A logical framework for incremental type-checking

Matthias Puech$^{1,2}$ Yann Régis-Gianas$^2$

$^1$Dept. of Computer Science, University of Bologna

$^2$University Paris 7, CNRS, and INRIA, PPS, team $\pi r^2$

★ January 2011 ★

PPS – Groupe de travail théorie des types et réalisabilité
A paradoxical situation

Observation
We have powerful tools to mechanize the metatheory of (proof) languages
A paradoxical situation

Observation
We have powerful tools to mechanize the metatheory of (proof) languages

... And yet,
Workflow of programming and formal mathematics is still largely inspired by legacy software development (emacs, make, svn, diffs...)
A paradoxical situation

Observation
We have powerful tools to mechanize the metatheory of (proof) languages

... And yet,
Workflow of programming and formal mathematics is still largely inspired by legacy software development (emacs, make, svn, diffs...)

Isn’t it time to make these tools metatheory-aware?
Q: Do you spend more time writing code or editing code?

Today, we use:
- separate compilation
- dependency management
- version control on the scripts
- interactive toplevel with rollback (Coq)
Incrementality in programming & proof languages
Incrementality in programming & proof languages
Incrementality in programming & proof languages

Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
| O => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity.
Qed.

Theorem not_eq_S : forall n : nat, n <> m -> S n <> S m.
Proof.
  red in |- *; auto.
Qed.

(** Injectivity of successor *)

Theorem eq_add_S : forall n m : nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n-m) with (pred (S n) - pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | O => False
  | S _ => True
Incrementality in programming & proof languages

Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
  | O => n
  | S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity.
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *; auto.
Qed.

(** Injectivity of successor *)

Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  rewrite Sn_eq_Sm, trivial.
  rewrite pred_Sn.
  auto using pred_Sn.
Qed.

Definition isSucc (n:nat) : Prop :=
match n with
  | O => False
  | S n => True
end.

Ready

Line: 54 Char: 5
Incrementality in programming & proof languages

Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
  | 0 => n
  | S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity.
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *; auto.
Qed.

(** Injectivity of successor *)

Theorem eq_add_S : forall n m:nat, S n = S m -> n = n.
Proof.
  intros n m Sn_eq_Sm.
  replace (n-m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S n => True
end.
Incrementality in programming & proof languages

Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
   | 0 => n
   | S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity. (* simple proof *)
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *; auto.
Qed.

(** Injectivity of successor *)

Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intro n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
   | 0 => False
   | S n => True
end.
Incrementality in programming & proof languages

Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity. (* simple proof *)
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *; auto.
Qed.

(** Injectivity of successor *)

Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n-m) with (pred (S n) - pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S n => True
Incrementality in programming & proof languages

```coq
Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity. (* simple proof *)
Qed.

Theorem nat_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *; auto.
Qed.

(** Injectivity of successor *)

Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S n => True
```

Ready
In an ideal world...

- Edition should be possible anywhere
- The impact of changes visible “in real time”
- No need for separate compilation, dependency management
In an ideal world...

- Edition should be possible anywhere
- The impact of changes visible “in real time”
- No need for separate compilation, dependency management

*Types are good witnesses of this impact*
In an ideal world...

- Edition should be possible anywhere
- The impact of changes visible “in real time”
- No need for separate compilation, dependency management

Types are good witnesses of this impact

Applications

- non-linear user interaction
- tactic languages
- type-directed programming
- typed version control systems
The big picture

Our approach
  Why not memoization?
  A popular storage model for repositories
  Logical framework
  Positionality

The language
  From LF to NLF
  NLF: Syntax, typing, reduction

Architecture
The big picture

Our approach
Why not memoization?
A popular storage model for repositories
Logical framework
Positionality

The language
From LF to NLF
NLF: Syntax, typing, reduction

Architecture
A logical framework for incremental type-checking

Yes, we’re speaking about (any) typed language.

A type-checker

\[
\textbf{val} \; \text{check} : \text{env} \rightarrow \text{term} \rightarrow \text{types} \rightarrow \text{bool}
\]

- builds and checks the derivation (on the stack)
- conscientiously discards it
A logical framework for incremental type-checking

Yes, we’re speaking about (any) typed language.

A type-checker

```
val check : env → term → types → bool
```

- builds and checks the derivation (on the stack)
- conscientiously discards it

\[
\begin{array}{c}
\frac{A \rightarrow B, B \rightarrow C, A \vdash B \rightarrow C}{A \rightarrow B, B \rightarrow C, A \vdash B} \quad Ax \\
\frac{A \rightarrow B, B \rightarrow C, A \vdash A \rightarrow B}{A \rightarrow B, B \rightarrow C, A \vdash A} \quad Ax \\
\frac{A \rightarrow B, B \rightarrow C, A \vdash B}{A \rightarrow B, B \rightarrow C, A \vdash C} \quad \rightarrow i \\
\frac{A \rightarrow B, B \rightarrow C \vdash A \rightarrow C}{A \rightarrow B \vdash (B \rightarrow C) \rightarrow A \rightarrow C} \quad \rightarrow i \\
\vdash (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C
\end{array}
\]
A logical framework for incremental type-checking

Yes, we’re speaking about (any) typed language.

A type-checker

\[
\text{val check : env } \rightarrow \text{ term } \rightarrow \text{ types } \rightarrow \text{ bool}
\]

- builds and checks the derivation (on the stack)
- conscientiously discards it

true
Goal  Type-check a large derivation taking advantage of the knowledge from type-checking previous versions

Idea  Remember all derivations!
A logical framework for **incremental** type-checking

**Goal**  Type-check a large derivation taking advantage of the knowledge from type-checking previous versions

**Idea**  Remember all derivations!

**Q**  Do we really need faster type-checkers?

**A**  Yes, since we implemented these ad-hoc fixes.
The big picture

version management

script files

parsing

type-checking
The big picture

version management

script files

parsing

type-checking
The big picture

script files

version management

parsing

type-checking
The big picture

script files

parsing

version management

type-checking

- AST representation
The big picture

- script files
  - parsing
    - version management
    - type-checking

- AST representation
The big picture

- user interaction
- parsing
- version management
- type-checking

- AST representation
The big picture

- user interaction
  - parsing
    - type-checking
      - version management

- AST representation
- Typing annotations
The big picture

- user interaction
- parsing
  - incremental type-checking
  - version management

