Zaawansowane programowanie funkcyjne

Daria Walukiewicz-Chrząszcz

28 lutego 2017

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Zaawansowane programowanie funkcyjne

Plan wykładu:

- moja część: dowodzenie twierdzeń i programowanie z typami zależnymi - Idris, Coq
- część Marcina Benke

Zasady zaliczania (podobnie jak w zeszłym roku), czyli:

- laboratorium: zadanie z Coqa + projekt 1-3 osobowy
- egzamin ustny, którego istotną częścią jest prezentacja projektu
- alternatywna forma zaliczenia: referat (koniecznie ciekawy!) ...możliwe także inne formy

Motywacja dla dowodzenia twierdzeń: poprawność

- Dowodzenie własności programów

Przykład z JPP

ciężko testować np funkcję insert

```
...
insert :: a -> [a] -> [a]
...
```

czy dla dowolnej listy xs i elementu z spełnia własność:

```
...
sorted (xs) -> sorted (insert z xs)
...
```

Motywacja dla dowodzenia twierdzeń: poprawność

- Dowodzenie własności programów

Przykład z JPP

ciężko testować np funkcję insert

```
...
insert :: a -> [a] -> [a]
...
```

czy dla dowolnej listy xs i elementu z spełnia własność:

```
...
sorted (xs) -> sorted (insert z xs)
...
```

Motywacja dla dowodzenia twierdzeń: poprawność

- Dowodzenie własności programów

Przykład z JPP

ciężko testować np funkcję insert

```
...
insert :: a -> [a] -> [a]
...
```

czy dla dowolnej listy xs i elementu z spełnia własność:

```
...
sorted (xs) -> sorted (insert z xs)
...
```

Motywacja dla typów zależnych - specyfikacje

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

Motywacja dla typów zależnych - specyfikacje

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

Motywacja dla typów zależnych - specyfikacje

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

Motywacja dla typów zależnych - specyfikacje

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ

$$\forall n : nat.M_{n,n}$$

- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
  typ `ftree n` zależy od wartości `n` (typy zależne)
  typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat.ftree\ n$$

$$\Pi n : nat.ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat` $\to$ `bool`

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ

$$\forall n : nat. M_{n,n}$$

- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
  typ `ftree n` zależy od wartości `n` (typy zależne)
  typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

$$\texttt{forall n : nat, ftree n}$$

Uwaga: $\texttt{forall n : nat, bool} \equiv \texttt{nat} \to \texttt{bool}$

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ
$$\forall n : nat.M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat.ftree\ n$$

$$\Pi n : nat.ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat → bool`

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat` $\to$ `bool`

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat` $\to$ `bool`

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. \mathit{ftree}\, n$$

$$\Pi n : nat. \mathit{ftree}\, n$$

$$\texttt{forall n : nat, ftree n}$$

Uwaga: $\texttt{forall n : nat, bool} \equiv \texttt{nat} \to \texttt{bool}$

Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach n na n ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że e jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. \mathit{ftree}\, n$$

$$\Pi n : nat. \mathit{ftree}\, n$$

$$\texttt{forall n : nat, ftree n}$$

Uwaga: $\texttt{forall n : nat, bool} \quad \equiv \quad \texttt{nat} \to \texttt{bool}$

Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typy zależne - potrzebują dowodów

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity 0 = Even 0
parity (S 0) = Odd 0
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typ `Even (S j)` to `Parity((S j) + (S j))` a oczekiwany jest `Parity(S (S k))` gdzie `k` to `j+j`.
Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

Typy zależne - potrzebują dowodów

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typ `Even (S j)` to `Parity((S j) + (S j))` a oczekiwany jest `Parity(S (S k))` gdzie `k` to `j+j`.
Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

Typy zależne - potrzebują dowodów

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Typ `Even (S j)` to `Parity((S j) + (S j))` a oczekiwany jest `Parity(S (S k))` gdzie `k` to `j+j`.
Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

Idris

- rozwijany od około 2008 roku
- http://www.idris-lang.org/
- główny projektant i wykonawca to Edwin Brady z University of St Andrews.
- https://edwinb.wordpress.com/
- autor powstającej na bieżąco książki “Type-driven development with Idris”
- Idris (chyba) jeszcze nie ma dobrze sprawdzonych podstaw teoretycznych

Idris

- rozwijany od około 2008 roku
- http://www.idris-lang.org/
- główny projektant i wykonawca to Edwin Brady z University of St Andrews.
- https://edwinb.wordpress.com/
- autor powstającej na bieżąco książki "Type-driven development with Idris"
- Idris (chyba) jeszcze nie ma dobrze sprawdzonych podstaw teoretycznych

Idris

- rozwijany od około 2008 roku
- http://www.idris-lang.org/
- główny projektant i wykonawca to Edwin Brady z University of St Andrews.
- https://edwinb.wordpress.com/
- autor powstającej na bieżąco książki "Type-driven development with Idris"
- Idris (chyba) jeszcze nie ma dobrze sprawdzonych podstaw teoretycznych

Idris

- rozwijany od około 2008 roku
- http://www.idris-lang.org/
- główny projektant i wykonawca to Edwin Brady z University of St Andrews.
- https://edwinb.wordpress.com/
- autor powstającej na bieżąco książki "Type-driven development with Idris"
- Idris (chyba) jeszcze nie ma dobrze sprawdzonych podstaw teoretycznych

