrow P y$

Отношение  $P$  соблюдает отношение  $\_rel\_$  .

$\_Respects_2\_ : \forall \{A : Set\} \rightarrow Rel_2 A \rightarrow Rel_2 A \rightarrow Set$   
 $P \text{ Respects}_2 \_rel\_ =$   
 $(\forall \{x\} \rightarrow P x \text{ Respects } \_rel_) \times$   
 $(\forall \{y\} \rightarrow \text{flip } P y \text{ Respects } \_rel_)$

Тип данных для обобщенного отношения меньше или равно.

$\text{data } \_<=_ \{A : Set\} \{ \_<\_ : Rel_2 A\} \{ \_==\_ : Rel_2 A\} : Rel_2 A \text{ where}$   
 $\text{le} : \forall \{x\} y \rightarrow x < y \rightarrow x <= y$   
 $\text{eq} : \forall \{x\} y \rightarrow x == y \rightarrow x <= y$

Обобщенные функции минимум и максимум.

$\text{min max} : \{A : Set\} \{ \_<\_ : Rel_2 A\} \{ \_==\_ : Rel_2 A\}$   
 $\rightarrow (cmp : Cmp \_<\_ \_==_) \rightarrow A \rightarrow A \rightarrow A$   
 $\text{min } cmp \ x \ y \text{ with } cmp \ x \ y$   
 $\dots \mid \text{tri} < \_ \_ \_ = x$   
 $\dots \mid \_ = y$   
 $\text{max } cmp \ x \ y \text{ with } cmp \ x \ y$   
 $\dots \mid \text{tri} > \_ \_ \_ = x$   
 $\dots \mid \_ = y$

Лемма: элемент меньше или равный двух других элементов меньше или равен минимума из них.

```

lemma-<=min : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A}
  {cmp : Cmp _<_ _==_} {a b c : A}
  → (_<=_ {_<_ = _<_} {_==_} a b)
  → (_<=_ {_<_ = _<_} {_==_} a c)
  → (_<=_ {_<_ = _<_} {_==_} a (min cmp b c))

```

Функция — минимум из трех элементов.

```

min3 : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A}
  → (cmp : Cmp _<_ _==_) → A → A → A → A
min3 cmp x y z with cmp x y
... | tri< _ _ _ = min cmp x z
... | _ = min cmp y z

```

Аналогичная предыдущей лемма для минимума из трех элементов.

```

lemma-<=min3 : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A}
  {cmp : Cmp _<_ _==_} {x a b c : A}
  → (_<=_ {_<_ = _<_} {_==_} x a)
  → (_<=_ {_<_ = _<_} {_==_} x b)
  → (_<=_ {_<_ = _<_} {_==_} x c)
  → (_<=_ {_<_ = _<_} {_==_} x (min3 cmp a b c))
lemma-<=min3 {cmp = cmp} {x} {a} {b} {c} xa xb xc with cmp a b
... | tri< _ _ _ = lemma-<=min {cmp = cmp} xa xc
... | tri= _ _ _ = lemma-<=min {cmp = cmp} xb xc
... | tri> _ _ _ = lemma-<=min {cmp = cmp} xb xc

```

Леммы `lemma-<=min` и `lemma-<=min3` понадобятся при доказательстве соотношений между элементами, из которых составляются новые кучи при их обработке.

Отношение `_<=_` соблюдает отношение равен-



ства  $\_ == \_$ , с помощью которого оно определено.

```

resp<= : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A}
  → (resp : _<_ Respects2 _==_) → (trans== : Trans _==_)
  → (sym== : Symmetric _==_) → (_<=_ {A}{_<_}{_==_}) Respects2 _==_
resp<= {A}{_<_}{_==_} resp trans sym = left , right where
left : ∀ {a b c : A} → b == c → a <= b → a <= c
left b=c (le a<b) = le (fst resp b=c a<b)
left b=c (eq a=b) = eq (trans a=b b=c)
right : ∀ {a b c : A} → b == c → b <= a → c <= a
right b=c (le a<b) = le (snd resp b=c a<b)
right b=c (eq a=b) = eq (trans (sym b=c) a=b)