- AST representation
- Typing annotations
The big picture

Our approach
  Why not memoization?
  A popular storage model for repositories
  Logical framework
  Positionality

The language
  From LF to NLF
  NLF: Syntax, typing, reduction

Architecture
Memoization maybe?

```ocaml
let rec check env t a =
  match t with
  | ... → ... false
  | ... → ... true

and infer env t =
  match t with
  | ... → ... None
  | ... → ... Some a
```
Memoization maybe?

```ocaml
let table = ref ([] : environ × term × types) in
let rec check env t a =
    if List.mem (env,t,a) !table then true else
    match t with
    | ... → ... false
    | ... → ... table := (env,t,a)::! table; true
and infer env t =
try List.assoc (env,t) !table with Not_found →
match t with
| ... → ... None
| ... → ... table := (env,t,a)::! table; Some a
```
Memoization maybe?

+ lightweight
Memoization maybe?

+ lightweight
+ efficient implementation

\[
\Gamma \vdash J \text{wf} \Rightarrow \Gamma_1 \vdash J_1 \text{wf} \Rightarrow \Gamma_2 \vdots \Gamma_{n-1} \vdash J_{n-1} \text{wf} \Rightarrow \Gamma_n \vdash J_n \\Gamma_1 \vdash J \text{wf} \Rightarrow \Gamma_n [J_n] [J]
\]

– external to the logic (meta-cut)
– introduces a dissymmetry

What if I want e.g. the weakening property to be taken into account?

– syntactic comparison
– still no trace of the derivation

+ gives good reasons to go on
Memoization maybe?

+ lightweight
+ efficient implementation
- imperative
Memoization maybe?

+ lightweight
+ efficient implementation
- imperative

What does it mean logically?

\[
\begin{align*}
J & \in \Gamma \\
\Gamma & \vdash J \text{ wf } \Rightarrow \Gamma \\
\Gamma_1 & \vdash J_1 \text{ wf } \Rightarrow \Gamma_2 \\
\vdots \quad \Gamma_{n-1}[J_{n-1}] & \vdash J_n \text{ wf } \Rightarrow \Gamma_n \\
\Gamma_1 & \vdash J \text{ wf } \Rightarrow \Gamma_n[J_n][J]
\end{align*}
\]
Memoization maybe?

+ lightweight
+ efficient implementation

- imperative

What does it mean logically?

\[
\frac{J \in \Gamma}{\Gamma \vdash J \text{ wf} \Rightarrow \Gamma}
\]

\[
\frac{\Gamma_1 \vdash J_1 \text{ wf} \Rightarrow \Gamma_2 \quad \ldots \quad \Gamma_{n-1}[J_{n-1}] \vdash J_n \text{ wf} \Rightarrow \Gamma_n}{\Gamma_1 \vdash J \text{ wf} \Rightarrow \Gamma_n[J_n][J]}
\]

- external to the logic (meta-cut)
Memoization maybe?

+ lightweight
+ efficient implementation

- imperative

What does it mean logically?

\[
J \in \Gamma \\
\Gamma \vdash J \text{ wf } \Rightarrow \Gamma
\]

\[
\Gamma_1 \vdash J_1 \text{ wf } \Rightarrow \Gamma_2 \quad \ldots \quad \Gamma_{n-1}[J_{n-1}] \vdash J_n \text{ wf } \Rightarrow \Gamma_n
\]

\[
\Gamma_1 \vdash J \text{ wf } \Rightarrow \Gamma_n[J_n][J]
\]

- external to the logic (meta-cut)
- introduces a dissymmetry
Memoization maybe?

+ lightweight
+ efficient implementation
- imperative

What does it mean logically?

\[ \frac{J \in \Gamma}{\Gamma \vdash J \text{ wf} \Rightarrow \Gamma} \]

\[ \frac{\Gamma_1 \vdash J_1 \text{ wf} \Rightarrow \Gamma_2 \quad \ldots \quad \Gamma_{n-1}[J_{n-1}] \vdash J_n \text{ wf} \Rightarrow \Gamma_n}{\Gamma_1 \vdash J \text{ wf} \Rightarrow \Gamma_n[J_n][J]} \]

- external to the logic (meta-cut)
- introduces a dissymmetry

What if I want *e.g.* the weakening property to be taken into account?
Memoization maybe?

+ lightweight
+ efficient implementation
- imperative

What does it mean logically?

\[ J \in \Gamma \]
\[ \Gamma \vdash J \text{wf} \Rightarrow \Gamma \]

\[ \Gamma_1 \vdash J_1 \text{wf} \Rightarrow \Gamma_2 \ldots \Gamma_{n-1}[J_{n-1}] \vdash J_n \text{wf} \Rightarrow \Gamma_n \]
\[ \Gamma_1 \vdash J \text{wf} \Rightarrow \Gamma_n[J_n][J] \]

- external to the logic (meta-cut)
- introduces a dissymmetry

What if I want e.g. the weakening property to be taken into account?

- syntactic comparison
Memoization maybe?

+ lightweight
+ efficient implementation
- imperative

What does it mean logically?

\[
J \in \Gamma \\
\Gamma \vdash J \text{ wf } \Rightarrow \Gamma
\]

\[
\Gamma_1 \vdash J_1 \text{ wf } \Rightarrow \Gamma_2 \quad \ldots \quad \Gamma_{n-1}[J_{n-1}] \vdash J_n \text{ wf } \Rightarrow \Gamma_n \\
\Gamma_1 \vdash J \text{ wf } \Rightarrow \Gamma_n[J_n][J]
\]

- external to the logic (meta-cut)
- introduces a dissymmetry

What if I want e.g. the weakening property to be taken into account?

- syntactic comparison
- still no trace of the derivation
Memoization maybe?

+ lightweight
+ efficient implementation
- imperative

What does it mean logically?

\[ J \in \Gamma \]
\[ \Gamma \vdash J \text{wf } \Rightarrow \Gamma \]

\[ \Gamma_1 \vdash J_1 \text{wf } \Rightarrow \Gamma_2 \quad \ldots \quad \Gamma_{n-1}[J_{n-1}] \vdash J_n \text{wf } \Rightarrow \Gamma_n \]
\[ \Gamma_1 \vdash J \text{wf } \Rightarrow \Gamma_n[J_n][J] \]

- external to the logic (meta-cut)
- introduces a dissymmetry

What if I want e.g. the weakening property to be taken into account?

