A logical framework for incremental type-checking

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

\textsuperscript{1}Dept. of Computer Science, University of Bologna

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

June 2011

INRIA – Gallium
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?
Incrementality in programming & proof languages

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 global rollback (Coq)
Incrementality in programming & proof languages

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 global rollback (*Coq*)

... ad-hoc tools, code duplication, hacks...

Examples
- `diff`’s language-specific options, lines of context...
- `git`’s merge heuristics
- `ocamldep` *vs.* `ocaml` module system
- `coqtop`’s rigidity
In an ideal world... 

- Edition should be incrementally communicated to the tool
- The impact of changes visible “in real time”
- No need for separate compilation, dependency management...
In an ideal world...

- Edition should be incrementally communicated to the tool
- 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 incrementally communicated to the tool 
- 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
   Tools
      The LF logical framework
      Monadic LF
   Typing by annotating
   The typing/committing process
      What does it do?
      Example
      Regaining version management
Menu

The big picture
  Incremental type-checking
  Why not memoization?

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

A metalanguage of repository
  Tools
    The LF logical framework
    Monadic LF
  Typing by annotating
  The typing/committing process
    What does it do?
    Example
    Regaining version management
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

- AST representation
The big picture

- AST representation
The big picture

- AST representation
The big picture

- AST representation
- Typing annotations
The big picture

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

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

A type-checker

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

- builds and checks the derivation (on the stack)
- conscientiously discards it
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 $\mathcal{R} : repository$ and a $\delta : delta$ and

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

an incremental type-checker

$$tc : repository \rightarrow delta \rightarrow bool$$

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

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

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

an incremental type-checker

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

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

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

```

to

```
| s | tc | \( \mathcal{R} \) |
```

\( \mathcal{R} \)
let rec check env t a =
mismatch t with
| ... → ... false
| ... → ... true

and infer env t =
mismatch 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
Memoization maybe?

Syntactically

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

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

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

− 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?

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

- 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
  Tools
    The LF logical framework
    Monadic LF
  Typing by annotating
  The typing/committing process
    What does it do?
    Example
    Regaining version management
Two-passes type-checking

\[ s \xrightarrow{\delta_{LF}} ti \xrightarrow{\delta} tc \xrightarrow{R'} R \]

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

\[ s \rightarrow ti \overset{\delta_{LF}}{\rightarrow} tc \rightarrow R' \]

\( ti = \text{type inference} = \text{derivation delta synthesis} \)

\( tc = \text{type checking} = \text{derivation delta checking} \)

\( \delta = \text{program delta} \)

\( \delta_{LF} = \text{derivation delta} \)

\( R = \text{repository of derivations} \)

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

```
/v1
/foobar
/fooa /barb /barc
/
/bar
/barc'
```
A popular storage model for directories

```
/v1
/foo
/bar
/foo/a
/bar/b
/bar/c
/
/bar
/bar/c'
/v2
```

```
/v1
/foo
/bar
/foo/a
/bar/b
/bar/c
/
/bar
/bar/c'
/v2
```
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
...```
A popular storage model for directories

