

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

Введение	5
Глава 1. Обзор	6
1.1 Лямбда-исчисление	6
1.2 Лямбда-исчисление с простыми типами	7
1.3 Алгебраические типы данных и сопоставление с образцом . .	7
1.4 Теория типов	7
1.4.1 Отношение конвертабельности	8
1.4.2 Интуиционистская теория типов	8
1.5 Функциональное программирование	9
1.5.1 Сопоставление с образцом	9
1.5.2 ???	9
1.6 Унификация	10
1.7 Индуктивные семейства	10
1.8 Использование индуктивных семейств в структурах данных .	10
1.9 Agda	11
1.10 Выводы по главе 1	12
Глава 2. Описание реализованной структуры данных	13
2.1 Постановка задачи	13
2.2 Структура данных «двоичная куча»	13
2.3 Вспомогательные определения	13
2.3.1 Общие определения	13
2.3.2 Определение отношений и доказательства их свойств .	17
2.4 Модуль Heap	23
2.4.1 Расширение исходного типа	23
2.4.2 Тип данных Heap	27
2.4.3 Функции вставки в кучу	30
2.4.4 Удаление минимума из полной кучи	31
2.4.5 Удаление минимума из неполной кучи	32
2.5 Выводы по главе 2	45

Литература	46
-----------------------------	-----------

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 (λ -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 (α -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 (β -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 .¹

Два лямбда-терма A и B называются *конвертабельными*, когда существует две последовательности бета-редукций, приводящих их к обще-

¹В 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 .

му 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. Лямбда-исчисление с простыми типами

TODO написать про λ^{\rightarrow} [ChurchSTLC, 2]

1.3. Алгебраические типы данных и сопоставление с образцом

Алгебраический тип данных — вид составного типа, то есть типа, сформированного комбинированием других типов. Комбинирование осуществляется с помощью алгебраических операций — сложения и умножения.

Сумма типов A и B — дизъюнктное объединение исходных типов. Значения типа-суммы обычно создаются с помощью *конструкторов*.

Произведение типов A и B — прямое произведение исходных типов, кортеж типов.

TODO написать про μ типы.

1.4. Теория типов

Теория типов — раздел математики изучающий отношения типизации вида $M : \tau$ и их свойства. M называется *термом* или *выражением*, а τ — типом терма M .

Теория типов также изучает правила для *переписывания* термов — замены подтермов в выражениях другими термами. Такие правила также называют правилами *редукции* или *конверсии* термов. Редукцию терма x в терм y записывают: $x \rightarrow y$. Также рассматривают транзитивное замыкание отношения редукции: \rightarrow^* . Например, термы $2 + 1$ и 3 — разные термы, но первый

редуцируется во второй: $2 + 1 \xrightarrow{*} 3$. Если для терма x не существует терма y , для которого $x \rightarrow y$, то говорят, что терм x — в *нормальной форме*.

1.4.1. Отношение конвертабельности

Два терма x и y называются *конвертабельными*, если существует терм z такой, что $x \xrightarrow{*} z$ и $y \xrightarrow{*} z$. Обозначают $x \longleftrightarrow^* y$. 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. Конвертабельность — рефлексивно-транзитивно-симметричное замыкание отношения редукции.

1.4.2. 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 пара. Например, тип $\sum_{n:\mathbb{N}} \text{Vec}(\mathbb{R}, n)$ — тип пары из числа n и кортежа из n вещественных чисел.
- TODO : здесь будет написано про W типы

Базовые типы в ИТТ: \perp или 0 — пустой тип, не содержащий ни одного элемента; \top или 1 — единичный тип, содержащий единственный элемент.

Индуктивный или *рекурсивный* тип — тип данных, который может содержать значения своего типа.

1.5. 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 [4], Curry [5], Agda [6], диалекты LISP [7—9], SML [10], OCaml [11]), но существуют и функциональные языки явно не основанные на этом формализме (например, препроцессор языка C и шаблоны в C++).

1.5.1. Сопоставление с образцом

Сопоставление с образцом — способ обработки объектов алгебраических типов данных, который идентифицирует значения по конструктору и извлекает данные в соответствии с представленным образцом.

1.5.2. ???

TODO Здесь будет написано про дата-тайпы, pattern matching и унификацию в Agda

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {k} → A → Vec A k → Vec A (succ k)
```

соответствует

$$\begin{aligned}
& \forall n. \\
& [] : (n \equiv zero) \rightarrow Vec\ A\ n \\
& _ :: _ : \forall k. A \rightarrow Vec\ A\ k \rightarrow (n \equiv succ\ k) \rightarrow Vec\ A\ n
\end{aligned}$$

1.6. Унификация

Унификатор для термов A и B — подстановка S , действующая на их свободные переменные, такая что $S(A) \equiv S(B)$.

Унификация — процесс поиска унификатора.

1.7. Индуктивные семейства

Определение 1.1. *Индуктивное семейство* [12, 13] — это индуктивный тип данных, который может 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.

1.8. Использование индуктивных семейств в структурах данных

В работах [14, 15] приведены различные подходы в использовании индуктивных семейств в реализации структур данных и доказательств их свойств.

Пример задания структуры данных и инвариантов — тип данных AVL-дерева и тип данных для хранения баланса в AVL-дереве [16].

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
```

В работе [15] представлен способ обобщения упорядоченных структур данных (таких как отсортированные списки и деревья поиска) и использование этого метода для реализации 2-3 деревьев.

1.9. Agda

Agda [6] — 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 ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

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

```
_+_ : ℕ → ℕ → 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) : ℕ → Set where
  nil  : Vec A zero
  cons : ∀ {n} → A → Vec A n → 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:

```
head : ∀ {A} {n} → Vec A (succ n) → A
```

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

```
head (cons a as) = a
```

1.10. 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 [17] — 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. Вспомогательные определения

2.3.1. Общие определения

Часть общеизвестных определений заимствована из стандартной библиотеки Agda [18].

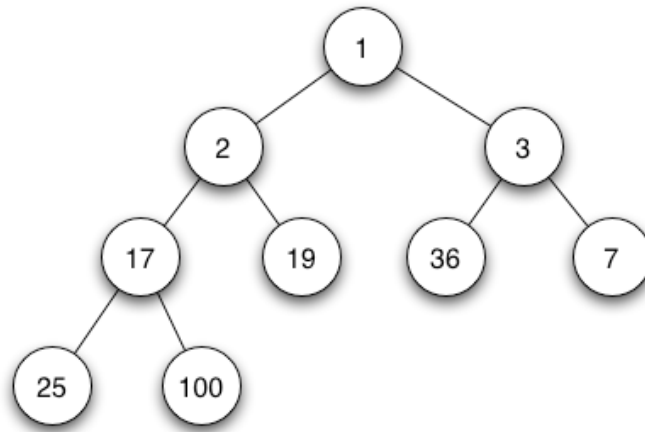


Рис. 2.1. Пример заполненной кучи для минимума

```
module HeapModule where
```

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

```
data ⊥ : Set where
```

```
module Level where
```

```
  postulate Level : Set
```

```
  postulate lzero : Level
```

```
  postulate lsucc : Level → Level
```

```
  postulate _⊔_ : Level → Level → Level
```

```
  infixl 6 _⊔_
```

```
  {-# BUILTIN LEVEL Level #-}
```

```
  {-# BUILTIN LEVELZERO lzero #-}
```

```
  {-# BUILTIN LEVELSUC lsucc #-}
```

```
  {-# BUILTIN LEVELMAX _⊔_ #-}
```

```
open Level
```

```
module Function where
```

Композиция функций.