- syntactic comparison
- still no trace of the derivation
+ gives good reasons to go on
A popular storage model for repositories

```
/v1
/foobar
/foo/a /bar/b /bar/c
```

```
/v1
/foobar
/foo /bar
```

```
/v1
```

```
```

```
```
A popular storage model for repositories

```
/ /foo/a
|   /bar/b
|   /bar/c
|   /foo
|   /bar
|   /foo/a
|   /bar/b
|   /bar/c'
|   /foo
|   /bar
|   /foo/a
|   /bar/b
|   /bar/c'
```

v1
A popular storage model for repositories
A popular storage model for repositories
A popular storage model for repositories
A popular storage model for repositories

```
/ /foo
  /a
    99dcf1d

/ /bar
  /b
    46f9c2
    923ace3
    6e4f99a
    d54c809

/ /bar
  /c
    cf189a6
    46f9c2
    c328e8f

/ /bar
  /c'
    v1
    v2
    820e7ab
    a85f0b77

99dcf1d → "Hello World"
d54c809 → 46f9c2, 923ace3
6e4f99a → 99dcf1d
4244e8a → 6e4f99a, d54c809
cf189a6 → 46f9c2, c328e8f
...```
A popular storage model for repositories

\[\begin{align*}
/\text{foo}/a & \quad 99\text{dcf1d} \\
/\text{bar}/b & \quad 46\text{f9c2} \\
/\text{bar}/c & \quad 92\text{3ace3} \\
\end{align*}\]
A popular storage model for repositories

The repository $R$ is a pair $(\Delta, x)$:

$$\Delta : x \mapsto (\text{Commit } (x \times y) \mid \text{Tree } \vec{x} \mid \text{Blob } \text{string})$$

with the invariants:

- if $(x, \text{Commit } (y, z)) \in \Delta$ then
  - $(y, \text{Tree } t) \in \Delta$
  - $(z, \text{Commit } (t, v)) \in \Delta$

- if $(x, \text{Tree}(\vec{y})) \in \Delta$ then
  for all $y_i$, either $(y_i, \text{Tree}(\vec{z}))$ or $(y_i, \text{Blob}(s)) \in \Delta$
A popular storage model for repositories

The repository $R$ is a pair $(\Delta, x)$:

$$
\Delta : x \mapsto (\text{Commit } (x \times y) \mid \text{Tree } \vec{x} \mid \text{Blob } \text{string})
$$

with the invariants:

- if $(x, \text{Commit } (y, z)) \in \Delta$ then
  - $(y, \text{Tree } t) \in \Delta$
  - $(z, \text{Commit } (t, v)) \in \Delta$
- if $(x, \text{Tree}(\vec{y})) \in \Delta$ then
  for all $y_i$, either $(y_i, \text{Tree}(\vec{z}))$ or $(y_i, \text{Blob}(s)) \in \Delta$

Let’s do the same with proofs
A *typed* repository of proofs

\[ \pi_1 : A \land B \vdash C \quad \pi_2 : \vdash A \quad \pi_3 : \vdash B \]

\[ \lambda : \vdash (A \land B) \to C \quad \land, : \vdash A \land B \]

\[ \vdash C \]

\[ \nu 1 \]
A *typed* repository of proofs
A typed repository of proofs
A *typed* repository of proofs

\[ \pi_1 : A \land B \vdash C \]
\[ \pi_2 : \vdash A \]
\[ \pi_3 : \vdash B \]
\[ \lambda - : \vdash (A \land B) \rightarrow C \]
\[ \vdash -,- : A \land B \]
\[ \pi_1 : A \land B \vdash C \]
\[ \pi_2 : \vdash A \]
\[ \pi_3 : \vdash B \]
\[ \pi_3' : \vdash B \]
A *typed* repository of proofs
A *typed* repository of proofs

\[
x = \ldots : A \land B \vdash C
\]

\[
y = \ldots : \vdash A
\]

\[
z = \ldots : \vdash B
\]

\[
t = \lambda a^{A \land B} \cdot x : \vdash A \land B \to C
\]

\[
u = (y, z) : \vdash A \land B
\]

\[
v = t \ u : \vdash C
\]

\[
w = \text{Commit}(v, w1) : \text{Version}
\]
A typed repository of proofs

\[ x = \ldots : A \land B \vdash C \]
\[ y = \ldots : \vdash A \]
\[ z = \ldots : \vdash B \]
\[ t = \lambda a^{A \land B} \cdot x : \vdash A \land B \rightarrow C \]
\[ u = (y, z) : \vdash A \land B \]
\[ v = t \ u : \vdash C \]
\[ w = \text{Commit}(v, w_1) : \text{Version} \quad , \quad w \]
A typed repository of proofs

\[ x = \ldots : A \land B \vdash C \]
\[ y = \ldots : \vdash A \]
\[ z = \ldots : \vdash B \]
\[ t = \lambda x.A \land B \cdot x : \vdash A \land B \rightarrow C \]
\[ u = (y, z) : \vdash A \land B \]
\[ v = t \ u : \vdash C \]
\[ w = \text{Commit}(v, w1) : \text{Version} \]
\[ p = \ldots : \vdash B \]
\[ q = (y, p) : \vdash A \land B \]
\[ r = t \ q : \vdash C \]
\[ s = \text{Commit}(r, w) : \text{Version} \]
A *typed* repository of proofs

\[
x = \ldots : A \land B \vdash C
\]
\[
y = \ldots : \vdash A
\]
\[
z = \ldots : \vdash B
\]
\[
t = \lambda a^{A \land B} \cdot x : \vdash A \land B \rightarrow C
\]
\[
u = (y, z) : \vdash A \land B
\]
\[
v = t \ u : \vdash C
\]
\[
w = \text{Commit}(v, w1) : \text{Version}
\]
\[
p = \ldots : \vdash B
\]
\[
q = (y, p) : \vdash A \land B
\]
\[
r = t \ q : \vdash C
\]
\[
s = \text{Commit}(r, w) : \text{Version}
\]
A *typed* repository of proofs

