Generic Bidirectional Typing for Dependent Type Theories

Thiago Felicissimo

LMF Non-Permanent Members Seminar

November 30, 2023

Dependent type theory, in a nutshell

In dependent type theory:

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have types

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

  where Γ is a *context* of variables $x_1 : A_1, \ldots, x_k : A_k$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

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

  where Γ is a *context* of variables $x_1 : A_1, \ldots, x_k : A_k$

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

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

where Γ is a *context* of variables $x_1 : A_1, \ldots, x_k : A_k$

- Functions can be *dependent*

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

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

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

  where Γ is a *context* of variables $x_1 : A_1, \ldots, x_k : A_k$

- Functions can be *dependent*

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

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

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

  where Γ is a *context* of variables $x_1 : A_1, \ldots, x_k : A_k$

- Functions can be *dependent*

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

Dependent type theory, in a nutshell

In dependent type theory:

- Terms have *dependent* types

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

where Γ is a *context* of variables $x_1 : A_1, \ldots, x_k : A_k$

- Functions can be *dependent*

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

- Types are equal modulo computation

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

Why dependent type theory?

Curry-Howard correspondence Deep link between type theory and logic

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

Why dependent type theory?

Curry-Howard correspondence Deep link between type theory and logic

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

Foundation of popular proof assistants: Coq, Lean, Agda,...

Why dependent type theory?

Curry-Howard correspondence Deep link between type theory and logic

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

Foundation of 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?

Curry-Howard correspondence Deep link between type theory and logic

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

Foundation of popular proof assistants: Coq, Lean, Agda,...

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

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

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

Type annotations in dependent type theory

In its most primitive form, syntax is extremely annotated and verbose

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

~~One application~~ One for each domain A and codomain B (think of semantics).

Type annotations in dependent type theory

In its most primitive form, syntax is extremely annotated and verbose

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

~~One application~~ One for each domain A and codomain B (think of semantics).

Same for $x ::_A l$ and $\langle t, u \rangle_{x:A.B}$ and …

Type annotations in dependent type theory

In its most primitive form, syntax is extremely annotated and verbose

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

~~One application~~ One for each domain A and codomain B (think of semantics).

Same for $x ::_A l$ and $\langle t, u \rangle_{x:A.B}$ and … **Unusable in practice**

Type annotations in dependent type theory

In its most primitive form, syntax is extremely annotated and verbose

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

~~One application~~ One for each domain A and codomain B (think of semantics).

Same for $x ::_A l$ and $\langle t, u \rangle_{x:A.B}$ and ... **Unusable in practice**

Most presentation restore usability by omitting type annotations from syntax

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

Type annotations in dependent type theory

In its most primitive form, syntax is extremely annotated and verbose

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

~~One application~~ One for each domain A and codomain B (think of semantics).

Same for $x ::_A l$ and $\langle t, u \rangle_{x:A.B}$ and … **Unusable in practice**

Most presentation restore usability by omitting type annotations from syntax

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

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 type theory 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 type theory 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 type theory 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 type theory 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 type theory 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 type theory 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 $\alpha_0 = (\alpha_5 \to \alpha_3) \to \alpha_5 \to \alpha_3$

Why does Hindley-Miler type inference works?

In simple type theory, types are *1st order*

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

Why does Hindley-Miler type inference works?

In simple type theory, 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?

In simple type theory, 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?

In simple type theory, 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?

In simple type theory, 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

However, general guidelines have remained informal, no unified framework

Contribution

Bidirectional type systems have been studied and proposed for many theories

However, general guidelines have remained informal, no unified framework

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

Contribution

Bidirectional type systems have been studied and proposed for many theories

However, general guidelines have remained informal, no unified framework

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

However, general guidelines have remained informal, no unified framework

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

However, general guidelines have remained informal, no unified framework

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

However, general guidelines have remained informal, no unified framework

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

The theories

One syntax for all!

One syntax for all!

$$
\begin{aligned}
t, u, T, U ::= & \mid x && \text{(variables)} \\
& \mid \mathsf{x}\{u_1, \ldots, u_k\} && \text{(metavariables)} \\
& \mid c(\vec{x}_1.u_1, \ldots, \vec{x}_k.u_k) && \text{(constructor application)} \\
& \mid d(t;\ \vec{x}_1.u_1, \ldots, \vec{x}_k.u_k) && \text{(destructor application)}
\end{aligned}
$$

Symbols separated between *constructors* c (intros) and *destructors* d (elims)

In $d(t; \ldots)$, we call t the *principal argument*.

One syntax for all!

$$
\begin{aligned}
t, u, T, U \ ::= &\mid x && \text{(variables)} \\
&\mid \mathtt{x}\{u_1, \dots, u_k\} && \text{(metavariables)} \\
&\mid c(\vec{x}_1.u_1, \dots, \vec{x}_k.u_k) && \text{(constructor application)} \\
&\mid d(t;\ \vec{x}_1.u_1, \dots, \vec{x}_k.u_k) && \text{(destructor application)}
\end{aligned}
$$

Symbols separated between *constructors* c (intros) and *destructors* d (elims)

In $d(t; \dots)$, we call t the *principal argument*.