```

Транзитивность отношения  $\_<=_$ .

```

trans<= : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A}
  → _<_ Respects2 _==_ → Symmetric _==_ → Trans _==_ → Trans _<_
  → Trans (_<=_ {A}{_<_}{_==_})
trans<= r s t== t< (le a<b) (le b<c) = le (t< a<b b<c)
trans<= r s t== t< (le a<b) (eq b=c) = le (fst r b=c a<b)
trans<= r s t== t< (eq a=b) (le b<c) = le (snd r (s a=b) b<c)
trans<= r s t== t< (eq a=b) (eq b=c) = eq (t== a=b b=c)

```

### 2.3.3. КучКучаа

Модуль, в котором мы определим структуру данных куча, параметризован исходным типом, двумя отношениями, определенными для этого типа,  $\_<=_$  и  $\_==\_$ . Также требуется симметричность и транзитивность  $\_==\_$ , транзитивность  $\_<=_$ , соблюдение отношением  $\_<=_$  отношения  $\_==\_$  и

```

module TryHeap (A : Set) (_<_ _==_ : Rel2 A) (cmp : Cmp _<_ _==_)
  (sym== : Symmetric _==_) (trans== : Trans _==_)
  (trans< : Trans _<_) (resp : _<_ Respects2 _==_)
  where

```

Будем индексировать кучу минимальным элементом в ней, для того, чтобы можно было строить инварианты порядка на куче исходя из этих индексов. Так как в пустой куче нет элементов, то мы не можем выбрать элемент, который нужно указать в индексе. Чтобы решить эту проблему, расширим исходный тип данных, добавив элемент, больший всех остальных. Тип данных для расширения исходного типа.

```

data expanded (A : Set) : Set where
  # : A → expanded A - (# x) - элемент исходного типа
  top : expanded A - элемент расширение

```

Теперь нам нужно аналогичным образом расширить отношения заданные на множестве исходного типа. Тип данных для расширения отношения меньше.

```

data _<E_ : Rel2 (expanded A) where
  base : ∀ {x y : A} → x < y → (# x) <E (# y)
  ext  : ∀ {x : A} → (# x) <E top

```

Вспомогательная лемма, извлекающая доказательство для отношения элементов исходного типа из отношения для элементов расширенного типа.

```

lemma-<E : ∀ {x} {y} → (# x) <E (# y) → x < y
lemma-<E (base r) = r

```

Расширенное отношение меньше — транзитивно.

```
trans<E : Trans _<E_
trans<E {# _} {# _} {# _} a<b b<c =
  base (trans< (lemma-<E a<b) (lemma-<E b<c))
trans<E {# _} {# _} {top} _ _ = ext
```

Тип данных расширенного отношения равенства.

```
data _=E_ : Rel2 (expanded A) where
  base : ∀ {x y} → x == y → (# x) =E (# y)
  ext : top =E top
```

Расширенное отношение равенства — симметрично и транзитивно.

```
sym=E : Symmetric _=E_
sym=E (base a=b) = base (sym== a=b)
sym=E ext = ext
trans=E : Trans _=E_
trans=E (base a=b) (base b=c) = base (trans== a=b b=c)
trans=E ext ext = ext
```

Отношение  $\_<E\_$  соблюдает отношение  $\_=E\_$ .

```
respE : _<E_ Respects2 _=E_
respE = left , right where
  left : ∀ {a b c : expanded A} → b =E c → a <E b → a <E c
  left {# _} {# _} {# _} (base r1) (base r2) = base (fst resp r1 r2)
  left {# _} {top} {top} ext ext = ext
```

$\text{right} : \forall \{a \ b \ c : \text{expanded } A\} \rightarrow b =_E c \rightarrow b <_E a \rightarrow c <_E a$   
 $\text{right } \{\# \_ \} \{\# \_ \} \{\# \_ \} (\text{base } r1) (\text{base } r2) = \text{base } (\text{snd } \text{resp } r1 \ r2)$   
 $\text{right } \{\text{top}\} \{\# \_ \} \{\# \_ \} \_ \text{ext} = \text{ext}$

Отношение меньше-равно для расширенного типа.

$\_ \leq \_ : \text{Rel}_2 (\text{expanded } A)$   
 $\_ \leq \_ = \_ <= \_ \{ \text{expanded } A \} \{ \_ <_E \_ \} \{ \_ =_E \_ \}$

Транзитивность меньше-равно следует из свойств отношений  $\_ =_E \_$  и  $\_ <_E \_$ :

$\text{trans} \leq : \text{Trans } \_ \leq \_$   
 $\text{trans} \leq = \text{trans} <= \text{resp } E \text{ sym} =_E \text{trans} =_E \text{trans} <_E$   
 $\text{resp} \leq : \_ \leq \_ \text{Respects}_2 \_ =_E \_$   
 $\text{resp} \leq = \text{resp} <= \text{resp } E \text{trans} =_E \text{sym} =_E$

Вспомогательная лемма, извлекающая доказательство равенства элементов исходного типа из равенства элементов расширенного типа.

