

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: ассистент кафедры ТП Я. М. Малаховски
Санкт-Петербург

2014

Содержание

Введение	4
Глава 1. Обзор	5
1.1 Функциональное программирование	5
1.1.1 Концепции	5
1.1.2 Сопоставление с образцом	5
1.2 Теория типов	5
1.2.1 Отношение конвертабельности	6
1.2.2 Интуиционистская теория типов	6
1.3 Унификация	7
1.4 Индуктивные семейства	7
1.5 Agda	8
1.6 Выводы по главе 1	9
Глава 2. Описание реализованной структуры данных	10
2.1 Постановка задачи	10
2.2 Структура данных «двоичная куча»	10
2.3 Реализация	10
2.3.1 Вспомогательные определения	10
2.3.2 Определение отношений и их свойств	12
2.3.3 Куча	17
2.4 Выводы по главе 2	26
Список литературы	27

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

1.1. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании) [1]. В функциональном программировании избегается использование изменяемого глобального состояния и изменяемых данных.

1.1.1. Концепции

Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции [2]. *Чистые функции* — функции, которые не имеют побочных эффектов ввода-вывода и изменения памяти, они зависят только от своих параметров и возвращают только свой результат.

1.1.2. Сопоставление с образцом

Сопоставление с образцом — способ обработки структур данных, при котором аргументы функций сравниваются (по значению или по структуре) с образцом такого же типа.

1.2. 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*.

1.2.1. 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.

1.2.2. Интуиционистская теория типов

Интуиционистская теория типов основана на математическом конструктивизме [3].

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.
- Σ -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: $\text{refl} : \prod_{a:A} a \equiv a$.

1.3. 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 $x \mapsto 2, y \mapsto cons(2, nil)$.

1.4. ИНДУКТИВНЫЕ СЕМЕЙСТВА

Определение 1.1. *Индуктивное семейство* [4] — 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 [5, 6] 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 [7].

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
```

1.5. AGDA

Agda [8] — 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`:

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

1.6. 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 [9] — 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 [10].

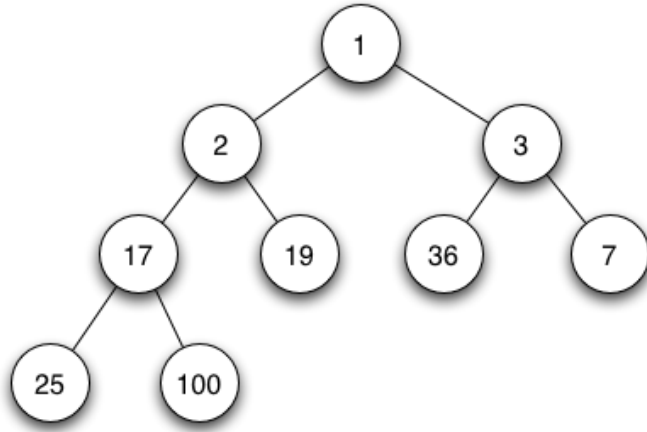


Рис. 2.1. Пример заполненной кучи для минимума

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

Логическое отрицание.

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

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) 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_ : ∀ {ℓ} {A : Set} → (A → Set ℓ) → Rel2 A → Set`

$P \text{ Respects } _rel_ = \forall \{x\ y\} \rightarrow x \text{ rel } y \rightarrow P\ x \rightarrow P\ y$

Отношение P соблюдает отношение $_rel_$.

$_Respects_2_ : \forall \{A : \text{Set}\} \rightarrow \text{Rel}_2\ A \rightarrow \text{Rel}_2\ A \rightarrow \text{Set}$
 $P \text{ Respects}_2\ _rel_ =$
 $(\forall \{x\} \rightarrow P\ x \rightarrow \text{Respects } _rel_) \times$
 $(\forall \{y\} \rightarrow \text{flip } P\ y \rightarrow \text{Respects } _rel_)$

Тип данных для обобщенного отношения меньше или равно.

$\text{data } _<=_ \{A : \text{Set}\} \{ _<_ : \text{Rel}_2\ A \} \{ _==_ : \text{Rel}_2\ A \} : \text{Rel}_2\ A \text{ where}$
 $\text{le} : \forall \{x\ y\} \rightarrow x < y \rightarrow x <=_ y$
 $\text{eq} : \forall \{x\ y\} \rightarrow x == y \rightarrow x <=_ y$

Обобщенные функции минимум и максимум.

$\text{min max} : \{A : \text{Set}\} \{ _<_ : \text{Rel}_2\ A \} \{ _==_ : \text{Rel}_2\ A \} \rightarrow (cmp : \text{Cmp } _<_ _==_)$
 $\text{min } cmp\ x\ y \text{ with } cmp\ x\ y$
 $\dots \mid \text{tri}< _ _ _ = x$
 $\dots \mid _ = y$
 $\text{max } cmp\ x\ y \text{ with } cmp\ x\ y$
 $\dots \mid \text{tri}> _ _ _ = x$
 $\dots \mid _ = y$

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

$\text{lemma-}<=_\text{min} : \{A : \text{Set}\} \{ _<_ : \text{Rel}_2\ A \} \{ _==_ : \text{Rel}_2\ A \}$
 $\{ cmp : \text{Cmp } _<_ _==_ \} \{ a\ b\ c : A \}$
 $\rightarrow (_<=_ \{ _<_ = _<_ \} \{ _==_ \} a\ b) \rightarrow (_<=_ \{ _<_ = _<_ \} \{ _==_ \}$
 $\rightarrow (_<=_ \{ _<_ = _<_ \} \{ _==_ \} a\ (\text{min } cmp\ b\ c))$

Функция — минимум из трех элементов.

