A Framework for Computational Type Theories with Erased Syntax and Bidirectional Typing

Thiago Felicissimo

29th International Conference on Types for Proofs and Programs

June 12, 2023

The goal of this talk

CompLF Logical framework for defining dependent type theories

Capture their usual presentation, in particular non-annotated syntaxes

Generic bidirectional algorithm can be instantiated with various theories

Logical frameworks Unified formalisms for defining type theories

Logical frameworks Unified formalisms for defining type theories

Theoretical interest

- One unified notion of theory, of model, etc
- Theorems proven once and for all

Practical interest

- One unified implementation
- Prototyping new systems (like with rewrite rules in Agda)
- Independent typecheckers for proof assistants (as in Dedukti)

LFs can be classified in two groups

Pure LFs

Equational LFs

LFs can be classified in two groups

Pure LFs

- Fixed definitional equality, ~~terms~~ derivations encoded as terms

Equational LFs

LFs can be classified in two groups

Pure LFs

- Fixed definitional equality, ~~terms~~ derivations encoded as terms

✓ Good for formalizing metatheory (Twelf, Beluga)

Equational LFs

LFs can be classified in two groups

Pure LFs

- Fixed definitional equality, ~~terms~~ derivations encoded as terms

✓ Good for formalizing metatheory (Twelf, Beluga)

✗ Typechecker for ~~the theory~~ its judgment derivations

Equational LFs

LFs can be classified in two groups

Pure LFs

- Fixed definitional equality, ~~terms~~ derivations encoded as terms

✓ Good for formalizing metatheory (Twelf, Beluga)

✗ Typechecker for ~~the theory~~ its judgment derivations

Equational LFs

✓ Customizable definitional equality, allows defining type theories directly

LFs can be classified in two groups

Pure LFs

- Fixed definitional equality, ~~terms~~ derivations encoded as terms

✓ Good for formalizing metatheory (Twelf, Beluga)

✗ Typechecker for ~~the theory~~ its judgment derivations

Equational LFs

✓ Customizable definitional equality, allows defining type theories directly

✗ Customizable definitional equality, how to implement?

Dedukti

- Idea: restrict supported equalities, making implementation tractable
  Only *computational* theories, that is, equality generated only by rewriting

Dedukti

- Idea: restrict supported equalities, making implementation tractable
  Only *computational* theories, that is, equality generated only by rewriting

✓ Rewriting allows (fast!) theory-agnostic equality checking
  (experience rechecking big proof libraries confirms this)

Dedukti

- Idea: restrict supported equalities, making implementation tractable
  Only *computational* theories, that is, equality generated only by rewriting

✓ Rewriting allows (fast!) theory-agnostic equality checking
  (experience rechecking big proof libraries confirms this)

However, mismatch between syntax of your type theory vs syntax of it in Dedukti

Dedukti

- Idea: restrict supported equalities, making implementation tractable
  Only *computational* theories, that is, equality generated only by rewriting

✓ Rewriting allows (fast!) theory-agnostic equality checking
(experience rechecking big proof libraries confirms this)

However, mismatch between syntax of your type theory vs syntax of it in Dedukti

✗ "Bureaucratic" meaningless terms, not in the image of translation function

✓ $\lambda(x.@(t, x))$ ✗ $\lambda(@(t))$ ✗ $\lambda((z.z)(@, t))$

Dedukti

- Idea: restrict supported equalities, making implementation tractable
  Only *computational* theories, that is, equality generated only by rewriting

✓ Rewriting allows (fast!) theory-agnostic equality checking
(experience rechecking big proof libraries confirms this)

However, mismatch between syntax of your type theory vs syntax of it in Dedukti

✗ "Bureaucratic" meaningless terms, not in the image of translation function
✓ $\lambda(x.@(t,x))$ ✗ $\lambda(@(t))$ ✗ $\lambda((z.z)(@,t))$

✗ Only supports fully annotated syntax: $\langle t,u\rangle \implies \langle t,u\rangle_{A,x.B}$
  - Impacts performance and user experience
  - Makes difficult to relate to standard non-annotated presentation
  - Excess of annotations interacts badly with rewriting, adds non-linearity

1st Contribution | propose CompLF

1st Contribution I propose CompLF

- A logical framework for computational type theories
  Like in Dedukti, easy theory-agnostic equality checking with rewriting