```

_◦_ : ∀ {a b c}
  → {A : Set a} {B : Set b} {C : Set c}
  → (B → C) → (A → B) → (A → C)
f ◦ g = λ x → f (g x)
flip : ∀ {a b c}
  → {A : Set a} {B : Set b} {C : A → B → Set c}
  → ((x : A) → (y : B) → C x y)
  → ((y : B) → (x : A) → C x y)
flip f x y = f y x

```

```

open Function public
module Logic where

```

Из элемента пустого типа следует что-угодно.

```

⊥-elim : ∀ {a} {Whatever : Set a} → ⊥ → Whatever
⊥-elim ()

```

Логическое отрицание.

```

¬ : ∀ {a} → Set a → Set a
¬ P = P → ⊥

```

```

private
module DummyAB {a b} {A : Set a} {B : Set b} where

```

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

```

contradiction : A → ¬ A → B

```

contradiction $a \neg a = \perp\text{-elim } (\neg a a)$

Контрапозиция

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

contraposition = flip $_ \circ _$

open DummyAB public

open Logic public

Определения интуиционистской теории типов.

module MLTT where

Пропозициональное равенство из интуиционистской теории типов [3].

infix 4 $_ \equiv _$

data $_ \equiv _$ {a} {A : Set a} (x : A) : A → Set a where

refl : $x \equiv x$

{-# BUILTIN EQUALITY $_ \equiv _$ #-}

{-# BUILTIN REFL refl #-}

Тип-сумма — зависимая пара.

record Σ {a b} (A : Set a) (B : A → Set b) : Set (a \sqcup b) where

constructor $_,_$

field fst : A ; snd : B fst

open Σ public

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

$_ \times _$: $\forall \{a b\} (A : Set a) \rightarrow (B : Set b) \rightarrow Set (a \sqcup b)$

$A \times B = \Sigma A (\lambda _ \rightarrow B)$

```
infixr 5 _×_ _,_
```

```
module ≡-Prop where
```

```
private
```

```
module DummyA {a b} {A : Set a} {B : Set b} where
```

```
-- _≡_ is congruent
```

Конгруэнтность пропозиционального равенства.

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

```
cong f refl = refl
```

```
open DummyA public
```

```
open ≡-Prop public
```

```
open MLTT public
```

2.3.2. Определение отношений и доказательства их свойств

Чтобы задать порядок элементов в куче, нужно уметь сравнивать элементы. Зададим 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
```

Задавать высоту кучи будем натуральными числами.

```
data N : Set where
  zero : N
  succ : N → N
{-# BUILTIN NATURAL N #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC succ #-}
```

Тип данных для отношения меньше или равно на натуральных числах.

```
data _N≤_ : Rel2 N where
  z≤n : ∀ {n} → zero N≤ n
```

$$s \leq s : \forall \{n\} \{m\} \rightarrow n \mathbb{N} \leq m \rightarrow \text{succ } n \mathbb{N} \leq \text{succ } m$$

Все остальные отношения определяются через $_ \mathbb{N} \leq _$.

$$_ \mathbb{N} < _ _ \mathbb{N} \geq _ _ \mathbb{N} > _ : \text{Rel}_2 \mathbb{N}$$

$$n \mathbb{N} < m = \text{succ } n \mathbb{N} \leq m$$

$$n \mathbb{N} > m = m \mathbb{N} < n$$

$$n \mathbb{N} \geq m = m \mathbb{N} \leq n$$

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

$$\text{lemma-succ-}\equiv : \forall \{n\} \{m\} \rightarrow \text{succ } n \equiv \text{succ } m \rightarrow n \equiv m$$

$$\text{lemma-succ-}\equiv \text{ refl} = \text{refl}$$

$$\text{lemma-succ-}\leq : \forall \{n\} \{m\} \rightarrow \text{succ } (\text{succ } n) \mathbb{N} \leq \text{succ } m \rightarrow \text{succ } n \mathbb{N} \leq m$$

$$\text{lemma-succ-}\leq (s \leq s \ r) = r$$

$$\text{cmpN} : \text{Cmp } \{\mathbb{N}\} _ \mathbb{N} < _ \equiv _$$

$$\text{cmpN } \text{zero } (\text{zero}) = \text{tri} = (\lambda ()) \text{ refl } (\lambda ())$$

$$\text{cmpN } \text{zero } (\text{succ } y) = \text{tri} < (s \leq s \ z \leq n) (\lambda ()) (\lambda ())$$

$$\text{cmpN } (\text{succ } x) \text{ zero} = \text{tri} > (\lambda ()) (\lambda ()) (s \leq s \ z \leq n)$$

$$\text{cmpN } (\text{succ } x) (\text{succ } y) \text{ with } \text{cmpN } x \ y$$

$$\dots \mid \text{tri} < \ a \ \neg b \ \neg c = \text{tri} < (s \leq s \ a) (\text{contraposition lemma-succ-}\equiv \ \neg b)$$

$$(\text{contraposition lemma-succ-}\leq \ \neg c)$$

$$\dots \mid \text{tri} > \ \neg a \ \neg b \ c = \text{tri} > (\text{contraposition lemma-succ-}\leq \ \neg a)$$

$$(\text{contraposition lemma-succ-}\equiv \ \neg b) (s \leq s \ c)$$

$$\dots \mid \text{tri} = \ \neg a \ b \ \neg c = \text{tri} = (\text{contraposition lemma-succ-}\leq \ \neg a)$$

$$(\text{cong succ } b) (\text{contraposition lemma-succ-}\leq \ \neg c)$$

Транзитивность отношения.

$\text{Trans} : \{A : \text{Set}\} \rightarrow \text{Rel}_2 A \rightarrow \text{Set}$

$\text{Trans } \{A\} \text{ _rel_} = \{a \ b \ c : A\} \rightarrow (a \text{ rel } b) \rightarrow (b \text{ rel } c) \rightarrow (a \text{ rel } c)$

Симметричность отношения.

$\text{Symmetric} : \forall \{A : \text{Set}\} \rightarrow \text{Rel}_2 A \rightarrow \text{Set}$

$\text{Symmetric } \text{ _rel_} = \forall \{a \ b\} \rightarrow a \text{ rel } b \rightarrow b \text{ rel } a$

Предикат P учитывает (соблюдает) отношение _rel_ .

$\text{ _Respects_} : \forall \{\ell\} \{A : \text{Set}\} \rightarrow (A \rightarrow \text{Set } \ell) \rightarrow \text{Rel}_2 A \rightarrow \text{Set } _$

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

Отношение P соблюдает отношение _rel_ .

$\text{ _Respects}_2 : \forall \{A : \text{Set}\} \rightarrow \text{Rel}_2 A \rightarrow \text{Rel}_2 A \rightarrow \text{Set}$

$P \text{ Respects}_2 \text{ _rel_} =$

$(\forall \{x\} \rightarrow P \ x \ \text{Respects } \text{ _rel_}) \times$

$(\forall \{y\} \rightarrow \text{flip } P \ y \ \text{Respects } \text{ _rel_})$

Тип данных для обобщенного отношения меньше или равно.