Idris

- rozwijany od około 2008 roku
- http://www.idris-lang.org/
- główny projektant i wykonawca to Edwin Brady z University of St Andrews.
- https://edwinb.wordpress.com/
- autor powstającej na bieżąco książki “Type-driven development with Idris”
- Idris (chyba) jeszcze nie ma dobrze sprawdzonych podstaw teoretycznych

Idris

- rozwijany od około 2008 roku
- http://www.idris-lang.org/
- główny projektant i wykonawca to Edwin Brady z University of St Andrews.
- https://edwinb.wordpress.com/
- autor powstającej na bieżąco książki "Type-driven development with Idris"
- Idris (chyba) jeszcze nie ma dobrze sprawdzonych podstaw teoretycznych

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - czysty język funkcyjny

- rozbudowany system typów służy specyfikowaniu i lepszemu/łatwiejszemu implementowaniu funkcji
- znajdowanie programu spełniającego specyfikację jest jak "solving a puzzle"
- prawdziwy język programowania (można kompilować i uruchamiać programy:)
- składniowo dosyć podobny do Haskella, ale są zamienione znaczenia : i ::, type classes nazywają się interfejsami ...
- gorliwa strategia obliczania, choć możliwe jest wprowadzenie leniwości
- z powodu typów zależnych potrzebna jest ewaluacja przy sprawdzaniu typów (powiedzmy taka jak w Coqu w konwersji)
- funkcje potrzebne do typowania muszą być całkowite (total) i terminujące (spełniające "size change principle")
- przy generowaniu programu wykonywalnego kompilator "odrzuca" części potrzebne tylko do typowania

Idris - sprawdzanie typów, kompilacja

- wejście do repl poprzez idris plik.idr
- tam można używać komend, np :t, :q (:? to ich lista)
- kompilacja porzez idris -o plik plik.idr
- lub w repl :c plik i :exec

Idris - sprawdzanie typów, kompilacja

- wejście do repl poprzez idris plik.idr
- tam można używać komend, np :t, :q (:? to ich lista)
- kompilacja porzez idris -o plik plik.idr
- lub w repl :c plik i :exec

Idris - sprawdzanie typów, kompilacja

- wejście do repl poprzez idris plik.idr
- tam można używać komend, np :t, :q (:? to ich lista)
- kompilacja porzez idris -o plik plik.idr
- lub w repl :c plik i :exec

Idris - sprawdzanie typów, kompilacja

- wejście do repl poprzez idris plik.idr
- tam można używać komend, np :t, :q (:? to ich lista)
- kompilacja porzez idris -o plik plik.idr
- lub w repl :c plik i :exec

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Idris i typy zależne - przykłady

- Hello.idr
- HelloHole.idr
- FCTypes.idr
- Generic.idr
- HOF.idr
- Let_Where.idr
- Vectors.idr
- TCVects.idr
- AppendVec.idr
- WordLength_vec.idr
- Adder.idr
- removeElem.idr
- plus_theorem.idr
- Void.idr
- parity.idr
- binary.idr
- AppendVecRew.idr

Typy zależne i zależny pattern-matching

- typy zależą od wartości
- nie ma syntaktycznej różnicy między typem i wartością
- typy są wartościami
- można pisać funkcje, dla których typ wyniku zależy od wartości argumentu

Typy zależne i zależny pattern-matching

- typy zależą od wartości
- nie ma syntaktycznej różnicy między typem i wartością
- typy są wartościami
- można pisać funkcje, dla których typ wyniku zależy od wartości argumentu

Typy zależne i zależny pattern-matching

- typy zależą od wartości
- nie ma syntaktycznej różnicy między typem i wartością
- typy są wartościami
- można pisać funkcje, dla których typ wyniku zależy od wartości argumentu

Typy zależne i zależny pattern-matching

- typy zależą od wartości
- nie ma syntaktycznej różnicy między typem i wartością
- typy są wartościami
- można pisać funkcje, dla których typ wyniku zależy od wartości argumentu

Sprawdzanie totalności

Funkcja jest *total* jeśli:

- jej definicja pokrywa wszystkie przypadki wejść,
- jest dobrze ufundowana - przy wywołaniu rekurencyjnym argumenty maleją
- nie używa typów danych, które nie są "strictly positive"
- nie odwołuje się do nietotalnych funkcji

Sprawdzanie totalności

Funkcja jest *total* jeśli:

- jej definicja pokrywa wszystkie przypadki wejść,
- jest dobrze ufundowana - przy wywołaniu rekurencyjnym argumenty maleją
- nie używa typów danych, które nie są "strictly positive"
- nie odwołuje się do nietotalnych funkcji

Sprawdzanie totalności

Funkcja jest *total* jeśli:

- jej definicja pokrywa wszystkie przypadki wejść,
- jest dobrze ufundowana - przy wywołaniu rekurencyjnym argumenty maleją
- nie używa typów danych, które nie są “strictly positive”
- nie odwołuje się do nietotalnych funkcji

Sprawdzanie totalności

Funkcja jest *total* jeśli:

- jej definicja pokrywa wszystkie przypadki wejść,
- jest dobrze ufundowana - przy wywołaniu rekurencyjnym argumenty maleją
- nie używa typów danych, które nie są “strictly positive”
- nie odwołuje się do nietotalnych funkcji