Idris

Daria Walukiewicz-Chrząszcz

Advanced Functional Programming

12 March 2024

Advanced functional programming

My part of the lecture:
theorem proving and programming with dependent types

Plan:

- Idris (1 lecture)
- Coq (6 lectures)
  Coq project (grades)

Advanced functional programming

My part of the lecture:
theorem proving and programming with dependent types

Plan:

- Idris (1 lecture)
- Coq (6 lectures)
  Coq project (grades)

Advanced functional programming

My part of the lecture:
theorem proving and programming with dependent types

Plan:

- Idris (1 lecture)
- Coq (6 lectures)
  Coq project (grades)

Advanced functional programming

My part of the lecture:
theorem proving and programming with dependent types

Plan:

- Idris (1 lecture)
- Coq (6 lectures)
  Coq project (grades)

Motivation for dependent types: specifications

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

Inductive $ftree$: $nat \to$ 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 better specify the properties of the function

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

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

Motivation for dependent types: specifications

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

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

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

Motivation for dependent types: specifications

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

Inductive $ftree : nat \to$ 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 better specify the properties of the function

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

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

Dependent types - introduction

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

$$\forall n : nat. Array\, n$$

Notations:

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

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

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

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

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

Dependent types - introduction

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

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

Dependent types - introduction

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

Notations:

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

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

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

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

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

Notations:

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

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

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

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

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

Notations:

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

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

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

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

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

Notations:

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

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

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

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

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

Notations:

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

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

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

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

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

Dependent types - introduction

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

- M : *Array* n means that M is an array of size n,
- *Array* : $nat \to \star$ is a type constructor,
- *Zeroes* n : *Array* n is an array of n zeroes,
- mapping $n \mapsto$ *Zeroes* n has functional dependent type

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

Notations:

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

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

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

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

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

Dependencies in types

type `List A` depends on a type `A` (polimorphism)
type `ftree n` depends on a value `n` (dependent type)
type `vector A n` depends on a type `A` and value `n` (dependent type)

Dependent types - computations in types

```
ftree (2+2) ≡ 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

`ftree (2+2)` ≡ `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

`ftree (2+2)` $\equiv$ `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 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 - 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 - 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 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 0 = Even 0
parity (S 0) = Odd 0
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)   ?: S (S (j+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 0 = Even 0
parity (S 0) = Odd 0
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)   ?: S (S (j+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 0 = Even 0
parity (S 0) = Odd 0
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)   ?: S (S (j+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 0 = Even 0
parity (S 0) = Odd 0
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)   ?: S (S (j+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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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
- Idris1 based on core Type Theory ("Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation", Journal of Functional Programming 2013)
- Idris2 based on Quantitative Type Theory ("The Syntax and Semantics of Quantitative Type Theory", Robert Atkey, LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom)

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

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 is 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 arguments to functions and constructors bound with quantity/multiplicity 0; erased arguments are still relevant at compile time.

Idris - getting started

- installation: see https://www.idris-lang.org/pages/download.html
- `idris2 foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris2 --cg racket -o foo foo.idr`
  executable in `build/exec`

Idris - getting started

- installation: see https://www.idris-lang.org/pages/download.html
- `idris2 foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris2 --cg racket -o foo foo.idr`
  executable in `build/exec`

Idris - getting started

- installation: see https://www.idris-lang.org/pages/download.html
- `idris2 foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris2 --cg racket -o foo foo.idr`
  executable in `build/exec`

Idris - getting started

- installation: see https://www.idris-lang.org/pages/download.html
- `idris2 foo.idr` enters the interactive environment, similar to ghci
- commands, `:t`, `:q` (type `:?` for full list of commands)
- compilation: `idris2 --cg racket -o foo foo.idr`
  executable in `build/exec`

Idris and dependent types - examples

- Hello.idr
- Generic.idr
- Let_Where.idr
- FCTypes.idr
- Vectors.idr
- TCVects.idr
- WordLength_vec.idr
- ApplyVec.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

Multiplicities

Allow to control

- linearity (used exactly once)
- erasure (not used at runtime)
- and unrestricted use.

(see Multiplicities.idr)

Multiplicities

Allow to control

- linearity (used exactly once)
- erasure (not used at runtime)
- and unrestricted use.

(see Multiplicities.idr)

Multiplicities

Allow to control

- linearity (used exactly once)
- erasure (not used at runtime)
- and unrestricted use.

(see Multiplicities.idr)