A logical framework for incremental type-checking

Matthias Puech\textsuperscript{1,2}  Yann Régis-Gianas\textsuperscript{2}

\textsuperscript{1}Dept. of Computer Science, University of Bologna
\textsuperscript{2}University Paris 7, CNRS, and INRIA, PPS, team $\pi r^2$

May 2011

CEA LIST
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
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.
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
```

Ready
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 = 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

Ready
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 n  => n
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
  | O => False
  | S n => True
end.
Incrementality in programming & proof languages
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|batch) user interaction
- typed version control systems
- type-directed programming
- tactic languages
In this talk, we focus on...

... building a procedure to type-check *local changes*

- What data structure for storing type derivations?
- What language for expressing changes?
Menu

The big picture
  Incremental type-checking
  Why not memoization?

Our approach
  Two-passes type-checking
  The data-oriented way

A metalanguage of repository
  The LF logical framework
  Monadic LF
  Committing to MLF
The big picture
  Incremental type-checking
  Why not memoization?

Our approach
  Two-passes type-checking
  The data-oriented way

A metalanguage of repository
  The LF logical framework
  Monadic LF
  Committing to MLF
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

- AST representation
The big picture

- AST representation
The big picture

- AST representation
- Typing annotations
The big picture

user interaction

parsing

incremental type-checking

version management

- AST representation
- Typing annotations
A logical framework for incremental type-checking

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

A type-checker

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

- 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

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

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

\[
\begin{align*}
A \rightarrow B, B \rightarrow C, A \vdash B \rightarrow C & \quad \text{Ax} \\
A \rightarrow B, B \rightarrow C, A \vdash A \rightarrow B & \quad \text{Ax} \\
A \rightarrow B, B \rightarrow C, A \vdash A \rightarrow A & \quad \text{Ax} \\
\rightarrow e & \\
A \rightarrow B, B \rightarrow C, A \vdash B & \\
\rightarrow i & \\
A \rightarrow B, B \rightarrow C, A \vdash C & \\
\rightarrow i & \\
A \rightarrow B \vdash (B \rightarrow C) \rightarrow A \rightarrow C & \\
\vdash (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C
\end{align*}
\]
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
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!
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!

More precisely

Given a well-typed $\mathcal{R} : \text{repository}$ and a $\delta : \text{delta}$ and

$$\text{apply} : \text{repository} \rightarrow \text{delta} \rightarrow \text{derivation},$$

an incremental type-checker

$$\text{tc} : \text{repository} \rightarrow \text{delta} \rightarrow \text{bool}$$

decides if $\text{apply}(\delta, \mathcal{R})$ is well-typed in $O(|\delta|)$.

(and not $O(|\text{apply}(\delta, \mathcal{R})|)$)
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!

More precisely

Given a well-typed $R : repository$ and a $\delta : delta$ and

$$apply : repository \rightarrow delta \rightarrow derivation,$$

an incremental type-checker