```
let x = ... : is (cons (conj A B) nil) C in
let y = ... : is nil A in
let z = ... : is nil B in
  let t = lam (conj A B) x : is nil (arr (conj A B) C) in
  let u = pair y z : is nil (conj A B) in
  let v = app t u : is nil C in
  let w = commit v w1 : version in
  w
```
A typed repository of proofs

let x = ... : is (cons (conj A B) nil) C in
let y = ... : is nil A in
let z = ... : is nil B in
  let t = lam (conj A B) x : is nil (arr (conj A B) C) in
  let u = pair y z : is nil (conj A B) in
  let v = app t u : is nil C in
  let w = commit v w1 : version in
let p = ... : is nil B
  let q = pair y p : is nil (conj A B) in
  let r = t q : is nil C
  let s = commit r w : version in
  s
A *typed* repository of proofs

... 

\textbf{val} \ is \ : \ \text{env} \to \ \text{prop} \to \ \text{type}  
\textbf{val} \ \textbf{conj} \ : \ \text{prop} \to \ \text{prop} \to \ \text{prop}  
\textbf{val} \ \textbf{pair} \ : \ \text{is} \ \alpha \ \beta \to \ \text{is} \ \alpha \ \gamma \to \ \text{is} \ \alpha \ (\ \text{conj} \ \beta \ \gamma)  
\textbf{val} \ \textbf{version} \ : \ \text{type}  
\textbf{val} \ \textbf{commit} \ : \ \text{is} \ \text{nil} \ \text{C} \to \ \text{version} \to \ \text{version}  

... 

\begin{align*} 
\textbf{let} \ x = \ldots & : \ \text{is} \ (\ \text{cons} \ (\ \text{conj} \ \alpha \ \beta) \ \text{nil}) \ \text{C} \ \textbf{in} \\
\textbf{let} \ y = \ldots & : \ \text{is} \ \text{nil} \ \alpha \ \textbf{in} \\
\textbf{let} \ z = \ldots & : \ \text{is} \ \text{nil} \ \beta \ \textbf{in} \\
\textbf{let} \ t = \ \text{lam} \ (\ \text{conj} \ \alpha \ \beta) \ x : \ \text{is} \ \text{nil} \ (\ \text{arr} \ (\ \text{conj} \ \alpha \ \beta) \ \text{C}) \ \textbf{in} \\
\textbf{let} \ u = \ \text{pair} \ y \ z : \ \text{is} \ \text{nil} \ (\ \text{conj} \ \alpha \ \beta) \ \textbf{in} \\
\textbf{let} \ v = \ \text{app} \ t \ u : \ \text{is} \ \text{nil} \ \text{C} \ \textbf{in} \\
\textbf{let} \ w = \ \text{commit} \ v \ w1 : \ \text{version} \ \textbf{in} \\
\textbf{let} \ p = \ldots : \ \text{is} \ \text{nil} \ \beta \\
\textbf{let} \ q = \ \text{pair} \ y \ p : \ \text{is} \ \text{nil} \ (\ \text{conj} \ \alpha \ \beta) \ \textbf{in} \\
\textbf{let} \ r = t \ q : \ \text{is} \ \text{nil} \ \text{C} \\
\textbf{let} \ s = \ \text{commit} \ r \ w : \ \text{version} \ \textbf{in} \\
\end{align*}
A logical framework for incremental type-checking

LF [Harper et al. 1992] provides a way to represent and validate syntax, rules and proofs by means of a typed λ-calculus. But we need a little bit more:

\[
\begin{align*}
\text{let } u &= \text{pair } y z : \text{is nil (conj } A B) \text{ in} \\
&\hspace{1cm} \text{let } v &= \text{app } t u : \text{is nil } C \text{ in} \\
\end{align*}
\]

\[
\ldots
\]
A logical framework for incremental type-checking

LF [Harper et al. 1992] provides a way to represent and validate syntax, rules and proofs by means of a typed λ-calculus. But we need a little bit more:

\begin{verbatim}
... 
let u = pair y z : is nil (conj A B) in 
  let v = app t u : is nil C in 
...
1. definitions / explicit substitutions
\end{verbatim}
A logical framework for incremental type-checking

LF [Harper et al. 1992] provides a way to represent and validate syntax, rules and proofs by means of a typed λ-calculus. But we need a little bit more:

... 
let u = pair y z : is nil (conj A B) in
  let v = app t u : is nil C in
...

1. definitions / explicit substitutions
2. type annotations on application spines
A logical framework for incremental type-checking

LF [Harper et al. 1992] provides a way to represent and validate syntax, rules and proofs by means of a typed λ-calculus. But we need a little bit more:

\[
\begin{align*}
\textbf{let } u &= \text{pair } y \ z : \text{is nil (conj } A \ B) \ \textbf{in} \\
&\hspace{1em} \textbf{let } v &= \text{app } t \ u : \text{is nil } C \ \textbf{in} \\
\end{align*}
\]

...  

1. definitions / explicit substitutions  
2. type annotations on application spines  
3. fully applied constants / η-long NF
A logical framework for incremental type-checking

LF [Harper et al. 1992] provides a way to represent and validate syntax, rules and proofs by means of a typed $\lambda$-calculus. But we need a little bit more:

\[
\text{let } u = \text{pair } y \ z : \text{is nil (conj } A \ B) \text{ in}
\]
\[
\text{let } v = \text{app } t \ u : \text{is nil } C \text{ in}
\]

1. definitions / explicit substitutions
2. type annotations on application spines
3. fully applied constants / $\eta$-long NF
4. Naming of all application spines / A-normal form
   (= construction of syntax/proofs)
$R =$

\[
\begin{align*}
\text{let } x &= \ldots : \text{is } (\text{cons } (\text{conj } A B) \text{ nil }) \text{ C in} \\
\text{let } y &= \ldots : \text{is } \text{ nil } A \text{ in} \\
\text{let } z &= \ldots : \text{is } \text{ nil } B \text{ in} \\