Example

$$
\begin{aligned}
\Sigma_{\lambda\Pi} = \quad & \Pi(\mathsf{A}, \mathsf{B}\{x\}),\ \lambda(\mathtt{t}\{x\}), \dots && \text{(constructors)} \\
& @(\mathtt{t}; \mathtt{u}) && \text{(destructors)}
\end{aligned}
$$

$$
t, u, A, B ::= x \mid \mathtt{x}\{\vec{t}\} \mid @(t; u) \mid \lambda(x.t) \mid \Pi(A, x.B) \mid \dots
$$

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 Sorts are terms that can type other terms[1]

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 Sorts are terms that can type other terms[1]
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 Sorts are terms that can type other terms[1]

Used to define the *judgment forms* of the theory.

Example: In MLTT, 2 judgment forms: "□ type" and "□ : A" for a type A.

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

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

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

Constructor rules

Constructors are bidirectionally typed in mode check (its sort is an input)

The sort of the rule should be a *pattern* T^{P}, allowing to recover omitted annotations

Constructor rules

Constructors are bidirectionally typed in mode check (its sort is an input)

The sort of the rule should be a *pattern* T^{P}, allowing to recover omitted annotations

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

Constructor rules

Constructors are bidirectionally typed in mode check (its sort is an input)

The sort of the rule should be a *pattern* T^{P}, allowing to recover omitted annotations

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

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

Destructor rules

Destructors are bidirectionally typed in mode infer (the sort is an output)

The sort of the *principal argument* $\mathtt{t} : T^{\mathsf{P}}$ should be a pattern

Destructor rules

Destructors are bidirectionally typed in mode infer (the sort is an output)

The sort of the *principal argument* $\mathtt{t} : T^{\mathsf{P}}$ should be a pattern

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

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.

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 ☺

Type systems

Each theory $\mathbb{T}$ defines two type systems.

Type systems

Each theory $\mathbb{T}$ defines two type systems.

Declarative system The "usual" type system, presented in papers

$$\Gamma \vdash t : T$$

More abstract and better for theoretic study

Type systems

Each theory $\mathbb{T}$ defines two type systems.

Declarative system The "usual" type system, presented in papers

$$\Gamma \vdash t : T$$

More abstract and better for theoretic study

However, worse for implementation: we need to "guess" omitted annotations

Type systems

Each theory $\mathbb{T}$ defines two type systems.

Declarative system The "usual" type system, presented in papers

$$\Gamma \vdash t : T$$

More abstract and better for theoretic study

However, worse for implementation: we need to "guess" omitted annotations

Bidirectional system Implementation-friendly, no need for any guessing

$$\Gamma \vdash t \Leftarrow T \qquad \text{and} \qquad \Gamma \vdash t \Rightarrow T$$

Type systems

Each theory $\mathbb{T}$ defines two type systems.

Declarative system The "usual" type system, presented in papers

$$\Gamma \vdash t : T$$

More abstract and better for theoretic study

However, worse for implementation: we need to "guess" omitted annotations

Bidirectional system Implementation-friendly, no need for any guessing

$$\Gamma \vdash t \Leftarrow T \qquad \text{and} \qquad \Gamma \vdash t \Rightarrow T$$

We will see each system and show that they are equivalent

Declarative type system

Declarative typing rules

Declarative typing rules

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

$\rightsquigarrow$

$\rightsquigarrow$

Declarative typing rules

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

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

$\rightsquigarrow$

Declarative typing rules

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

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

$$\rightsquigarrow$$

Declarative typing rules

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

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

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

Declarative typing rules

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

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

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

Bidirectional type system

Matching modulo rewriting

Suppose we want to infer the sort of $@(t; u)$

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

Matching modulo rewriting

Suppose we want to infer the sort of $@(t;u)$

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

We know

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

but they are no syntactically equal... How to recover annotations A and B from U?

Matching modulo rewriting

Suppose we want to infer the sort of $@(t; u)$

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

We know

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

but they are no syntactically equal... How to recover annotations A and B from U?

Given t^{P} and u, we define a 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 a metavariable substitution s.t. $t^{\mathsf{P}}[\vec{x}_1.t_1/\mathsf{x}_1, \ldots, \vec{x}_k.t_k/\mathsf{x}_k] \equiv u$.

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

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

Limitation not specific to bidirectional typing, undecidable in general!

Inferable and checkable terms

Not all unannotated terms can be algorithmically typed

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

Limitation not specific to bidirectional typing, undecidable in general!

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

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

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

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

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

Limitation not specific to bidirectional typing, undecidable in general!