$$tc : repository \rightarrow delta \rightarrow repository\; option$$

decides if $apply(\delta, R)$ is well-typed in $O(|\delta|)$.

(and not $O(|apply(\delta, R)|)$)
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!

from

\[ t \rightarrow tc \]

to

\[ s \rightarrow tc \rightarrow k' \]

\[ k \rightarrow \]
let rec check env t a =
  match t with
  | ... → ... false
  | ... → ... true

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

Syntactically

+ lightweight, efficient implementation

What if I want e.g. weakening or permutation to be taken into account?

Semantically – external to the type system (meta-cut)

What does it mean logically?

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

– imperative (introduces a dissymmetry)

Mixes two goals: derivation synthesis & object reuse
Memoization maybe?

Syntactically

+ lightweight, efficient implementation
+ *repository* = *table*, *delta* = t
Memoization maybe?

Syntactically

- lightweight, efficient implementation
- \( repository = \text{table}, \ delta = t \)
- syntactic comparison (no quotient on judgements)
  
  What if I want \( e.g. \) weakening or permutation to be taken into account?
Memoization maybe?

Syntactically

+ lightweight, efficient implementation
+ repository = table, delta = t

- syntactic comparison (no quotient on judgements)
  What if I want e.g. weakening or permutation to be taken into account?

Semantically

- external to the type system (meta-cut)
  What does it mean logically?

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

**Syntactically**

- lightweight, efficient implementation
- repository = table, delta = t
- syntactic comparison (no quotient on judgements)
  What if I want e.g. weakening or permutation to be taken into account?

**Semantically**

- external to the type system (meta-cut)
  What does it mean logically?

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

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

- imperative (introduces a dissymmetry)
Memoization maybe?

**Syntactically**

+ lightweight, efficient implementation
+ *repository* = table, *delta* = t

  - syntactic comparison (no quotient on judgements)
    
    What if I want *e.g.* weakening or permutation to be taken into account?

**Semantically**

  - external to the type system (meta-cut)
    
    What does it mean logically?

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

  - imperative (introduces a dissymmetry)

Mixes two goals: derivation synthesis & object reuse
Menu

The big picture
  Incremental type-checking
  Why not memoization?

Our approach
  Two-passes type-checking
  The data-oriented way

A metalanguage of repository
  The LF logical framework
  Monadic LF
  Committing to MLF
Two-passes type-checking

\[ \delta \] = program delta
\[ \delta_{LF} \] = derivation delta
\[ \mathcal{R} \] = repository of derivations

\[ \text{ti} = \text{type inference} = \text{derivation delta synthesis} \]
\[ \text{tc} = \text{type checking} = \text{derivation delta checking} \]
Two-passes type-checking

\[
\begin{align*}
s & \xrightarrow{\delta} ti \xrightarrow{\delta_{LF}} tc \xrightarrow{\mathcal{R}'} \\
\mathcal{R} & \xrightarrow{\delta} ti \xrightarrow{\delta_{LF}} tc
\end{align*}
\]

- \textbf{ti} = type inference = derivation delta synthesis
- \textbf{tc} = type checking = derivation delta checking
- \(\delta\) = program delta
- \(\delta_{LF}\) = derivation delta
- \(\mathcal{R}\) = repository of derivations

\textit{Shift of trust:} ti (complex, ad-hoc algorithm) \(\rightarrow\) tc (simple, generic kernel)
A popular storage model for directories

