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 абстрагирования обработки данных. Свойства структуры данных происходят из инвариантов, которые эта структура данных соблюдает.

Практика показывает, что тривиальные структуры данных и их инварианты 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. Функциональное программирование

Функциональное программирование — парадигма программирования, являющаяся разновидностью декларативного программирования, в которой программу представляют в виде функций (в 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 [1], Curry [2], Agda [3], диалекты LISP [4—6], SML [7], OCaml[8]), но существуют и функциональные языки явно не основанные на этом формализме (например, препроцессор языка C и шаблоны в C++).

1.2. 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.

Множество пре-лямбда-термов обозначают Λ^-. *Лямбда-термы* — это пре-лямбда-термы, факторизованные по отношению *альфа-эквивалентности*. Обозначение: $\Lambda = \Lambda^- / =_\alpha$.

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]

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 [9] и [10].

1.3. Лямбда-исчисление с простыми типами

Определение 1.1. Пусть U — бесконечное счетное множество, элементы которого мы будем называть *переменными типов*. Множество *простых типов* Π — множество, определенное грамматикой:

$$\Pi ::= U \mid (\Pi \to \Pi)$$

Для обозначения элементов множества Π используют буквы греческого алфавита: $\sigma, \tau \ldots$.

Определение 1.2. Множество контекстов C — это множество всех множеств

[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.

пар такого вида:

$$\{x_1 : \tau_1, \dots, x_n : \tau_n\}$$

где $\tau_1, \dots, \tau_n \in \Pi$, а $x_1, \dots, x_n \in V$ (переменные из Λ) и $x_i \neq x_j$ если $i \neq j$.

Определение 1.3. *Домен* контекста $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$:

$$\mathrm{dom}(\Gamma) = \{x_1, \dots, x_n\}$$

и $x_i \neq x_j$ при $i \neq j$.

Определение 1.4. Отношение *типизации* (typability) $\vdash$ на множестве $C \times \Lambda \times \Pi$ определяется следующими правилами:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

В первом и втором правиле мы требуем $x \notin \mathrm{dom}(\Gamma)$.

Определение 1.5. *Лямбда-исчисление с простыми типами* или $\lambda^{\to}$ — это тройка $(\Lambda, \Pi, \vdash)$. Чтобы отличать данное в этой работе определение системы $\lambda^{\to}$ от других вариантов, эту систему называют лямбда-исчисление с простыми типами *по Карри*.

За более подробной информацией об этом формализме следует обращаться к [11] и [10].

1.4. 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.

1.4.1. Рекурсивные типы данных

Рекурсивный тип данных — тип данных, в определении которого содержится определяемый тип данных. Например, список элементов типа A:

$$List\ A = Nil + (A \times List\ A)$$

В теории [12] для введения рекурсивных типов используются μ-типы. *Сырые* μ-типы вводятся с помощью оператора μ: $\mu X.T$. При этом T может содержать X.

Определение 1.6. Сырой μ-тип T называется *сократимым* (contractive), если для любого подвыражения T вида $\mu X.\mu X_1 \ldots \mu X_n.S$ тело S не равняется X.

Сырой μ-тип называется просто *μ-типом* (μ -type), если он сократим.

Пример: список элементов типа A: $List\ A = \mu X.Nil + (A \times X)$.

1.4.2. 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. 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*.

1.5.1. 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.

1.5.2. Интуиционистская теория типов

Интуиционистская теория типов (теория типов Мартина-Лёфа) основана на математическом конструктивизме [13].

Операторы для типов в ИТТ:

- Π-тип (пи-тип) — зависимое произведение, обобщение типов функций ($X \to Y$), в которых тип результата зависит от значения аргумента: $\Pi_{x:X} Y(x)$. Например, если $\mathrm{Vec}(A, n)$ — тип кортежей из n элементов типа A, $\mathbb{N}$ — тип натуральных чисел, то $\Pi_{n:\mathbb{N}} \mathrm{Vec}(A, n)$ — тип функции, которая по натуральному числу n возвращает кортеж из n элементов типа A.
- Σ-тип — зависимая пара $\Sigma_{x:A} B(x)$. Второй элемент в зависимой паре зависит от первого. Например, тип $\Sigma_{n:\mathbb{N}} \mathrm{Vec}(A, n)$ — тип пары из числа n и кортежа из n элементов типа A.
- Пусть A — множество конструкторов, B — селектор на A. Элементы множества A представляют разные способы сформировать элемент в $W_{a:A} B(a)$, а $B(a)$ представляют части дерева, сформированные с помощью a. $W_{a:A} B(a)$ — рекурсивный тип, построенный с помощью конструкторов $B(a)$, который можно представить в виде *фундированных деревьев* (well-founded trees) [14].

