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

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 синтаксическим 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 связанности.

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]

[1] 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 связанности. 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

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.

1.3.1. 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.

1.4. 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: $x \to y$. Также рассматривают транзитивное замыкание отношения редукции: $\xrightarrow{*}$. 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 $x \to y$, 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 $x \xleftrightarrow{*} y$. 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 [11].

Операторы для типов в ИТТ:

- Π-тип (пи-тип) — зависимое произведение. Например, если $\mathrm{Vec}(\mathbb{R}, n)$ — тип кортежей из n вещественных чисел, $\mathbb{N}$ — тип натуральных чисел, то $\Pi_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$ — тип функции, которая по натуральному числу n возвращает кортеж из n вещественных чисел.
- Σ-тип — зависимая пара. Например, тип $\Sigma_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$ — тип пары из числа n и кортежа из n вещественных чисел.

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. Унификация

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

Унификация — процесс поиска унификатора.

1.6. 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.

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

Пример задания структуры данных и инвариантов — тип данных AVL-дерева и 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
```

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 — 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
```

1.8. 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 [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. Реализация

2.3.1. Вспомогательные определения

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

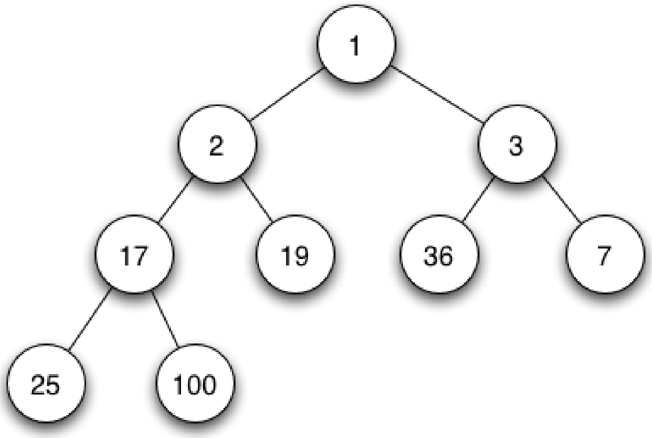

Рис. 2.1. Пример заполненной кучи для минимума

```
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

  _∘_ : ∀ {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 ¬a = ⊥-elim (¬a a)
```

Контрапозиция

```
contraposition : (A → B) → (¬ B → ¬ A)
contraposition = flip _∘_

open DummyAB public
open Logic public
module MLTT where
  infix 4 _≡_
```

Пропозициональное равенство из интуционистской теории типов.

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

  {-# BUILTIN EQUALITY _≡_ #-}
  {-# BUILTIN REFL refl #-}
```

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

  open Σ public
```

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)

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 доказательство 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
```

```
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≤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_ : ∀ {ℓ} {A : Set} → (A → Set ℓ) → Rel₂ A → Set _
P Respects _rel_ = ∀ {x y} → x rel y → P x → P y
```

Отношение P соблюдает отношение _rel_ .

```
_Respects₂_ : ∀ {A : Set} → Rel₂ A → Rel₂ A → Set
P Respects₂ _rel_ =
  (∀ {x} → P x  Respects _rel_) ×
  (∀ {y} → flip P y Respects _rel_)
```

Тип данных для обобщенного отношения меньше или равно.

```
data _<=_ {A : Set} {_<_ : Rel₂ A} {_==_ : Rel₂ A} : Rel₂ A where
  le : ∀ {x y} → x < y → x <= y
  eq : ∀ {x y} → x == y → x <= y
```

Обобщенные функции минимум и максимум.

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

```
... | tri> _ _ _ = x
... | _ = 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))

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 котороых составляются новые кучи при их обработке.

Отношение _<=_ 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)
```

2.3.3. 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 Heap (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

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.

```
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

  left {_} {# _} {top} () _
  left {_} {top} {# _} () _
  left {top} {_} {_}  _ ()

  right : ∀ {a b c : expanded A} → b =E c → b <E a → c <E a
  right {# _} {# _} {# _} (base r1) (base r2) = base (snd resp r1 r2)
  right {top} {# _} {# _} _ ext = ext

  right {_} {# _} {top} () _
  right {_} {top} {_} _ ()
```

Отношение меньше-равно для расширенного типа.