$\text{data } \text{ _<=} \{A : \text{Set}\} \{ \text{ _<_} : \text{Rel}_2 A \} \{ \text{ _==_} : \text{Rel}_2 A \} : \text{Rel}_2 A \text{ where}$

$\text{le} : \forall \{x \ y\} \rightarrow x < y \rightarrow x \text{ _<=} y$

$\text{eq} : \forall \{x \ y\} \rightarrow x == y \rightarrow x \text{ _<=} y$

Обобщенные функции минимум и максимум.

$\text{min max} : \{A : \text{Set}\} \{ \text{ _<_} : \text{Rel}_2 A \} \{ \text{ _==_} : \text{Rel}_2 A \}$

$\rightarrow (\text{cmp} : \text{Cmp } \text{ _<_} \text{ _==_}) \rightarrow A \rightarrow A \rightarrow A$

$\text{min } \text{cmp } x \ y \text{ with } \text{cmp } x \ y$

$\dots \mid \text{tri<} \text{ _ _ _} = x$

... | _ = y
max cmp x y with cmp x y
... | tri> _ _ _ = x
... | _ = y

Лемма: элемент меньше или равный двух других элементов меньше или равен минимуму 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))

lemma-<=min { cmp = cmp } { _ } { b } { c } ab ac with cmp b c
... | tri< _ _ _ = ab
... | tri= _ _ _ = ac
... | tri> _ _ _ = ac

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 `_<=_` с `_==_` соблюдает отношение равенства 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)

```

2.4. Модуль Heap

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

```

2.4.1. Расширение исходного типа

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 -- 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
trans<E {# _} {top} {# _} _ _ ()
trans<E {top} {# _} {# _} _ _ ()
```

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.

$\text{sym}=\text{E} : \text{Symmetric } _=\text{E}_\text{}$
 $\text{sym}=\text{E} (\text{base } a=b) = \text{base } (\text{sym}== a=b)$
 $\text{sym}=\text{E } \text{ext} = \text{ext}$
 $\text{trans}=\text{E} : \text{Trans } _=\text{E}_\text{}$
 $\text{trans}=\text{E} (\text{base } a=b) (\text{base } b=c) = \text{base } (\text{trans}== a=b b=c)$
 $\text{trans}=\text{E } \text{ext } \text{ext} = \text{ext}$

Отношение $_<\text{E}_\text{}$ соблюдает отношение $_=\text{E}_\text{}$.

$\text{respE} : _<\text{E}_\text{ } \text{Respects}_2 _=\text{E}_\text{}$
 $\text{respE} = \text{left} , \text{right } \text{where}$
 $\text{left} : \forall \{a \ b \ c : \text{expanded } A\} \rightarrow b =\text{E } c \rightarrow a <\text{E } b \rightarrow a <\text{E } c$
 $\text{left } \{ \# _ \} \{ \# _ \} \{ \# _ \} (\text{base } r1) (\text{base } r2) = \text{base } (\text{fst } \text{resp } r1 \ r2)$
 $\text{left } \{ \# _ \} \{ \text{top} \} \{ \text{top} \} \text{ext } \text{ext} = \text{ext}$

 $\text{left } \{ _ \} \{ \# _ \} \{ \text{top} \} () _$
 $\text{left } \{ _ \} \{ \text{top} \} \{ \# _ \} () _$
 $\text{left } \{ \text{top} \} \{ _ \} \{ _ \} _ ()$

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

 $\text{right } \{ _ \} \{ \# _ \} \{ \text{top} \} () _$
 $\text{right } \{ _ \} \{ \text{top} \} \{ _ \} _ ()$

Отношение меньше-равно для расширенного типа.

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

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
```

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.

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

2.4.2. Тип данных Heap

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

$\text{data HeapState} : \text{Set where}$
 $\text{full almost} : \text{HeapState}$

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

$\text{data Heap} : (\text{expanded } A) \rightarrow (h : \mathbb{N}) \rightarrow \text{HeapState} \rightarrow \text{Set where}$

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

$\text{eh} : \text{Heap top zero full}$

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 .

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

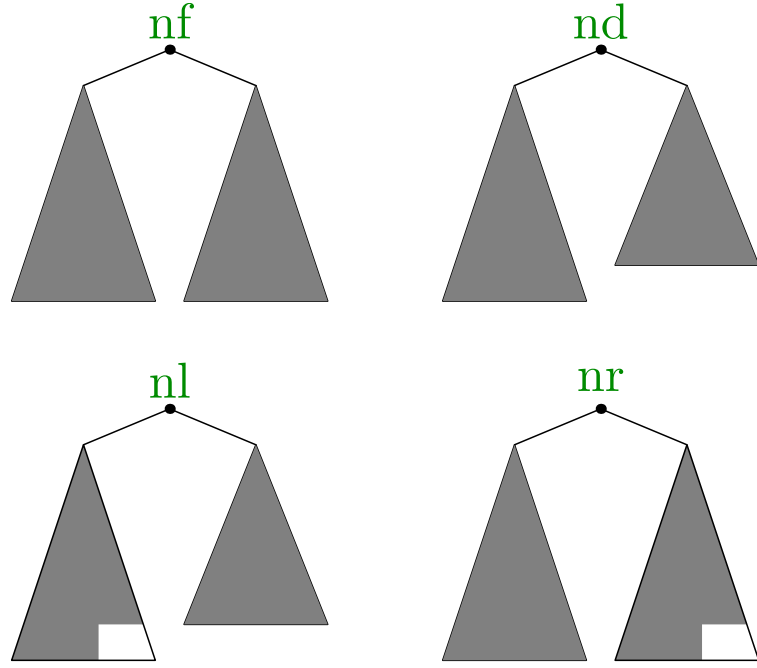


Рис. 2.2. Конструкторы типа данных `Heap`

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 $n + 1$.

```

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

```

Замечание: высота любой неполной кучи больше нуля.

```

lemma-almost-height : ∀ {m h} → Heap m h almost → h ℕ > 0

```

```

lemma-almost-height (nd _ _ _ _ _) = s ≤ s z ≤ n

```

```

lemma-almost-height (nl _ _ _ _ _) = s ≤ s z ≤ n

```

```

lemma-almost-height (nr _ _ _ _ _) = s ≤ s z ≤ n

```

Функция — просмотр минимума в куче.

```
peekMin : ∀ {m h s} → Heap m h s → (expanded A)
peekMin eh = top
peekMin (nd p _ _ _ _) = # p
peekMin (nf p _ _ _ _) = # p
peekMin (nl p _ _ _ _) = # p
peekMin (nr p _ _ _ _) = # p
```