```
/v1
/15
```

```
/foo
/bar
```

```
/foo/a
/bar/b
/bar/c
```

```
```
A popular storage model for directories
A popular storage model for directories

```
/v1
/foo
/bar

/foo/a
/bar/b
/bar/c

/bar/c'
```
A popular storage model for directories
A popular storage model for directories
A popular storage model for directories

```
99dcf1d → "Hello World"
d54c809 → 46f9c2, 923ace3
6ef99a → 99dcf1d
4244e8a → 6e4f99a, d54c809
cf189a6 → 46f9c2, c328e8f
c328e8f
```
A popular storage model for directories
A popular storage model for directories

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

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

Operations

- commit $\delta$
  - extend the database with Tree/Blob objects
  - add a Commit object
  - update head

- checkout $v$
  - follow $v$ all the way to the Blobs

- diff $v_1$ $v_2$
  - follow simultaneously $v_1$ and $v_2$
  - if object names are equal, stop (content is equal)
  - otherwise continue

...
A popular storage model for directories

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

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

Invariants

- $\Delta$ forms a DAG
- 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 directories

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

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

Invariants

- $\Delta$ forms a DAG
- 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

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

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

\[
\text{v1}
\]
A *typed* repository of proofs
A *typed* repository of proofs

\[
\begin{align*}
\pi_1 &: A \land B \vdash C \\
\pi_2 &: \vdash A \\
\pi_3 &: \vdash B \\
\lambda - &: \vdash (A \land B) \rightarrow C \\
-,- &: \vdash A \land B \\
v1 &:
\end{align*}
\]
A typed repository of proofs
A *typed* repository of proofs

\[
\begin{align*}
\pi_1 &: A \land B \vdash C \\
\pi_2 &: \vdash A \\
\pi_3 &: \vdash B \\
\lambda - &: \vdash (A \land B) \rightarrow C \\
\pi_3' &: \vdash B \\
-,- &: \vdash A \land B \\
-,- &: \vdash A \land B \\
\pi_1 &: A \land B \vdash C \\
\pi_2 &: \vdash A \\
\pi_3 &: \vdash B \\
\pi_3' &: \vdash B \\
-,- &: \vdash A \land B \\
\end{align*}
\]
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

\[\begin{align*}
x &= \ldots : A \land B \vdash C \\
y &= \ldots : \vdash A \\
z &= \ldots : \vdash B \\
t &= \lambda a : A \land B \cdot x : \vdash A \land B \to C \\
u &= (y, z) : \vdash A \land B \\
v &= t \ u : \vdash C \\
w &= \text{Commit}(v, w1) : \text{Version} \ , \ w
\end{align*}\]
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

\[ x = \ldots : A \land B \vdash C \]
\[ y = \ldots : \vdash A \]
\[ z = \ldots : \vdash B \]
\[ t = \lambda a : A \land B \cdot x : \vdash A \land B \rightarrow C \]
\[ u = (y, z) : \vdash A \land B \]
\[ v = t u : \vdash C \]
\[ w = \text{Commit}(v, w1) : \text{Version} \]
\[ p = \ldots : \vdash B \]
\[ q = (y, p) : \vdash A \land B \]
\[ r = t q : \vdash C \]
\[ s = \text{Commit}(r, w) : \text{Version} \quad , \quad s \]
A data-oriented notion of delta

The first-order case

A delta is a term $t$ with variables $x, y$, defined in the repository
A data-oriented notion of delta

The binder case

A delta is a term $t$ with variables $x, y$ and boxes $[t]_{y,n}^{x/u}$ to jump over binders in the repository.
A data-oriented notion of delta

The binder case

A delta is a term $t$ with variables $x, y$ and boxes $[t]_{y,n}$ to jump over binders in the repository
