Idris

Daria Walukiewicz-Chrząszcz

Advanced Functional Programming

5 march 2019

Advanced functional programming

Plan of my part:

- theorem proving and programming with dependent types
- Idris (1 lecture)
- Coq (6 lectures)
- Coq project (grades)

Advanced functional programming

Plan of my part:

- theorem proving and programming with dependent types
- Idris (1 lecture)
- Coq (6 lectures)
- Coq project (grades)

Advanced functional programming

Plan of my part:

- theorem proving and programming with dependent types
- Idris (1 lecture)
- Coq (6 lectures)
- Coq project (grades)

Advanced functional programming

Plan of my part:

- theorem proving and programming with dependent types
- Idris (1 lecture)
- Coq (6 lectures)
- Coq project (grades)

Advanced functional programming

Plan of my part:

- theorem proving and programming with dependent types
- Idris (1 lecture)
- Coq (6 lectures)
- Coq project (grades)

Motivation for dependent types: specifications

- types become more precise
- finer types specify better the properties of the function

Inductive $ftree : nat \rightarrow$ Set :=
| $Leaf$: $ftree\ 0$
| $Node : \forall\ n{:}\ nat,\ Z \rightarrow ftree\ n \rightarrow ftree\ n \rightarrow ftree\ (S\ n)$.

Definition $root\ (n : nat)(t : ftree(S\ n)) : Z$:=
  match t with
  | $Node\ n\ k\ l\ r \Rightarrow k$
  end.

Motivation for dependent types: specifications

- types become more precise
- finer types specify better the properties of the function

Inductive $ftree : nat \to \text{Set} :=$
$| \; Leaf : ftree \; 0$
$| \; Node : \forall \; n{:} \; nat, Z \to ftree \; n \to ftree \; n \to ftree \; (S \; n).$

Definition $root \; (n : nat)(t : ftree(S \; n)) : Z :=$
  match t with
  $| \; Node \; n \; k \; l \; r \Rightarrow k$
  end.

Motivation for dependent types: specifications

- types become more precise
- finer types specify better the properties of the function

Inductive $\mathit{ftree} : \mathit{nat} \to \mathrm{Set} :=$
$|\ \mathit{Leaf} : \mathit{ftree}\ 0$
$|\ \mathit{Node} : \forall\ n : \mathit{nat},\ Z \to \mathit{ftree}\ n \to \mathit{ftree}\ n \to \mathit{ftree}\ (S\ n).$

Definition $\mathit{root}\ (n : \mathit{nat})(t : \mathit{ftree}(S\ n)) : Z :=$
match t with
$|\ \mathit{Node}\ n\ k\ l\ r \Rightarrow k$
end.

Motivation for dependent types: specifications

- types become more precise
- finer types specify better the properties of the function

Inductive $\mathit{ftree} : \mathit{nat} \to \texttt{Set} :=$
$|\ \mathit{Leaf} : \mathit{ftree}\ 0$
$|\ \mathit{Node} : \forall\ n{:}\ \mathit{nat},\ Z \to \mathit{ftree}\ n \to \mathit{ftree}\ n \to \mathit{ftree}\ (S\ n).$

Definition $\mathit{root}\ (n : \mathit{nat})(t : \mathit{ftree}(S\ n)) : Z :=$
  match t with
  $|\ \mathit{Node}\ n\ k\ l\ r \Rightarrow k$
  end.

Motivation for dependent types: specifications

- types become more precise
- finer types specify better the properties of the function

$\texttt{Inductive}\ \mathit{ftree} : \mathit{nat} \to \texttt{Set} :=$
$\mid \mathit{Leaf} : \mathit{ftree}\ 0$
$\mid \mathit{Node} : \forall\ n{:}\ \mathit{nat},\ Z \to \mathit{ftree}\ n \to \mathit{ftree}\ n \to \mathit{ftree}\ (S\ n).$

$\texttt{Definition}\ \mathit{root}\ (n : \mathit{nat})(t : \mathit{ftree}(S\ n)) : Z :=$
$\quad \texttt{match}\ t\ \texttt{with}$
$\quad \mid \mathit{Node}\ n\ k\ l\ r \Rightarrow k$
$\quad \texttt{end}.$

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type

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

- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$\texttt{(n : nat)} \to \texttt{ftree n}$$