2.4.3. Функции вставки в кучу

Функция вставки элемента в полную кучу.

```
finsert : ∀ {h m} → (z : A) → Heap m h full
→ Σ HeapState (Heap (minE m (# z)) (succ h))
finsert {0} z eh = full , nf z (le ext) (le ext) eh eh
finsert {1} z (nf p i j eh eh) with cmp p z
... | tri < p < z _ _ = almost ,
    nd p (le (base p < z)) j (nf z (le ext) (le ext) eh eh) eh
... | tri = _ p = z _ _ = almost ,
    nd z (eq (base (sym == p = z))) (le ext) (nf p i j eh eh) eh
... | tri > _ _ z < p = almost ,
    nd z (le (base z < p)) (le ext) (nf p i j eh eh) eh
finsert z (nf p i j (nf x il jl a b) c) with cmp p z
finsert z (nf p i j (nf x il jl a b) c) | tri < p < z _ _
    with finsert z (nf x il jl a b)
    | lemma-<=minE {# p} {# x} {# z} i (le (base p < z))
... | full , newleft | ll = almost , nd p ll j newleft c
... | almost , newleft | ll = almost , nl p ll j newleft c
finsert z (nf p i j (nf x il jl a b) c) | tri = _ p = z _
    with finsert p (nf x il jl a b)
```

```

| lemma-<=minE {# z} {# x} {# p}
  (snd resp≤ (base p=z) i) (eq (base (sym== p=z)))
| snd resp≤ (base p=z) j
... | full , newleft | l1 | l2 = almost , nd z l1 l2 newleft c
... | almost , newleft | l1 | l2 = almost , nl z l1 l2 newleft c

```

TODO из-за непонятного бага в LaTeX некоторые строки на Agda не отрендерены

Вставка элемента в неполную кучу.

```

ainsert : ∀ {h m} → (z : A) → Heap m h almost
  → Σ HeapState (Heap (minE m (# z)) h)
ainsert z (nd p i j a b) with cmp p z
ainsert z (nd p i j a b) | tri< p<z _ _
  with fininsert z b | lemma-<=minE j (le (base p<z))
... | full , nb | l1 = full , nf p i l1 a nb
... | almost , nb | l1 = almost , nr p i l1 a nb

```

2.4.4. Удаление минимума из полной кучи

Вспомогательный тип данных.

```

data OR (A B : Set) : Set where
  orA : A → OR A B
  orB : B → OR A B

```

Слияние двух полных куч одной высоты.

```

fmerge : ∀ {x y h} → Heap x h full → Heap y h full
  → OR (Heap x zero full × (x ≡ y) × (h ≡ zero))
  (Heap (minE x y) (succ h) almost)

```

$\text{fmerge } \text{eh } \text{eh} = \text{orA } (\text{eh} , \text{refl} , \text{refl})$
 $\text{fmerge } (\text{nf } x \ i_1 \ j_1 \ a \ b) (\text{nf } y \ i_2 \ j_2 \ c \ d) \text{ with } \text{cmp } x \ y$
 $\text{fmerge } (\text{nf } x \ i_1 \ j_1 \ a \ b) (\text{nf } y \ i_2 \ j_2 \ c \ d) \mid \text{tri} < x < y _ _ \text{ with } \text{fmerge } a \ b$
 $\dots \mid \text{orA } (\text{eh} , \text{refl} , \text{refl}) = \text{orB } (\text{nd } x \ (\text{le } (\text{base } x < y)) \ j_1 (\text{nf } y \ i_2 \ j_2 \ c \ d) \ \text{eh})$
 $\dots \mid \text{orB } ab = \text{orB}$
 $(\text{nr } x \ (\text{le } (\text{base } x < y))) (\text{lemma-} \leq \text{minE } i_1 \ j_1) (\text{nf } y \ i_2 \ j_2 \ c \ d) \ ab)$

$\text{fmerge } (\text{nf } x \ i_1 \ j_1 \ a \ b) (\text{nf } y \ i_2 \ j_2 \ c \ d) \mid \text{tri} > _ _ y < x \text{ with } \text{fmerge } c \ d$
 $\dots \mid \text{orA } (\text{eh} , \text{refl} , \text{refl}) = \text{orB } (\text{nd } y \ (\text{le } (\text{base } y < x)) \ j_2 (\text{nf } x \ i_1 \ j_1 \ a \ b) \ \text{eh})$
 $\dots \mid \text{orB } cd = \text{orB}$
 $(\text{nr } y \ (\text{le } (\text{base } y < x))) (\text{lemma-} \leq \text{minE } i_2 \ j_2) (\text{nf } x \ i_1 \ j_1 \ a \ b) \ cd)$

Извлечение минимума из полной кучи.

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

2.4.5. Удаление минимума из неполной кучи

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 **nd**.

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

$\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \text{ with } \text{cmp } x \ y$
 $\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} < x < y _ _ \text{ with } \text{fmerge } a \ b$
 $\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} < x < y _ _ \mid \text{orA } (_, _, ())$
 $\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} < x < y _ _ \mid \text{orB } x_l =$
 $\text{nl } x (\text{lemma-}\leq\text{minE } i \ j) (\text{le } (\text{base } x < y)) x_l (\text{nf } y \ i_l \ j_l \ c \ d)$

$\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} = _ \ x = y _ \text{ with } \text{fmerge } c \ d$
 $\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} = _ \ x = y _ \mid \text{orA } (\text{eh}, \text{refl}, \text{refl})$
 $\text{with } \text{fmerge } a \ b$
 $\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} = _ \ x = y _ \mid \text{orA } (\text{eh}, \text{refl}, \text{refl})$
 $\mid \text{orA } (\text{eh}, \text{refl}, ())$
 $\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} = _ \ x = y _ \mid \text{orA } (\text{eh}, \text{refl}, \text{refl})$
 $\mid \text{orB } ab = \text{nl } y (\text{lemma-resp } x=y \ i \ j) (\text{eq } (\text{base } (\text{sym} == x=y)))$
 $ab (\text{nf } x (\text{le } \text{ext}) (\text{le } \text{ext}) \text{eh } \text{eh})$

$\text{ndmerge } (\text{nf } x \ i \ j \ a \ b) (\text{nf } y \ i_l \ j_l \ c \ d) \mid \text{tri} = _ \ x = y _ \mid \text{orB } cd \text{ with } \text{fmerge } a \ b$


```

ndmerge (nf x i j a b) (nf y il jl c d) | tri= _ x=y _ | orB cd | orA ( _ , _ , ())
ndmerge (nf x i j a b) (nf y il jl c d) | tri= _ x=y _ | orB cd | orB ab =
  nl y (lemma-resp x=y i j) (lemma-<=min3E il jl (eq (base (sym== x=y))))
  ab (makeH x c d)
ndmerge (nf x i j a b) (nf y il jl c d) | tri> _ _ y<x with fmerge a b
ndmerge (nf x i j a b) (nf y il jl c d) | tri> _ _ y<x | orA ( _ , _ , ())
ndmerge (nf x i j a b) (nf y il jl c d) | tri> _ _ y<x | orB ab =
  nl y (lemma-trans y<x i j) (lemma-<=min3E il jl (le (base y<x)))
  ab (makeH x c d)