Towards a metalanguage of proof repository

Repository language

1. name all proof steps
2. annotate them by judgement

Delta language

1. address sub-proofs (variables)
2. instantiate lambdas (boxes)
3. check against $\mathcal{R}$
Towards a metalanguage of proof repository

Repository language

1. name all proof steps
2. annotate them by judgement

Delta language

1. address sub-proofs (variables)
2. instantiate lambdas (boxes)
3. check against $\mathcal{R}$

$\rightsquigarrow$ Need extra-logical features!
Menu

The big picture
  Incremental type-checking
  Why not memoization?

Our approach
  Two-passes type-checking
  The data-oriented way

A metalanguage of repository
  The LF logical framework
  Monadic LF
  Committing to MLF
A logical framework for incremental type-checking

LF [Harper et al. 1992] (a.k.a. $\lambda\Pi$) provides a meta-logic to represent and validate syntax, rules and proofs of an object language, by means of a typed $\lambda$-calculus.

dependent types to express object-judgements
signature to encode the object language
higher-order abstract syntax to easily manipulate hypothesis
A logical framework for incremental type-checking

LF [Harper et al. 1992] (a.k.a. λΠ) provides a meta-logic to represent and validate syntax, rules and proofs of an object language, by means of a typed λ-calculus.

dependent types to express object-judgements
signature to encode the object language
higher-order abstract syntax to easily manipulate hypothesis

Examples

\[
\begin{align*}
[x : A] \\
\mid \quad \vdash \quad t : B \\
\hline
\lambda x \cdot t : A \rightarrow B
\end{align*}
\]

\[
\begin{align*}
[x : N] \\
\mid \quad \vdash \quad \lambda x \cdot x : N \rightarrow N
\end{align*}
\]

**is-lam :** \( \Pi A, B : ty \cdot \Pi t : tm \rightarrow tm \cdot (\Pi x : tm \cdot is \ x A \rightarrow is \ (t x) B) \rightarrow is \ (lam \ A \ (\lambda x \cdot t x))(arr \ A \ B) \)

**is-lam nat nat (\lambda x \cdot x) (\lambda yz \cdot z) :** \( is \ (lam \ nat \ (\lambda x \cdot x)) \ (arr \ nat \ nat) \)
A logical framework for incremental type-checking

Syntax

\[
\begin{align*}
K & ::= \Pi x : A \cdot K \mid * \\
A & ::= \Pi x : A \cdot A \mid a(l) \\
t & ::= \lambda x : A \cdot t \mid x(l) \mid c(l) \\
l & ::= \mathbf{\cdot} \mid t, l \\
\Sigma & ::= \mathbf{\cdot} \mid \Sigma[c : A] \mid \Sigma[a : K]
\end{align*}
\]

Judgements

- \( \Gamma \vdash_{\Sigma} K \)
- \( \Gamma \vdash_{\Sigma} A : K \)
- \( \Gamma \vdash_{\Sigma} t : A \)
- \( \vdash \Sigma \)
The delta language

Syntax

\[ K ::= \Pi x : A \cdot K | * \]
\[ A ::= \Pi x : A \cdot A | a(l) \]
\[ t ::= \lambda x : A \cdot t | x(l) | c(l) | [t]_{x.n}^{x/t} \]
\[ l ::= \cdot | t, l \]
\[ \Sigma ::= \cdot | \Sigma[c : A] | \Sigma[a : K] \]

Informally

- \( R, \Gamma \vdash \Sigma x \Rightarrow \mathcal{R} \) means “I am what \( x \) stands for, in \( \Gamma \) or in \( R \) (and produce \( \mathcal{R} \)).”
- \( R, \Gamma \vdash \Sigma [t]_{y.n}^{x/u} \Rightarrow \mathcal{R}’ \) means “Variable \( y \) has the form \( \ldots (v_1 \ldots v_{n-1}(\lambda x \cdot \mathcal{R}'') \ldots) \) in \( \mathcal{R} \). Make all variables in \( \mathcal{R}'' \) in scope for \( t \), taking \( u \) for \( x \). \( t \) will produce \( \mathcal{R}'' \)”

Judgements

- \( R, \Gamma \vdash \Sigma K \Rightarrow \mathcal{R} \)
- \( R, \Gamma \vdash \Sigma A : K \Rightarrow \mathcal{R} \)
