Generic Bidirectional Typing for Dependent Type Theories

Thiago Felicissimo

August 21, 2024

Mas antes de falar de ciência...

Engenheiro pela Télécom Paris

Mestrado pelo Master Parisien de Recherche en Informatique (MPRI)

Mas antes de falar de ciência...

Engenheiro pela Télécom Paris

Mestrado pelo Master Parisien de Recherche en Informatique (MPRI)

Doutorando no final do 3o ano (defesa em menos de um mês), na Deducteam

Sob a direção de Gilles Dowek e Frédéric Blanqui

Mas antes de falar de ciência...

Engenheiro pela Télécom Paris

Mestrado pelo Master Parisien de Recherche en Informatique (MPRI)

Doutorando no final do 3o ano (defesa em menos de um mês), na Deducteam

Sob a direção de Gilles Dowek e Frédéric Blanqui

Mas antes de tudo isso…

Mas antes de falar de ciência...

Engenheiro pela Télécom Paris e pela UFMG

Mestrado pelo Master Parisien de Recherche en Informatique (MPRI)

Doutorando no final do 3o ano (defesa em menos de um mês), na Deducteam

Sob a direção de Gilles Dowek e Frédéric Blanqui

Mas antes de tudo isso…

Estudei engenharia elétrica na UFMG, antes de ir pro duplo diploma na Télécom

Fiz uma iniciação científica no DCC, no LECOM

Generic Bidirectional Typing for Dependent Type Theories

Thiago Felicissimo

August 21, 2024

Dependent type theory, in a nutshell

In dependent type theory:

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have types

$$[0, 1, 2] : \mathsf{List\ Nat}$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathrm{Vec}\ \mathrm{Nat}\ 3$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathrm{Vec}\ \mathrm{Nat}\ 3$$

- Functions can be *dependent*

$$\lambda n.[1, \dots, n] : ?$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3$$

- Functions can be *dependent*

$$\lambda n.[1, \ldots, n] : \mathsf{Nat} \to \mathsf{List}\ \mathsf{Nat}$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3$$

- Functions can be *dependent*

$$\lambda n.[1, \ldots, n] : \Pi(n : \mathsf{Nat}).\mathsf{Vec}\ \mathsf{Nat}\ n$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3$$

- Functions can be *dependent*

$$\lambda n.[1, \ldots, n] : \Pi(n : \mathsf{Nat}).\mathsf{Vec}\ \mathsf{Nat}\ n$$

- Types are equal modulo computation

$$\frac{[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3}{\quad}$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3$$

- Functions can be *dependent*

$$\lambda n.[1, \ldots, n] : \Pi(n : \mathsf{Nat}).\mathsf{Vec}\ \mathsf{Nat}\ n$$

- Types are equal modulo computation

$$\frac{[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3 \qquad 3 \equiv 2 + 1}{}$$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

$$[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3$$

- Functions can be *dependent*

$$\lambda n.[1, \ldots, n] : \Pi(n : \mathsf{Nat}).\mathsf{Vec}\ \mathsf{Nat}\ n$$

- Types are equal modulo computation

$$\frac{[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ 3 \qquad 3 \equiv 2 + 1}{[0, 1, 2] : \mathsf{Vec}\ \mathsf{Nat}\ (2 + 1)}$$

Why dependent type theory?

A foundation for mathematics, by the **Curry-Howard correspondence**

Propositions as types, proofs as programs. Proof/type theory dictionary

Why dependent type theory?

A foundation for mathematics, by the **Curry-Howard correspondence**

Propositions as types, proofs as programs. Proof/type theory dictionary

Used in many popular *proof assistants*, such as Coq, Lean and Agda

Why dependent type theory?

A foundation for mathematics, by the **Curry-Howard correspondence**

Propositions as types, proofs as programs. Proof/type theory dictionary

Used in many popular *proof assistants*, such as Coq, Lean and Agda

Dependently-typed programming Dependent types allow to write both data and specification in the *same* language

(* pre-condition: list not empty *)

$\mathsf{hd} : \mathsf{List\ Nat} \longrightarrow \mathsf{Nat}$

$\mathsf{hd} \quad (x :: l) = x$

$\mathsf{hd} \quad [] = \textbf{FAIL}$

Why dependent type theory?

A foundation for mathematics, by the **Curry-Howard correspondence**

Propositions as types, proofs as programs. Proof/type theory dictionary

Used in many popular *proof assistants*, such as Coq, Lean and Agda

Dependently-typed programming Dependent types allow to write both data and specification in the *same* language

$$\mathsf{hd} : \Pi(n : \mathsf{Nat}).\ \mathsf{Vec}\ \mathsf{Nat}\ (n + 1) \longrightarrow \mathsf{Nat}$$
$$\mathsf{hd}\ n\ (x :: l) = x$$

The syntax of type theory

When defining syntax of type theories, many choices:

The syntax of type theory

When defining syntax of type theories, many choices:

Fully-annotated syntax keeps track of all annotations

$$t \mathbin{@}_{A,x.B} u \qquad \langle t, u \rangle_{A,x.B} \qquad t ::_A l \qquad \ldots$$

Arguably the most canonical choice

The syntax of type theory

When defining syntax of type theories, many choices:

Fully-annotated syntax keeps track of all annotations

$$t \mathbin{@}_{A,x.B} u \qquad \langle t, u \rangle_{A,x.B} \qquad t ::_A l \qquad \dots$$

Arguably the most canonical choice, but the syntax is unusable in practice…

The syntax of type theory

When defining syntax of type theories, many choices:

Fully-annotated syntax keeps track of all annotations

$$t \mathbin{@_{A,x.B}} u \qquad \langle t, u \rangle_{A,x.B} \qquad t ::_A l \qquad \dots$$

Arguably the most canonical choice, but the syntax is unusable in practice…

Non-annotated syntax restores usability by eliding parameter annotations

$$t\ u \qquad \langle t, u \rangle \qquad t :: l \qquad \dots$$

The syntax of type theory

When defining syntax of type theories, many choices:

