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

DEDUC⊢
⊢EAM

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

Dependent type theory, in a nutshell

In dependent type theory:

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have types

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

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*

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

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

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

- Functions can be *dependent*

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

- Types are equal modulo computation

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

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: Coq, Lean, 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: Coq, Lean, Agda,...

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

(* pre-condition: list not empty *)

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

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

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

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: Coq, Lean, Agda,...

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

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

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

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

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

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

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

Non-annotated syntax restores usability by eliding parameter annotations

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

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)

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

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)

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

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*

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

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*

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 A *sort*[1] is a term T that can appear in the right of typing judgment $t : T$

Used to represent the judgment forms of the theory (as in LFs and GATs)

[1] I avoid calling them "types" to prevent a name clash with the types of the object theories

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 A *sort*[1] is a term T that can appear in the right of typing judgment $t : T$

Used to represent the judgment forms of the theory (as in LFs and GATs)

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

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

[1] I avoid calling them "types" to prevent a name clash with the types of the object theories

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 A *sort*[1] is a term T that can appear in the right of typing judgment $t : T$

Used to represent the judgment forms of the theory (as in LFs and GATs)

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

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

Formally, of the form $c(\Theta)$ sort, with Θ a *metavariable context* representing premises

Example in formal notation: $\mathrm{Ty}(\cdot)$ sort and $\mathrm{Tm}(\mathsf{A} : \mathrm{Ty})$ sort

[1] I avoid calling them "types" to prevent a name clash with the types of the object theories

The theories

Constructor rules In bidirectional typing, constructors support *type-checking*, so missing annotations recovered from the sort given as input

The theories

Constructor rules In bidirectional typing, constructors support *type-checking*, so missing annotations recovered from the sort given as input

Two groups of premises: Θ_1 erased and Θ_2 kept in the syntax

Sort of the rule should be a *pattern* U^{P} containing the metavariables of Θ_1

The theories

Constructor rules In bidirectional typing, constructors support *type-checking*, so missing annotations recovered from the sort given as input

Two groups of premises: Θ_1 erased and Θ_2 kept in the syntax

Sort of the rule should be a *pattern* U^{P} containing the metavariables of Θ_1

$$\frac{\mathsf{A} : \mathbf{Ty} \qquad x : \mathbf{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathbf{Ty}}{\Pi(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathbf{Ty}} \qquad \frac{\begin{array}{c}\mathsf{A} : \mathbf{Ty} \qquad x : \mathbf{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathbf{Ty} \\ x : \mathbf{Tm}(\mathsf{A}) \vdash \mathtt{t} : \mathbf{Tm}(\mathsf{B}\{x\})\end{array}}{\lambda(x.\mathtt{t}\{x\}) : \mathbf{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

The theories

Constructor rules In bidirectional typing, constructors support *type-checking*, so missing annotations recovered from the sort given as input

Two groups of premises: Θ_1 erased and Θ_2 kept in the syntax

Sort of the rule should be a *pattern* U^{P} containing the metavariables of Θ_1

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

Formally, constructor rules of the form $c(\Theta_1; \Theta_2) : U^{\mathsf{P}}$, with U^{P} pattern on Θ_1

Example in formal notation: $\Pi(\cdot;\ \mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty}$ and $\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathtt{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$

The theories

Destructor rules In bidirectional typing, destructors support *type-inference*, so missing arguments are recovered by inferring a *principal argument*

The theories

Destructor rules In bidirectional typing, destructors support *type-inference*, so missing arguments are recovered by inferring a *principal argument*

Two groups of premises: Θ_1 erased and Θ_2 kept in the syntax

And a principal argument $\mathsf{x} : T^{\mathsf{P}}$, whose sort T^{P} is a pattern on Θ_1

The theories

Destructor rules In bidirectional typing, destructors support *type-inference*, so missing arguments are recovered by inferring a *principal argument*

Two groups of premises: Θ_1 erased and Θ_2 kept in the syntax

And a principal argument $\mathsf{x} : T^{\mathsf{P}}$, whose sort T^{P} is a pattern on Θ_1

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

The theories

Destructor rules In bidirectional typing, destructors support *type-inference*, so missing arguments are recovered by inferring a *principal argument*