```

min3 : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A} → (cmp : Cmp _<_ _==_)
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)
→ (_<=_ {_<_ = _<_} {_==_} 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 равенства `_==_`, с помощью которого оно определено.

```

resp<= : {A : Set} {_<_ : Rel2 A} {_==_ : Rel2 A} → (resp : _<_ 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. Куча

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 _==_) (resp : _<_ Respects₂ _==_) (trans< : Trans _<_
  (trans== : Trans _==_)
  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 ре-

шить 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)
```

$\text{ext} : \text{top} =_E \text{top}$

Расширенное отношение равенства — симметрично и транзитивно.

$\text{sym}=_E : \text{Symmetric } _ =_E _$

$\text{sym}=_E (\text{base } a=b) = \text{base } (\text{sym}== a=b)$

$\text{sym}=_E \text{ ext} = \text{ext}$

$\text{trans}=_E : \text{Trans } _ =_E _$

$\text{trans}=_E (\text{base } a=b) (\text{base } b=c) = \text{base } (\text{trans}== a=b \ b=c)$

$\text{trans}=_E \text{ ext } \text{ext} = \text{ext}$

Отношение $_ <_E _$ соблюдает отношение $_ =_E _$.

$\text{resp}E : _ <_E _ \text{Respects}_2 _ =_E _$

$\text{resp}E = \text{left} , \text{right} \text{ where}$

$\text{left} : \forall \{a \ b \ c : \text{expanded } A\} \rightarrow b =_E c \rightarrow a <_E b \rightarrow a <_E c$

$\text{left } \{ \# _ \} \{ \# _ \} \{ \# _ \} (\text{base } r1) (\text{base } r2) = \text{base } (\text{fst } \text{resp } r1 \ r2)$

$\text{left } \{ \# _ \} \{ \text{top} \} \{ \text{top} \} \text{ext } \text{ext} = \text{ext}$

$\text{right} : \forall \{a \ b \ c : \text{expanded } A\} \rightarrow b =_E c \rightarrow b <_E a \rightarrow c <_E a$

$\text{right } \{ \# _ \} \{ \# _ \} \{ \# _ \} (\text{base } r1) (\text{base } r2) = \text{base } (\text{snd } \text{resp } r1 \ r2)$

$\text{right } \{ \text{top} \} \{ \# _ \} \{ \# _ \} _ \text{ext} = \text{ext}$

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 $_ <E _$:

```
trans≤ : Trans _≤_
trans≤ = trans<= respE sym=E trans=E trans<E
resp≤ : _≤_ Respects₂ _=E_
resp≤ = resp<= respE trans=E sym=E
```

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

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

Трихотомичность для $_ <E _$ и $_ =E _$.

```
cmpE : Cmp {expanded A} _<E_ _=E_
cmpE (# x) (# y) with cmp x y
cmpE (# x) (# y) | tri< a b c =
  tri< (base a) (contraposition lemma-=E b) (contraposition lemma-<E c)
cmpE (# x) (# y) | tri= a b c =
  tri= (contraposition lemma-<E a) (base b) (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`.