Fully-annotated syntax keeps track of all annotations

$$t \mathbin{@_{A,x.B}} u \qquad \langle t, u \rangle_{A,x.B} \qquad t ::_A l \qquad \ldots$$

Arguably the most canonical choice, but the syntax is unusable in practice…

Non-annotated syntax restores usability by eliding parameter annotations

$$t\ u \qquad \langle t, u \rangle \qquad t :: l \qquad \ldots$$

Syntax so common that many don't realize that an omission is being made

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[u/x]}$$

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

A solution for simple types Store the constraints on unknown types, then solve them using unification (Hindley-Milner type inference)

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

A solution for simple types Store the constraints on unknown types, then solve them using unification (Hindley-Milner type inference)

$$\frac{}{\vdash \lambda f.\lambda x.f\ x : \alpha_0}$$

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

A solution for simple types Store the constraints on unknown types, then solve them using unification (Hindley-Milner type inference)

$$\alpha_0 = \alpha_1 \to \alpha_2 \to \alpha_3 \; \frac{\dfrac{}{f : \alpha_1, x : \alpha_2 \vdash f\ x : \alpha_3}}{\vdash \lambda f.\lambda x.f\ x : \alpha_0}$$

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

A solution for simple types Store the constraints on unknown types, then solve them using unification (Hindley-Milner type inference)

$$\alpha_0 = \alpha_1 \to \alpha_2 \to \alpha_3 \; \frac{\alpha_4 = \alpha_5 \to \alpha_3 \; \dfrac{\dfrac{}{f : \alpha_1, x : \alpha_2 \vdash f : \alpha_4} \qquad \dfrac{}{f : \alpha_1, x : \alpha_2 \vdash x : \alpha_5}}{f : \alpha_1, x : \alpha_2 \vdash f\ x : \alpha_3}}{\vdash \lambda f.\lambda x.f\ x : \alpha_0}$$

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

A solution for simple types Store the constraints on unknown types, then solve them using unification (Hindley-Milner type inference)

$$\alpha_0 = \alpha_1 \to \alpha_2 \to \alpha_3 \frac{\alpha_4 = \alpha_5 \to \alpha_3 \dfrac{\alpha_1 = \alpha_4 \dfrac{}{f : \alpha_1, x : \alpha_2 \vdash f : \alpha_4} \qquad \alpha_2 = \alpha_5 \dfrac{}{f : \alpha_1, x : \alpha_2 \vdash x : \alpha_5}}{f : \alpha_1, x : \alpha_2 \vdash f\ x : \alpha_3}}{\vdash \lambda f.\lambda x.f\ x : \alpha_0}$$

Typechecking without annotations

Omission has a cost Knowing annotations is needed for typing

$$\frac{\Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash t\ u : ?}$$

How to verify program $t\ u$ is typed if A and B are not stored in syntax?

A solution for simple types Store the constraints on unknown types, then solve them using unification (Hindley-Milner type inference)

$$\alpha_0 = \alpha_1 \to \alpha_2 \to \alpha_3 \frac{\alpha_4 = \alpha_5 \to \alpha_3 \dfrac{\alpha_1 = \alpha_4 \dfrac{}{f : \alpha_1, x : \alpha_2 \vdash f : \alpha_4} \qquad \alpha_2 = \alpha_5 \dfrac{}{f : \alpha_1, x : \alpha_2 \vdash x : \alpha_5}}{f : \alpha_1, x : \alpha_2 \vdash f\ x : \alpha_3}}{\vdash \lambda f.\lambda x.f\ x : \alpha_0}$$

Unification succeeds, with $\alpha_0 \mapsto (\alpha_5 \to \alpha_3) \to \alpha_5 \to \alpha_3$

Why does Hindley-Miler type inference works?

Simple types are *1st order*

$$A, B ::= \mathrm{Nat} \mid A \rightarrow B$$

Why does Hindley-Miler type inference works?

Simple types are *1st order*

$$A, B ::= \mathsf{Nat} \mid A \to B$$

1st order unification is decidable and has most general unifiers

Why does Hindley-Miler type inference works?

Simple types are *1st order*

$$A, B ::= \mathsf{Nat} \mid A \to B$$

1st order unification is decidable and has most general unifiers

Problem In dependent type theory, terms appear in types

Why does Hindley-Miler type inference works?

Simple types are *1st order*

$$A, B ::= \mathsf{Nat} \mid A \to B$$

1st order unification is decidable and has most general unifiers

Problem In dependent type theory, terms appear in types

Therefore, we would need *higher-order* unification, which is undecidable...

Why does Hindley-Miler type inference works?

Simple types are *1st order*

$$A, B ::= \mathsf{Nat} \mid A \to B$$

1st order unification is decidable and has most general unifiers

Problem In dependent type theory, terms appear in types

Therefore, we would need *higher-order* unification, which is undecidable...

We need a different solution

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference	$\Gamma \vdash t \Rightarrow A$	(inputs: Γ, t) (outputs: A)
Checking	$\Gamma \vdash t \Leftarrow A$	(inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference	$\Gamma \vdash t \Rightarrow A$	(inputs: Γ, t) (outputs: A)
Checking	$\Gamma \vdash t \Leftarrow A$	(inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^* \Pi(x : A).B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\ u \Rightarrow B[u/x]}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow ?}{\Gamma \vdash t\ u \Rightarrow ?}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C}{\Gamma \vdash t\ u \Rightarrow ?}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^{*} \Pi(x : A).B}{\Gamma \vdash t\ u \Rightarrow ?}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^* \Pi(x : A).B \qquad \Gamma \vdash u \Leftarrow ?}{\Gamma \vdash t\ u \Rightarrow ?}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^* \Pi(x : A).B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\ u \Rightarrow ?}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference	$\Gamma \vdash t \Rightarrow A$	(inputs: Γ, t) (outputs: A)
Checking	$\Gamma \vdash t \Leftarrow A$	(inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^* \Pi(x : A).B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\ u \Rightarrow B[u/x]}$$