```

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

```

afmerge : ∀ {h x y} → Heap x (succ (succ h)) almost
→ OR (Heap y (succ h) full) (Heap y (succ (succ h)) full)
→ OR (Heap (minE x y) (succ (succ h)) full)
  (Heap (minE x y) (succ (succ (succ h))) almost)

afmerge (nd x i j (nf p il jl eh eh) eh) (orA (nf y i2 j2 eh eh)) with cmp x y
... | tri< x<y _ _ = orA (nf x i (le (base x<y))
  (nf p il jl eh eh) (nf y i2 j2 eh eh))
... | tri= _ x=y _ = orA (nf y (eq (base (sym== x=y)))
  (snd resp≤ (base x=y) i) (nf x (le ext) (le ext) eh eh) (nf p il jl eh eh))
... | tri> _ _ y<x = orA (nf y (le (base y<x))
  (trans≤ (le (base y<x)) i) (nf x j j eh eh) (nf p jl jl eh eh))

afmerge (nd x i j (nf p1 il jl a1 b1) (nf p2 i2 j2 a2 b2)) (orA (nf y i3 j3 c d))
  with cmp x y | ndmerge (nf p1 il jl a1 b1) (nf p2 i2 j2 a2 b2)
... | tri< x<y _ _ | ab = orB (nl x (lemma-<=minE i j) (le (base x<y))
  ab (nf y i3 j3 c d))
... | tri= _ x=y _ | ab = orB (nl y (lemma-resp x=y i j)

```

```

(lemma-<=min3E i3 j3 (eq (base (sym== x=y)))) ab (makeH x c d))
... | tri> _ _ y<x | ab = orB (nl y (lemma-trans y<x i j)
(lemma-<=min3E i3 j3 (le (base y<x)))) ab (makeH x c d))
afmerge (nl x i j (nd p1 i1 j1 a1 b1) (nf p2 i2 j2 a2 b2)) (orA (nf y i3 j3 c d))
with cmp x y | afmerge (nd p1 i1 j1 a1 b1) (orA (nf p2 i2 j2 a2 b2))
... | tri< x<y _ _ | orA ab =
orA (nf x (lemma-<=minE i j) (le (base x<y))) ab (nf y i3 j3 c d))
... | tri< x<y _ _ | orB ab =
orB (nl x (lemma-<=minE i j) (le (base x<y))) ab (nf y i3 j3 c d))
... | tri= _ x=y _ | orA ab = orA
(nf y (lemma-resp x=y i j) (lemma-<=min3E i3 j3 (eq (base (sym== x=y))))
ab (makeH x c d))
... | tri= _ x=y _ | orB ab = orB
(nl y (lemma-resp x=y i j) (lemma-<=min3E i3 j3 (eq (base (sym== x=y))))
ab (makeH x c d))
... | tri> _ _ y<x | orA ab = orA
(nf y (lemma-trans y<x i j) (lemma-<=min3E i3 j3 (le (base y<x))))
ab (makeH x c d))
... | tri> _ _ y<x | orB ab = orB
(nl y (lemma-trans y<x i j) (lemma-<=min3E i3 j3 (le (base y<x))))
ab (makeH x c d))

afmerge (nl x i j (nl p1 i1 j1 a1 b1) (nf p2 i2 j2 a2 b2)) (orA (nf y i3 j3 c d))
with cmp x y | afmerge (nl p1 i1 j1 a1 b1) (orA (nf p2 i2 j2 a2 b2))
... | tri< x<y _ _ | orA ab =
orA (nf x (lemma-<=minE i j) (le (base x<y))) ab (nf y i3 j3 c d))
... | tri< x<y _ _ | orB ab =
orB (nl x (lemma-<=minE i j) (le (base x<y))) ab (nf y i3 j3 c d))
... | tri= _ x=y _ | orA ab = orA (nf y (lemma-resp x=y i j)
(lemma-<=min3E i3 j3 (eq (base (sym== x=y)))) ab (makeH x c d))

```

... | tri= _ x=y _ | orB ab = orB (nl y (lemma-resp x=y i j)
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri> _ _ y<x | orA ab = orA (nf y (lemma-trans y<x i j)
 (lemma-<=min3E i₃ j₃ (le (base y<x)))) ab (makeH x c d))
 ... | tri> _ _ y<x | orB ab = orB (nl y (lemma-trans y<x i j)
 (lemma-<=min3E i₃ j₃ (le (base y<x)))) ab (makeH x c d))

afmerge (nl x i j (nr p₁ i₁ j₁ a₁ b₁) (nf p₂ i₂ j₂ a₂ b₂)) (orA (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nr p₁ i₁ j₁ a₁ b₁) (orA (nf p₂ i₂ j₂ a₂ b₂))
 ... | tri< x<y _ _ | orA ab =
 orA (nf x (lemma-<=minE i j) (le (base x<y)) ab (nf y i₃ j₃ c d))
 ... | tri< x<y _ _ | orB ab =
 orB (nl x (lemma-<=minE i j) (le (base x<y)) ab (nf y i₃ j₃ c d))
 ... | tri= _ x=y _ | orA ab = orA (nf y (lemma-resp x=y i j)
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri= _ x=y _ | orB ab = orB (nl y (lemma-resp x=y i j)
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri> _ _ y<x | orA ab = orA (nf y (lemma-trans y<x i j)
 (lemma-<=min3E i₃ j₃ (le (base y<x)))) ab (makeH x c d))
 ... | tri> _ _ y<x | orB ab = orB (nl y (lemma-trans y<x i j)
 (lemma-<=min3E i₃ j₃ (le (base y<x)))) ab (makeH x c d))

afmerge (nr x i j (nf p₁ i₁ j₁ a₁ b₁) (nd p₂ i₂ j₂ a₂ b₂)) (orA (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nd p₂ i₂ j₂ a₂ b₂) (orB (nf p₁ i₁ j₁ a₁ b₁))
 ... | tri< x<y _ _ | (orA ab) =
 orA (nf x (le (base x<y)) (lemma-<=minE j i) (nf y i₃ j₃ c d) ab)
 ... | tri< x<y _ _ | (orB ab) =
 orB (nl x (lemma-<=minE j i) (le (base x<y)) ab (nf y i₃ j₃ c d))
 ... | tri= _ x=y _ | (orA ab) = orA (nf y (lemma-resp x=y j i)
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))

... | tri= _ x=y _ | (orB ab) = orB (nl y (lemma-resp x=y j i))
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri> _ _ y<x | (orA ab) = orA (nf y (lemma-trans y<x j i))
 (lemma-<=min3E i₃ j₃ (le (base y<x))) ab (makeH x c d))
 ... | tri> _ _ y<x | (orB ab) = orB (nl y (lemma-trans y<x j i))
 (lemma-<=min3E i₃ j₃ (le (base y<x))) ab (makeH x c d))

afmerge (nr x i j (nf p₁ i₁ j₁ a₁ b₁) (nl p₂ i₂ j₂ a₂ b₂)) (orA (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nl p₂ i₂ j₂ a₂ b₂) (orB (nf p₁ i₁ j₁ a₁ b₁))
 ... | tri< x<y _ _ | (orA ab) =
 orA (nf x (le (base x<y))) (lemma-<=minE j i) (nf y i₃ j₃ c d) ab)
 ... | tri< x<y _ _ | (orB ab) =
 orB (nl x (lemma-<=minE j i) (le (base x<y))) ab (nf y i₃ j₃ c d))
 ... | tri= _ x=y _ | (orA ab) = orA (nf y (lemma-resp x=y j i))
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri= _ x=y _ | (orB ab) = orB (nl y (lemma-resp x=y j i))
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri> _ _ y<x | (orA ab) = orA (nf y (lemma-trans y<x j i))
 (lemma-<=min3E i₃ j₃ (le (base y<x))) ab (makeH x c d))
 ... | tri> _ _ y<x | (orB ab) = orB (nl y (lemma-trans y<x j i))
 (lemma-<=min3E i₃ j₃ (le (base y<x))) ab (makeH x c d))