\text{let } t &= \text{lam } (\text{conj } A B) x : \text{is } \text{ nil } (\text{arr } (\text{conj } A B) \text{ C}) \text{ in} \\
\text{let } u &= \text{pair } y \ z : \text{is } \text{ nil } (\text{conj } A B) \text{ in} \\
\text{let } v &= \text{app } t \ u : \text{is } \text{ nil } C \text{ in} \\
\text{let } w &= \text{commit } v \ w1 : \text{version} \text{ in} \\
w
\end{align*}
\]
\[ R = \]

\[
\begin{align*}
\text{let } & \quad x = \ldots : \text{is (cons (conj A B) nil) C in} \\
\text{let } & \quad y = \ldots : \text{is nil A in} \\
\text{let } & \quad z = \ldots : \text{is nil B in} \\
\text{let } & \quad t = \text{lam (conj A B) x : is nil (arr (conj A B) C) in} \\
\text{let } & \quad u = \text{pair y z : is nil (conj A B) in} \\
\text{let } & \quad v = \text{app t u : is nil C in} \\
\text{let } & \quad w = \text{commit v w1 : version in} \\
\end{align*}
\]

- Expose the *head* of the term
Positionality

\[ R = \]

\[
\begin{align*}
\text{let } x &= \ldots : \text{is } (\text{cons } (\text{conj } A \ B) \ \text{nil}) \ C \ \text{in} \\
\text{let } y &= \ldots : \text{is } \text{nil } A \ \text{in} \\
\text{let } z &= \ldots : \text{is } \text{nil } B \ \text{in} \\
\text{let } t &= \text{lam } (\text{conj } A \ B) \ x : \text{is } \text{nil } (\text{arr } (\text{conj } A \ B) \ C) \ \text{in} \\
\text{let } u &= \text{pair } y \ z : \text{is } \text{nil } (\text{conj } A \ B) \ \text{in} \\
\text{let } v &= \text{app } t \ u : \text{is } \text{nil } C \ \text{in} \\
\text{let } w &= \text{commit } v \ w1 : \text{version } \text{in} \\
\end{align*}
\]

-Expose the head of the term

\[(\lambda x.\lambda y. T) \ U \ V\]
Positionality

\[ R = \]

\[
\text{let } x = \ldots : \text{is cons (conj A B) nil ) C in}
\]

\[
\text{let } y = \ldots : \text{is nil A in}
\]

\[
\text{let } z = \ldots : \text{is nil B in}
\]

\[
\text{let } t = \text{lam (conj A B) x : is nil (arr (conj A B) C) in}
\]

\[
\text{let } u = \text{pair y z : is nil (conj A B) in}
\]

\[
\text{let } v = \text{app t u : is nil C in}
\]

\[
\text{let } w = \text{commit v w1 : version in}
\]

\[
w
\]

- Expose the head of the term

\[
(\lambda x.\lambda y.T) \ U \ V
\]

- Abstract from the positions of the binders
  (from inside and from outside)
Menu

The big picture

Our approach
- Why not memoization?
- A popular storage model for repositories
- Logical framework
- Positionality

The language
- From LF to NLF
- NLF: Syntax, typing, reduction

Architecture
Presentation (of the ongoing formalization)

- alternative syntax for LF
- a datastructure of LF derivations
- the repository storage model

**Motto:** *Take control of the environment*

$$\lambda_{LF} \xrightarrow{\text{LJ-style annotation}} \text{XLF} \xrightarrow{\text{heads exposed, forget positions}} \text{NLF}$$
From LF to XLF

\[ K ::= \Pi x^A \cdot K \mid * \]
\[ A ::= \Pi x^A \cdot A \mid A \ t \mid a \]
\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t \ t \mid x \mid c \]
\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]
\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from standard $\lambda_{LF}$ with definitions
From LF to XLF

\[ K ::= \Pi x^A \cdot K \mid * \]
\[ A ::= \Pi x^A \cdot A \mid A[l] \mid a[l] \]
\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] \mid x[l] \mid c[l] \]
\[ l ::= \cdot \mid t; l \]
\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]
\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from standard \( \lambda_{LF} \) with definitions
- sequent calculus-like applications (\( \bar{\lambda} \))
From LF to XLF

\[ K ::= \Pi x^A \cdot K \mid \ast \]
\[ A ::= \Pi x^A \cdot A \mid A[l] : K \mid a[l] : K \]
\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A \]
\[ l ::= \cdot \mid t; l \]
\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]
\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from standard \( \lambda_{LF} \) with definitions
- sequent calculus-like applications (\( \bar{\lambda} \))
- type annotations on application spines
From LF to XLF

\[ K ::= \Pi x^A \cdot K | * \]
\[ A ::= \Pi x^A \cdot A | A[l] : K | a[l] : K \]
\[ t ::= \lambda x^A \cdot t | \text{let } x = t \text{ in } t | t[l] : A | x[l] : A | c[l] : A \]
\[ l ::= \cdot | t ; l \]
\[ \Gamma ::= \cdot | \Gamma [x : A] | \Gamma [x = t] \]
\[ \Sigma ::= \cdot | \Sigma [c : A] | \Sigma [a : K] \]

- start from standard \( \lambda_{LF} \) with definitions
- sequent calculus-like applications (\( \bar{\lambda} \))
- type annotations on application spines
From LF to XLF

\[ K ::= \Pi x^A \cdot K \mid * \]
\[ A ::= \Pi x^A \cdot A \mid A[l] : K \mid a[l] : K \]
\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A \]
\[ l ::= \cdot \mid t; l \]
\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]
\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from standard \( \lambda_{LF} \) with definitions
- sequent calculus-like applications (\( \bar{\lambda} \))
- type annotations on application spines