```
_≤_ : Rel₂ (expanded A)
_≤_ = _<=_ {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 элементов в поддеревьях. Мы можем упростить этот инвариант, сравнивая элемент в узле только с корнями поддеревьев. Порядок кучи задается с помощью двух элементов отношения _≤_: i и j, которые говорят от том, что значение в корне меньше-равно значений в корнях левого и правого поддеревьев соответственно. На рисунке 2.2 схематичное 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.

$$\mathsf{nd} : \forall\ \{n\}\ \{x\ y\} \to (p : A) \to (i : (\#\ p) \leq x) \to (j : (\#\ p) \leq y)$$
$$\to (a : \mathsf{Heap}\ x\ (\mathsf{succ}\ n)\ \mathsf{full})$$
$$\to (b : \mathsf{Heap}\ y\ n\ \mathsf{full})$$
$$\to \mathsf{Heap}\ (\#\ p)\ (\mathsf{succ}\ (\mathsf{succ}\ n))\ \mathsf{almost}$$

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.

$$\mathsf{nl} : \forall\ \{n\}\ \{x\ y\} \to (p : A) \to (i : (\#\ p) \leq x) \to (j : (\#\ p) \leq y)$$
$$\to (a : \mathsf{Heap}\ x\ (\mathsf{succ}\ n)\ \mathsf{almost})$$
$$\to (b : \mathsf{Heap}\ y\ n\ \mathsf{full})$$
$$\to \mathsf{Heap}\ (\#\ p)\ (\mathsf{succ}\ (\mathsf{succ}\ n))\ \mathsf{almost}$$

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 $n+1$.

$$\mathsf{nr} : \forall\ \{n\}\ \{x\ y\} \to (p : A) \to (i : (\#\ p) \leq x) \to (j : (\#\ p) \leq y)$$
$$\to (a : \mathsf{Heap}\ x\ (\mathsf{succ}\ n)\ \mathsf{full})$$
$$\to (b : \mathsf{Heap}\ y\ (\mathsf{succ}\ n)\ \mathsf{almost})$$
$$\to \mathsf{Heap}\ (\#\ p)\ (\mathsf{succ}\ (\mathsf{succ}\ n))\ \mathsf{almost}$$

Замечание: высота любой неполной кучи больше нуля.

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

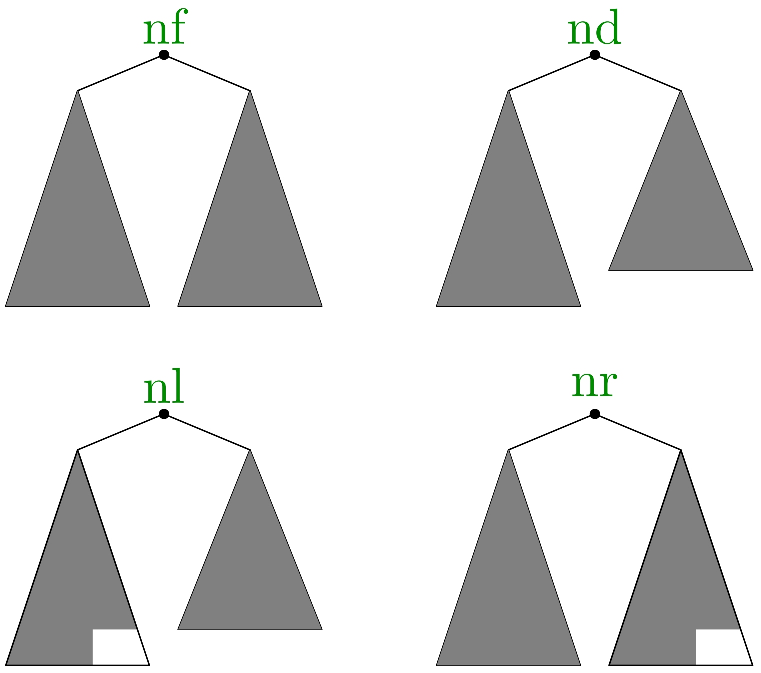

Рис. 2.2. Конструкторы типа данных Heap

$\mathsf{lemma\text{-}almost\text{-}height}\ (\mathsf{nd}\ \_\ \_\ \_\ \_\ \_\ \_) = \mathsf{s{\leq}s}\ \mathsf{z{\leq}n}$

$\mathsf{lemma\text{-}almost\text{-}height}\ (\mathsf{nl}\ \_\ \_\ \_\ \_\ \_\ \_) = \mathsf{s{\leq}s}\ \mathsf{z{\leq}n}$

$\mathsf{lemma\text{-}almost\text{-}height}\ (\mathsf{nr}\ \_\ \_\ \_\ \_\ \_\ \_) = \mathsf{s{\leq}s}\ \mathsf{z{\leq}n}$

Функция — просмотр минимума в куче.