```
nf : ∀ {n} {x y} → (p : A) → (i : (# p) ≤ x) → (j : (# p) ≤ y)
  → (a : Heap x n full)
  → (b : Heap y n full)
  → Heap (# p) (succ n) full
```

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 $n + 1$.

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

Замечание: высота любой неполной кучи больше нуля.

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

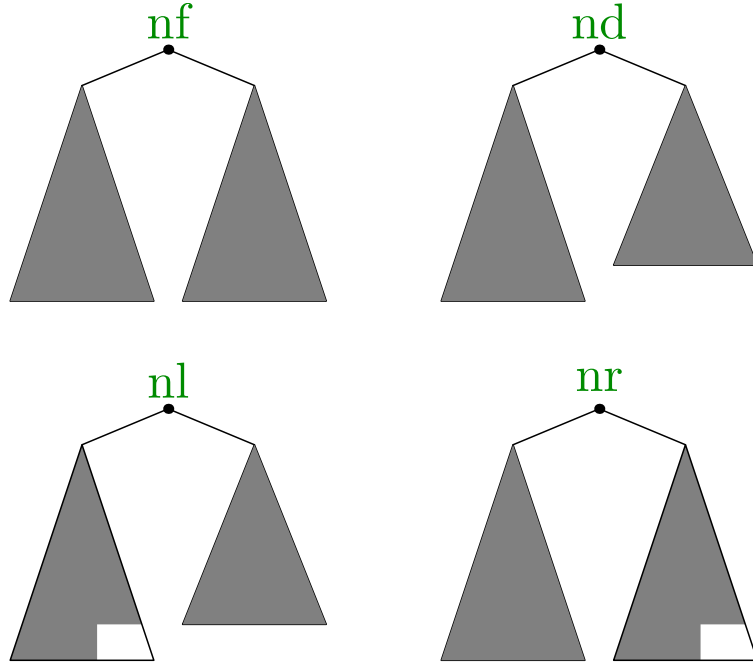


Рис. 2.2. Конструкторы типа данных **Heap**

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.

$\text{lemma-}\leq\text{minE} : \forall \{a\ b\ c\} \rightarrow a \leq b \rightarrow a \leq c \rightarrow a \leq (\text{minE}\ b\ c)$

$\text{lemma-}\leq\text{minE}\ ab\ ac = \text{lemma-}\leq\text{min}\ \{\text{expanded}\ A\}\{_<\text{E}_ \}\{_=\text{E}_ \}\{\text{cmpE}\}\ c$

$\text{lemma-}\leq\text{min3E} : \forall \{x\ a\ b\ c\} \rightarrow x \leq a \rightarrow x \leq b \rightarrow x \leq c \rightarrow x \leq (\text{min3E}\ a\ b\ c)$

$\text{lemma-}\leq\text{min3E} = \text{lemma-}\leq\text{min3}\ \{\text{expanded}\ A\}\{_<\text{E}_ \}\{_=\text{E}_ \}\{\text{cmpE}\}$

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.

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

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 эле-

мента.

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

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 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 $h+2$.

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

Извлечение минимума из неполной кучи.

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

2.4. Выводы по главе 2

Разработаны типы данных для представления структуры данных двоичная куча.

Список литературы

1. Functional programming - Wikipedia. https://en.wikipedia.org/wiki/Functional_programming.
2. *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs. MIT Press, 1985. ISBN: 0-262-51036-7.
3. *Martin-Löf P.* Intuitionistic Type Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.
4. *Dybjer P.* Inductive Families // Formal Asp. Comput. 1994. №4. С. 440–465.
5. *Okasaki C.* Purely Functional Data Structures. Докт. дисс. Pittsburgh, PA 15213, 1996.
6. *McBride C.* How to Keep Your Neighbours in Order. <https://personal.cis.strath.ac.uk/conor.mcbride/Pivotal>
7. *McBride C., Norell U., Danielsson N. A.* The Agda standard library — AVL trees. <http://agda.github.io/agda-stdlib/html/Data.AVL.html>.
8. Agda language. <http://wiki.portal.chalmers.se/agda/pmwiki.php>.
9. *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.
10. The Agda standard library. <http://agda.github.io/agda-stdlib/html/README.html>.