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

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. Agda

Agda [3] — 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
```

$\mathrm{cons} : \forall \{n\} \to A \to \mathrm{Vec}\ A\ n \to \mathrm{Vec}\ A\ (\mathrm{succ}\ n)$

В фигурные скобки заключаются неявные аргументы.

1.7.1. Сопоставление с образцом по типам с индексами

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

$\mathrm{head} : \forall \{A\}\ \{n\} \to \mathrm{Vec}\ A\ (\mathrm{succ}\ n) \to A$

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:

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

Перепишем тип данных Vec в немного другом виде — заменим индекс на параметр:

```
data Vec-ni (A : Set) (n : ℕ) : Set where
  nil  : (n ≡ zero) → Vec-ni A n
  cons : ∀ {k} → (n ≡ succ k) → A → Vec-ni A k → Vec-ni A n
```

Теперь конструкторы nil и cons явно требуют доказательства о длине вектора n. Agda при сопоставлении с образцом на индексированных типах генерирует эти доказательства с помощью унификации [15, 16]. В определении функции head тип аргумента унифицируется с типами конструкторов типа данных Vec и, так как не существует k такого, что zero $\equiv$ succ k, сопоставление производится только по конструктору cons.

1.8. Индуктивные семейства

Определение 1.7. *Индуктивное семейство* [17, 18] — 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.

1.9. Использование индуктивных семейств в структурах данных

В работах [19, 20] 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 высоты поддеревьев в AVL-дереве [21].

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

1.10. Выводы по главе 1

В этой главе были рассмотрены некоторые существующие подходы к построению структур данных с использованием индуктивных семейств. Кратко описаны особенности языка программирования *Agda*, который использован в этой работе.

Глава 2. Описание реализованной структуры данных

2.1. Постановка задачи

Целью данной работы является разработка типов данных для представления структуры данных и инвариантов, а также доказательства этих 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 [22] — 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

Некоторые общеизвестные определения заимствованы из стандартной библиотеки Agda [23].

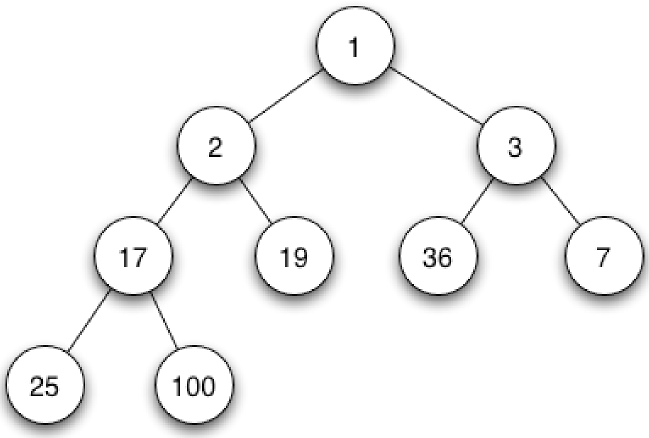

Рис. 2.1. Пример заполненной кучи для минимума

```
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 ¬a = ⊥-elim (¬a a)
```

Контрапозиция

```
      contraposition : (A → B) → (¬ B → ¬ A)
      contraposition = flip _∘_
    open DummyAB public
open Logic public
```

Определения интуиционистской теории типов Матрина-Лёфа [13].

```
module MLTT where
```

Пропозициональное равенство из интуиционистской теории типов.

```
  infix 4 _≡_
  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
```

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≤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)
```

Определим типы данных для задания свойств отношений. Транзитивность отношения.

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

$\text{Trans}\ \{A\}\ \_rel\_ = \{a\ b\ c : A\} \to (a\ rel\ b) \to (b\ rel\ c) \to (a\ rel\ c)$

Симметричность отношения.