Bidirectional typing

Decompose typing judgment $\Gamma \vdash t : A$ into two modes:

Inference $\Gamma \vdash t \Rightarrow A$ (inputs: Γ, t) (outputs: A)

Checking $\Gamma \vdash t \Leftarrow A$ (inputs: Γ, t, A) (outputs: none)

New judgments allow to specify flow of type information in typing rules

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^* \Pi(x : A).B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\ u \Rightarrow B[u/x]}$$

Complements unannotated syntax, *locally* explains how to recover annotations

Contribution

Bidirectional type systems have been studied and proposed for many theories

But general guidelines have remained mostly informal and not fully developed

Contribution

Bidirectional type systems have been studied and proposed for many theories

But general guidelines have remained mostly informal and not fully developed

This work *Generic* account of bidirectional typing for class of type theories

Contribution

Bidirectional type systems have been studied and proposed for many theories

But general guidelines have remained mostly informal and not fully developed

This work *Generic* account of bidirectional typing for class of type theories

Roadmap

Contribution

Bidirectional type systems have been studied and proposed for many theories

But general guidelines have remained mostly informal and not fully developed

This work *Generic* account of bidirectional typing for class of type theories

Roadmap

1. We give a general definition of type theories (or equivalently, a *logical framework*) supporting non-annotated syntaxes

Contribution

Bidirectional type systems have been studied and proposed for many theories

But general guidelines have remained mostly informal and not fully developed

This work *Generic* account of bidirectional typing for class of type theories

Roadmap

1. We give a general definition of type theories (or equivalently, a *logical framework*) supporting non-annotated syntaxes
2. For each theory, we define declarative and bidirectional type systems

Contribution

Bidirectional type systems have been studied and proposed for many theories

But general guidelines have remained mostly informal and not fully developed

This work *Generic* account of bidirectional typing for class of type theories

Roadmap

1. We give a general definition of type theories (or equivalently, a *logical framework*) supporting non-annotated syntaxes
2. For each theory, we define declarative and bidirectional type systems
3. We show, in a theory-independent fashion, their equivalence

BiTTs: A theory-independent bidirectional type-checker

Our framework not only of theoretic interest, can also have practical applications

BiTTs: A theory-independent bidirectional type-checker

Our framework not only of theoretic interest, can also have practical applications

Implemented in the theory-independent bidirectional type-checker BiTTs

```
constructor List () (A : Ty) : Ty
constructor nil (A : Ty) () : Tm(List(A))
constructor cons (A : Ty) (a : Tm(A), l : Tm(List(A))) : Tm(List(A))

destructor ind_List    (A : Ty) [l : Tm(List(A))] (P {x : Tm(List(A))} : Ty, l_nil : Tm(P{nil}),
                        l_cons {a : Tm(A), l : Tm(List(A)), pl : Tm(P{l})} : Tm(P{cons(a, l)}))
                       : Tm(P{l})

equation ind_List(nil, l. P{l}, l_nil, a l pl. l_cons{a, l, pl}) --> l_nil
equation ind_List(cons(a, l), l. P{l}, l_nil, a l pl. l_cons{a, l, pl}) -->
    l_cons{a, l, ind_List(l, l. P{l}, l_nil, a l pl. l_cons{a, l, pl})}

let 0::1::2::3::nil : Tm(List(ℕ)) := cons(0, cons(S(0), cons(S(S(0)), cons(S(S(S(0))), nil))))

let sum_of_list : Tm(Π(List(ℕ), _. ℕ)) := λ(l. ind_List(l, _. ℕ, 0, x _ acc. @(@(+, x), acc)))

assert @(sum_of_list, 0::1::2::3::nil) = S(S(S(S(S(S(0))))))
```

BiTTs: A theory-independent bidirectional type-checker

Our framework not only of theoretic interest, can also have practical applications

Implemented in the theory-independent bidirectional type-checker BiTTs

```
constructor List () (A : Ty) : Ty
constructor nil (A : Ty) () : Tm(List(A))
constructor cons (A : Ty) (a : Tm(A), l : Tm(List(A))) : Tm(List(A))

destructor ind_List     (A : Ty) [l : Tm(List(A))] (P {x : Tm(List(A))} : Ty, l_nil : Tm(P{nil}),
                         l_cons {a : Tm(A), l : Tm(List(A)), pl : Tm(P{l})} : Tm(P{cons(a, l)}))
                        : Tm(P{l})

equation ind_List(nil, l. P{l}, l_nil, a l pl. l_cons{a, l, pl}) --> l_nil
equation ind_List(cons(a, l), l. P{l}, l_nil, a l pl. l_cons{a, l, pl}) -->
    l_cons{a, l, ind_List(l, l. P{l}, l_nil, a l pl. l_cons{a, l, pl})}

let 0::1::2::3::nil : Tm(List(ℕ)) := cons(0, cons(S(0), cons(S(S(0)), cons(S(S(S(0))), nil))))


let sum_of_list : Tm(Π(List(ℕ), _. ℕ)) := λ(l. ind_List(l, _. ℕ, 0, x _ acc. @(@(+, x), acc)))

assert @(sum_of_list, 0::1::2::3::nil) = S(S(S(S(S(S(0))))))
```

Many theories supported: flavours of MLTT, HOL, etc (see the implementation)

BiTTs: A theory-independent bidirectional type-checker

Our framework not only of theoretic interest, can also have practical applications

