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

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Рыбак Андрей Викторович

Представление структур данных индуктивными семействами и доказательства их свойств

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

2014

Содержание

Введение **4**

Глава 1. Обзор **5**

1.1 Функциональное программирование 5
1.1.1 Концепции 5
1.1.2 Сопоставление с образцом 5
1.2 Теория типов 5
1.2.1 Отношение конвертабельности 6
1.2.2 Интуиционистская теория типов 6
1.3 Унификация 7
1.4 Индуктивные семейства 7
1.5 Agda 8
1.6 Выводы по главе 1 9

Глава 2. Описание реализованной структуры данных **10**

2.1 Постановка задачи 10
2.2 Структура данных «двоичная куча» 10
2.3 Выводы по главе 2 23

Список литературы **24**

Введение

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

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

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

Глава 1. Обзор

1.1. Функциональное программирование

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

1.1.1. Концепции

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

1.1.2. Сопоставление с образцом

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

1.2. Теория типов

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

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

1.2.1. Отношение конвертабельности

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

1.2.2. Интуиционистская теория типов

Интуиционистская теория типов основана на математическом конструктивизме [3].

Операторы для типов в ИТТ:

- 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 сумма (пара). Например, тип $\Sigma_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$ — тип пары из числа n и кортежа из n вещественных чисел.

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

1.3. Унификация

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

1.4. Индуктивные семейства

Определение 1.1. *Индуктивное семейство* [4]— это семейство типов данных, которые могут зависеть от других типов и значений.

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

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

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

В работах [5, 6] приведены различные подходы к построению функциональных структур данных.

Пример задания структуры данных и инвариантов — тип данных AVL-дерева и для хранения баланса в AVL-дереве [7].

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

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

1.5. Agda

Agda [8] — чистый функциональный язык программирования с зависимыми типами. В Agda есть поддержка модулей:

```
module AgdaDescription where
```

В коде на Agda широко используются символы Unicode. Тип натуральных чисел — 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:

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

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

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

1.6. Выводы по главе 1

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

Глава 2. Описание реализованной структуры данных

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

2.1. Постановка задачи

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

Требования к данной работе:

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

2.2. Структура данных «двоичная куча»

Определение 2.1. Двоичная куча или пирамида [9] — такое двоичное подвешенное дерево, для которого выполнены следующие три условия:

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

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

```
contradiction : A → ¬ A → B
contradiction a ¬a = ⊥-elim (¬a a)
```

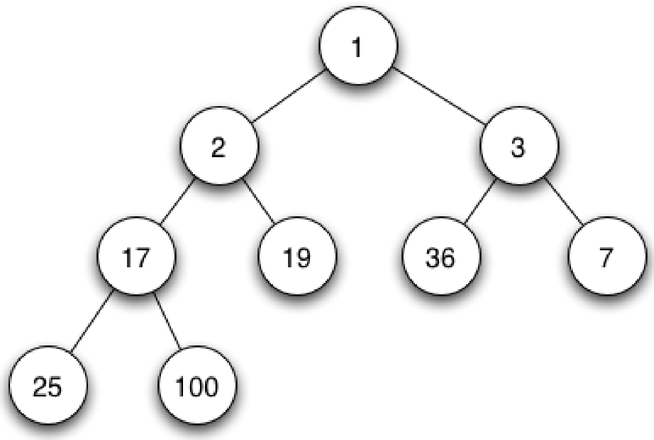

Рис. 2.1. Пример заполненной кучи для минимума

Контрапозиция

$$\mathsf{contraposition} : (A \to B) \to (\neg\ B \to \neg\ A)$$
$$\mathsf{contraposition} = \mathsf{flip}\ \_\circ'\_$$

Пропозициональное равенство из ИТТ.

$$\mathsf{data}\ \_\equiv\_\ \{a\}\ \{A : \mathsf{Set}\ a\}\ (x : A) : A \to \mathsf{Set}\ a\ \mathsf{where}$$
$$\quad \mathsf{refl} : x \equiv x$$

Тип-сумма — зависимая пара.

$$\mathsf{record}\ \Sigma\ \{a\ b\}\ (A : \mathsf{Set}\ a)\ (B : A \to \mathsf{Set}\ b) : \mathsf{Set}\ (a \sqcup b)\ \mathsf{where}$$
$$\quad \mathsf{constructor}\ \_,\_$$
$$\quad \mathsf{field}\ \mathsf{fst} : A\ ;\ \mathsf{snd} : B\ \mathit{fst}$$

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

$$\_\times\_ : \forall\ \{a\ b\}\ (A : \mathsf{Set}\ a) \to (B : \mathsf{Set}\ b) \to \mathsf{Set}\ (a \sqcup b)$$
$$A \times B = \Sigma\ A\ (\lambda\ \_ \to B)$$

Конгруэнтность пропозиционального равенства.

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

Для сравнения элементов нужно задать отношения на этих элементах.

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

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

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

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

```
flip₁ : ∀ {A B : Set} {C : Set₁} → (A → B → C) → B → A → C
flip₁ f a b = f b a
```

```
Cmp : {A : Set} → Rel₂ A → Rel₂ A → Set
Cmp {A} _<_ _==_ = ∀ (x y : A) → Tri (_<_) (_==_) (flip₁ _<_) x y
```

Тип данных для отношения меньше или равно на натуральных числах.