\[
\frac{\Gamma \vdash t : A \quad \Gamma [x = t], B \vdash l : C}{\Gamma, \Pi x^A \cdot B \vdash t; l : C}
\]
From LF to XLF

\[ K ::= \Pi x^A \cdot K \mid * \]

\[ A ::= \Pi x^A \cdot A \mid A[l] : K \mid a[l] : K \]

\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A \]

\[ l ::= \cdot \mid x = t ; l \]

\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]

\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from standard \( \lambda_{LF} \) with definitions
- sequent calculus-like applications (\( \bar{\lambda} \))
- type annotations on application spines
- named arguments

\[
\frac{
\Gamma \vdash t : A \quad \Gamma [x = t], B \vdash l : C}
{\Gamma, \Pi x^A \cdot B \vdash x = t ; l : C}
\]
From LF to XLF

\[
K ::= \Pi x^A \cdot K \mid *
\]
\[
A ::= \Pi x^A \cdot A \mid A[l] : K \mid a[l] : K
\]
\[
t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A
\]
\[
l ::= \cdot \mid x = t ; l
\]
\[
\Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t]
\]
\[
\Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K]
\]

- start from standard $\lambda_{LF}$ with definitions
- sequent calculus-like applications ($\bar{\lambda}$)
- type annotations on application spines
- named arguments

\[
\text{FV}(t[l] : A) = \text{FV}(t) \cup \text{FV}(l) \cup (\text{FV}(A) - \text{FV}(l))
\]
\[
\text{FV}(x = t ; l) = \text{FV}(t) \cup (\text{FV}(l) - \{x\})
\]
XLF: Properties

- LJ-style application
- type annotation on application spines
- named arguments (labels)

**Lemma (Conservativity)**

- \( \Gamma \vdash_{\text{LF}} K \text{ kind} \quad \text{iff} \quad |\Gamma| \vdash_{\text{XLF}} |K| \text{ kind} \)
- \( \Gamma \vdash_{\text{LF}} A \text{ type} \quad \text{iff} \quad |\Gamma| \vdash_{\text{XLF}} |A| \text{ type} \)
- \( \Gamma \vdash_{\text{LF}} t : A \quad \text{iff} \quad |\Gamma| \vdash_{\text{XLF}} |t| : |A| \)
From XLF to NLF

\[
K ::= \Pi x^A \cdot K \mid \ast
\]

\[
A ::= \Pi x^A \cdot A \mid A[l] : K \mid a[l] : K
\]

\[
t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A
\]

\[
l ::= \cdot \mid x = t ; l
\]

\[
\Gamma ::= \cdot \mid \Gamma[x : A] \mid \Gamma[x = t]
\]

\[
\Sigma ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K]
\]

▶ start from XLF
From XLF to NLF

\[ \begin{align*}
K & ::= \Pi x^A \cdot K \mid h_K \\
\Sigma & ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \\
A & ::= \Pi x^A \cdot A \mid A[l] : K \mid a[l] : K \\
t & ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A \\
l & ::= \cdot \mid x = t ; l \\
\Gamma & ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t]
\end{align*} \]

- start from XLF
- isolate heads (non-binders)
From XLF to NLF

\[
K ::= \Pi x^A \cdot K \mid h_K
\]

\[
h_K ::= *
\]

\[
A ::= \Pi x^A \cdot A \mid h_A
\]

\[
h_A ::= A[l] : K \mid a[l] : K
\]

\[
t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid t[l] : A \mid x[l] : A \mid c[l] : A
\]

\[
l ::= \cdot \mid x = t;l
\]

\[
\Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t]
\]

\[
\Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K]
\]

- start from XLF
- isolate heads (non_binders)
From XLF to NLF

\[ K ::= \Pi x^A \cdot K \mid h_K \]

\[ h_K ::= * \]

\[ A ::= \Pi x^A \cdot A \mid h_A \]

\[ h_A ::= A[l] : K \mid a[l] : K \]

\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid h_t \]

\[ h_t ::= t[l] : A \mid x[l] : A \mid c[l] : A \]

\[ l ::= \cdot \mid x = t; l \]

\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]

\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from XLF
- isolate heads (non-binders)
From XLF to NLF

\[ K ::= \Pi x^A \cdot K \mid h_K \]

\[ h_K ::= * \]

\[ A ::= \Pi x^A \cdot A \mid h_A \]

\[ h_A ::= A[l] : h_K \mid a[l] : h_K \]

\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid h_t \]

\[ h_t ::= t[l] : A \mid x[l] : A \mid c[l] : A \]

\[ l ::= \cdot \mid x = t; l \]

\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]

\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from XLF
- isolate heads (non-binders)
- enforce \( \eta \)-long forms by annotating with heads
From XLF to NLF

\[ K ::= \Pi x^A \cdot K \mid h_K \]
\[ h_K ::= * \]
\[ A ::= \Pi x^A \cdot A \mid h_A \]
\[ h_A ::= A[l] : h_K \mid a[l] : h_K \]
\[ t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid h_t \]
\[ h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A \]
\[ l ::= \cdot \mid x = t; l \]
\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]
\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from XLF
- isolate heads (non-binders)
- enforce \( \eta \)-long forms by annotating with heads
From XLF to NLF

\[
K ::= \Pi x^A \cdot K \mid h_K \\
\text{where } h_K ::= * \\
A ::= \Pi x^A \cdot A \mid h_A \\
h_A ::= A[l] : h_K \mid a[l] : h_K \\
t ::= \lambda x^A \cdot t \mid \text{let } x = t \text{ in } t \mid h_t \\
h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A \\
l ::= \cdot \mid x = t; l \\
\Gamma ::= \cdot \mid \Gamma[x : A] \mid \Gamma[x = t] \\