$\mathsf{peekMin} : \forall\ \{m\ h\ s\} \to \mathsf{Heap}\ m\ h\ s \to (\mathsf{expanded}\ A)$

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

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

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

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

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

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

$\mathsf{min3E} : (\mathsf{expanded}\ A) \to (\mathsf{expanded}\ A) \to (\mathsf{expanded}\ A) \to (\mathsf{expanded}\ A)$

```
min3E x y z = min3 cmpE x y z
```

Леммы для сравнения с минимумами для элементов расширенного типа.

```
lemma-<=minE : ∀ {a b c} → a ≤ b → a ≤ c → a ≤ (minE b c)
lemma-<=minE = lemma-<=min {expanded A}{_<E_}{_=E_}{cmpE}
lemma-<=min3E : ∀ {x a b c} → x ≤ a → x ≤ b → x ≤ c → x ≤ (min3E a b c)
lemma-<=min3E = lemma-<=min3 {expanded A}{_<E_}{_=E_}{cmpE}
```

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 e
... | 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 i1 j1 a b) c) with cmp p z
finsert z (nf p i j (nf x i1 j1 a b) c) | tri< p<z _ _
  with finsert z (nf x i1 j1 a b)
  | lemma-<=minE {# p} {# x} {# z} i (le (base p<z))
... | full , newleft | l1 = almost , nd p l1 j newleft c
... | almost , newleft | l1 = almost , nl p l1 j newleft c
finsert z (nf p i j (nf x i1 j1 a b) c) | tri= _ p=z _
  with finsert p (nf x i1 j1 a b)
  | lemma-<=minE {# z} {# x} {# p} (snd resp≤ (base p=z) i) (eq (base (sym=
... | full , newleft | l1 | l2 = almost , nd z l1 l2 newleft c
... | almost , newleft | l1 | l2 = almost , nl z l1 l2 newleft c
finsert z (nf p i j (nf x i1 j1 a b) c) | tri> _ _ z<p
```

```
    with finsert p (nf x i₁ j₁ a b)
    | lemma-<=minE {# z} {# x} {# p} (trans≤ (le (base z<p)) i) (le (base z<p))
  ... | full   , newleft | l1 = almost , nd z l1 (trans≤ (le (base z<p)) j) newleft c
  ... | almost , newleft | l1 = almost , nl z l1 (trans≤ (le (base z<p)) j) newleft c
```

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 finsert z b | lemma-<=minE j (le (base
  ... | full   , nb | l1 = full   , nf p i l1 a nb
  ... | almost , nb | l1 = almost , nr p i l1 a nb
  ainsert z (nd p i j a b) | tri= _ p=z _ with finsert p b | snd resp≤ (base p=z) i | le
  ... | full   , nb | l1 | l2 = full   , nf z l1 l2 a nb
  ... | almost , nb | l1 | l2 = almost , nr z l1 l2 a nb
  ainsert z (nd p i j a b) | tri> _ _ z<p with finsert p b | trans≤ (le (base z<p)) i |
  ... | full   , nb | l1 | l2 = full   , nf z l1 l2 a nb
  ... | almost , nb | l1 | l2 = almost , nr z l1 l2 a nb

  ainsert z (nl p i j a b) with cmp p z
  ainsert z (nl p i j a b) | tri< p<z _ _ with ainsert z a | lemma-<=minE i (le (base
  ... | full   , na | l1 = almost , nd p l1 j na b
  ... | almost , na | l1 = almost , nl p l1 j na b
  ainsert z (nl p i j a b) | tri= _ p=z _ with ainsert p a | lemma-<=minE (snd resp≤
  ... | full   , na | l1 | l2 = almost , nd z l1 l2 na b
  ... | almost , na | l1 | l2 = almost , nl z l1 l2 na b
  ainsert z (nl p i j a b) | tri> _ _ z<p with ainsert p a | lemma-<=minE (trans≤ (l
  ... | full   , na | l1 | l2 = almost , nd z l1 l2 na b
  ... | almost , na | l1 | l2 = almost , nl z l1 l2 na b
```

```
ainsert z (nr p i j a b) with cmp p z
ainsert z (nr p i j a b) | tri< p<z _ _ with ainsert z b | lemma-<=minE j (le (base
... | full   , nb | l1 = full   , nf p i l1 a nb
... | almost , nb | l1 = almost , nr p i l1 a nb
ainsert z (nr p i j a b) | tri= _ p=z _ with ainsert p b | snd resp≤ (base p=z) i | le
... | full   , nb | l1 | l2 = full   , nf z l1 l2 a nb
... | almost , nb | l1 | l2 = almost , nr z l1 l2 a nb
ainsert z (nr p i j a b) | tri> _ _ z<p with ainsert p b | trans≤ (le (base z<p)) i |
... | full   , nb | l1 | l2 = full   , nf z l1 l2 a nb
... | almost , nb | l1 | l2 = almost , nr z l1 l2 a nb
```