Implemented in the theory-independent bidirectional type-checker BiTTs

```
constructor List () (A : Ty) : Ty
constructor nil (A : Ty) () : Tm(List(A))
constructor cons (A : Ty) (a : Tm(A), l : Tm(List(A))) : Tm(List(A))

destructor ind_List     (A : Ty) [l : Tm(List(A))] (P {x : Tm(List(A))} : Ty, l_nil : Tm(P{nil}),
                         l_cons {a : Tm(A), l : Tm(List(A)), pl : Tm(P{l})} : Tm(P{cons(a, l)}))
                        : Tm(P{l})

equation ind_List(nil, l. P{l}, l_nil, a l pl. l_cons{a, l, pl}) --> l_nil
equation ind_List(cons(a, l), l. P{l}, l_nil, a l pl. l_cons{a, l, pl}) -->
    l_cons{a, l, ind_List(l, l. P{l}, l_nil, a l pl. l_cons{a, l, pl})}

let 0::1::2::3::nil : Tm(List(ℕ)) := cons(0, cons(S(0), cons(S(S(0)), cons(S(S(S(0))), nil))))


let sum_of_list : Tm(Π(List(ℕ), _. ℕ)) := λ(l. ind_List(l, _. ℕ, 0, x _ acc. @(@(+, x), acc)))

assert @(sum_of_list, 0::1::2::3::nil) = S(S(S(S(S(S(0))))))
```

Many theories supported: flavours of MLTT, HOL, etc (see the implementation)

Can be used as independent proof verified, like in the Dedukti project

But support for non-annotated syntax can allow for better performances

The theories

The theories

A *theory* $\mathbb{T}$ is made of *schematic typing rules* and *rewrite rules*.

1

The theories

A *theory* $\mathbb{T}$ is made of *schematic typing rules* and *rewrite rules*.

3 schematic typing rules: *sort rules*, *constructor rules* and *destructor rules*

1

The theories

A *theory* $\mathbb{T}$ is made of *schematic typing rules* and *rewrite rules.*

3 schematic typing rules: *sort rules*, *constructor rules* and *destructor rules*

Sort rules[1] Sorts are terms T that can appear to the right in judgment $t : T$

Used to define the *judgment forms* of the theory

[1] We use the name "sort" instead of "type" to avoid a name clash with the types of the theory

The theories

A *theory* $\mathbb{T}$ is made of *schematic typing rules* and *rewrite rules*.

3 schematic typing rules: *sort rules*, *constructor rules* and *destructor rules*

Sort rules[1] Sorts are terms T that can appear to the right in judgment $t : T$
Used to define the *judgment forms* of the theory

Example: In MLTT, 2 judgment forms: "$\square$ type" and "$\square : A$" for a type A.

[1]We use the name "sort" instead of "type" to avoid a name clash with the types of the theory

The theories

A *theory* $\mathbb{T}$ is made of *schematic typing rules* and *rewrite rules*.

3 schematic typing rules: *sort rules*, *constructor rules* and *destructor rules*

Sort rules[1] Sorts are terms T that can appear to the right in judgment $t : T$
Used to define the *judgment forms* of the theory

Example: In MLTT, 2 judgment forms: "$\square$ type" and "$\square : A$" for a type A.

$$\frac{}{\mathrm{Ty}\ \mathsf{sort}} \qquad\qquad A\ \mathsf{type} \quad \rightsquigarrow \quad A : \mathrm{Ty}$$

$$\frac{\mathsf{A} : \mathrm{Ty}}{\mathrm{Tm}(\mathsf{A})\ \mathsf{sort}} \qquad\qquad t : A \quad \rightsquigarrow \quad t : \mathrm{Tm}(A)$$

[1] We use the name "sort" instead of "type" to avoid a name clash with the types of the theory

Constructor rules

Constructor rules In bidirectional typing, constructors support *type-checking*

Missing information recovered from the sort given as input

The sort of the rule should be a *pattern* T^{P} on the missing arguments[2]

[2] Here, pattern = a *Miller* pattern that is *linear* and contains no destructors

Constructor rules

Constructor rules In bidirectional typing, constructors support *type-checking*

Missing information recovered from the sort given as input

The sort of the rule should be a *pattern* T^{P} on the missing arguments[2]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\Pi(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathrm{Ty}}$$

[2]Here, pattern = a *Miller* pattern that is *linear* and contains no destructors

Constructor rules

Constructor rules In bidirectional typing, constructors support *type-checking*

Missing information recovered from the sort given as input

The sort of the rule should be a *pattern* T^{P} on the missing arguments[2]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\Pi(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathrm{Ty}}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

[2]Here, pattern = a *Miller* pattern that is *linear* and contains no destructors

Destructor rules

Destructor rules In bidirectional typing, destructors support *type-inference*

Missing arguments are recovered by inferring the *principal argument* $\texttt{t} : T^{\mathsf{P}}$

Sort of the principal argument $\texttt{t} : T^{\mathsf{P}}$ should be a pattern on the missing arguments

Destructor rules

Destructor rules In bidirectional typing, destructors support *type-inference*

Missing arguments are recovered by inferring the *principal argument* $\mathtt{t} : T^{\mathsf{P}}$

Sort of the principal argument $\mathtt{t} : T^{\mathsf{P}}$ should be a pattern on the missing arguments

$$\frac{\mathsf{A} : \mathbf{Ty} \qquad x : \mathbf{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathbf{Ty} \qquad \mathtt{t} : \mathbf{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathbf{Tm}(\mathsf{A})}{@(\mathtt{t}, \mathsf{u}) : \mathbf{Tm}(\mathsf{B}\{\mathsf{u}\})}$$

Destructor rules

Destructor rules In bidirectional typing, destructors support *type-inference*

Missing arguments are recovered by inferring the *principal argument* $\mathtt{t} : T^{\mathsf{P}}$

Sort of the principal argument $\mathtt{t} : T^{\mathsf{P}}$ should be a pattern on the missing arguments

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathtt{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathtt{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})}$$

Destructors written in orange, constructors written in blue

Rewrite rules

In type theory, terms should compute

Rewrite rules The computational rules of the theory

$$@(\lambda(x.\mathtt{t}\{x\}), \mathtt{u}) \longmapsto \mathtt{t}\{\mathtt{u}\}$$

In general, of the form $d(t^{\mathsf{P}}, \vec{x}_1.t_1^{\mathsf{P}}, \dots, \vec{x}_k.t_k^{\mathsf{P}}) \longmapsto r$, with left-hand-side linear

Rewrite rules

In type theory, terms should compute

Rewrite rules The computational rules of the theory

$$@(\lambda(x.\mathtt{t}\{x\}), \mathtt{u}) \longmapsto \mathtt{t}\{\mathtt{u}\}$$

In general, of the form $d(t^{\mathsf{P}}, \vec{x}_1.t_1^{\mathsf{P}}, \ldots, \vec{x}_k.t_k^{\mathsf{P}}) \longmapsto r$, with left-hand-side linear

Condition: no two left-hand sides unify

Therefore, rewrite systems are *orthogonal*, hence *confluent* by construction!