1st Contribution I propose CompLF

- A logical framework for computational type theories
  Like in Dedukti, easy theory-agnostic equality checking with rewriting

But, faithful representation of syntax

1st Contribution I propose CompLF

- A logical framework for computational type theories
  Like in Dedukti, easy theory-agnostic equality checking with rewriting

But, faithful representation of syntax

✓ No bureaucratic terms, only meaningful ones
  $\cancel{\lambda((z.z)(@, t))}$ $\quad$ $\cancel{\lambda(@(t))}$ $\quad$ $\lambda(x.@(t, x))$

1st Contribution I propose CompLF

- A logical framework for computational type theories
  Like in Dedukti, easy theory-agnostic equality checking with rewriting

But, faithful representation of syntax

✓ No bureaucratic terms, only meaningful ones
  $\cancel{\lambda((z.z)(@, t))}$ $\quad \cancel{\lambda(@(t))}$ $\quad \lambda(x.@(t, x))$

✓ Supports theories with non-annotated syntaxes

Example Minimalistic MLTT defined by $\mathbb{T}_{\lambda\Pi} := (\Sigma_{\lambda\Pi}, \mathcal{R}_{\lambda\Pi})$

$$\mathcal{R}_{\lambda\Pi} := \quad \mathtt{@}(\lambda(x.\mathtt{t}(x), \mathtt{u})) \longmapsto \mathtt{t}(\mathtt{u})$$

$$\Sigma_{\lambda\Pi} :=$$

$$\begin{aligned}
\mathsf{Ty} &: \ \square \\
\mathsf{Tm} &: \ (\mathtt{A} : \mathsf{Ty}) \to \square \\
\Pi &: \ (\mathtt{A} : \mathsf{Ty})(\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}) \to \mathsf{Ty} \\
\lambda &: \ \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} \\
&\quad\ (\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x))) \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x))) \\
\mathtt{@} &: \ \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} \\
&\quad\ (\mathtt{t} : \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x))))(\mathtt{u} : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(\mathtt{u}))
\end{aligned}$$

Example Minimalistic MLTT defined by $\mathbb{T}_{\lambda\Pi} := (\Sigma_{\lambda\Pi}, \mathcal{R}_{\lambda\Pi})$

$$\mathcal{R}_{\lambda\Pi} := \quad \mathtt{@}(\lambda(x.\mathtt{t}(x), \mathtt{u})) \longmapsto \mathtt{t}(\mathtt{u})$$

$\Sigma_{\lambda\Pi} :=$

$$\begin{aligned}
\mathsf{Ty} &: \ \square \\
\mathsf{Tm} &: \ (\mathtt{A} : \mathsf{Ty}) \to \square \\
\Pi &: \ (\mathtt{A} : \mathsf{Ty})(\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}) \to \mathsf{Ty} \\
\lambda &: \ \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} \\
&\quad (\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x))) \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x))) \\
\mathtt{@} &: \ \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} \\
&\quad (\mathtt{t} : \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x))))(\mathtt{u} : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(\mathtt{u}))
\end{aligned}$$

$\varsigma_{\lambda\Pi} :=$

$$\begin{aligned}
\mathsf{Ty} &:: \ \square \\
\mathsf{Tm} &:: \ (\mathtt{A} :: \mathsf{ty}) \to \square \\
\Pi &:: \ (\mathtt{A} :: \mathsf{ty})(\mathtt{B} :: (x :: \mathsf{tm}) \to \mathsf{ty}) \to \mathsf{ty} \\
\lambda &:: \ (\mathtt{t} :: (x :: \mathsf{tm}) \to \mathsf{tm}) \to \mathsf{tm} \\
\mathtt{@} &:: \ (\mathtt{t} :: \mathsf{tm})(\mathtt{u} :: \mathsf{tm}) \to \mathsf{tm}
\end{aligned}$$

$\Sigma_{\lambda\Pi}$ describes typing rules, $\varsigma_{\lambda\Pi}$ describes syntax