afmerge (nr x i j (nf p₁ i₁ j₁ a₁ b₁) (nr p₂ i₂ j₂ a₂ b₂)) (orA (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nr p₂ i₂ j₂ a₂ b₂) (orB (nf p₁ i₁ j₁ a₁ b₁))
 ... | tri< x<y _ _ | (orA ab) =
 orA (nf x (le (base x<y))) (lemma-<=minE j i) (nf y i₃ j₃ c d) ab)
 ... | tri< x<y _ _ | (orB ab) =
 orB (nl x (lemma-<=minE j i) (le (base x<y))) ab (nf y i₃ j₃ c d))
 ... | tri= _ x=y _ | (orA ab) = orA (nf y (lemma-resp x=y j i))
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))

... | tri= _ x=y _ | (orB ab) = orB (nl y (lemma-resp x=y j i))
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) ab (makeH x c d))
 ... | tri> _ _ y<x | (orA ab) = orA (nf y (lemma-trans y<x j i))
 (lemma-<=min3E i₃ j₃ (le (base y<x))) ab (makeH x c d))
 ... | tri> _ _ y<x | (orB ab) = orB (nl y (lemma-trans y<x j i))
 (lemma-<=min3E i₃ j₃ (le (base y<x))) ab (makeH x c d))

afmerge (nd x i j (nf p i₁ j₁ eh eh) eh) (orB (nf y i₂ j₂ c d)) with cmp x y
 ... | tri< x<y _ _ =
 orB (nd x (le (base x<y)) i (nf y i₂ j₂ c d) (nf p i₁ j₁ eh eh))
 ... | tri= _ x=y _ = orB (nd y
 (lemma-<=min3E i₂ j₂ (eq (base (sym== x=y)))) (snd resp≤ (base x=y) i)
 (makeH x c d) (nf p i₁ j₁ eh eh))
 ... | tri> _ _ y<x = orB (nd y (lemma-<=min3E i₂ j₂ (le (base y<x)))
 (trans≤ (le (base y<x)) i) (makeH x c d) (nf p i₁ j₁ eh eh))

afmerge (nd x i j (nf p₁ i₁ j₁ a₁ b₁) (nf p₂ i₂ j₂ a₂ b₂)) (orB (nf y i₃ j₃ c d))
 with cmp x y | ndmerge (nf p₁ i₁ j₁ a₁ b₁) (nf p₂ i₂ j₂ a₂ b₂)
 ... | tri< x<y _ _ | ab =
 orB (nr x (le (base x<y)) (lemma-<=minE i j) (nf y i₃ j₃ c d) ab)
 ... | tri= _ x=y _ | ab = orB (nr y
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y))))
 (lemma-resp x=y i j) (makeH x c d) ab)
 ... | tri> _ _ y<x | ab = orB (nr y
 (lemma-<=min3E i₃ j₃ (le (base y<x)))
 (lemma-trans y<x i j) (makeH x c d) ab)

afmerge (nl x i j (nd p₁ i₁ j₁ a₁ b₁) (nf p₂ i₂ j₂ a₂ b₂)) (orB (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nd p₁ i₁ j₁ a₁ b₁) (orA (nf p₂ i₂ j₂ a₂ b₂))
 ... | tri< x<y _ _ | (orA ab) = orB (nd x (le (base x<y))

(lemma-<=minE i j) (nf y i₃ j₃ c d) ab)
 ... | tri< x<y _ _ | (orB ab) = orB (nr x (le (base x<y)))
 (lemma-<=minE i j) (nf y i₃ j₃ c d) ab)
 ... | tri= _ x=y _ | (orA ab) = orB (nd y
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y))))) (lemma-resp x=y i j)
 (makeH x c d) ab)
 ... | tri= _ x=y _ | (orB ab) = orB (nr y
 (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y))))) (lemma-resp x=y i j)
 (makeH x c d) ab)
 ... | tri> _ _ y<x | (orA ab) = orB (nd y
 (lemma-<=min3E i₃ j₃ (le (base y<x)))) (lemma-trans y<x i j) (makeH x c d) ab)
 ... | tri> _ _ y<x | (orB ab) = orB (nr y
 (lemma-<=min3E i₃ j₃ (le (base y<x)))) (lemma-trans y<x i j) (makeH x c d) ab)
 afmerge (nl x i j (nl p₁ i₁ j₁ a₁ b₁) (nf p₂ i₂ j₂ a₂ b₂)) (orB (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nl p₁ i₁ j₁ a₁ b₁) (orA (nf p₂ i₂ j₂ a₂ b₂))
 ... | tri< x<y _ _ | (orA ab) = orB (nd x (le (base x<y)))
 (lemma-<=minE i j) (nf y i₃ j₃ c d) ab)
 ... | tri< x<y _ _ | (orB ab) = orB (nr x (le (base x<y)))
 (lemma-<=minE i j) (nf y i₃ j₃ c d) ab)
 ... | tri= _ x=y _ | (orA ab) = orB
 (nd y (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))))
 (lemma-resp x=y i j) (makeH x c d) ab)
 ... | tri= _ x=y _ | (orB ab) = orB
 (nr y (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))))
 (lemma-resp x=y i j) (makeH x c d) ab)
 ... | tri> _ _ y<x | (orA ab) = orB
 (nd y (lemma-<=min3E i₃ j₃ (le (base y<x))))
 (lemma-trans y<x i j) (makeH x c d) ab)
 ... | tri> _ _ y<x | (orB ab) = orB
 (nr y (lemma-<=min3E i₃ j₃ (le (base y<x))))