Two groups of premises: Θ_1 erased and Θ_2 kept in the syntax

And a principal argument $\mathsf{x} : T^{\mathsf{P}}$, whose sort T^{P} is a pattern on Θ_1

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

Formally, of the form $d(\Theta_1;\ \mathsf{x} : T^{\mathsf{P}};\ \Theta_2) : U$, with T^{P} a pattern on Θ_1

Example in formal notation:

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

The theories

Rewrite rules Specify the definitional equality (aka conversion) $\equiv$ of the theory

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

In general, of the form $d(\mathbf{t}^{\mathrm{P}}) \longmapsto r$

The theories

Rewrite rules Specify the definitional equality (aka conversion) $\equiv$ of the theory

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

In general, of the form $d(\mathbf{t}^{\mathrm{P}}) \longmapsto r$

Condition: no two left-hand sides unify

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

The theories

Rewrite rules Specify the definitional equality (aka conversion) $\equiv$ of the theory

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

In general, of the form $d(\mathbf{t}^{\mathrm{P}}) \longmapsto r$

Condition: no two left-hand sides unify

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

Full example Theory $\mathbb{T}_{\lambda\Pi}$

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

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

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

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

Declarative type system

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

Declarative type system

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

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

$$\frac{d(\Xi_1; \mathsf{x} : T; \Xi_2) : U \in \mathbb{T} \qquad \Theta; \Gamma \vdash \mathbf{t}, t, \mathbf{u} : \Xi_1.(\mathsf{x} : T).\Xi_2}{\Theta; \Gamma \vdash d(\mathbf{t}, t, \mathbf{u}) : U[\mathbf{t}, t, \mathbf{u}]}\ \textsc{Dest}$$

Declarative type system

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

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

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

(for $@(\mathtt{A} : \mathrm{Ty},\ \mathtt{B}\{x : \mathrm{Tm}(\mathtt{A})\} : \mathrm{Ty};\ \mathtt{t} : \mathrm{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}\{x\}));\ \mathtt{u} : \mathrm{Tm}(\mathtt{A})) : \mathrm{Tm}(\mathtt{B}\{\mathtt{u}\}) \in \mathbb{T}_{\lambda\Pi}$)

Declarative type system

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

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

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

(for $@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}) \in \mathbb{T}_{\lambda\Pi}$)

Note that reading bottom-up, requires guessing A and B

Declarative type system

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

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

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

(for $@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}) \in \mathbb{T}_{\lambda\Pi}$)

Note that reading bottom-up, requires guessing A and B

Properties of the declarative system Weakening, substitution property, …

Bidirectional typing system

Matching modulo rewriting

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

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

2

Matching modulo rewriting

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

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

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?

2

Matching modulo rewriting

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 We define an algorithmic[2] matching judgment $T^{\mathsf{P}} < U \rightsquigarrow \mathbf{v}$

We have $T^{\mathsf{P}}[\mathbf{v}] \equiv U$ iff $T^{\mathsf{P}} < U \rightsquigarrow \mathbf{v}'$ for some $\mathbf{v}' \equiv \mathbf{v}$

[2]Decidable when U is normalizing

Bidirectional syntax

Not all unannotated terms can be algorithmically typed

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

Bidirectional syntax

Not all unannotated terms can be algorithmically typed

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

Avoided by defining bidirectional system only for *inferrable* and *checkable* terms

$$\begin{aligned}
\boxed{\mathsf{Tm}^{\mathsf{i}}} \ni \quad & t^{\mathsf{i}}, u^{\mathsf{i}} ::= x \mid d(t^{\mathsf{i}}, \mathbf{t}^{\mathsf{c}}) \\
\boxed{\mathsf{Tm}^{\mathsf{c}}} \ni \quad & t^{\mathsf{c}}, u^{\mathsf{c}} ::= c(\mathbf{t}^{\mathsf{c}}) \mid \underline{t}^{\mathsf{i}} \\
\boxed{\mathsf{MSub}^{\mathsf{c}}} \ni \quad & \mathbf{t}^{\mathsf{c}}, \mathbf{u}^{\mathsf{c}} ::= \epsilon \mid \mathbf{t}^{\mathsf{c}}, \vec{x}.t^{\mathsf{c}}
\end{aligned}$$