```
<table>
<thead>
<tr>
<th>99dcf1d  → &quot;Hello World&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>d54c809  → 46f9c2, 923ace3</td>
</tr>
<tr>
<td>6ef99a  → 99dcf1d</td>
</tr>
<tr>
<td>4244e8a → 6e4f99a, d54c809</td>
</tr>
<tr>
<td>cf189a6 → 46f9c2, c328e8f</td>
</tr>
</tbody>
</table>
```
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

\[
\begin{array}{c}
\pi_1 : A \land B \vdash C \\
\pi_2 : \vdash A \\
\pi_3 : \vdash B
\end{array}
\]

\[
\begin{array}{c}
\lambda : \vdash (A \land B) \to C \\
\_\_ : \vdash A \land B
\end{array}
\]

\[
\_\_ : \vdash C
\]
A typed repository of proofs

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

\[ \lambda - : \vdash (A \land B) \to C \quad \land_2 : \vdash A \land B \quad \pi_1 : A \land B \vdash C \quad \pi_2 : \vdash A \quad \pi_3' : \vdash B \]

\[ \land_1 : \vdash C \quad \lambda - : \vdash (A \land B) \to C \quad \land_2 : \vdash A \land B \]

\[ \land_1 : \vdash C \]
A typed repository of proofs
A *typed* repository of proofs

\[
\begin{align*}
\pi_1 &: A \land B \vdash C \\
\pi_2 &: \vdash A \\
\pi_3 &: \vdash B \\
\lambda &: \vdash (A \land B) \to C \\
\pi_1 &: A \land B \\
\pi_2 &: A \land B \\
\pi_3 &: A \land B \\
\pi_3' &: A \land B \\
\vdash &: C \\
v_1 &: C \\
v_2 &: C
\end{align*}
\]
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

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

, 

\[w\]
A typed repository of proofs

\[ x = \ldots : A \land B \vdash C \]
\[ y = \ldots : \vdash A \]
\[ z = \ldots : \vdash B \]
\[ t = \lambda a : A \land B \cdot x : \vdash A \land B \rightarrow C \]
\[ u = (y, z) : \vdash A \land B \]
\[ v = t \; u : \vdash C \]
\[ w = \text{Commit}(v, w_1) : \text{Version} \]
\[ p = \ldots : \vdash B \]
\[ q = (y, p) : \vdash A \land B \]
\[ r = t \; q : \vdash C \]
\[ s = \text{Commit}(r, w) : \text{Version} \]
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 data-oriented notion of delta

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

- A delta is a term $t$ with variables $x, y$, defined in the repository
- A repository $\mathcal{R}$ is a flattened, annotated term with a head
A data-oriented notion of delta

- A delta is a term $t$ with variables $x, y$, defined in the repository
- A repository $R$ is a flattened, annotated term with a head
- Incrementality by sharing common subterms

$\pi$, $\pi'$, $\delta(\pi, \pi')$
A data-oriented notion of delta

- A delta is a term $t$ with variables $x, y$, defined in the repository
- A repository $\mathcal{R}$ is a flattened, annotated term with a head
- Incrementality by sharing common subterms

![Diagram of delta and invariants](image)

Invariants

- $\mathcal{R}$ forms a DAG
- Annotations are valid wrt. proof rules
Higher-order notion of delta

Problem
Proofs are higher-order objects by nature:

Example

We can’t allow sharing in \( \vdash t : B \) without instantiating \( \vdash x : A \) (scope escape)
Higher-order notion of delta

Solutions

- “first-orderize” your logic (de Bruijn indices, $\Gamma$ is a list...) 
  + we’re done 
  - weakening, permutation, substitution etc. become explicit operations 
  - delta application possibly has to rewrite the repository (lift) 
  - dull dull dull...

- “let meta handle it” (the delta language) 
  + known technique (HOAS) 
  + implicit environments = weakening, permutation, substitution for free 
  - have to add an instantiation operator
Higher-order notion of delta

Solution

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

Repository language

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

Delta language

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

The big picture
  Incremental type-checking
  Why not memoization?

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

A metalanguage of repository
  Tools
    The LF logical framework
    Monadic LF
  Typing by annotating