\Sigma ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K]
\]

- start from XLF
- isolate heads (non-binders)
- enforce \(\eta\)-long forms by annotating with heads

\[
t[l] : \Pi x^A \cdot B \quad \xrightarrow{\eta} \quad \lambda x^A \cdot (t[x = x; l] : B) \quad x \not\in \text{FV}(t)
\]
From XLF to NLF

\[
K ::= \Pi x^A \cdot K \mid h_K
\]
\[
h_K ::= \ast
\]
\[
A ::= \Pi x^A \cdot A \mid h_A
\]
\[
h_A ::= A[l] : h_K \mid a[l] : h_K
\]
\[
t ::= \Gamma \vdash h_t
\]
\[
h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A
\]
\[
l ::= \cdot \mid x = t;l
\]
\[
\Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t]
\]
\[
\Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K]
\]

- start from XLF
- isolate heads (non-binders)
- enforce \(\eta\)-long forms by annotating with heads
- factorize binders and environments
From XLF to NLF

\[
K ::= \Pi x^A \cdot K \mid h_K
\]
\[
h_K ::= *
\]
\[
A ::= \Gamma \vdash h_A
\]
\[
h_A ::= A[l] : h_K \mid a[l] : h_K
\]
\[
t ::= \Gamma \vdash h_t
\]
\[
h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A
\]
\[
l ::= \cdot \mid x = t; l
\]
\[
\Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t]
\]
\[
\Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K]
\]

▶ start from XLF
▶ isolate heads (non-binders)
▶ enforce \(\eta\)-long forms by annotating with heads
▶ factorize binders and environments
From XLF to NLF

\[ K ::= \Gamma \vdash h_k \]

\[ h_K ::= * \]

\[ A ::= \Gamma \vdash h_A \]

\[ h_A ::= A[l] : h_K \mid a[l] : h_K \]

\[ t ::= \Gamma \vdash h_t \]

\[ h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A \]

\[ l ::= \cdot \mid x = t; l \]

\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]

\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from XLF
- isolate heads (non-binders)
- enforce \( \eta \)-long forms by annotating with heads
- factorize binders and environments
From XLF to NLF

\[ K ::= \Gamma \vdash h_k \]

\[ h_K ::= \ast \]

\[ A ::= \Gamma \vdash h_A \]

\[ h_A ::= A[l] : h_K \mid a[l] : h_K \]

\[ t ::= \Gamma \vdash h_t \]

\[ h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A \]

\[ l ::= \Gamma \]

\[ \Gamma ::= \cdot \mid \Gamma [x : A] \mid \Gamma [x = t] \]

\[ \Sigma ::= \cdot \mid \Sigma [c : A] \mid \Sigma [a : K] \]

- start from XLF
- isolate heads (non-binders)
- enforce \( \eta \)-long forms by annotating with heads
- factorize binders and environments
From XLF to NLF

\[ K ::= \Gamma \vdash h_k \]
\[ h_K ::= \ast \]
\[ A ::= \Gamma \vdash h_A \]
\[ h_A ::= A[l] : h_K \mid a[l] : h_K \]
\[ t ::= \Gamma \vdash h_t \]
\[ h_t ::= t[l] : h_A \mid x[l] : h_A \mid c[l] : h_A \]
\[ l ::= \Gamma \]
\[ \Gamma : x \mapsto ([x : A] \mid [x = t]) \]
\[ \Sigma ::= \cdot \mid \Sigma \left[ c : A \right] \mid \Sigma \left[ a : K \right] \]

- start from XLF
- isolate heads (non-binders)
- enforce \( \eta \)-long forms by annotating with heads
- factorize binders and environments
- abstract over environment datastructure (maps)
NLF

Syntax

\[ K ::= \Gamma \text{ kind} \]
\[ A ::= \Gamma \vdash h_A \text{ type} \]
\[ h_A ::= a \Gamma \]
\[ t ::= \Gamma \vdash h_t : h_A \]
\[ h_t ::= t \Gamma \mid x \Gamma \mid c \Gamma \]
\[ \Gamma : x \mapsto ([x : a] \mid [x = t]) \]

Judgements

- \( \Gamma \text{ kind} \)
- \( \Gamma \vdash h_A \text{ type} \)
- \( \Gamma \vdash h_t : h_A \)
NLF

Syntax

\[ K ::= \Gamma \text{ kind} \]
\[ A ::= \Gamma \vdash h_A \text{ type} \]
\[ h_A ::= a \Gamma \]
\[ t ::= \Gamma \vdash h_t : h_A \]
\[ h_t ::= t \Gamma | x \Gamma | c \Gamma \]
\[ \Gamma : x \mapsto ([x : a] | [x = t]) \]

Judgements

- \( K \)
- \( A \)
- \( t \)
NLF

Syntax

\[ K ::= \Gamma \text{ kind} \]
\[ A ::= \Gamma \vdash h_A \text{ type} \]
\[ h_A ::= a \Gamma \]
\[ t ::= \Gamma \vdash h_t : h_A \]
\[ h_t ::= t \Gamma \mid x \Gamma \mid c \Gamma \]
\[ \Gamma : x \mapsto ([x : a] \mid [x = t]) \]

Judgements

- \( K \) \text{ wf}
- \( A \) \text{ wf}
- \( t \) \text{ wf}
Notations

- "h_A" for "\( \vdash h_A \)"
- "h_A" for "\( \emptyset \vdash h_A \) type"
- "a" for "a \emptyset"

Example

\[
\lambda f^{A \rightarrow B} \cdot \lambda x^{A} \cdot f \ x : (A \rightarrow B) \rightarrow A \rightarrow B \equiv \\
[f : [a : A] \vdash B \ type] [x : A] \vdash f \ [a = x] : B
\]
Some more examples

\[ A : \emptyset \text{ kind} \]

\[ \text{vec} : \quad \lbrack \text{len} : \mathbb{N} \rbrack \text{ kind} \]

\[ \text{nil} : \quad \vdash \text{vec} \lbrack \text{len} = \vdash 0 : \mathbb{N} \rbrack \text{ type} \]

\[ \text{cons} : \quad \lbrack l : \mathbb{N} \rbrack \lbrack \text{hd} : A \rbrack \lbrack \text{tl} : \quad \vdash \text{vec} \lbrack \text{len} = \vdash l : \mathbb{N} \rbrack \text{ type} \rbrack \vdash \]
\[ \text{vec} \lbrack \text{len} = \vdash s \lbrack n = \vdash l : \mathbb{N} \rbrack : \mathbb{N} \rbrack \text{ type} \]

\[ \equiv \Pi l^\mathbb{N} \cdot A \rightarrow \Pi tl^\text{vec} l \cdot \text{vec} (s l : \mathbb{N}) : * \]