$\text{lemma} =_E : \forall \{x\} \{y\} \rightarrow (\# x) =_E (\# y) \rightarrow x == y$   
 $\text{lemma} =_E (\text{base } r) = r$

Трихотомичность для  $\_ <_E \_$  и  $\_ =_E \_$ .

$\text{cmp } E : \text{Cmp } \{ \text{expanded } A \} \_ <_E \_ \_ =_E \_$   
 $\text{cmp } E (\# x) (\# y) \text{ with } \text{cmp } x \ y$   
 $\text{cmp } E (\# x) (\# y) \mid \text{tri} < a \ b \ c =$   
 $\quad \text{tri} < (\text{base } a) (\text{contraposition lemma} =_E b) (\text{contraposition lemma} <_E c)$   
 $\text{cmp } E (\# x) (\# y) \mid \text{tri} = a \ b \ c =$   
 $\quad \text{tri} = (\text{contraposition lemma} <_E a) (\text{base } b) (\text{contraposition lemma} <_E c)$

```

cmpE (# x) (# y) | tri> a b c =
  tri> (contraposition lemma-<E a) (contraposition lemma-=E b) (base c)
cmpE (# x) top = tri< ext (λ ()) (λ ())
cmpE top (# y) = tri> (λ ()) (λ ()) ext
cmpE top top = tri= (λ ()) ext (λ ())

```

Функция — минимум для расширенного типа.

```

minE : (x y : expanded A) → expanded A
minE = min cmpE

```

Вспомогательный тип данных для индексации кучи — куча полная или почти заполненная.

```

data HeapState : Set where
  full almost : HeapState

```

Тип данных для кучи, проиндексированный минимальным элементом кучи, натуральным числом — высотой — и заполненностью.

```

data Heap : (expanded A) → (h : ℕ) → HeapState → Set where

```

У пустой кучи минимальный элемент — `top`, высота — ноль. Пустая куча — полная.

```

eh : Heap top zero full

```

Полная куча высотой  $n + 1$  состоит из корня и двух куч высотой  $n$ . Мы хотим в непустых кучах задавать порядок на элементах — элемент в узле меньше либо равен элементам в поддеревьях. Мы можем упростить этот инвариант, сравнивая элемент в узле только с корнями поддеревьев. Порядок кучи задается с помощью двух элементов отношения

$\_ \leq \_$ :  $i$  и  $j$ , которые говорят о том, что значение в корне меньше-равно значений в корнях левого и правого поддеревьев соответственно. На рисунке 2.2 схематично изображены конструкторы типа данных **Heap**.

$$\begin{aligned}
\text{nf} : & \forall \{n\} \{x\ y\} \rightarrow (p : A) \rightarrow (i : (\# \ p) \leq x) \rightarrow (j : (\# \ p) \leq y) \\
& \rightarrow (a : \text{Heap } x \ n \ \text{full}) \\
& \rightarrow (b : \text{Heap } y \ n \ \text{full}) \\
& \rightarrow \text{Heap } (\# \ p) \ (\text{succ } n) \ \text{full}
\end{aligned}$$

Куча высотой  $n + 2$ , у которой нижний ряд заполнен до середины, состоит из корня и двух полных куч: левая высотой  $n + 1$  и правая высотой  $n$ .

$$\begin{aligned}
\text{nd} : & \forall \{n\} \{x\ y\} \rightarrow (p : A) \rightarrow (i : (\# \ p) \leq x) \rightarrow (j : (\# \ p) \leq y) \\
& \rightarrow (a : \text{Heap } x \ (\text{succ } n) \ \text{full}) \\
& \rightarrow (b : \text{Heap } y \ n \ \text{full}) \\
& \rightarrow \text{Heap } (\# \ p) \ (\text{succ } (\text{succ } n)) \ \text{almost}
\end{aligned}$$

Куча высотой  $n + 2$ , у которой нижний ряд заполнен меньше, чем до середины, состоит из корня и двух куч: левая неполная высотой  $n + 1$  и правая полная высотой  $n$ .