  The typing/committing process
    What does it do?
    Example
    Regaining version management
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
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.

**Examples**

1. \[ [x : A] \]
   \[
   \begin{array}{c}
   \vdots \\
   t : B \\
   \lambda x \cdot t : A \to B
   \end{array}
   \]
   \[ \leadsto \]

   \( \text{is-lam : } \; \Pi A, B : \text{ty} \cdot \Pi t : \text{tm} \to \text{tm} \cdot \\
   (\Pi x : \text{tm} \cdot \text{is } x \; A \to \text{is } (t \; x) \; B) \to \\
   \text{is } (\text{lam } A (\lambda x \cdot t \; x))(\text{arr } A \; B) \)

2. \[ [x : N] \]
   \[
   \begin{array}{c}
   \vdots \\
   \lambda x \cdot x : N \to N
   \end{array}
   \]

   \[ \leadsto \]

   \( \text{is-lam nat nat } (\lambda x \cdot x) (\lambda y z \cdot z) \\
   : \text{is } (\text{lam nat } (\lambda x \cdot x))(\text{arr nat nat}) \)
A logical framework for incremental type-checking

Syntax

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

\[ A ::= \Pi x : A \cdot A \mid a(l) \]

\[ t ::= \lambda x \cdot t \mid x(l) \mid c(l) \]

\[ l ::= \cdot \mid t, l \]

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

Judgements

- \( \Gamma \vdash \Sigma K \)
- \( \Gamma \vdash \Sigma A \)
- \( \Gamma \vdash \Sigma t : A \)
- \( \Gamma, A \vdash \Sigma l : A \)
- \( \vdash \Sigma \)

\[
\begin{align*}
\Gamma \vdash t : A \quad \Gamma, B[x/t] \vdash l : B \\
\hline
\Gamma, \Pi x : A \cdot B \vdash t, l : C
\end{align*}
\]

Remarks

- the spine-form, canonical flavor (\( \beta \) and \( \eta \)-long normal)
- substitution is hereditary (i.e. cut-admissibility / big-step reduction)
Naming of proof steps

Remark
In LF, proof step = term application spine

Example is-lam nat nat (\(x \cdot x\)) (\(yz \cdot z\))
Naming of proof steps

Remark
In LF, proof step = term application spine
Example is-lam nat nat \((\lambda x \cdot x) \ (\lambda y z \cdot z)\)

Monadic Normal Form (MNF)

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


t ::= \lambda x \cdot t \mid t(l) \mid x \quad \Rightarrow \quad t ::= \text{ret } v \mid \text{let } x = v(l) \text{ in } t \mid v(l)

l ::= \cdot \mid t, l

v ::= x \mid \lambda x \cdot t

Examples

- \(f(g(x)) \notin \text{MNF}\)
- \(\lambda x \cdot f(g(\lambda y \cdot y, x)) \quad \Rightarrow \quad \text{ret } (\lambda x \cdot \text{let } a = g(\lambda y \cdot y, x) \text{ in } f(a))\)
Naming of proof steps

Positionality inefficiency

Order of lets is irrelevant, we just want non-cyclicity and fast access.

\[
\begin{align*}
\text{let } x & = \ldots \text{ in} \\
\text{let } y & = \ldots \text{ in} \\
\text{let } z & = \ldots \text{ in} \\
\vdots
\end{align*}
\]

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

Order of \texttt{let}s is irrelevant, we just want non-cyclicity and fast access.

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

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

Non-positional monadic calculus

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

Positionality inefficiency

Order of lets is irrelevant, we just want non-cyclicity and fast access.

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

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

Non-positional monadic calculus

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

Positionality inefficiency

Order of lets is irrelevant, we just want non-cyclicity and fast access.

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

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

Non-positional monadic calculus

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

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

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

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

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

Definition

\[ \ast : LF \to \text{monadic LF} \]