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

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type

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

- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$(\texttt{n} : \texttt{nat}) \to \texttt{ftree n}$$

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

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type

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

- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$\mathtt{(n : nat) \to ftree\ \ n}$$

Convention: $\mathtt{forall\ n : nat, bool} \quad \equiv \quad \mathtt{nat \to bool}$

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type

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

- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$\mathtt{(n : nat) \to ftree\ n}$$

Convention: $\mathtt{forall\ n : nat, bool} \quad \equiv \quad \mathtt{nat \to bool}$

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type

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

- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$\mathtt{(n : nat) \to ftree\ n}$$

Convention: $\mathtt{forall\ n : nat, bool} \quad \equiv \quad \mathtt{nat \to bool}$

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type

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

- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$(\mathtt{n : nat}) \to \mathtt{ftree\ n}$$

Convention: $\mathtt{forall\ n : nat, bool} \equiv \mathtt{nat} \to \mathtt{bool}$

Dependent types - introduction

Functional dependent type - type of a function whose codomain depends on an argument

- mapping $n \mapsto I_{n,n}$ that assigns an identity matrix of size n to a natural number n has type
$$\forall n : nat.M_{n,n}$$
- in monoid $(A, e, \cdot)$, operation $\cdot$ is a function that has type $A \to A \to A$, and such that e is its identity element
- type `List A` depends on a type `A` (polimorphism)
  type `ftree n` depends on a value `n` (dependent types)
  type `vector A n` depends on a type `A` and value `n`(dependent types)

Notations:

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

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

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

$$(\texttt{n} : \texttt{nat}) \to \texttt{ftree n}$$

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

Dependent types - computations in types

$$\texttt{ftree (2+2)} \equiv \texttt{ftree (4)}$$

these types are *convertible* - should be regarded as internally equal

Attention:
for $+$ defined by pattern matching on first argument:

```
0 + y = y
(S x) + y = S (x+y)
```

- 2+2 computes to 4
- 0+n computes to n
- but n+0 does not compute to n
  (equality can be proved by induction)

Dependent types - computations in types

$$\texttt{ftree (2+2)} \equiv \texttt{ftree (4)}$$

these types are *convertible* - should be regarded as internally equal

Attention:
for $+$ defined by pattern matching on first argument:

```
0 + y = y
(S x) + y = S (x+y)
```

- 2+2 computes to 4
- 0+n computes to n
- but n+0 does not compute to n
  (equality can be proved by induction)

Dependent types - computations in types

$$\texttt{ftree (2+2)} \equiv \texttt{ftree (4)}$$

these types are *convertible* - should be regarded as internally equal

Attention:
for $+$ defined by pattern matching on first argument:

```
0 + y = y
(S x) + y = S (x+y)
```

- 2+2 computes to 4
- 0+n computes to n
- but n+0 does not compute to n
  (equality can be proved by induction)

Dependent types - computations in types

$$\texttt{ftree (2+2)} \equiv \texttt{ftree (4)}$$

these types are *convertible* - should be regarded as internally equal

Attention:
for $+$ defined by pattern matching on first argument:

```
0 + y = y
(S x) + y = S (x+y)
```

- 2+2 computes to 4
- `0+n` computes to `n`
- but `n+0` does not compute to `n`
  (equality can be proved by induction)

Dependent types - computations in types

$$\texttt{ftree (2+2)} \equiv \texttt{ftree (4)}$$

these types are *convertible* - should be regarded as internally equal

Attention:
for $+$ defined by pattern matching on first argument:

```
0 + y = y
(S x) + y = S (x+y)
```

- 2+2 computes to 4
- `0+n` computes to `n`
- but `n+0` does not compute to `n`
  (equality can be proved by induction)

Dependent types - in simplified Idris

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

hence `Even i : Parity (i+i)` for a given `i : nat`

```
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)
```

Dependent types - in simplified Idris

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

hence `Even i : Parity (i+i)` for a given `i : nat`

```
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)
```

Dependent types - in simplified Idris

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

hence `Even i : Parity (i+i)` for a given `i : nat`

```
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)
```

Dependent types naturally need proofs

```
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)
```

Type of `Even (S j)` is `Parity((S j) + (S j))`, but expected type is `Parity(S (S k))` where k is j+j.
Conclusion: we need a proof that `S (j+(S j)` equals `S (S (j+j))`

Dependent types naturally need proofs

```
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)
```

Type of `Even (S j)` is `Parity((S j) + (S j))`, but expected type is `Parity(S (S k))` where k is j+j.
Conclusion: we need a proof that `S (j+(S j)` equals `S (S (j+j))`

Dependent types naturally need proofs

```
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)
```

Type of `Even (S j)` is `Parity((S j) + (S j))`, but expected type is `Parity(S (S k))` where `k` is `j+j`.