\[ \text{fill} : \quad \lbrack n : \mathbb{N} \rbrack \vdash \text{vec} \lbrack \text{len} = \vdash n : \mathbb{N} \rbrack \text{ type} \]

\[ \equiv \Pi n^\mathbb{N} \cdot (\text{vec} n : *) \]

\[ \text{empty} : \quad \lbrack e : \text{vec} \lbrack \text{len} = 0 \rbrack \rbrack \text{ kind} \]

\[ \equiv \text{vec} 0 \rightarrow * \]

\[ \lbrack e = \vdash \text{fill} \lbrack n = 0 \rbrack : \text{vec} \lbrack \text{len} = n \rbrack \rbrack \text{ type} \]

\[ \equiv \text{empty} (\text{fill} 0 : \text{vec} 0) \]
Environments

... double as labeled *directed acyclic graphs* of dependencies:

**Definition (environment)**

\[ \Gamma = (V, E) \] directed acyclic where:

- \( V \subseteq X \times (t \cup A) \) and
- \((x, y) \in E \) (\( x \) depends on \( y \)) if \( y \in FV(\Gamma(x)) \)

**Definition (lookup)**

\[ \Gamma(x) : A \] if \((x, A) \in E \)

\[ \Gamma(x) = t \] if \((x, t) \in E \)

**Definition (bind)**

\[ \Gamma [x : A] = (V \cup (x, A), E \cup \{(x, y) \mid y \in FV(A)\}) \]

\[ \Gamma [x = t] = (V \cup (x, A), E \cup \{(x, y) \mid y \in FV(t)\}) \]
Environments

... double as labeled *directed acyclic graphs* of dependencies:

**Definition (decls, defs)**

\[ \text{decls}(\Gamma) = [x_1, \ldots, x_n] \text{ s.t. } \Gamma(x_i) : A_i \text{ topologically sorted wrt. } \Gamma \]

\[ \text{defs}(\Gamma) = [x_1, \ldots, x_n] \text{ s.t. } \Gamma(x_i) = t_i \text{ topologically sorted wrt. } \Gamma \]

**Definition (merge)**

\[ \Gamma \cdot \Delta = \Gamma \cup \Delta \text{ s.t.} \]

- if \( \Gamma(x) : A \) and \( \Gamma(x) = t \) then \( \Gamma \cdot \Delta(x) = t \)
- undefined otherwise
Reduction

Definition (A-contexts)

$$\Delta^* = \{ [x = x] \mid x \in \text{defs}(\Delta) \}$$

$$\Gamma \vdash (\Delta \vdash h_t : h_A) \Xi : _- \quad \xrightarrow{\text{“} \beta \text{”}} \quad \Gamma \cdot \Delta \cdot \Xi \vdash h_t : h_A$$

$$\Gamma \vdash c \Delta : h_A \quad \longrightarrow \quad \Gamma \cdot \Delta \vdash c \Delta^* : h_A \quad \text{if } \Delta \neq \Delta^*$$

$$\Gamma \vdash c \Xi^* : a \Delta \quad \longrightarrow \quad \Gamma \cdot \Delta \vdash c \Xi^* : a \Delta^* \quad \text{if } \Delta \neq \Delta^*$$
Typing

\[
\begin{align*}
\text{FAM} & \quad \Sigma(a) : (\exists \text{ kind}) & \quad \Gamma \vdash \Delta : \exists \\
& \quad \Gamma \vdash a \Delta \text{ type} \\
\text{OBJC} & \quad \Sigma(c) : (\exists \vdash h_A \text{ type}) & \quad \Gamma \vdash \Delta : \exists \\
& \quad \Gamma \cdot \exists \cdot \Delta \vdash h'_A \equiv h_A \text{ type} \\
& \quad \Gamma \vdash c \Delta : h'_A \\
\text{OBJX} & \quad \Gamma(x) = (\exists \vdash h_t : h_A) & \quad \Gamma \vdash \Delta : \exists \\
& \quad \Gamma \cdot \exists \cdot \Delta \vdash h'_A \equiv h_A \text{ type} \\
& \quad \Gamma \cdot \exists \cdot \Delta \vdash h_t : h_A \\
& \quad \Gamma \vdash x \Delta : h'_A \\
\text{ARGS} & \quad \forall x \in \text{decls}(\exists) \\
& \quad \Delta(x) = (\Delta' \vdash h_t : h_A) & \quad \exists(x) : (\exists' \vdash h'_A \text{ type}) \\
& \quad \Gamma \cdot \Delta \cdot \Delta' \vdash h_t : h_A \\
& \quad \Gamma \cdot \exists \cdot \Delta \cdot \exists' \cdot \Delta' \vdash h'_A \equiv h_A \text{ type} \\
& \quad \Gamma \vdash \Delta : \exists
\end{align*}
\]
Properties

Translation functions

- $|·| : K_{LF} \rightarrow \Gamma_{NLF} \rightarrow K_{NLF}$ option
- $|·| : A_{LF} \rightarrow \Gamma_{NLF} \rightarrow A_{NLF}$ option
- $|·| : t_{LF} \rightarrow \Gamma_{NLF} \rightarrow t_{NLF}$ option
- ... and their inverses $|·|^{-1}$

Conjecture (Conservativity)

- $\vdash_{LF} K \text{ kind } \iff (|K|_{\emptyset}) \text{ wf}$
- $\vdash_{LF} A \text{ type } \iff (|A|_{\emptyset}) \text{ wf}$
- $\vdash_{LF} t : A \iff (|t|_{\emptyset}) \text{ wf}$
Menu

The big picture

Our approach
  Why not memoization?
  A popular storage model for repositories
  Logical framework
  Positionality

The language
  From LF to NLF
  NLF: Syntax, typing, reduction

Architecture
$ ./gasp init hol.elf
$ ./gasp init hol.elf

[holtype : kind]
[i : holtype]
[o : holtype]
[arr : [x2 : holtype][x1 : holtype] ⊩ holtype type]

Fatal error: exception Assert_failure("src/NLF.ml", 61, 13)
Checkout

$ ./gasp checkout v42

if

\[ t = \Gamma \vdash v_{52} : \text{Version} \quad \text{and} \quad \Gamma(v_{42}) = \text{Commit}(v_{41}, h) \]

then

\[ |\Gamma(h)|^{-1} \]

is the LF term representing v42
Commit

$ ./gasp commit term.elf

if

\[ t = \Gamma \vdash v_{52} : \text{Version} \quad \text{and} \quad |\text{term.elf}|_\Gamma = \Delta \vdash h_t : h_A \]

then

\[ \Delta [v_{53} = \text{Commit} [prev = v_{52}] [this = h_t]] \vdash v_{53} : \text{Version} \]

is the new repository
Further work

- still some technical & metatheoretical unknowns
- from derivations to terms (proof search? views?)
- diff on terms or derivations
- type errors handling and recovery
- mimick other operations from VCS (Merge)