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

$\text{Symmetric}\ \_rel\_ = \forall\ \{a\ b\} \to a\ rel\ b \to b\ rel\ a$

Предикат P учитывает (соблюдает) отношение _rel_ .

$\_\text{Respects}\_ : \forall\ \{\ell\}\ \{A : \text{Set}\} \to (A \to \text{Set}\ \ell) \to \text{Rel}_2\ A \to \text{Set}\ \_$

$P\ \text{Respects}\ \_rel\_ = \forall\ \{x\ y\} \to x\ rel\ y \to P\ x \to P\ y$

Отношение P соблюдает отношение _rel_ .

$\_\text{Respects}_2\_ : \forall\ \{A : \text{Set}\} \to \text{Rel}_2\ A \to \text{Rel}_2\ A \to \text{Set}$

$P\ \text{Respects}_2\ \_rel\_ =$

$\quad (\forall\ \{x\} \to P\ x\quad \text{Respects}\ \_rel\_) \times$

$\quad (\forall\ \{y\} \to \text{flip}\ P\ y\ \text{Respects}\ \_rel\_)$

Тип данных для обобщенного отношения меньше или равно.

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

$\quad \text{le} : \forall\ \{x\ y\} \to x < y \to x <= y$

$\quad \text{eq} : \forall\ \{x\ y\} \to x == y \to x <= y$

Обобщенные функции минимум и максимум.

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

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

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

```
... | 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 которых составляются новые кучи при их обработке.

Отношение _<=_ соблюдает отношение равенства _==_, с помощью которого оно определено.

```
resp<= : {A : Set} {_<_ : Rel₂ A} {_==_ : Rel₂ A}
  → (resp : _<_ Respects₂ _==_) → (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.

```
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
```

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

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

Леммы для сравнения с минимумами для элементов расширенного типа.

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

2.4.2. Тип данных Heap

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

```
data HeapState : Set where
  full almost : HeapState
```

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

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

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

```
eh : Heap top zero full
```

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 схематично изображены конструкторы типа данных Heap.

Полная куча высотой $n + 1$ состоит из корня и двух куч высотой n.

```
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.

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

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

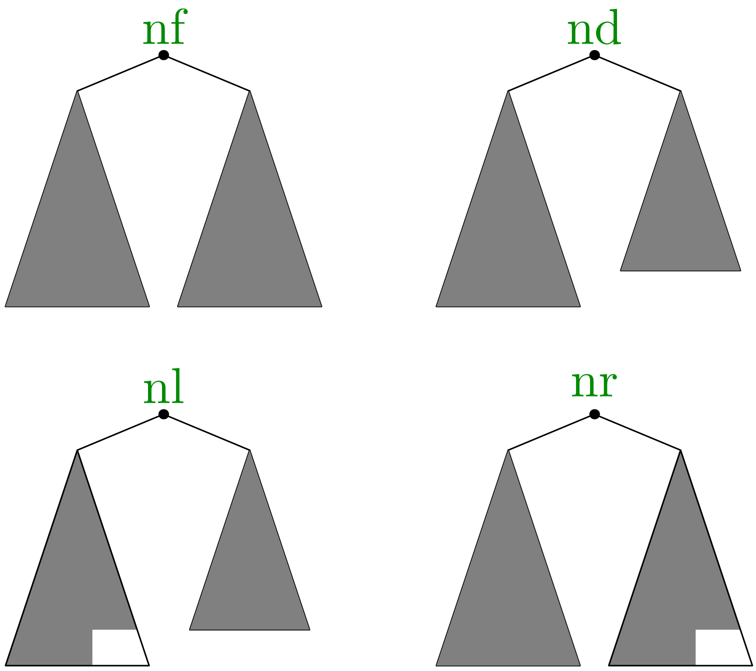

Рис. 2.2. Конструкторы типа данных Heap

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 i_1 j_1 a b) c) with *cmp* p z
finsert z (nf p i j (nf x i_1 j_1 a b) c) | tri< p<z _ _
with finsert z (nf x i_1 j_1 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 i_1 j_1 a b) c) | tri= _ p=z _