$$\begin{aligned}
\text{nl} : & \forall \{n\} \{x\ y\} \rightarrow (p : A) \rightarrow (i : (\# \ p) \leq x) \rightarrow (j : (\# \ p) \leq y) \\
& \rightarrow (a : \text{Heap } x \ (\text{succ } n) \ \text{almost}) \\
& \rightarrow (b : \text{Heap } y \ n \ \text{full}) \\
& \rightarrow \text{Heap } (\# \ p) \ (\text{succ } (\text{succ } n)) \ \text{almost}
\end{aligned}$$

Неполная куча высотой  $n + 2$ , у которой нижний ряд заполнен больше, чем до середины, состоит из корня и двух куч: левая полная высотой  $n + 1$  и правая неполная высотой  $n + 1$ .

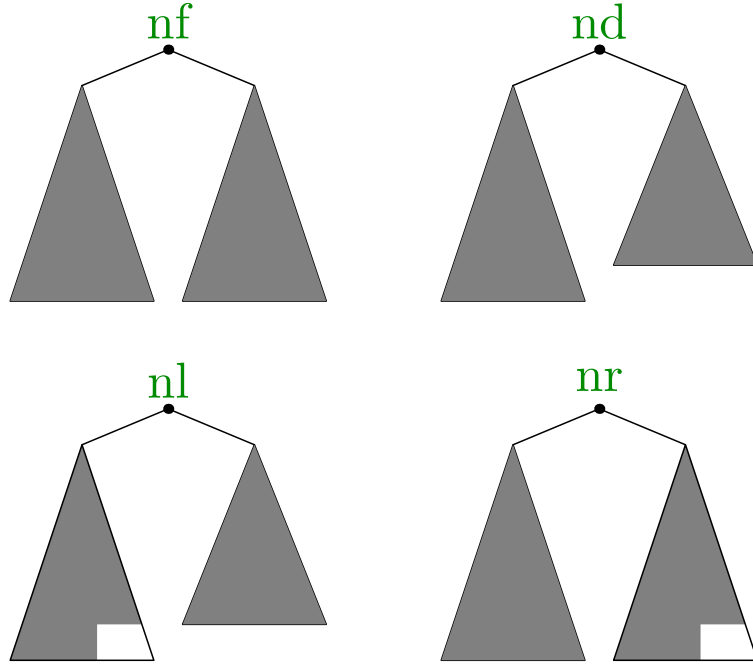


Рис. 2.2. Конструкторы типа данных **Heap**

$$\begin{aligned}
 \text{nr} : & \forall \{n\} \{x\ y\} \rightarrow (p : A) \rightarrow (i : (\# \ p) \leq x) \rightarrow (j : (\# \ p) \leq y) \\
 & \rightarrow (a : \text{Heap } x \ (\text{succ } n) \ \text{full}) \\
 & \rightarrow (b : \text{Heap } y \ (\text{succ } n) \ \text{almost}) \\
 & \rightarrow \text{Heap } (\# \ p) \ (\text{succ } (\text{succ } n)) \ \text{almost}
 \end{aligned}$$

*Замечание:* высота любой неполной кучи больше нуля.

$$\text{lemma-almost-height} : \forall \{m\ h\} \rightarrow \text{Heap } m\ h \ \text{almost} \rightarrow h \ \mathbb{N} > 0$$

Функция — просмотр минимума в куче.

$$\text{peekMin} : \forall \{m\ h\ s\} \rightarrow \text{Heap } m\ h\ s \rightarrow (\text{expanded } A)$$

$$\text{peekMin } \text{eh} = \text{top}$$

$$\text{peekMin } (\text{nd } p \ \_ \ \_ \ \_ \ \_) = \# \ p$$

$$\text{peekMin } (\text{nf } p \ \_ \ \_ \ \_ \ \_) = \# \ p$$

$$\text{peekMin } (\text{nl } p \ \_ \ \_ \ \_ \ \_) = \# \ p$$

$$\text{peekMin } (\text{nr } p \ \_ \ \_ \ \_ \ \_) = \# \ p$$

Функция — минимум из трех элементов расширенного типа — частный случай ранее определенной общей функции.

$\text{min3E} : (\text{expanded } A) \rightarrow (\text{expanded } A) \rightarrow (\text{expanded } A) \rightarrow (\text{expanded } A)$   
 $\text{min3E } x \ y \ z = \text{min3 cmpE } x \ y \ z$

Леммы для сравнения с минимумами для элементов расширенного типа.