Full example, in the *formal* notation

Previous inference-rule notation can be desugared into a formal notation

Full example, in the *formal* notation

Previous inference-rule notation can be desugared into a formal notation

Theory $\mathbb{T}_{\lambda\Pi}$ A minimalistic type theory with only dependent functions

$\mathrm{Ty}(\cdot)$ sort

$\mathrm{Tm}(\mathsf{A} : \mathrm{Ty})$ sort

$\Pi(\cdot;\ \ \mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty}$

$\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \ \mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$

$@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$

$@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$

Full example, in the *formal* notation

Previous inference-rule notation can be desugared into a formal notation

Theory $\mathbb{T}_{\lambda\Pi}$ A minimalistic type theory with only dependent functions

$\mathrm{Ty}(\cdot)$ sort

$\mathrm{Tm}(\mathsf{A} : \mathrm{Ty})$ sort

$\Pi(\cdot;\ \ \mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty}$

$\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \ \mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$

$@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$

$@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$

In the rest of the talk, we use the inference-rule notation for readability ☺

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\rightsquigarrow$$

$$\rightsquigarrow$$

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow$$

$$\rightsquigarrow$$

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t : \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B))}$$

$$\rightsquigarrow$$

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t : \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B))}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow$$

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \quad\quad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \quad\quad \Gamma, x : \mathrm{Tm}(A) \vdash t : \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B))}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \quad\quad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \quad\quad \Gamma \vdash t : \mathrm{Tm}(\Pi(A, x.B)) \qquad \Gamma \vdash u : \mathrm{Tm}(A)}{\Gamma \vdash @(t, u) : \mathrm{Tm}(B[u/x])}$$

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t : \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B))}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \qquad \Gamma \vdash t : \mathrm{Tm}(\Pi(A, x.B)) \qquad \Gamma \vdash u : \mathrm{Tm}(A)}{\Gamma \vdash @(t, u) : \mathrm{Tm}(B[u/x])}$$

Note that typing terms bottom-up requires guessing A and B

Declarative type system

Each theory $\mathbb{T}$ defines a declarative type system, with main judgment $\Gamma \vdash t : T$

Main typing rules instantiate the schematic rules of $\mathbb{T}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(\mathsf{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t : \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B))}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow \quad \frac{\Gamma \vdash A : \mathrm{Ty} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \qquad \Gamma \vdash t : \mathrm{Tm}(\Pi(A, x.B)) \qquad \Gamma \vdash u : \mathrm{Tm}(A)}{\Gamma \vdash @(t, u) : \mathrm{Tm}(B[u/x])}$$

Note that typing terms bottom-up requires guessing A and B

Properties of the declarative system Weakening, substitution property, …

Bidirectional type system

Matching modulo for recovering arguments

In bidirectional typing, we need matching modulo to recover missing arguments

$$\frac{\Gamma \vdash t \Rightarrow U \qquad \ldots}{\Gamma \vdash @(t, u) \Rightarrow}$$

Matching modulo for recovering arguments

In bidirectional typing, we need matching modulo to recover missing arguments

$$\frac{\Gamma \vdash t \Rightarrow U \qquad \ldots}{\Gamma \vdash @(t, u) \Rightarrow}$$

If $@(t, u)$ is well-typed (in the declarative system), for some A, B we have

$$U \equiv \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))[A/\mathsf{A},\ x.B/\mathsf{B}]$$

but how to recover A and B from U?

Matching modulo for recovering arguments

In bidirectional typing, we need matching modulo to recover missing arguments

$$\frac{\Gamma \vdash t \Rightarrow U \qquad \ldots}{\Gamma \vdash @(t, u) \Rightarrow}$$

If $@(t, u)$ is well-typed (in the declarative system), for some A, B we have

$$U \equiv \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))[A/\mathsf{A},\ x.B/\mathsf{B}]$$

but how to recover A and B from U?

Solution Given T^{P} and U, we define an algorithmic[3] matching judgment

$$T^{\mathsf{P}} \prec U \rightsquigarrow \vec{x}_1.t_1/\mathsf{x}_1, \ldots, \vec{x}_k.t_k/\mathsf{x}_k$$

that tries to compute $t_1, \ldots, t_k$ s.t. $T^{\mathsf{P}}[\vec{x}_1.t_1/\mathsf{x}_1, \ldots, \vec{x}_k.t_k/\mathsf{x}_k] \equiv U$

[3] Decidable when U is normalizing

Inferable and checkable terms

Not all unannotated terms can be algorithmically typed

$$\dfrac{\dfrac{?}{\Gamma \vdash \lambda(x.t) \Rightarrow ?} \qquad \cdots}{\Gamma \vdash @(\lambda(x.t), u) \Rightarrow ?}$$

Inferable and checkable terms

Not all unannotated terms can be algorithmically typed

$$\dfrac{\dfrac{?}{\Gamma \vdash \lambda(x.t) \Rightarrow ?} \qquad \ldots}{\Gamma \vdash @(\lambda(x.t), u) \Rightarrow ?}$$

Limitation not specific to bidirectional typing, undecidable in general!