```
  with finsert p (nf x i₁ j₁ a b)
  | lemma-<=minE (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
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 (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 p<z))
... | 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
  | lemma-<=minE (snd resp≤ (base p=z) j) (eq (base (sym== p=z)))
... | 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
  | lemma-<=minE (trans≤ (le (base z<p)) j) (le (base z<p))
... | 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 p<z))
... | 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≤ (base p=z) i)
    (eq (base (sym== p=z))) | snd resp≤ (base p=z) j
... | 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≤ (le (base z<p)) i)
    (le (base z<p)) | trans≤ (le (base z<p)) j
... | 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 p<z))
... | 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
  | lemma-<=minE (snd resp≤ (base p=z) j) (eq (base (sym== p=z)))
... | 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
  | lemma-<=minE (trans≤ (le (base z<p)) j) (le (base z<p))
```

```
... | full    , nb | l1 | l2 = full    , nf z l1 l2 a nb
... | almost , nb | l1 | l2 = almost , nr z l1 l2 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)
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 j1 a b) cd)


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 $y<x$)) j_2 (nf x i_1 j_1 a b) eh)

... | orB cd = orB

(nr y (le (base $y<x$)) (lemma-<=minE i_2 j_2) (nf x i_1 j_1 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) × ($m \leq x$)))

(Heap top h full)

fpop (nf _ _ _ eh eh) = orB eh

fpop (nf _ i j (nf x i_1 j_1 a b) (nf y i_2 j_2 c d))

with fmerge (nf x i_1 j_1 a b) (nf y i_2 j_2 c d)

... | orA (() , _ , _)

... | orB res = orA ((minE (# x) (# y)) , res , lemma-<=minE i j)

2.4.5. Удаление минимума из неполной кучи

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 i_1 j_1 c d) with cmp x y

makeH p (nf x i j a b) (nf y i_1 j_1 c d) | tri< $x<y$ _ _ with cmp x p

makeH p (nf x i j a b) (nf y i_1 j_1 c d) | tri< $x<y$ _ _ | tri< $x<p$ _ _

with makeH p a b

... | res = nf x (lemma-<=min3E i j (le (base $x<p$))) (le (base $x<y$))

res (nf y i_1 j_1 c d)

```
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | tri= _ x=p _ =
   nf p (eq (base (sym== x=p))) (le (base (snd resp x=p x<y)))
      (nf x i j a b) (nf y i₁ j₁ c d)
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri< x<y _ _ | tri> _ _ p<x =
   nf p (le (base p<x)) (le (base (trans< p<x x<y)))
      (nf x i j a b) (nf y i₁ j₁ c d)
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 (base (sym== x=y))) (lemma-<=min3E i₁ j₁ (le (base y<p)))
      (nf x i j a b) (makeH p c d)
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | tri= _ y=p _ =
   nf p (eq (base (trans== (sym== y=p) (sym== x=y))))
      (eq (base (sym== y=p))) (nf x i j a b) (nf y i₁ j₁ c d)
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri= _ x=y _ | tri> _ _ p<y =
   nf p (le (base (fst resp (sym== x=y) p<y))) (le (base p<y))
      (nf x i j a b) (nf y i₁ j₁ c d)


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 (base y<x)) (lemma-<=min3E i₁ j₁ (le (base y<p)))
      (nf x i j a b) (makeH p c d)
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | tri= _ y=p _ =
   nf p (le (base (snd resp y=p y<x))) (eq (base (sym== y=p)))
      (nf x i j a b) (nf y i₁ j₁ c d)
makeH p (nf x i j a b) (nf y i₁ j₁ c d) | tri> _ _ y<x | tri> _ _ p<y =
   nf p (le (base (trans< p<y y<x))) (le (base p<y))
      (nf x i j a b) (nf y i₁ j₁ c d)
```

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-<=minE i j) (le (base x<y)) x1 (nf y i1 j1 c d)