- \( R, \Gamma \vdash \Sigma t : A \Rightarrow \mathcal{R} \)
- \( \vdash \Sigma \)
Naming of proof steps

Remark
In LF, proof step = term application spine
Example is-lam nat nat (λx · x) (λyz · z)

Monadic Normal Form (MNF)

Program transformation, IR for FP compilers
**Goal:** sequentialize all computations by naming them (lets)

\[
\begin{align*}
t & ::= λx · t \mid t(l) \mid x \\
l & ::= · \mid t, l
\end{align*}
\]

\[
\begin{align*}
t & ::= \text{ret } v \mid \text{let } x = v(l) \text{ in } t \mid v(l) \\
l & ::= · \mid v, l \\
v & ::= x \mid λx · t
\end{align*}
\]

Examples

- \( f(g(x)) \) \( \notin \) MNF
- \( λx · f(g(λy · y, x)) \) \( \implies \) 
  \( \text{ret } (λx · \text{let } a = g(λy · y, x) \text{ in } f(a)) \)
Naming of proof steps

Positionality inefficiency

\[
\begin{align*}
\text{let } x &= \ldots \text{ in} \\
\text{let } y &= \ldots \text{ in} \\
\text{let } z &= \ldots \text{ in} \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
v(l)
\end{align*}
\]
Naming of proof steps

Positionality inefficiency

\[
\begin{align*}
\text{let } x = \ldots \text{ in} \\
\text{let } y = \ldots \text{ in} \\
\text{let } z = \ldots \text{ in} \\
\vdots \\
v(l)
\end{align*}
\]

\[\begin{pmatrix}
x = \ldots \\
y = \ldots \\
z = \ldots \\
\vdots
\end{pmatrix} \vdash v(l)\]
Naming of proof steps

Positionality inefficiency

\[
\begin{align*}
\text{let } x = \ldots \text{ in } \\
\text{let } y = \ldots \text{ in } \\
\text{let } z = \ldots \text{ in } \\
\vdots \\
v(l)
\end{align*}
\]

\[
\begin{pmatrix}
x = \ldots \\
y = \ldots \\
z = \ldots \\
\vdots
\end{pmatrix} \vdash v(l)
\]

Non-positional monadic calculus

\[
\begin{align*}
t & ::= \text{ret } v \mid \text{let } x = v(l) \text{ in } t \mid v(l) \\
l & ::= \cdot \mid v, l \\
v & ::= x \mid \lambda x \cdot t
\end{align*}
\]
Naming of proof steps

Positionality inefficiency

let \( x = \ldots \) in
let \( y = \ldots \) in
let \( z = \ldots \) in
\[\vdash v(l)\]

Non-positional monadic calculus

\[t ::= \text{ret } v \mid \sigma \vdash v(l)\]
\[l ::= \cdot \mid v, l\]
\[v ::= x \mid \lambda x \cdot t\]
\[\sigma ::= \cdot \mid \sigma[x = v(l)]\]
Naming of proof steps

Positionality inefficiency

\[
\begin{align*}
\text{let } x = \ldots \text{ in} \\
\text{let } y = \ldots \text{ in} \\
\text{let } z = \ldots \text{ in} \\
\vdots \\
v(l)
\end{align*}
\]

\[
\implies \left( \begin{array}{c}
\vdash
\end{array} \right)
\]

Non-positional monadic calculus

\[
\begin{align*}
\sigma & : x \mapsto v(l) \\
\sigma & ::= \text{ret } v | \sigma \vdash v(l) \\
l & ::= \cdot | v, l \\
v & ::= x | \lambda x \cdot t \\
\end{align*}
\]
Monadic LF

\[ K ::= \Pi x : A \cdot K \mid * \]
\[ A ::= \Pi x : A \cdot A \mid \sigma \vdash a(l) \]
\[ t ::= \text{ret} \; v \mid \sigma \vdash h(l) \]
\[ h ::= x \mid c \]
\[ l ::= \cdot \mid v, l \]
\[ v ::= c \mid x \mid \lambda x : A \cdot t \]
\[ \sigma : x \mapsto h(l) \]
\[ \Sigma ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K] \]
Monadic LF

\[
\begin{align*}
K & ::= \Pi x : A \cdot K \mid * \\
A & ::= \Pi x : A \cdot A \mid \sigma \vdash a(l) \\
t & ::= \text{ret } v \mid \sigma \vdash h(l) \\
h & ::= x \mid c \\
l & ::= \cdot \mid v, l \\
v & ::= c \mid x \mid \lambda x : A \cdot t \\
\sigma & : x \mapsto h(l) \\
\Sigma & ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K]
\end{align*}
\]
Monadic LF

\[ K ::= \Pi x : A \cdot K \mid * \]
\[ A ::= \Pi x : A \cdot A \mid \sigma \vdash a(l) \]
\[ t ::= \sigma \vdash h(l) \]