Inferable and checkable terms

Not all unannotated terms can be algorithmically typed

$$\dfrac{\dfrac{?}{\Gamma \vdash \lambda(x.t) \Rightarrow ?} \qquad \ldots}{\Gamma \vdash @(\lambda(x.t), u) \Rightarrow ?}$$

Limitation not specific to bidirectional typing, undecidable in general!

Avoided by defining bidirectional typing only for *inferrable* and *checkable* terms.

$$t^{\mathrm{i}}, u^{\mathrm{i}} ::= x \mid d(t^{\mathrm{i}}, \vec{x}_1.u_1^{\mathrm{c}}, \ldots, \vec{x}_k.u_k^{\mathrm{c}})$$
$$t^{\mathrm{c}}, u^{\mathrm{c}} ::= c(\vec{x}_1.u_1^{\mathrm{c}}, \ldots, \vec{x}_k.u_k^{\mathrm{c}}) \mid \underline{t}^{\mathrm{i}}$$

Inferable and checkable terms

Not all unannotated terms can be algorithmically typed

$$\dfrac{\dfrac{?}{\Gamma \vdash \lambda(x.t) \Rightarrow ?} \qquad \dots}{\Gamma \vdash @(\lambda(x.t), u) \Rightarrow ?}$$

Limitation not specific to bidirectional typing, undecidable in general!

Avoided by defining bidirectional typing only for *inferrable* and *checkable* terms.

$$t^{\mathrm{i}}, u^{\mathrm{i}} ::= x \mid d(t^{\mathrm{i}}, \vec{x}_1.u^{\mathrm{c}}_1, \dots, \vec{x}_k.u^{\mathrm{c}}_k)$$
$$t^{\mathrm{c}}, u^{\mathrm{c}} ::= c(\vec{x}_1.u^{\mathrm{c}}_1, \dots, \vec{x}_k.u^{\mathrm{c}}_k) \mid \underline{t}^{\mathrm{i}}$$

Principal argument of a destructor can only be variable or another destructor.

Inferable and checkable terms

Not all unannotated terms can be algorithmically typed

$$\dfrac{\dfrac{?}{\Gamma \vdash \lambda(x.t) \Rightarrow ?} \qquad \dots}{\Gamma \vdash @(\lambda(x.t), u) \Rightarrow ?}$$

Limitation not specific to bidirectional typing, undecidable in general!

Avoided by defining bidirectional typing only for *inferrable* and *checkable* terms.

$$t^{\mathrm{i}}, u^{\mathrm{i}} ::= x \mid d(t^{\mathrm{i}}, \vec{x}_1.u^{\mathrm{c}}_1, \dots, \vec{x}_k.u^{\mathrm{c}}_k)$$
$$t^{\mathrm{c}}, u^{\mathrm{c}} ::= c(\vec{x}_1.u^{\mathrm{c}}_1, \dots, \vec{x}_k.u^{\mathrm{c}}_k) \mid \underline{t}^{\mathrm{i}}$$

Principal argument of a destructor can only be variable or another destructor.

For most theories: $t^{\mathrm{c}}, u^{\mathrm{c}}, \dots$ = normal forms

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathrm{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathrm{i}} \Rightarrow T$

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$

The main typing rules instantiate the schematic rules of $\mathbb{T}$:[4]

$\rightsquigarrow$

$\rightsquigarrow$

[4]Given t^{i} or u^{c}, I write $\ulcorner t^{\mathsf{i}} \urcorner$ or $\ulcorner u^{\mathsf{c}} \urcorner$ for the underlying regular term.

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$

The main typing rules instantiate the schematic rules of $\mathbb{T}$:[4]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow$$

$$\rightsquigarrow$$

[4] Given t^{i} or u^{c}, I write $\ulcorner t^{\mathsf{i}} \urcorner$ or $\ulcorner u^{\mathsf{c}} \urcorner$ for the underlying regular term.

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$

The main typing rules instantiate the schematic rules of $\mathbb{T}$:[4]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec T \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t^{\mathsf{c}} \Leftarrow \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t^{\mathsf{c}}) \Leftarrow T}$$

$$\rightsquigarrow$$

[4] Given t^{i} or u^{c}, I write $\ulcorner t^{\mathsf{i}} \urcorner$ or $\ulcorner u^{\mathsf{c}} \urcorner$ for the underlying regular term.

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$

The main typing rules instantiate the schematic rules of $\mathbb{T}$:[4]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec T \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t^{\mathsf{c}} \Leftarrow \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t^{\mathsf{c}}) \Leftarrow T}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow$$

[4]Given t^{i} or u^{c}, I write $\ulcorner t^{\mathsf{i}} \urcorner$ or $\ulcorner u^{\mathsf{c}} \urcorner$ for the underlying regular term.

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$

The main typing rules instantiate the schematic rules of $\mathbb{T}$:[4]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec T \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t^{\mathsf{c}} \Leftarrow \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t^{\mathsf{c}}) \Leftarrow T}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow \quad \frac{\Gamma \vdash t^{\mathsf{i}} \Rightarrow T \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec T \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma \vdash u^{\mathsf{c}} \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t^{\mathsf{i}}, u^{\mathsf{c}}) \Rightarrow \mathrm{Tm}(B[\ulcorner u^{\mathsf{c}} \urcorner / x])}$$

[4] Given t^{i} or u^{c}, I write $\ulcorner t^{\mathsf{i}} \urcorner$ or $\ulcorner u^{\mathsf{c}} \urcorner$ for the underlying regular term.

Bidirectional typing rules

Each $\mathbb{T}$ defines a bidirectional system. Main judgments: $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$

The main typing rules instantiate the schematic rules of $\mathbb{T}$:[4]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))} \quad \rightsquigarrow \quad \frac{\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec T \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma, x : \mathrm{Tm}(A) \vdash t^{\mathsf{c}} \Leftarrow \mathrm{Tm}(B)}{\Gamma \vdash \lambda(x.t^{\mathsf{c}}) \Leftarrow T}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})} \quad \rightsquigarrow \quad \frac{\Gamma \vdash t^{\mathsf{i}} \Rightarrow T \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec T \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma \vdash u^{\mathsf{c}} \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t^{\mathsf{i}}, u^{\mathsf{c}}) \Rightarrow \mathrm{Tm}(B[\ulcorner u^{\mathsf{c}} \urcorner / x])}$$

Note that no more need to guess A and B!