$(\text{lemma-trans } y < x \ i \ j) (\text{makeH } x \ c \ d) \ ab)$

$\text{afmerge } (\text{nl } x \ i \ j (\text{nr } p_1 \ i_1 \ j_1 \ a_1 \ b_1) (\text{nf } p_2 \ i_2 \ j_2 \ a_2 \ b_2)) (\text{orB } (\text{nf } y \ i_3 \ j_3 \ c \ d))$
 $\text{with } \text{cmp } x \ y \mid \text{afmerge } (\text{nr } p_1 \ i_1 \ j_1 \ a_1 \ b_1) (\text{orA } (\text{nf } p_2 \ i_2 \ j_2 \ a_2 \ b_2))$
 $\dots \mid \text{tri} < x < y \ _ _ \mid (\text{orA } ab) = \text{orB}$
 $(\text{nd } x (\text{le } (\text{base } x < y))) (\text{lemma-} \leq \text{minE } i \ j) (\text{nf } y \ i_3 \ j_3 \ c \ d) \ ab)$
 $\dots \mid \text{tri} < x < y \ _ _ \mid (\text{orB } ab) = \text{orB}$
 $(\text{nr } x (\text{le } (\text{base } x < y))) (\text{lemma-} \leq \text{minE } i \ j) (\text{nf } y \ i_3 \ j_3 \ c \ d) \ ab)$
 $\dots \mid \text{tri} = _ \ x = y \ _ \mid (\text{orA } ab) = \text{orB}$
 $(\text{nd } y (\text{lemma-} \leq \text{min3E } i_3 \ j_3 (\text{eq } (\text{base } (\text{sym} == x = y))))$
 $(\text{lemma-resp } x = y \ i \ j) (\text{makeH } x \ c \ d) \ ab)$
 $\dots \mid \text{tri} = _ \ x = y \ _ \mid (\text{orB } ab) = \text{orB}$
 $(\text{nr } y (\text{lemma-} \leq \text{min3E } i_3 \ j_3 (\text{eq } (\text{base } (\text{sym} == x = y))))$
 $(\text{lemma-resp } x = y \ i \ j) (\text{makeH } x \ c \ d) \ ab)$
 $\dots \mid \text{tri} > _ _ \ y < x \mid (\text{orA } ab) = \text{orB}$
 $(\text{nd } y (\text{lemma-} \leq \text{min3E } i_3 \ j_3 (\text{le } (\text{base } y < x))))$
 $(\text{lemma-trans } y < x \ i \ j) (\text{makeH } x \ c \ d) \ ab)$
 $\dots \mid \text{tri} > _ _ \ y < x \mid (\text{orB } ab) = \text{orB}$
 $(\text{nr } y (\text{lemma-} \leq \text{min3E } i_3 \ j_3 (\text{le } (\text{base } y < x))))$
 $(\text{lemma-trans } y < x \ i \ j) (\text{makeH } x \ c \ d) \ ab)$

$\text{afmerge } (\text{nr } x \ i \ j (\text{nf } p_1 \ i_1 \ j_1 \ a_1 \ b_1) (\text{nd } p_2 \ i_2 \ j_2 \ a_2 \ b_2)) (\text{orB } (\text{nf } y \ i_3 \ j_3 \ c \ d))$
 $\text{with } \text{cmp } x \ y \mid \text{afmerge } (\text{nd } p_2 \ i_2 \ j_2 \ a_2 \ b_2) (\text{orB } (\text{nf } p_1 \ i_1 \ j_1 \ a_1 \ b_1))$
 $\dots \mid \text{tri} < x < y \ _ _ \mid (\text{orA } ab) = \text{orB}$
 $(\text{nd } x (\text{le } (\text{base } x < y))) (\text{lemma-} \leq \text{minE } j \ i) (\text{nf } y \ i_3 \ j_3 \ c \ d) \ ab)$
 $\dots \mid \text{tri} < x < y \ _ _ \mid (\text{orB } ab) = \text{orB}$
 $(\text{nr } x (\text{le } (\text{base } x < y))) (\text{lemma-} \leq \text{minE } j \ i) (\text{nf } y \ i_3 \ j_3 \ c \ d) \ ab)$
 $\dots \mid \text{tri} = _ \ x = y \ _ \mid (\text{orA } ab) = \text{orB}$
 $(\text{nd } y (\text{lemma-} \leq \text{min3E } i_3 \ j_3 (\text{eq } (\text{base } (\text{sym} == x = y))))$
 $(\text{lemma-resp } x = y \ j \ i) (\text{makeH } x \ c \ d) \ ab)$

... | tri= _ x=y _ | (orB ab) = orB
 (nr y (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y))))
 (lemma-resp x=y j i) (makeH x c d) ab)
 ... | tri> _ _ y<x | (orA ab) = orB
 (nd y (lemma-<=min3E i₃ j₃ (le (base y<x)))) (lemma-trans y<x j i)
 (makeH x c d) ab)
 ... | tri> _ _ y<x | (orB ab) = orB
 (nr y (lemma-<=min3E i₃ j₃ (le (base y<x)))) (lemma-trans y<x j i)
 (makeH x c d) ab)
 afmerge (nr x i j (nf p₁ i₁ j₁ a₁ b₁) (nl p₂ i₂ j₂ a₂ b₂)) (orB (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nl p₂ i₂ j₂ a₂ b₂) (orB (nf p₁ i₁ j₁ a₁ b₁))
 ... | tri< x<y _ _ | (orA ab) = orB
 (nd x (le (base x<y))) (lemma-<=minE j i) (nf y i₃ j₃ c d) ab)
 ... | tri< x<y _ _ | (orB ab) = orB
 (nr x (le (base x<y))) (lemma-<=minE j i) (nf y i₃ j₃ c d) ab)
 ... | tri= _ x=y _ | (orA ab) = orB
 (nd y (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) (lemma-resp x=y j i)
 (makeH x c d) ab)
 ... | tri= _ x=y _ | (orB ab) = orB
 (nr y (lemma-<=min3E i₃ j₃ (eq (base (sym== x=y)))) (lemma-resp x=y j i)
 (makeH x c d) ab)
 ... | tri> _ _ y<x | (orA ab) = orB
 (nd y (lemma-<=min3E i₃ j₃ (le (base y<x)))) (lemma-trans y<x j i)
 (makeH x c d) ab)
 ... | tri> _ _ y<x | (orB ab) = orB
 (nr y (lemma-<=min3E i₃ j₃ (le (base y<x)))) (lemma-trans y<x j i)
 (makeH x c d) ab)
 afmerge (nr x i j (nf p₁ i₁ j₁ a₁ b₁) (nr p₂ i₂ j₂ a₂ b₂)) (orB (nf y i₃ j₃ c d))
 with cmp x y | afmerge (nr p₂ i₂ j₂ a₂ b₂) (orB (nf p₁ i₁ j₁ a₁ b₁))
 ... | tri< x<y _ _ | (orA ab) = orB


```

(nd x (le (base x<y)) (lemma-<=minE j i) (nf y i3 j3 c d) ab)
... | tri< x<y _ _ | (orB ab) = orB
(nr x (le (base x<y)) (lemma-<=minE j i) (nf y i3 j3 c d) ab)
... | tri= _ x=y _ | (orA ab) = orB
(nd y (lemma-<=min3E i3 j3 (eq (base (sym== x=y))))
(lemma-resp x=y j i) (makeH x c d) ab)
... | tri= _ x=y _ | (orB ab) = orB
(nr y (lemma-<=min3E i3 j3 (eq (base (sym== x=y))))
(lemma-resp x=y j i) (makeH x c d) ab)
... | tri> _ _ y<x | (orA ab) = orB
(nd y (lemma-<=min3E i3 j3 (le (base y<x))))
(lemma-trans y<x j i) (makeH x c d) ab)
... | tri> _ _ y<x | (orB ab) = orB
(nr y (lemma-<=min3E i3 j3 (le (base y<x))))
(lemma-trans y<x j i) (makeH x c d) ab)

```

Извлечение минимума из неполной кучи.

```

apop : ∀ {m h} → Heap m (succ h) almost
→ OR (Σ (expanded A) (λ x → (Heap x (succ h) almost) × (m ≤ x)))
(Σ (expanded A) (λ x → (Heap x h full) × (m ≤ x)))