\[ h ::= x \mid c \]
\[ l ::= \cdot \mid v, l \]
\[ v ::= c \mid x \mid \lambda x : A \cdot t \]
\[ \sigma ::= x \mapsto h(l) \]
\[ \Sigma ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K] \]
Type annotation

Remark
In LF, judgement annotation = type annotation

Example
is-lam nat nat \((\lambda x \cdot x) (\lambda yz \cdot z)\)
: is \((\text{lam nat } (\lambda x \cdot x)) (\text{arr nat nat})\)
Type annotation

Remark
In LF, judgement annotation = type annotation

Example
is-lam nat nat (\lambda x \cdot x) (\lambda yz \cdot z) : is (lam nat (\lambda x \cdot x)) (arr nat nat)

K ::= \Pi x : A \cdot K | *
A ::= \Pi x : A \cdot A | \sigma \vdash a(l)
t ::= \sigma \vdash h(l) : a(l)
h ::= x \mid a
l ::= \cdot \mid v, l
v ::= c \mid x \mid \lambda x : A \cdot t
\sigma ::= x \mapsto h(l) : a(l)
\Sigma ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K]
The repository language

Remark
In LF, judgement annotation = type annotation

Example
is-lam nat nat \( (\lambda x \cdot x) (\lambda y z \cdot z) \)
: is \((\text{lam nat } (\lambda x \cdot x)) \ (\text{arr nat nat})\)

\[
K ::= \prod x : A \cdot K \mid *
A ::= \prod x : A \cdot A \mid \sigma \vdash a(l)
\mathcal{R} ::= \sigma \vdash h(l) : a(l)
\]

\[
h ::= x \mid a
l ::= \cdot \mid v, l
v ::= c \mid x \mid \lambda x : A \cdot \mathcal{R}
\]

\[
\sigma : x \mapsto h(l) : a(l)
\]

\[
\Sigma ::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K]
\]
Commit (WIP)

\[ \mathcal{R}^-, \cdot - \vdash_{\Sigma^-} t^- : A^+ \Rightarrow \mathcal{R}^+ \]

What does it do?

- type-checks \( t \) wrt. \( \mathcal{R} \) (in \( O(t) \))
- puts \( t \) in non-pos. MNF
- annotate with type
- with the adapted rules for variable & box:

\[
\begin{align*}
\text{VAR} & : \quad \Gamma(x) = A \quad \text{or} \quad \sigma(x) : A \\
& \quad (\sigma \vdash \cdot : \cdot), \Gamma \vdash_{\Sigma} x : A \Rightarrow (\sigma \vdash x : A)
\end{align*}
\]

Box

\[
\begin{align*}
\sigma(x).i &= \lambda y : B \cdot (\rho \vdash H'') \\
& \quad (\sigma \vdash H), \Gamma \vdash u : B \Rightarrow (\theta \vdash H') \\
& \quad (\rho \cup \theta[y = H'] \vdash H''), \Gamma \vdash t : A \Rightarrow \mathcal{R} \\
& \quad (\sigma \vdash H), \Gamma \vdash [t]_{x.i}^{\{y/u\}} : A \Rightarrow \mathcal{R}
\end{align*}
\]
Example

Signature

\[ A \ B \ C \ D : \ast \]
\[ a : (D \rightarrow B) \rightarrow C \rightarrow A \quad b \ b' : C \rightarrow B \]
\[ c : D \rightarrow C \quad d : D \]

Terms

\[ t_1 = a(\lambda x : D \cdot b(c(x)), c(d)) \]
\[ \mathcal{R}_1 = [v = c(d) : C] \vdash a(\lambda x : D \cdot [w = c(x) : C] \vdash b(w) : B, v) : A \]
\[ t_2 = a(\lambda y : D \cdot [b'(w)]_{1}^{\{x/y\}}) \]
\[ \mathcal{R}_2 = [v = c(d) : C] \vdash a(\lambda y : D \cdot [x = y][w = c(x) : C] \vdash b'(w) : B, v) : A \]
Regaining version management

Just add to the signature $\Sigma$:

$$\begin{align*}
\text{Version} &: \ast \\
\text{Commit0} &: \text{Version} \\
\text{Commit} &: \Pi t : \text{tm} \cdot \text{is}(t, \text{unit}) \rightarrow \text{Version} \rightarrow \text{Version}
\end{align*}$$

Commit $t$

if $\mathcal{R} = \sigma \vdash v : \text{Version}$ and $\mathcal{R}, \cdot \vdash \Sigma t : \text{is}(p, \text{unit}) \Rightarrow (\rho \vdash k)$

then

$$\rho[x = \text{Commit}(p, k, v)] \vdash x : \text{Version}$$

is the new repository
Further work

- implementation & metatheory of Commit
- from terms to derivations (ti)
- diff on terms
- mimick other operations from VCS (Merge)