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

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

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

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

Bidirectional typing rules

Bidirectional typing rules

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

$\rightsquigarrow$

$\rightsquigarrow$

Bidirectional typing rules

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

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

$$\rightsquigarrow$$

Bidirectional typing rules

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

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

$$\rightsquigarrow$$

Bidirectional typing rules

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

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

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

Bidirectional typing rules

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

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

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

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 $\Gamma \vdash t^{\mathrm{i}} \Rightarrow T$ then $\Gamma \vdash t : T$.

If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathrm{c}} \Leftarrow T$ then $\Gamma \vdash t : T$.

Equivalence with declarative typing

Suppose underlying theory $\mathbb{T}$ is valid.

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

Completeness For t^{i} inferable, if $\Gamma \vdash t : T$ then $\Gamma \vdash t^{\mathsf{i}} \Rightarrow U$ with $T \equiv U$.
For t^{c} checkable, if $\Gamma \vdash t : T$ 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 $\Gamma \vdash t : T$.
If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ then $\Gamma \vdash t : T$.

Completeness For t^{i} inferable, if $\Gamma \vdash t : T$ then $\Gamma \vdash t^{\mathsf{i}} \Rightarrow U$ with $T \equiv U$.
For t^{c} checkable, if $\Gamma \vdash t : T$ then $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$.

Decidability If $\mathbb{T}$ weak 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 $\Gamma \vdash t : T$.
If $\Gamma \vdash T$ sort and $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$ then $\Gamma \vdash t : T$.

Completeness For t^{i} inferable, if $\Gamma \vdash t : T$ then $\Gamma \vdash t^{\mathsf{i}} \Rightarrow U$ with $T \equiv U$.
For t^{c} checkable, if $\Gamma \vdash t : T$ then $\Gamma \vdash t^{\mathsf{c}} \Leftarrow T$.

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

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

More examples

Dependent sums

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

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

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

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

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

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

Lists

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

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

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

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

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

W types

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

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

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

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

Universes

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

$$\frac{}{\vdash \mathrm{U}(\cdot) : \mathrm{Ty}} \qquad \frac{\vdash \mathsf{a} : \mathrm{Tm}(\mathrm{U})}{\vdash \mathrm{El}(\mathsf{a};\cdot) : \mathrm{Ty}}$$

Tarski-style Adds codes for all types

$$\frac{}{\vdash \mathrm{u}(\cdot) : \mathrm{Tm}(\mathrm{U})} \qquad \mathrm{El}(\mathrm{u};\varepsilon) \longmapsto \mathrm{U}$$

$$\frac{\vdash \mathsf{a} : \mathrm{Tm}(\mathrm{U}) \qquad x : \mathrm{Tm}(\mathrm{El}(\mathsf{a})) \vdash \mathsf{b} : \mathrm{Tm}(\mathrm{U})}{\vdash \pi(\mathsf{a},\mathsf{b}) : \mathrm{Tm}(\mathrm{U})}$$

$$\mathrm{El}(\pi(\mathsf{a}, x.\mathsf{b}\{x\});\varepsilon) \longmapsto \Pi(\mathrm{El}(\mathsf{a};\varepsilon), x.\mathrm{El}(\mathsf{b}\{x\};\varepsilon))$$

(Weak) Coquand-style

Adds a code constructor c

$$\frac{\vdash \mathsf{A} : \mathrm{Ty}}{\vdash \mathrm{c}(\mathsf{A}) : \mathrm{Tm}(\mathrm{U})}$$

$$\mathrm{El}(\mathrm{c}(\mathsf{A});\varepsilon) \longmapsto \mathsf{A}$$

Conclusion

Conclusion

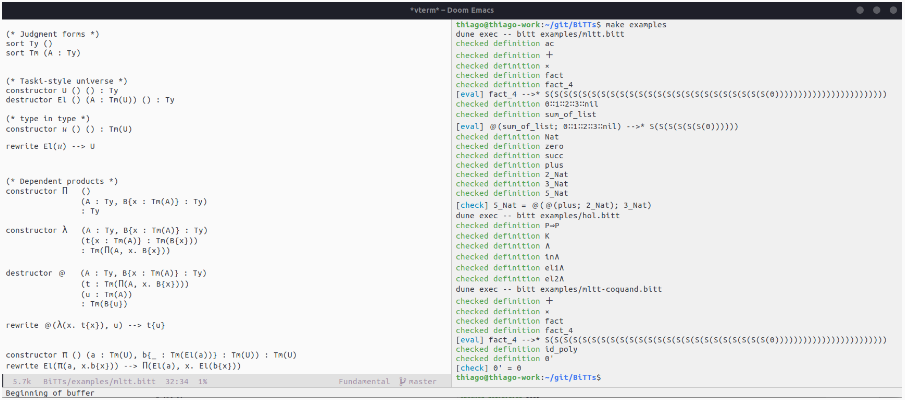
Generic account of bidirectional typing for class of dependent type theories

Conclusion

Generic account of bidirectional typing for class of dependent type theories

Bidirectional system implemented in a prototype, available at

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



Thank you for your attention!