Conclusion: we need a proof that `S (j+(S j)` equals `S (S (j+j))`

Dependent types naturally need proofs

```
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)
```

Type of `Even (S j)` is `Parity((S j) + (S j))`, but expected type is `Parity(S (S k))` where `k` is `j+j`.
Conclusion: we need a proof that `S (j+(S j)` equals `S (S (j+j))`

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- started in 2008
- http://www.idris-lang.org/
- development led by Edwin Brady at the University of St Andrews
- https://edwinb.wordpress.com/
- "Type-driven development with Idris" Edwin Brady, published by Manning, March 2017
- Idris based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- some of its metatheoretic properties are conjectured (not yet proved)

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as “solving a puzzle”: the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as “solving a puzzle”: the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as “solving a puzzle”: the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as “solving a puzzle”: the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as “solving a puzzle”: the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris

- general purpose pure functional programming language with dependent types
- syntax similar to Haskell, but the meanings of : and :: are interchanged
- type declarations required
- eager evaluation, lazy computations are possible
- dependent types
- types are first class language constructs (can be arguments to functions, returned from functions)
- dependent types provide better specifications of functions
- but writing a function that satisfies its specification may need proofs
- type-driven development treats programming as "solving a puzzle": the program is the solution to the puzzle, the type the goal of the puzzle
- because of dependent types, evaluation is needed at type-checking
- functions used in evaluation must be total and terminating
- compiler gets rid of the parts needed only for type checking

Idris - getting started

- installation: `cabal update; cabal install idris`
- `idris foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris -o foo foo.idr`
- or using `:c foo` and `:exec` commands

Idris - getting started

- installation: `cabal update; cabal install idris`
- `idris foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris -o foo foo.idr`
- or using `:c foo` and `:exec` commands

Idris - getting started

- installation: `cabal update; cabal install idris`
- `idris foo.idr` enters the interactive environment, similar to ghci
- commands, `:t, :q` (type `:?` for full list of commands)
- compilation: `idris -o foo foo.idr`
- or using `:c foo` and `:exec` commands

Idris - getting started

- installation: `cabal update; cabal install idris`
- `idris foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris -o foo foo.idr`
- or using `:c foo` and `:exec` commands

Idris - getting started

- installation: `cabal update; cabal install idris`
- `idris foo.idr` enters the interactive environment, similar to ghci
- commands, `:t, :q` (type `:?` for full list of commands)
- compilation: `idris -o foo foo.idr`
- or using `:c foo` and `:exec` commands

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Idris and dependent types - examples

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

Interfaces

- similar to type classes in Haskell
- there can be many implementations for one type

(see Eq.idr Tree.idr)

Equality in Idris

- == is not adequate
- equality defined at the level of types

(see EqNat.idr, ExactLength.idr)

Totality checking

Function is *total* if it

- covers all possible inputs
- is well-founded (in recursive calls arguments are decreasing)
- does not use any data types which are not strictly positive
- does not call any non-total functions

Totality checking

Function is *total* if it

- covers all possible inputs
- is well-founded (in recursive calls arguments are decreasing)
- does not use any data types which are not strictly positive
- does not call any non-total functions

Totality checking

Function is *total* if it

- covers all possible inputs
- is well-founded (in recursive calls arguments are decreasing)
- does not use any data types which are not strictly positive
- does not call any non-total functions

Totality checking

Function is *total* if it

- covers all possible inputs
- is well-founded (in recursive calls arguments are decreasing)
- does not use any data types which are not strictly positive
- does not call any non-total functions

Totality checking

Function is *total* if it

- covers all possible inputs
- is well-founded (in recursive calls arguments are decreasing)
- does not use any data types which are not strictly positive
- does not call any non-total functions