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 fmerge a b
ndmerge (nf x i j a b) (nf y i1 j1 c d) | tri= _ x=y _ | orA (eh , refl , refl)
  | orA (eh , refl , ())
ndmerge (nf x i j a b) (nf y i1 j1 c d) | tri= _ x=y _ | orA (eh , refl , refl)
  | orB ab = nl y (lemma-resp x=y i j) (eq (base (sym== x=y)))
```

ab (nf *x* (le ext) (le ext) eh eh)

ndmerge (nf *x i j a b*) (nf *y* i_1 j_1 *c d*) | tri= _ *x*=*y* _ | orB *cd* with fmerge *a b*

ndmerge (nf *x i j a b*) (nf *y* i_1 j_1 *c d*) | tri= _ *x*=*y* _ | orB *cd* | orA (_ , _ , ())

ndmerge (nf *x i j a b*) (nf *y* i_1 j_1 *c d*) | tri= _ *x*=*y* _ | orB *cd* | orB *ab* =

nl *y* (lemma-resp *x*=*y i j*) (lemma-<=min3E i_1 j_1 (eq (base (*sym== x*=*y*))))

ab (makeH *x c d*)

ndmerge (nf *x i j a b*) (nf *y* i_1 j_1 *c d*) | tri> _ _ *y*<*x* with fmerge *a b*

ndmerge (nf *x i j a b*) (nf *y* i_1 j_1 *c d*) | tri> _ _ *y*<*x* | orA (_ , _ , ())

ndmerge (nf *x i j a b*) (nf *y* i_1 j_1 *c d*) | tri> _ _ *y*<*x* | orB *ab* =

nl *y* (lemma-trans *y*<*x i j*) (lemma-<=min3E i_1 j_1 (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* i_1 j_1 eh eh) eh) (orA (nf *y* i_2 j_2 eh eh)) with *cmp x y*

... | tri< *x*<*y* _ _ = orA (nf *x i* (le (base *x*<*y*))

(nf *p* i_1 j_1 eh eh) (nf *y* i_2 j_2 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* i_1 j_1 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* j_1 j_1 eh eh))

afmerge (nd *x i j* (nf p_1 i_1 j_1 a_1 b_1) (nf p_2 i_2 j_2 a_2 b_2)) (orA (nf *y* i_3 j_3 *c d*))

with *cmp x y* | ndmerge (nf p_1 i_1 j_1 a_1 b_1) (nf p_2 i_2 j_2 a_2 b_2)

```
... | 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* i_3 j_3 *c d*))
... | tri= _ *x*=*y* _ | (orA *ab*) = orA (nf *y* (lemma-resp *x*=*y j i*)
(lemma-<=min3E i_3 j_3 (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 i_3 j_3 (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 i_3 j_3 (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 i_3 j_3 (le (base *y*<*x*))) *ab* (makeH *x c d*))

afmerge (nd *x i j* (nf *p* i_1 j_1 eh eh) eh) (orB (nf *y* i_2 j_2 *c d*)) with *cmp x y*
... | tri< *x*<*y* _ _ =
orB (nd *x* (le (base *x*<*y*)) *i* (nf *y* i_2 j_2 *c d*) (nf *p* i_1 j_1 eh eh))
... | tri= _ *x*=*y* _ = orB (nd *y*
(lemma-<=min3E i_2 j_2 (eq (base (*sym*== *x*=*y*)))) (snd resp≤ (base *x*=*y*) *i*)
(makeH *x c d*) (nf *p* i_1 j_1 eh eh))
... | tri> _ _ *y*<*x* = orB (nd *y* (lemma-<=min3E i_2 j_2 (le (base *y*<*x*)))
(trans≤ (le (base *y*<*x*)) *i*) (makeH *x c d*) (nf *p* i_1 j_1 eh eh))

afmerge (nd *x i j* (nf p_1 i_1 j_1 a_1 b_1) (nf p_2 i_2 j_2 a_2 b_2)) (orB (nf *y* i_3 j_3 *c d*))
with *cmp x y* | ndmerge (nf p_1 i_1 j_1 a_1 b_1) (nf p_2 i_2 j_2 a_2 b_2)
... | tri< *x*<*y* _ _ | *ab* =
orB (nr *x* (le (base *x*<*y*)) (lemma-<=minE *i j*) (nf *y* i_3 j_3 *c d*) *ab*)
... | tri= _ *x*=*y* _ | *ab* = orB (nr *y*
(lemma-<=min3E i_3 j_3 (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 i_3 j_3 (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)))
  (lemma-trans y<x i j) (makeH x c d) ab)