[4] Given t^{i} or u^{c}, I write $\ulcorner t^{\mathsf{i}} \urcorner$ or $\ulcorner u^{\mathsf{c}} \urcorner$ for the underlying regular term.

Equivalence with declarative typing

(Suppose underlying theory $\mathbb{T}$ is *valid*)

Equivalence with declarative typing

(Suppose underlying theory $\mathbb{T}$ is *valid*)

Soundness If $\Gamma \vdash$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$

If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$

Equivalence with declarative typing

(Suppose underlying theory $\mathbb{T}$ is *valid*)

Soundness If $\Gamma \vdash$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$
If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$

Completeness If $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{i}} \Rightarrow U$ with $T \equiv U$
If $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$

Equivalence with declarative typing

(Suppose underlying theory $\mathbb{T}$ is *valid*)

Soundness If $\Gamma \vdash$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$
If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$

Completeness If $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{i}} \Rightarrow U$ with $T \equiv U$
If $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$

Decidability If $\mathbb{T}$ normalizing, then inference is decidable for inferable terms, and checking is decidable for checkable terms

Equivalence with declarative typing

(Suppose underlying theory $\mathbb{T}$ is *valid*)

Soundness If $\Gamma \vdash$ and $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$

If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ then $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$

Completeness If $\Gamma \vdash \ulcorner t^{\mathsf{i}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{i}} \Rightarrow U$ with $T \equiv U$

If $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$

Decidability If $\mathbb{T}$ normalizing, then inference is decidable for inferable terms, and checking is decidable for checkable terms

Not shown for one particular theory, but for *all* instances of our framework

More examples

Dependent sums

Extends $\mathbb{T}_{\lambda\Pi}$ with

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\Sigma(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathrm{Ty}} \qquad \frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ \mathsf{t} : \mathrm{Tm}(\mathsf{A}) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{B}\{\mathsf{t}\})\end{array}}{\mathrm{pair}(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\Sigma(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

$$\frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ \mathsf{t} : \mathrm{Tm}(\Sigma(\mathsf{A}, x.\mathsf{B}\{x\}))\end{array}}{\mathrm{proj}_1(\mathsf{t}) : \mathrm{Tm}(\mathsf{A})} \qquad \frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ \mathsf{t} : \mathrm{Tm}(\Sigma(\mathsf{A}, x.\mathsf{B}\{x\}))\end{array}}{\mathrm{proj}_2(\mathsf{t}) : \mathrm{Tm}(\mathsf{B}\{\mathrm{proj}_1(\mathsf{t})\})}$$

$$\mathrm{proj}_1(\mathrm{pair}(\mathsf{t}, \mathsf{u})) \longmapsto \mathsf{t} \qquad \mathrm{proj}_2(\mathrm{pair}(\mathsf{t}, \mathsf{u})) \longmapsto \mathsf{u}$$

Lists

Extends $\mathbb{T}_{\lambda\Pi}$ with

$$\frac{\mathtt{A} : \mathrm{Ty}}{\mathrm{List}(\mathtt{A}) : \mathrm{Ty}} \qquad \frac{\mathtt{A} : \mathrm{Ty}}{\mathrm{nil} : \mathrm{Tm}(\mathrm{List}(\mathtt{A}))} \qquad \frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{x} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{l} : \mathrm{Tm}(\mathrm{List}(\mathtt{A}))}{\mathrm{cons}(\mathtt{x}, \mathtt{l}) : \mathrm{Tm}(\mathrm{List}(\mathtt{A}))}$$

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{l} : \mathrm{Tm}(\mathrm{List}(\mathtt{A})) \qquad x : \mathrm{Tm}(\mathrm{List}(\mathtt{A})) \vdash \mathtt{P} : \mathrm{Ty} \qquad \mathtt{pnil} : \mathrm{Tm}(\mathtt{P}\{\mathrm{nil}\}) \qquad x : \mathrm{Tm}(\mathtt{A}), y : \mathrm{Tm}(\mathrm{List}(\mathtt{A})), z : \mathrm{Tm}(\mathtt{P}\{y\}) \vdash \mathtt{pcons} : \mathrm{Tm}(\mathtt{P}\{\mathrm{cons}(x, y)\})}{\mathrm{ListRec}(\mathtt{l}, x.\mathtt{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\}) : \mathrm{Tm}(\mathtt{P}\{\mathtt{l}\})}$$

$$\mathrm{ListRec}(\mathrm{nil}, x.\mathtt{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\}) \longmapsto \mathtt{pnil}$$

$$\mathrm{ListRec}(\mathrm{cons}(\mathtt{x}, \mathtt{l}), x.\mathtt{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\}) \longmapsto$$

$$\mathtt{pcons}\{\mathtt{x}, \mathtt{l}, \mathrm{ListRec}(\mathtt{l}; x.\mathtt{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\})\}$$