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

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

Все остальные отношения определяются через `_ℕ≤_` .

```
_ℕ<_ _ℕ≥_ _ℕ>_ : Rel₂ ℕ
n ℕ< m = succ n ℕ≤ m
n ℕ> m = m ℕ< n
n ℕ≥ m = m ℕ≤ n
```

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

```
lemma-succ-≡ : ∀ {n} {m} → succ n ≡ succ m → n ≡ m
lemma-succ-≡ refl = refl
lemma-succ-≤ : ∀ {n} {m} → succ (succ n) ℕ≤ succ m → succ n ℕ≤ m
lemma-succ-≤ (s≤s r) = r

cmpℕ : Cmp {ℕ} _ℕ<_ _≡_
cmpℕ zero (zero) = tri= (λ ()) refl (λ ())
cmpℕ zero (succ y) = tri< (s≤s z≤n) (λ ()) (λ ())
cmpℕ (succ x) zero = tri> (λ ()) (λ ()) (s≤s z≤n)
cmpℕ (succ x) (succ y) with cmpℕ x y
... | tri<  a ¬b ¬c = tri< (s≤s a) (contraposition lemma-succ-≡ ¬b) (contraposition
... | tri> ¬a ¬b  c = tri> (contraposition lemma-succ-≤ ¬a) (contraposition lemma-
... | tri= ¬a  b ¬c = tri= (contraposition lemma-succ-≤ ¬a) (cong succ b) (contrapo
```

```
Trans : {A : Set} → Rel₂ A → Set
Trans {A} _rel_ = {a b c : A} → (a rel b) → (b rel c) → (a rel c)
```

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

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

```
Symmetric : ∀ {A : Set} → Rel₂ A → Set
Symmetric _∼_ = ∀ {a b} → a ∼ b → b ∼ a
```

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

```
_Respects_ : ∀ {ℓ} {A : Set} → (A → Set ℓ) → Rel₂ A → Set _
P Respects _∼_ = ∀ {x y} → x ∼ y → P x → P y
```

Частный случай: отношение P соблюдает отношение $\_\sim\_$.

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

Обобщенное отношение 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
```

Лемма: число меньше-равное двух чисел меньше или равно минимума из них.

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

Функция минимума из трех элементов.

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

Аналогичная предыдущей лемма для минимума из трех элементов.

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

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

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

Транзитивность отношения _<=_.

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

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

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

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

```
data expanded (A : Set) : Set where
```

x — элемент исходного типа

```
  # : A → expanded A
```

top — элемент расширение

```
  top : expanded A
```

Теперь нам нужно аналагичным 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
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
trans<E {# _} {# _} {top} _ _ = ext
```

Тип данных расширенного отношения равенства.

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

Расширенное отношение равенства — симметрично и транзитивно.

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

Отношение _<E_ соблюдает отношение _=E_.

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


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) (cont
cmpE (# x) (# y) | tri= a b c = tri= (contraposition lemma-<E a) (base b) (contr
cmpE (# x) (# y) | tri> a b c = tri> (contraposition lemma-<E a) (contraposition
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.

```
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 неполная куча больше нуля.

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

Функция — просмотр минимума в куче.

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

```
peekMin eh = top
peekMin (nd p _ _ _ _) = # p
peekMin (nf p _ _ _ _) = # p
peekMin (nl p _ _ _ _) = # p
peekMin (nr p _ _ _ _) = # p
lemma-<=minE : ∀ {a b c} → a ≤ b → a ≤ c → a ≤ (minE b c)
lemma-<=minE ab ac = lemma-<=min {expanded A}{_<E_}{_=E_}{cmpE} a

min3E : (expanded A) → (expanded A) → (expanded A) → (expanded A)
min3E x y z = min3 cmpE x y z
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}

finsert : ∀ {h m} → (z : A) → Heap m h full
  → Σ HeapState (Heap (minE m (# z)) (succ h))

ainsert : ∀ {h m} → (z : A) → Heap m h almost
  → Σ HeapState (Heap (minE m (# z)) h)

fmerge : ∀ {x y h} → Heap x h full → Heap y h full → OR (Heap x zero full × (x

fpop : ∀ {m h} → Heap m (succ h) full
  → OR (Σ (expanded A) (λ x → (Heap x (succ h) almost) × (m ≤ x))) (Heap to
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
... | orA ((), _ , _)
```

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

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

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

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

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

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*)))

2.3. Выводы по главе 2

Разработаны типы данных для представления структуры данных двоичная куча.

Список литературы

1. Functional programming - Wikipedia. https://en.wikipedia.org/wiki/Functional_programming.
2. *Abelson H.*, *Sussman G. J.* Structure and Interpretation of Computer Programs. MIT Press, 1985. ISBN: 0-262-51036-7.
3. *Martin-Löf P.* Intuitionistic Type Theory. Bibliopolis, 1984. ISBN: 88-7088-105-9.
4. *Dybjer P.* Inductive Families // Formal Asp. Comput. 1994. №4. С. 440–465.
5. *Okasaki C.* Purely Functional Data Structures. Докт. дисс. Pittsburgh, PA 15213, 1996.
6. *McBride C.* How to Keep Your Neighbours in Order. https://personal.cis.strath.ac.uk/conor.mcbride/Pivotal.p
7. *McBride C.*, *Norell U.*, *Danielsson N. A.* The Agda standard library — AVL trees. http://agda.github.io/agda-stdlib/html/Data.AVL.html.
8. Agda language. http://wiki.portal.chalmers.se/agda/pmwiki.php.
9. *Cormen T. H.*, *Leiserson C. E.*, *Rivest R. L.*, *Stein C.* Introduction to Algorithms, Second Edition. The MIT Press и McGraw-Hill Book Company, 2001. ISBN: 0-262-03293-7, 0-07-013151-1.