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 cmp x y | afmerge (nr 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 (nr x i j (nf p1 i1 j1 a1 b1) (nd p2 i2 j2 a2 b2)) (orB (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) = 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) (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 i1 j1 a b) (nf y i2 j2 c d))
  with cmp x y | ndmerge (nf x i1 j1 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 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) (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) i₁)
    (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)) i₁)
  (nf x (le ext) (le ext) eh eh) (nf y (le ext) (le ext) eh eh) , j)

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

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) (orA (nf y i₂ j₂ 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 i1 j1 a b) (nf y i2 j2 c d))
   with cmp x y | afmerge (nr x i1 j1 a b) (orA (nf y i2 j2 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 i1 j1 a b) (nd y i2 j2 c d))
   with cmp y x | afmerge (nd y i2 j2 c d) (orB (nf x i1 j1 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 i1 j1 a b) (nl y i2 j2 c d))
   with cmp y x | afmerge (nl y i2 j2 c d) (orB (nf x i1 j1 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 i1 j1 a b) (nr y i2 j2 c d))
   with cmp y x | afmerge (nr y i2 j2 c d) (orB (nf x i1 j1 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.

Заключение

Представленный в данной работе подход к представлению инвариантов — по одному конструктору на каждый случай инварианта — приводит к неприятному разрастанию функций по обработке структуры данных. Но данный подход позволил написать простые доказательства с помощью интерактивной системы Agda-mode, использующей систему типов для указания типа требуемого терма. Хотелось бы уметь обобщать такие представления инвариантов для упрощения доказательств и уменьшения объема кода.

Литература

1. The Haskell Programming Language. http://www.haskell.org/haskellwiki/Haskell.
2. A Truly Integrated Functional Logic Language. http://www-ps.informatik.uni-kiel.de/currywiki/.
3. Agda language. http://wiki.portal.chalmers.se/agda/pmwiki.php.
4. *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.
5. *Hickey R.* The Clojure programming language / DLS. Под ред. Johan Brichau. ACM, 2008. С. 1. ISBN: 978-1-60558-270-2.
6. *Abelson H.*, *Sussman G. J.* Structure and Interpretation of Computer Programs. MIT Press, 1985. ISBN: 0-262-51036-7.
7. *Milner R.*, *Tofte M.*, *Macqueen D.* The Definition of Standard ML. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.
8. OCaml. http://ocaml.org/.
9. *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.
10. *Sørensen M. H. B.*, *Urzyczyn P.* Lectures on the Curry-Howard Isomorphism. 1998.
11. *Church A.* A Formulation of the Simple Theory of Types // J. Symb. Log. 1940. №2. С. 56—68.
12. *Pierce B. C.* Types and Programming Languages. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0-262-16209-1.
13. *Martin-Löf P.* Intuitionistic Type Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.
14. *Abbott M.*, *Altenkirch T.*, *Ghani N.* Representing Nested Inductive Types Using W-Types / ICALP. Под ред. Josep Díaz, Juhani Karhumäki, Arto Lepistö и Donald Sannella. Т. 3142. Lecture Notes in Computer Science. Springer, 2004. С. 59—71. ISBN: 3-540-22849-7.
15. *McBride C.*, *McKinna J.* The view from the left // J. Funct. Program. 2004. №1. С. 69—111.
16. *Pfenning F.* Unification and Anti-Unification in the Calculus of Constructions / In Sixth Annual IEEE Symposium on Logic in Computer Science. 1991. С. 74—85.
17. *Dybjer P.* Inductive Families // Formal Asp. Comput. 1994. №4. С. 440—465.
18. *Atkey R.*, *Johann P.*, *Ghani N.* Refining Inductive Types // Logical Methods in Computer Science. 2012. №2.
19. *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.
20. *McBride C.* How to Keep Your Neighbours in Order. https://personal.cis.strath.ac.uk/conor.mcbride/Pivotal.pdf.
21. *McBride C.*, *Norell U.*, *Danielsson N. A.* The Agda standard library — AVL trees. http://agda.github.io/agda-stdlib/html/Data.AVL.html.
22. *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.
23. The Agda standard library. http://agda.github.io/agda-stdlib/html/README.html.