Example Minimalistic MLTT defined by $\mathbb{T}_{\lambda\Pi} := (\Sigma_{\lambda\Pi}, \mathcal{R}_{\lambda\Pi})$

$$\mathcal{R}_{\lambda\Pi} := \quad @(\lambda(x.\mathtt{t}(x), \mathtt{u})) \longmapsto \mathtt{t}(\mathtt{u})$$

$$
\begin{array}{rlcrl}
\Sigma_{\lambda\Pi} := & & \xmapsto{|-|} & \varsigma_{\lambda\Pi} := & \\
\mathsf{Ty} : & \Box & \xmapsto{|-|} & \mathsf{Ty} :: & \Box \\
\mathsf{Tm} : & (\mathtt{A} : \mathsf{Ty}) \to \Box & \xmapsto{|-|} & \mathsf{Tm} :: & (\mathtt{A} :: \mathsf{ty}) \to \Box \\
\Pi : & (\mathtt{A} : \mathsf{Ty})(\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}) \to \mathsf{Ty} & \xmapsto{|-|} & \Pi :: & (\mathtt{A} :: \mathsf{ty})(\mathtt{B} :: (x :: \mathsf{tm}) \to \mathsf{ty}) \to \mathsf{ty} \\
\lambda : & \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} & \xmapsto{|-|} & \lambda :: & (\mathtt{t} :: (x :: \mathsf{tm}) \to \mathsf{tm}) \to \mathsf{tm} \\
& (\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x))) \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x))) & & & \\
@ : & \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} & \xmapsto{|-|} & @ :: & (\mathtt{t} :: \mathsf{tm})(\mathtt{u} :: \mathsf{tm}) \to \mathsf{tm} \\
& (\mathtt{t} : \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x))))(\mathtt{u} : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(\mathtt{u})) & & &
\end{array}
$$

$\Sigma_{\lambda\Pi}$ describes typing rules, $\varsigma_{\lambda\Pi}$ describes syntax

Dependency Erasure $|-|$ links specification of typing and syntax

Erased arguments marked with $\{-\}$, erased from the syntax but present in typing

$$\begin{aligned}\lambda :\ & \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} \\ & (\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x))) \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x)))\end{aligned} \quad \xmapsto{|-|} \quad \lambda :: \ (\mathtt{t} :: (x :: \mathsf{tm}) \to \mathsf{tm}) \to \mathsf{tm}$$

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

Erased arguments marked with $\{-\}$, erased from the syntax but present in typing

$$\begin{aligned}\lambda :\ & \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\} \\ & (\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x))) \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x)))\end{aligned} \quad \xmapsto{|-|} \quad \lambda :: (\mathtt{t} :: (x :: \mathsf{tm}) \to \mathsf{tm}) \to \mathsf{tm}$$

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

Problem They jeopardize decidability of typing. Guess arguments?

Bidirectional typing algorithms Alternate between two modes

$\boxed{\Gamma \vdash t \Leftarrow T}$ Check (input: Γ, t, T)

$\boxed{\Gamma \vdash t \Rightarrow T}$ Infer (input: Γ, t) (output: T)

Bidirectional typing algorithms Alternate between two modes

$\boxed{\Gamma \vdash t \Leftarrow T}$ Check (input: Γ, t, T)

$\boxed{\Gamma \vdash t \Rightarrow T}$ Infer (input: Γ, t) (output: T)

Allow specify flow of type information in typing rules

$$\frac{C \longrightarrow^{*} \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}(A) \vdash t \Leftarrow \mathsf{Tm}(B)}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}(C)}$$

Bidirectional typing algorithms Alternate between two modes

$\boxed{\Gamma \vdash t \Leftarrow T}$ Check (input: Γ, t, T)

$\boxed{\Gamma \vdash t \Rightarrow T}$ Infer (input: Γ, t) (output: T)

Allow specify flow of type information in typing rules

$$\frac{C \longrightarrow^{*} \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}(A) \vdash t \Leftarrow \mathsf{Tm}(B)}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}(C)}$$

Complement erased arguments very well, explains why they are redundant

Bidirectional typing algorithms Alternate between two modes

$\boxed{\Gamma \vdash t \Leftarrow T}$ Check (input: Γ, t, T)

$\boxed{\Gamma \vdash t \Rightarrow T}$ Infer (input: Γ, t) (output: T)

Allow specify flow of type information in typing rules

$$\frac{C \longrightarrow^* \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}(A) \vdash t \Leftarrow \mathsf{Tm}(B)}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}(C)}$$

Complement erased arguments very well, explains why they are redundant

Previous work Principles of (dependent) bidirectional typing well-known

However, no generic framework (as far as I know)

Bidirectional typing algorithms Alternate between two modes

$\boxed{\Gamma \vdash t \Leftarrow T}$ Check (input: Γ, t, T)

$\boxed{\Gamma \vdash t \Rightarrow T}$ Infer (input: Γ, t) (output: T)

Allow specify flow of type information in typing rules

$$\frac{C \longrightarrow^{*} \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}(A) \vdash t \Leftarrow \mathsf{Tm}(B)}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}(C)}$$

Complement erased arguments very well, explains why they are redundant

Previous work Principles of (dependent) bidirectional typing well-known

However, no generic framework (as far as I know)

LFs can be used for this!