Bidirectional syntax

Not all unannotated terms can be algorithmically typed

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

Avoided by defining bidirectional system only for *inferrable* and *checkable* terms

$$\boxed{\mathsf{Tm}^{\mathsf{i}}} \ni \qquad t^{\mathsf{i}}, u^{\mathsf{i}} ::= x \mid d(t^{\mathsf{i}}, \mathbf{t}^{\mathsf{c}})$$

$$\boxed{\mathsf{Tm}^{\mathsf{c}}} \ni \qquad t^{\mathsf{c}}, u^{\mathsf{c}} ::= c(\mathbf{t}^{\mathsf{c}}) \mid \underline{t}^{\mathsf{i}}$$

$$\boxed{\mathsf{MSub}^{\mathsf{c}}} \ni \qquad \mathbf{t}^{\mathsf{c}}, \mathbf{u}^{\mathsf{c}} ::= \epsilon \mid \mathbf{t}^{\mathsf{c}}, \vec{x}.t^{\mathsf{c}}$$

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

For most theories, $t^{\mathsf{c}}, u^{\mathsf{c}}, \dots$ are the normal forms

Bidirectional type system

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

Bidirectional type system

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 $\mathbb{T}$:[3]

$$d(\Xi_1; \mathsf{t} : T; \Xi_2) : U \in \mathbb{T} \quad \frac{\Gamma \vdash t^{\mathsf{i}} \Rightarrow T' \qquad T \prec T' \rightsquigarrow \mathbf{v} \qquad \Gamma \mid (\mathbf{v}, \ulcorner t^{\mathsf{i}} \urcorner) : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{u}^{\mathsf{c}} \Leftarrow \Xi_2}{\Gamma \vdash d(t^{\mathsf{i}}, \mathbf{u}^{\mathsf{c}}) \Rightarrow U[\mathbf{v}, \ulcorner t^{\mathsf{i}} \urcorner, \ulcorner \mathbf{u}^{\mathsf{c}} \urcorner]}\ \textsc{Dest}$$

[3]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 type system

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 $\mathbb{T}$:[3]

$$\frac{\Gamma \vdash t^{\mathsf{i}} \Rightarrow T' \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) < T' \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma \vdash u^{\mathsf{c}} \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t^{\mathsf{i}}, u^{\mathsf{c}}) \Rightarrow \mathrm{Tm}(B[\ulcorner u^{\mathsf{c}} \urcorner / x])}$$

(for $@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}) \in \mathbb{T}_{\lambda\Pi}$)

[3] 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 type system

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 $\mathbb{T}$:[3]

$$\frac{\Gamma \vdash t^{\mathsf{i}} \Rightarrow T' \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) < T' \rightsquigarrow A/\mathsf{A},\ x.B/\mathsf{B} \qquad \Gamma \vdash u^{\mathsf{c}} \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t^{\mathsf{i}}, u^{\mathsf{c}}) \Rightarrow \mathrm{Tm}(B[\ulcorner u^{\mathsf{c}} \urcorner / x])}$$

(for $@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}) \in \mathbb{T}_{\lambda\Pi}$)

Reading bottom-up, no more need to guess A and B!

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

Correctness with respect to declarative typing

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

Correctness with respect to declarative typing

(Supposing the 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 $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$

Correctness with respect to declarative typing

(Supposing the 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 $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T'$ with $T' \equiv T$
If $\Gamma \vdash \ulcorner t^{\mathsf{c}} \urcorner : T$ then $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$

Correctness with respect to declarative typing

(Supposing the 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 $\Gamma \vdash t^{\mathsf{i}} \Rightarrow T'$ with $T' \equiv T$

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

More examples

Dependent sums

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

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

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

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

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

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

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

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

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

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

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, 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 a prototype, available at

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

Conclusion

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

Bidirectional system implemented in a prototype, available at

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

Some of the additional features found in implementation/my PhD thesis:[4]

[4] 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 a prototype, available at

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

Some of the additional features found in implementation/my PhD thesis:[4]

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

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

[4] 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 a prototype, available at

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

Some of the additional features found in implementation/my PhD thesis:[4]

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

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

2. Equational premises, for defining indexed inductive types:

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

[4] 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 a prototype, available at

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

Some of the additional features found in implementation/my PhD thesis:[4]

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

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

2. Equational premises, for defining indexed inductive types:

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

[4] 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 a prototype, available at

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

Some of the additional features found in implementation/my PhD thesis:[4]

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!

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