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)

fmerge eh eh = orA (eh , refl , refl)
fmerge (nf x i₁ j₁ a b) (nf y i₂ j₂ c d) with cmp x y
fmerge (nf x i₁ j₁ a b) (nf y i₂ j₂ c d) | tri< x<y _ _ with fmerge a b
... | orA (eh , refl , refl) = orB (nd x (le (base x<y)) j₁ (nf y i₂ j₂ c d) eh)
... | orB ab = orB (nr x (le (base x<y)) (lemma-<=minE i₁ j₁) (nf y i₂ j₂ c d) ab)
fmerge (nf x i₁ j₁ a b) (nf y i₂ j₂ c d) | tri= _ x=y _ with fmerge c d
... | orA (eh , refl , refl) = orB (nd y (eq (base (sym== x=y))) j₂ (nf x i₁ j₁ a b) eh
... | orB cd = orB (nr y (eq (base (sym== x=y))) (lemma-<=minE i₂ j₂) (nf x i₁ j
```

```
fmerge (nf x i₁ j₁ a b) (nf y i₂ j₂ c d) | tri> _ _ y<x with fmerge c d
... | orA (eh , refl , refl) = orB (nd y (le (base y<x)) j₂ (nf x i₁ j₁ a b) eh)
... | orB cd = orB (nr y (le (base y<x)) (lemma-<=minE i₂ j₂) (nf x i₁ j₁ a b) cd)
```

Извлечение минимума из полной кучи.

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

fpop (nf _ _ _ eh eh) = orB eh
fpop (nf _ i j (nf x i₁ j₁ a b) (nf y i₂ j₂ c d))
  with fmerge (nf x i₁ j₁ a b) (nf y i₂ j₂ c d)
... | orA (() , _ , _)
... | orB res = orA ((minE (# x) (# y)) , res , lemma-<=minE i j)
```

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

```
makeH : ∀ {x y h} → (p : A) → Heap x h full → Heap y h full
  → Heap (min3E x y (# p)) (succ h) full

makeH p eh eh = nf p (le ext) (le ext) eh eh
makeH p (nf x i j a b) (nf y i₁ j₁ c d) with cmp x y
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ with cmp x p
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | tri< x<p _ _ with makeH
... | res = nf x (lemma-<=min3E i j (le (base x<p))) (le (base x<y)) res (nf y i₁ j₁
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | tri= _ x=p _ = nf p (eq (
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | tri> _ _ p<x = nf p (le (b

makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ with cmp y p
```

```
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | tri< y<p _ _ = nf y (eq (
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | tri= _ y=p _ = nf p (eq (
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | tri> _ _ p<y = nf p (le (l

makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x with cmp y p
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | tri< y<p _ _ = nf y (le (l
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | tri= _ y=p _ = nf p (le (l
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | tri> _ _ p<y = nf p (le (l
```

Вспомогательные леммы, использующие lemma-<=minE.

```
lemma-resp : ∀ {x y a b} → x == y → (# x) ≤ a → (# x) ≤ b
  → (# y) ≤ minE a b
lemma-resp x=y i j = lemma-<=minE (snd resp≤ (base x=y) i)
  (snd resp≤ (base x=y) j)
lemma-trans : ∀ {x y a b} → y < x → (# x) ≤ a → (# x) ≤ b
  → (# y) ≤ minE a b
lemma-trans y<x i j = lemma-<=minE (trans≤ (le (base y<x)) i)
  (trans≤ (le (base y<x)) j)
```