```

```

apop (nd {x = x} p i j a eh) = orB (x , a , i)
apop (nd _ i j (nf x il jl a b) (nf y i2 j2 c d))
  with cmp x y | ndmerge (nf x il jl a b) (nf y i2 j2 c d)
... | tri< _ _ _ | res = orA (# x , res , i)
... | tri= _ _ _ | res = orA (# y , res , j)
... | tri> _ _ _ | res = orA (# y , res , j)
apop (nl _ i j (nd x il jl (nf y _ _ eh eh) eh) (nf z _ _ eh eh))
  with cmp x z
... | tri< x<z _ _ = orB (# x , nf x il (le (base x<z)))

```

```

(nf y (le ext) (le ext) eh eh) (nf z (le ext) (le ext) eh eh) , i)
... | tri= _ x=z _ = orB (# z ,
  nf z (eq (base (sym== x=z))) (snd resp≤ (base x=z) i_l)
    (nf x (le ext) (le ext) eh eh) (nf y (le ext) (le ext) eh eh) , j)
... | tri> _ _ z<x = orB (# z , nf z
  (le (base z<x)) (trans≤ (le (base z<x)) i_l)
    (nf x (le ext) (le ext) eh eh) (nf y (le ext) (le ext) eh eh) , j)

apop (nl _ i j (nd x i_l j_l (nf y i_2 j_2 a_2 b_2) (nf z i_3 j_3 a_3 b_3)) (nf t i_4 j_4 c d))
  with cmp x t | ndmerge (nf y i_2 j_2 a_2 b_2) (nf z i_3 j_3 a_3 b_3)
... | tri< x<t _ _ | res = orA (# x , nl x
  (lemma-<=minE i_l j_l) (le (base x<t))
    res (nf t i_4 j_4 c d) , i)
... | tri= _ x=t _ | res = orA (# t , nl t
  (snd resp≤ (base x=t) (lemma-<=minE i_l j_l))
  (lemma-<=min3E i_4 j_4 (eq (base (sym== x=t)))) res (makeH x c d) , j)
... | tri> _ _ t<x | res = orA (# t , nl t
  (lemma-trans t<x i_l j_l)
  (lemma-<=min3E i_4 j_4 (le (base t<x))) res (makeH x c d) , j)

apop (nl _ i j (nl x i_l j_l a b) (nf y i_2 j_2 c d))
  with cmp x y | afmerge (nl x i_l j_l a b) (orA (nf y i_2 j_2 c d))
... | tri< _ _ _ | orA res = orB (# x , res , i)
... | tri= _ _ _ | orA res = orB (# y , res , j)
... | tri> _ _ _ | orA res = orB (# y , res , j)
... | tri< _ _ _ | orB res = orA (# x , res , i)
... | tri= _ _ _ | orB res = orA (# y , res , j)
... | tri> _ _ _ | orB res = orA (# y , res , j)
apop (nl _ i j (nr x i_l j_l a b) (nf y i_2 j_2 c d))
  with cmp x y | afmerge (nr x i_l j_l a b) (orA (nf y i_2 j_2 c d))

```

```

... | tri< _ _ _ | orA res = orB (# x , res , i)
... | tri= _ _ _ | orA res = orB (# y , res , j)
... | tri> _ _ _ | orA res = orB (# y , res , j)
... | tri< _ _ _ | orB res = orA (# x , res , i)
... | tri= _ _ _ | orB res = orA (# y , res , j)
... | tri> _ _ _ | orB res = orA (# y , res , j)
apop (nr _ i j (nf x il jl a b) (nd y i2 j2 c d))
  with cmp y x | afmerge (nd y i2 j2 c d) (orB (nf x il jl a b))
... | tri< _ _ _ | orA res = orB (# y , res , j)
... | tri= _ _ _ | orA res = orB (# x , res , i)
... | tri> _ _ _ | orA res = orB (# x , res , i)
... | tri< _ _ _ | orB res = orA (# y , res , j)
... | tri= _ _ _ | orB res = orA (# x , res , i)
... | tri> _ _ _ | orB res = orA (# x , res , i)
apop (nr _ i j (nf x il jl a b) (nl y i2 j2 c d))
  with cmp y x | afmerge (nl y i2 j2 c d) (orB (nf x il jl a b))
... | tri< _ _ _ | orA res = orB (# y , res , j)
... | tri= _ _ _ | orA res = orB (# x , res , i)
... | tri> _ _ _ | orA res = orB (# x , res , i)
... | tri< _ _ _ | orB res = orA (# y , res , j)
... | tri= _ _ _ | orB res = orA (# x , res , i)
... | tri> _ _ _ | orB res = orA (# x , res , i)
apop (nr _ i j (nf x il jl a b) (nr y i2 j2 c d))
  with cmp y x | afmerge (nr y i2 j2 c d) (orB (nf x il jl a b))
... | tri< _ _ _ | orA res = orB (# y , res , j)
... | tri= _ _ _ | orA res = orB (# x , res , i)
... | tri> _ _ _ | orA res = orB (# x , res , i)
... | tri< _ _ _ | orB res = orA (# y , res , j)
... | tri= _ _ _ | orB res = orA (# x , res , i)
... | tri> _ _ _ | orB res = orA (# x , res , i)

```

2.5. Выводы по главе 2

Разработаны типы данных для представления структуры данных двоичная куча. Реализованы функции для обработки кучи. Доказано сохранение инвариантов порядка на элементах и сбалансированности.

Литература

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. *Martin-Löf P.* Intuitionistic Type Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.
4. The Haskell Programming Language. <http://www.haskell.org/haskellwiki/Haskell>.
5. A Truly Integrated Functional Logic Language. <http://www-ps.informatik.uni-kiel.de/currywiki/>.
6. Agda language. <http://wiki.portal.chalmers.se/agda/pmwiki.php>.
7. *IEEE.* IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language. IEEE, 1991. ISBN: 1-55937-125-0. http://standards.ieee.org/reading/ieee/std_public/description/busarch/1178-1990_desc.html.
8. *Hickey R.* The Clojure programming language / DLS. Под ред. Johan Brichau. ACM, 2008. С. 1. ISBN: 978-1-60558-270-2.
9. *Abelson H., Sussman G. J.* Structure and Interpretation of Computer Programs. MIT Press, 1985. ISBN: 0-262-51036-7.
10. *Milner R., Tofte M., Macqueen D.* The Definition of Standard ML. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.
11. OCaml. <http://ocaml.org/>.
12. *Dybjer P.* Inductive Families // Formal Asp. Comput. 1994. №4. С. 440—465.
13. *Atkey R., Johann P., Ghani N.* Refining Inductive Types // Logical Methods in Computer Science. 2012. №2.
14. *Xi H., Pfenning F.* Dependent Types in Practical Programming / POPL. Под ред. Andrew W. Appel и Alex Aiken. ACM, 1999. С. 214—227. ISBN: 1-58113-095-3.
15. *McBride C.* How to Keep Your Neighbours in Order. <https://personal.cis.strath.ac.uk/conor.mcbride/Pivotal.pdf>.
16. *McBride C., Norell U., Danielsson N. A.* The Agda standard library — AVL trees. <http://agda.github.io/agda-stdlib/html/Data.AVL.html>.
17. *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.
18. The Agda standard library. <http://agda.github.io/agda-stdlib/html/README.html>.