$\text{lemma-}\leq\text{minE} : \forall \{a \ b \ c\} \rightarrow a \leq b \rightarrow a \leq c \rightarrow a \leq (\text{minE } b \ c)$   
 $\text{lemma-}\leq\text{minE} = \text{lemma-}\leq\text{min } \{\text{expanded } A\} \{ \_ < \text{E} \_ \} \{ \_ = \text{E} \_ \} \{ \text{cmpE} \}$   
 $\text{lemma-}\leq\text{min3E} : \forall \{x \ a \ b \ c\} \rightarrow x \leq a \rightarrow x \leq b \rightarrow x \leq c \rightarrow x \leq (\text{min3E } a \ b \ c)$   
 $\text{lemma-}\leq\text{min3E} = \text{lemma-}\leq\text{min3 } \{\text{expanded } A\} \{ \_ < \text{E} \_ \} \{ \_ = \text{E} \_ \} \{ \text{cmpE} \}$

Функция вставки элемента в полную кучу.

$\text{finsert} : \forall \{h \ m\} \rightarrow (z : A) \rightarrow \text{Heap } m \ h \ \text{full}$   
 $\rightarrow \Sigma \text{HeapState } (\text{Heap } (\text{minE } m \ (\# \ z)) \ (\text{succ } h))$

Вставка элемента в неполную кучу.

$\text{ainsert} : \forall \{h \ m\} \rightarrow (z : A) \rightarrow \text{Heap } m \ h \ \text{almost}$   
 $\rightarrow \Sigma \text{HeapState } (\text{Heap } (\text{minE } m \ (\# \ z)) \ h)$

Вспомогательный тип данных.

$\text{data OR } (A \ B : \text{Set}) : \text{Set where}$   
 $\text{orA} : A \rightarrow \text{OR } A \ B$   
 $\text{orB} : B \rightarrow \text{OR } A \ B$

Слияние двух полных куч одной высоты.

$\text{fmerge} : \forall \{x \ y \ h\} \rightarrow \text{Heap } x \ h \ \text{full} \rightarrow \text{Heap } y \ h \ \text{full}$   
 $\rightarrow \text{OR } (\text{Heap } x \ \text{zero} \ \text{full} \times (x \equiv y) \times (h \equiv \text{zero}))$



$(\text{Heap } (\text{minE } x \ y) \ (\text{succ } h) \ \text{almost})$

Извлечение минимума из полной кучи.

$\text{fpop} : \forall \{m \ h\} \rightarrow \text{Heap } m \ (\text{succ } h) \ \text{full}$   
 $\rightarrow \text{OR } (\Sigma \ (\text{expanded } A))$   
 $(\lambda x \rightarrow (\text{Heap } x \ (\text{succ } h) \ \text{almost}) \times (m \leq x))$   
 $(\text{Heap } \text{top } h \ \text{full})$

Составление полной кучи высотой  $h + 1$  из двух куч высотой  $h$  и одного элемента.

$\text{makeH} : \forall \{x \ y \ h\} \rightarrow (p : A) \rightarrow \text{Heap } x \ h \ \text{full} \rightarrow \text{Heap } y \ h \ \text{full}$   
 $\rightarrow \text{Heap } (\text{min3E } x \ y \ (\# \ p)) \ (\text{succ } h) \ \text{full}$

Вспомогательные леммы, использующие  $\text{lemma-}\leq\text{minE}$ .

$\text{lemma-resp} : \forall \{x \ y \ a \ b\} \rightarrow x == y \rightarrow (\# \ x) \leq a \rightarrow (\# \ x) \leq b$   
 $\rightarrow (\# \ y) \leq \text{minE } a \ b$

$\text{lemma-resp } x=y \ i \ j = \text{lemma-}\leq\text{minE } (\text{snd resp} \leq (\text{base } x=y) \ i)$   
 $(\text{snd resp} \leq (\text{base } x=y) \ j)$

$\text{lemma-trans} : \forall \{x \ y \ a \ b\} \rightarrow y < x \rightarrow (\# \ x) \leq a \rightarrow (\# \ x) \leq b$   
 $\rightarrow (\# \ y) \leq \text{minE } a \ b$

$\text{lemma-trans } y<x \ i \ j = \text{lemma-}\leq\text{minE } (\text{trans} \leq (\text{le } (\text{base } y<x)) \ i)$   
 $(\text{trans} \leq (\text{le } (\text{base } y<x)) \ j)$

Слияние поддеревьев из кучи, у которой последний ряд заполнен до середины, определенной конструктором  $\text{nd}$ .