2nd Contribution Theory-agnostic bidirectional typing algorithm

2nd Contribution Theory-agnostic bidirectional typing algorithm

Can be instantiated by turning theory into a *moded theory*

Each erased argument needs to appear in a *rigid* pattern

2nd Contribution Theory-agnostic bidirectional typing algorithm

Can be instantiated by turning theory into a *moded theory*

Each erased argument needs to appear in a *rigid* pattern

✓ Sound (assuming confluence, subject reduction and theory is well typed)

2nd Contribution Theory-agnostic bidirectional typing algorithm

Can be instantiated by turning theory into a *moded theory*

Each erased argument needs to appear in a *rigid* pattern

- ✓ Sound (assuming confluence, subject reduction and theory is well typed)
- ✓ Complete for *well-moded terms* (assuming also strong normalisation)

2nd Contribution Theory-agnostic bidirectional typing algorithm

Can be instantiated by turning theory into a *moded theory*

Each erased argument needs to appear in a *rigid* pattern

- ✓ Sound (assuming confluence, subject reduction and theory is well typed)
- ✓ Complete for *well-moded terms* (assuming also strong normalisation)

You, the theory designer, chooses amount of annotations and completeness

2nd Contribution Theory-agnostic bidirectional typing algorithm

Can be instantiated by turning theory into a *moded theory*

Each erased argument needs to appear in a *rigid* pattern

- ✓ Sound (assuming confluence, subject reduction and theory is well typed)
- ✓ Complete for *well-moded terms* (assuming also strong normalisation)

You, the theory designer, chooses amount of annotations and completeness

$\lambda^- : \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\}$

$(\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x)))^- \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x)))$

$$\frac{C \longrightarrow^* \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}(A) \vdash t \Leftarrow \mathsf{Tm}(B)}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}(C)}$$

Well-moded $=$ β-normal forms

2nd Contribution Theory-agnostic bidirectional typing algorithm

Can be instantiated by turning theory into a *moded theory*

Each erased argument needs to appear in a *rigid* pattern

- ✓ Sound (assuming confluence, subject reduction and theory is well typed)
- ✓ Complete for *well-moded terms* (assuming also strong normalisation)

You, the theory designer, chooses amount of annotations and completeness

$\lambda^- : \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\}$
$(\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x)))^- \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x)))$

$$\frac{C \longrightarrow^* \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}(A) \vdash t \Leftarrow \mathsf{Tm}(B)}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}(C)}$$

Well-moded = β-normal forms

$\lambda^+ : (\mathtt{A} : \mathsf{Ty})^-\{\mathtt{B} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Ty}\}$
$(\mathtt{t} : (x : \mathsf{Tm}(\mathtt{A})) \to \mathsf{Tm}(\mathtt{B}(x)))^+ \to \mathsf{Tm}(\Pi(\mathtt{A}, x.\mathtt{B}(x)))$