Слияние поддеревьев из кучи, у которой последний ряд заполнен до середины, определенной конструктором nd.

```
ndmerge : ∀ {x y h} → Heap x (succ (succ h)) full → Heap y (succ h) full
→ Heap (minE x y) (succ (succ (succ h))) almost

ndmerge (nf x i j a b) (nf y i₁ j₁ c d) with cmp x y
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ with fmerge a b
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | orA (_ , _ , ())
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | orB x₁ = nl x (lemma-<=m
```

```
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ with fmerge c d
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | orA (eh , refl , refl) with fm
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | orA (eh , refl , refl) | orA (e
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | orA (eh , refl , refl) | orB al

ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | orB cd with fmerge a b
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | orB cd | orA (_ , _ , ())
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | orB cd | orB ab = nl y (lem
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x with fmerge a b
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | orA (_ , _ , ())
ndmerge (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | orB ab = nl y (lemma-trans
```

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 i₁ j₁ 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 i₁ j₁ 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)
... | tri> _ _ y<x = orA (nf y (le (base y<x)) (trans≤ (le (base y<x)) i) (nf x j j

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

```
... | tri< x<y _ _ | orA ab = orA (nf x (lemma-<=minE i j) (le (base x<y)) ab (n
... | tri< x<y _ _ | orB ab = orB (nl x (lemma-<=minE i j) (le (base x<y)) ab (n
... | tri= _ x=y _ | orA ab = orA (nf y (lemma-resp x=y i j) (lemma-<=min3E i3
... | tri= _ x=y _ | orB ab = orB (nl y (lemma-resp x=y i j) (lemma-<=min3E i3
... | tri> _ _ y<x | orA ab = orA (nf y (lemma-trans y<x i j) (lemma-<=min3E i
... | tri> _ _ y<x | orB ab = orB (nl y (lemma-trans y<x i j) (lemma-<=min3E i

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 cm
... | tri< x<y _ _ | orA ab = orA (nf x (lemma-<=minE i j) (le (base x<y)) ab (n
... | tri< x<y _ _ | orB ab = orB (nl x (lemma-<=minE i j) (le (base x<y)) ab (n
... | tri= _ x=y _ | orA ab = orA (nf y (lemma-resp x=y i j) (lemma-<=min3E i3
... | tri= _ x=y _ | orB ab = orB (nl y (lemma-resp x=y i j) (lemma-<=min3E i3
... | tri> _ _ y<x | orA ab = orA (nf y (lemma-trans y<x i j) (lemma-<=min3E i
... | tri> _ _ y<x | orB ab = orB (nl y (lemma-trans y<x i j) (lemma-<=min3E i

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 cm
... | tri< x<y _ _ | orA ab = orA (nf x (lemma-<=minE i j) (le (base x<y)) ab (n
... | tri< x<y _ _ | orB ab = orB (nl x (lemma-<=minE i j) (le (base x<y)) ab (n
... | tri= _ x=y _ | orA ab = orA (nf y (lemma-resp x=y i j) (lemma-<=min3E i3
... | tri= _ x=y _ | orB ab = orB (nl y (lemma-resp x=y i j) (lemma-<=min3E i3
... | tri> _ _ y<x | orA ab = orA (nf y (lemma-trans y<x i j) (lemma-<=min3E i
... | tri> _ _ y<x | orB ab = orB (nl y (lemma-trans y<x i j) (lemma-<=min3E i

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 c
... | tri< x<y _ _ | (orA ab) = orA (nf x (le (base x<y)) (lemma-<=minE j i) (nf
... | tri< x<y _ _ | (orB ab) = orB (nl x (lemma-<=minE j i) (le (base x<y)) ab
... | tri= _ x=y _ | (orA ab) = orA (nf y (lemma-resp x=y j i) (lemma-<=min3E
... | tri= _ x=y _ | (orB ab) = orB (nl y (lemma-resp x=y j i) (lemma-<=min3E
... | tri> _ _ y<x | (orA ab) = orA (nf y (lemma-trans y<x j i) (lemma-<=min3E
... | tri> _ _ y<x | (orB ab) = orB (nl y (lemma-trans y<x j i) (lemma-<=min3E
```

```
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 cm
... | tri< x<y _ _ | (orA ab) = orA (nf x (le (base x<y)) (lemma-<=minE j i) (nf
... | tri< x<y _ _ | (orB ab) = orB (nl x (lemma-<=minE j i) (le (base x<y)) ab
... | tri= _ x=y _ | (orA ab) = orA (nf y (lemma-resp x=y j i) (lemma-<=min3E
... | tri= _ x=y _ | (orB ab) = orB (nl y (lemma-resp x=y j i) (lemma-<=min3E
... | tri> _ _ y<x | (orA ab) = orA (nf y (lemma-trans y<x j i) (lemma-<=min3E
... | tri> _ _ y<x | (orB ab) = orB (nl y (lemma-trans y<x j i) (lemma-<=min3E

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 cr
... | tri< x<y _ _ | (orA ab) = orA (nf x (le (base x<y)) (lemma-<=minE j i) (nf
... | tri< x<y _ _ | (orB ab) = orB (nl x (lemma-<=minE j i) (le (base x<y)) ab
... | tri= _ x=y _ | (orA ab) = orA (nf y (lemma-resp x=y j i) (lemma-<=min3E
... | tri= _ x=y _ | (orB ab) = orB (nl y (lemma-resp x=y j i) (lemma-<=min3E
... | tri> _ _ y<x | (orA ab) = orA (nf y (lemma-trans y<x j i) (lemma-<=min3E
... | tri> _ _ y<x | (orB ab) = orB (nl y (lemma-trans y<x j i) (lemma-<=min3E

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))))
... | tri> _ _ y<x = orB (nd y (lemma-<=min3E i2 j2 (le (base y<x))) (trans≤ (l

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 cr
... | tri< x<y _ _ | ab = orB (nr x (le (base x<y)) (lemma-<=minE i j) (nf y i3 j
... | tri= _ x=y _ | ab = orB (nr y (lemma-<=min3E i3 j3 (eq (base (sym== x=y
... | tri> _ _ y<x | ab = orB (nr y (lemma-<=min3E i3 j3 (le (base y<x))) (lemm

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 cr
... | tri< x<y _ _ | (orA ab) = orB (nd x (le (base x<y)) (lemma-<=minE i j) (nf
... | tri< x<y _ _ | (orB ab) = orB (nr x (le (base x<y)) (lemma-<=minE i j) (nf
```