$\text{ndmerge} : \forall \{x \ y \ h\} \rightarrow \text{Heap } x \ (\text{succ } (\text{succ } h)) \ \text{full} \rightarrow \text{Heap } y \ (\text{succ } h) \ \text{full}$   
 $\rightarrow \text{Heap } (\text{minE } x \ y) \ (\text{succ } (\text{succ } (\text{succ } h))) \ \text{almost}$

Слияние неполной кучи высотой  $h + 2$  и полной кучи высотой  $h + 1$  или  $h + 2$ .

$$\begin{aligned}
& \text{afmerge} : \forall \{h\} x y \rightarrow \text{Heap } x (\text{succ } (\text{succ } h)) \text{ almost} \\
& \rightarrow \text{OR } (\text{Heap } y (\text{succ } h) \text{ full}) (\text{Heap } y (\text{succ } (\text{succ } h)) \text{ full}) \\
& \rightarrow \text{OR } (\text{Heap } (\text{minE } x y) (\text{succ } (\text{succ } h)) \text{ full}) \\
& \quad (\text{Heap } (\text{minE } x y) (\text{succ } (\text{succ } (\text{succ } h))) \text{ almost})
\end{aligned}$$

Извлечение минимума из неполной кучи.

$$\begin{aligned}
& \text{ароп} : \forall \{m\} h \rightarrow \text{Heap } m (\text{succ } h) \text{ almost} \\
& \rightarrow \text{OR } (\Sigma (\text{expanded } A) (\lambda x \rightarrow (\text{Heap } x (\text{succ } h) \text{ almost}) \times (m \leq x))) \\
& \quad (\Sigma (\text{expanded } A) (\lambda x \rightarrow (\text{Heap } x h \text{ full}) \times (m \leq x)))
\end{aligned}$$

## 2.4. ВЫВОДЫ ПО ГЛАВЕ 2

Разработаны типы данных для представления структуры данных двоичная куча.

## Список литературы

1. *Thompson S.* Type theory and functional programming. International computer science series. Addison-Wesley, 1991. С. I–XV, 1–372. ISBN: 978-0-201-41667-1.
2. *Sørensen M. H. B., Urzyczyn P.* Lectures on the Curry-Howard Isomorphism. 1998.
3. The Haskell Programming Language. <http://www.haskell.org/haskellwiki/Haskell>.
4. A Truly Integrated Functional Logic Language. <http://www-ps.informatik.uni-kiel.de/currywiki/>.
5. Agda language. <http://wiki.portal.chalmers.se/agda/pmwiki.php>.
6. *IEEE.* IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language. 1991. С. 52. ISBN: 1-55937-125-0. [http://standards.ieee.org/reading/ieee/std\\_public/description/busarch/1178-1990\\_desc.html](http://standards.ieee.org/reading/ieee/std_public/description/busarch/1178-1990_desc.html).
7. *Hickey R.* The Clojure programming language / DLS. Под ред. Johan Brichau. ACM, 2008. С. 1. ISBN: 978-1-60558-270-2.
8. *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs. MIT Press, 1985. ISBN: 0-262-51036-7.
9. *Milner R., Tofte M., Macqueen D.* The Definition of Standard ML. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.
10. OCaml. <http://ocaml.org/>.
11. *Martin-Löf P.* Intuitionistic Type Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.
12. *Dybjer P.* Inductive Families // Formal Asp. Comput. 1994. №4. С. 440–465.
13. *Okasaki C.* Purely Functional Data Structures. Докт. дисс. Pittsburgh, PA 15213, 1996.
14. *McBride C.* How to Keep Your Neighbours in Order. <https://personal.cis.strath.ac.uk/conor.mcbride/Pivotal>.
15. *McBride C., Norell U., Danielsson N. A.* The Agda standard library — AVL trees. <http://agda.github.io/agda-stdlib/html/Data.AVL.html>.
16. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to Algorithms, Second Edition. The MIT Press и McGraw-Hill Book Company, 2001. ISBN: 0-262-03293-7, 0-07-013151-1.
17. The Agda standard library. <http://agda.github.io/agda-stdlib/html/README.html>.