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

Well-moded = all terms

```
(* Judgment forms *)
symbol Ty : *

symbol Tm (A : Ty)- : *


(* Dependent products (lambda not annotated) *)
symbol+ Π (A : Ty)- (B : (x : Tm A) Ty)- : Ty

symbol- λ {A : Ty} {B : (_ : Tm A) Ty} (t : (x : Tm A) Tm B(x))- : Tm Π(A, x. B(x))

symbol+ @ {A : Ty} {B : (_ : Tm A) Ty} (t : Tm Π(A, x. B(x)))+ (u : Tm A)- : Tm B(u)

rew @(λ(x. $t(x)), $u) --> $t($u)


symbol+ T : Ty (* Auxiliary base type *)

(* Example *)
let church1 : Tm Π(Π(T, _. T), _. Π(T, _. T)) := λ(f. λ(x. @(f, x)))
```

```
(* Gives error *)
(* let redex : Tm Π(T, _. T) := λ(x. @(λ(y.y), x)) *)

(* Dependent products (lambda annotated) *)
symbol+ Π' (A : Ty)- (B : (x : Tm A) Ty)- : Ty

symbol+ @' {A : Ty} {B : (_ : Tm A) Ty} (t : Tm Π'(A, x. B(x)))+ (u : Tm A)- : Tm B(u)

symbol+ λ' (A : Ty)- {B : (_ : Tm A) Ty} (t : (x : Tm A) Tm B(x))+ : Tm Π'(A, x. B(x))

rew @'(λ'($T, x. $t(x)), $u) --> $t($u)

(* Now it works! *)
type λ'(T, x. @'(λ'(T, y.y), x))
```

```
1.7k  complf/test/wg6.complf  15:0 21%                                  Fundamental (+4)
[type] λ'(T, x0. @'(λ'(T, x1. x1), x0)) : Tm(Π'(T, x0. T))
thiago@thiago-work:~/git/complf$
```

Beyond dependent products

```
(* Universe *)
symbol+ U : Ty
symbol+ El (A : Tm U)- : Ty

(* Equality type *)
symbol+ Eq (A : Ty)- (t : Tm A)- (u : Tm A)- : Ty

symbol- refl {A : Ty} {t : Tm A} : Tm Eq(A, t, t)

symbol+ J {A : Ty} {a : Tm A} {b : Tm A} (t : Tm Eq(A, a, b))+
       (P : (x : Tm A, y : Tm Eq(A, a, x)) Ty)- (prefl : Tm P(a, refl))- : Tm P(b, t)

rew J(refl, x y. $P(x, y), $prefl) --> $prefl

(* Code in U for Eq *)
symbol+ eq (a : Tm U)- (x : Tm El(a))- (y : Tm El(a))- : Tm U

rew El(eq($a, $x, $y)) --> Eq(El($a), $x, $y)

(* Properties of equality *)
let sym : Tm Π(U, a. Π(El(a), x. Π(El(a), y. Π(Eq(El(a), x, y), _. Eq(El(a), y, x)))))
    :=  λ(a. λ(x. λ(y. λ(p. J(p, z q. Eq(El(a), z, x), refl)))))

let transp : Tm Π(U, a. Π(U, b. Π(Eq(U, a, b), _. Π(El(a), _. El(b)))))
    := λ(a. λ(b. λ(p. λ(x. J(p, z q. El(z), x)))))
```

Beyond dependent products

```
(* Universe *)
symbol+ U : Ty
symbol+ El (A : Tm U)- : Ty

(* Equality type *)
symbol+ Eq (A : Ty)- (t : Tm A)- (u : Tm A)- : Ty

symbol- refl {A : Ty} {t : Tm A} : Tm Eq(A, t, t)

symbol+ J {A : Ty} {a : Tm A} {b : Tm A} (t : Tm Eq(A, a, b))+
       (P : (x : Tm A, y : Tm Eq(A, a, x)) Ty)- (prefl : Tm P(a, refl))- : Tm P(b, t)

rew J(refl, x y. $P(x, y), $prefl) --> $prefl

(* Code in U for Eq *)
symbol+ eq (a : Tm U)- (x : Tm El(a))- (y : Tm El(a))- : Tm U

rew El(eq($a, $x, $y)) --> Eq(El($a), $x, $y)

(* Properties of equality *)
let sym : Tm Π(U, a. Π(El(a), x. Π(El(a), y. Π(Eq(El(a), x, y), _. Eq(El(a), y, x)))))
    :=  λ(a. λ(x. λ(y. λ(p. J(p, z q. Eq(El(a), z, x), refl)))))

let transp : Tm Π(U, a. Π(U, b. Π(Eq(U, a, b), _. Π(El(a), _. El(b)))))
    := λ(a. λ(b. λ(p. λ(x. J(p, z q. El(z), x)))))
```

But also other types (Σ, List, Nat,...), cumulative universes, universe polymorphism, higher-order logic, etc

Conclusion

CompLF Logical framework for computational type theories

Support for non-annotated theories, faithful presentation of syntax

Customisable bidirectional typing algorithm

Prototype implementation at `https://github.com/thiagofelicissimo/complf`

Thank you for your attention!