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

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

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

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 i₁ j₁ a b) (nf y i₂ j₂ c d)) with cmp x y | ndmerge (nf x i₁ j₁ a
... | tri< _ _ _ | res = orA (# x , res , i)
... | tri= _ _ _ | res = orA (# y , res , j)
... | tri> _ _ _ | res = orA (# y , res , j)
apop (nl _ i j (nd x i₁ j₁ (nf y _ _ eh eh) eh) (nf z _ _ eh eh)) with cmp x z
... | tri< x<z _ _ = orB (# x , nf x i₁ (le (base x<z)) (nf y (le ext) (le ext) eh eh)
... | tri= _ x=z _ = orB (# z , nf z (eq (base (sym== x=z))) (snd resp≤ (base x
... | tri> _ _ z<x = orB (# z , nf z (le (base z<x)) (trans≤ (le (base z<x)) i₁) (n

apop (nl _ i j (nd x i₁ j₁ (nf y i₂ j₂ a₂ b₂) (nf z i₃ j₃ a₃ b₃)) (nf t i₄ j₄ c d)) with c
... | tri< x<t _ _ | res = orA (# x , nl x (lemma-<=minE i₁ j₁) (le (base x<t)) re
... | tri= _ x=t _ | res = orA (# t , nl t (snd resp≤ (base x=t) (lemma-<=minE i
... | tri> _ _ t<x | res = orA (# t , nl t (lemma-trans t<x i₁ j₁) (lemma-<=min3E
```

```
apop (nl _ i j (nl x i1 j1 a b) (nf y i2 j2 c d)) with cmp x y | afmerge (nl x i1 j1 a b
... | 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 i1 j1 a b) (nf y i2 j2 c d)) with cmp x y | afmerge (nr x i1 j1 a
... | 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 i1 j1 a b) (nd y i2 j2 c d)) with cmp y x | afmerge (nd y i2 j2 c
... | 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 i1 j1 a b) (nl y i2 j2 c d)) with cmp y x | afmerge (nl y i2 j2 c
... | 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 i1 j1 a b) (nr y i2 j2 c d)) with cmp y x | afmerge (nr y i2 j2 c
```

```
... | 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)
```

2.4. 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. C. 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 Languageu. 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. C. 52. ISBN: 1-55937-125-0. 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. C. 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. C. 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. C. 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
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.