W types

W types capture a class of inductive types

Extends $\mathbb{T}_{\lambda\Pi}$ with

$$\frac{\mathsf{A} : \mathbf{Ty} \qquad x : \mathbf{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathbf{Ty}}{\mathbf{W}(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathbf{Ty}} \qquad \frac{\begin{array}{c}\mathsf{A} : \mathbf{Ty} \qquad x : \mathbf{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathbf{Ty} \\ \mathsf{a} : \mathbf{Tm}(\mathsf{A}) \qquad \mathsf{f} : \mathbf{Tm}(\Pi(\mathsf{B}\{\mathsf{a}\}, \_.\mathbf{W}(\mathsf{A}, x.\mathsf{B}\{x\})))\end{array}}{\mathbf{sup}(\mathsf{a}, \mathsf{f}) : \mathbf{Tm}(\mathbf{W}(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

$$\frac{\begin{array}{c}\mathsf{A} : \mathbf{Ty} \qquad x : \mathbf{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathbf{Ty} \qquad \mathsf{t} : \mathbf{Tm}(\mathbf{W}(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad x : \mathbf{Tm}(\mathbf{W}(\mathsf{A}, x.\mathsf{B}\{x\})) \vdash \mathsf{P} : \mathbf{Ty} \\ x : \mathbf{Tm}(\mathsf{A}), y : \mathbf{Tm}(\Pi(\mathsf{B}\{x\}, x'.\mathbf{W}(\mathsf{A}, x.\mathsf{B}\{x\}))), z : \mathbf{Tm}(\Pi(\mathsf{B}\{x\}, x'.\mathsf{P}\{@(y, x')\})) \vdash \mathsf{p} : \mathbf{Tm}(\mathsf{P}\{\mathbf{sup}(x, y)\})\end{array}}{\mathbf{WRec}(\mathsf{t}, x.\mathsf{P}\{x\}, xyz.\mathsf{p}\{x, y, z\}) : \mathbf{Tm}(\mathsf{P}\{\mathsf{t}\})}$$

$$\mathbf{WRec}(\mathbf{sup}(\mathsf{a}, \mathsf{f}), x.\mathsf{P}\{x\}, xyz.\mathsf{p}\{x, y, z\}) \longmapsto \mathsf{p}\{\mathsf{a}, \mathsf{f}, \lambda(x.\mathbf{WRec}(@(\mathsf{f}, x), x.\mathsf{P}\{x\}, xyz.\mathsf{p}\{x, y, z\}))\}$$

Higher-Order Logic

Extends $\mathbb{T}_{\lambda\Pi}$ with

$$\frac{}{\mathrm{Prop} : \mathrm{Ty}} \qquad \frac{\mathsf{t} : \mathrm{Tm}(\mathrm{Prop})}{\mathrm{Prf}(\mathsf{t})\ \mathsf{sort}}$$

We can then add, for instance, universal quantification:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop})}{\forall(\mathsf{A}, x.\mathsf{P}\{x\}) : \mathrm{Tm}(\mathrm{Prop})} \qquad \frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{p} : \mathrm{Prf}(\mathsf{P})}{\forall_{\mathrm{i}}(x.\mathsf{p}\{x\}) : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\}))}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \qquad \mathsf{r} : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\})) \qquad \mathsf{t} : \mathrm{Tm}(\mathsf{A})}{\forall_{\mathrm{e}}(\mathsf{r}, \mathsf{t}) : \mathrm{Prf}(\mathsf{P}\{\mathsf{t}\})} \qquad \forall_{\mathrm{e}}(\forall_{\mathrm{i}}(x.\mathsf{p}\{x\}), \mathsf{t}) \longmapsto \mathsf{p}\{\mathsf{t}\}$$

Other examples

In the implementation at

`https://github.com/thiagofelicissimo/BiTTs`

you can also find:

- Indexed types: Equality, Vectors, etc
- Tarski-, Russell- and Coquand-style universes, with/without cumulativity and universe polymorphism
- Flavous of Observational Type Theory
- A variant of Exceptional Type Theory (type theory with exceptions)

Conclusion

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Bidirectional system implemented in the theory-independent type-checker BiTTs

Can be used as an independent proof verifier for a large class of theories

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Bidirectional system implemented in the theory-independent type-checker BiTTs

Can be used as an independent proof verifier for a large class of theories

Some of the additional features found in implementation/my PhD thesis:[5]

[5] Here I have presented the ESOP'24 version, which is simpler to understand

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Bidirectional system implemented in the theory-independent type-checker BiTTs
Can be used as an independent proof verifier for a large class of theories

Some of the additional features found in implementation/my PhD thesis:[5]

1. Ascriptions in the bidirectional system, allows for writing redexes:

$$@(\lambda(x.t) :: T, u)$$

[5] Here I have presented the ESOP'24 version, which is simpler to understand

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Bidirectional system implemented in the theory-independent type-checker BiTTs
Can be used as an independent proof verifier for a large class of theories

Some of the additional features found in implementation/my PhD thesis:[5]

1. Ascriptions in the bidirectional system, allows for writing redexes:

$$@(\lambda(x.t) :: T, u)$$

2. Equational premises, for defining indexed inductive types:

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{t} : \mathrm{Tm}(\mathtt{A})}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathtt{A}, \mathtt{t}, \mathtt{t}))}$$

[5] Here I have presented the ESOP'24 version, which is simpler to understand

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Bidirectional system implemented in the theory-independent type-checker BiTTs
Can be used as an independent proof verifier for a large class of theories

Some of the additional features found in implementation/my PhD thesis:[5]

1. Ascriptions in the bidirectional system, allows for writing redexes:

$$@(\lambda(x.t) :: T, u)$$

2. Equational premises, for defining indexed inductive types:

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{t} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{u} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{t} \equiv \mathtt{u}}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathtt{A}, \mathtt{t}, \mathtt{u}))}$$

[5]Here I have presented the ESOP'24 version, which is simpler to understand

Conclusion

We have given a generic account of bidirectional typing for a class of type theories

Bidirectional system implemented in the theory-independent type-checker BiTTs
Can be used as an independent proof verifier for a large class of theories

Some of the additional features found in implementation/my PhD thesis:[5]

1. Ascriptions in the bidirectional system, allows for writing redexes:

$$@(\lambda(x.t) :: T, u)$$

2. Equational premises, for defining indexed inductive types:

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{t} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{u} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{t} \equiv \mathtt{u}}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathtt{A}, \mathtt{t}, \mathtt{u}))}$$

Thank you for your attention!

[5] Here I have presented the ESOP'24 version, which is simpler to understand