One-pass, direct style version of [Danvy 2003]
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)
Type annotation

Remark
In LF, judgement annotation = type annotation

Example

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

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

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)

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

Remark
In LF, judgement annotation = type annotation

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

\[\begin{align*}
K & ::= \Pi x : A \cdot K | \ast \\
A & ::= \Pi x : A \cdot A | \sigma \vdash a(\_)

\mathcal{R} & ::= \sigma \vdash v : a(\_)
\end{align*}\]

←− σ DAG, binds in \(v\) and \(_\)

\[\begin{align*}
h & ::= x | a \\
l & ::= \cdot | v, l \\
v & ::= c | x | \lambda x : A \cdot \mathcal{R}

\sigma & \colon x \mapsto h(l) : a(\_)

\Sigma & ::= \cdot | \Sigma[c : A] | \Sigma[a : K]
\end{align*}\]
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 (lam nat (\lambda x \cdot x)) (arr nat nat)

\[
\begin{align*}
K & ::= \Pi x: A \cdot K \mid * \\
A & ::= \Pi x: A \cdot A \mid \sigma \vdash a(l) \\
\mathcal{R} & ::= \sigma \vdash v: a(l) \quad \leftarrow \sigma \text{ DAG, binds in } v \text{ and } l \\
h & ::= x \mid a \\
l & ::= \cdot \mid v, l \\
v & ::= c \mid x \mid \lambda x: A \cdot \mathcal{R} \\
\sigma & : x \mapsto h(l) : a(l) \quad \leftarrow \text{ named implementation} \\
\Sigma & ::= \cdot \mid \Sigma[c: A] \mid \Sigma[a: K]
\end{align*}
\]
The delta language

Syntax

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

Judgements

\[ \mathcal{R}, \Gamma \vdash K \rightarrow K \]
\[ \mathcal{R}, \Gamma \vdash A \rightarrow A \]
\[ \mathcal{R}, \Gamma \vdash t : A \rightarrow t \]
\[ \mathcal{R}, \Gamma, A \vdash l \rightarrow l : A \]
\[ \vdash \Sigma \rightarrow \Sigma \]

Informally

\[ \mathcal{R}, \Gamma \vdash_\Sigma x \Rightarrow \mathcal{R} \text{ means} \]
“I am what \( x \) stands for, in \( \Gamma \) or in \( \mathcal{R} \) (and produce \( \mathcal{R} \))”.

\[ \mathcal{R}, \Gamma \vdash_\Sigma [t]^u_{y.n} \Rightarrow \mathcal{R}' \text{ means} \]
“Variable \( y \) has the form \( c(v_1 \ldots v_{n-1}(\lambda x \cdot \mathcal{R}'') \ldots) \) in \( \mathcal{R} \).
Make all variables in \( \mathcal{R}'' \) in scope for \( t \), taking \( u \) for \( x \).
In this new scope, \( t \) will produce \( \mathcal{R}' \)”
The typing/committing process

\[ \mathcal{R}, \Gamma \vdash t : A \rightarrow t \]

What does it do?

- puts \( t \) in non-pos. MNF \((O(t))\)
- type-checks \( t \) wrt. \( \mathcal{R} \) and
- returns \( t \) i.e. \( t \) annotated with types \((O(t))\)
Typing by annotating

**partial translation**: monadic LF $\rightarrow$ annotated monadic LF

\[
\frac{
\mathcal{V}_{LAM} 
}{
\frac{
\mathcal{R}, \Gamma[x : A] \vdash t : B \rightarrow t 
}{
\mathcal{R}, \Gamma \vdash \lambda x \cdot t : \Pi x : A \cdot B \rightarrow \lambda x : A \cdot t
}}
\]
Typing by annotating

**partial translation**: monadic LF $\rightarrow$ annotated monadic LF

\[
\text{HVAR} \\
\frac{\Gamma(x) : A \quad \text{or} \quad \sigma(x) : A}{(\sigma \vdash v), \Gamma \vdash x \rightarrow A}
\]
Typing by annotating

**partial translation**: monadic LF \(\rightarrow\) annotated monadic LF

\[
\text{LCons} \\
\frac{\mathcal{R}, \Gamma \vdash v : A \rightarrow v}{\mathcal{R}, \Gamma, B[x/v] \vdash l \rightarrow l : a(l)} \\
\frac{\mathcal{R}, \Gamma, \Pi x : A \cdot B \vdash v, l \rightarrow v, l : a(l)}
\]
Typing by annotating

**partial translation**: monadic LF $\rightarrow$ annotated monadic LF

\[
\begin{align*}
\text{OBox} \\
\mathcal{R}|_p &= \lambda x : B \cdot \mathcal{R}' \\
\mathcal{R}, \Gamma \vdash u : B \rightarrow (\sigma \vdash h : a(l)) \\
\mathcal{R}' \cup \sigma[x = h : a(l)], \Gamma \vdash t : A \rightarrow t \\
\hline
\mathcal{R}, \Gamma \vdash [t]_p^u : A \rightarrow t
\end{align*}
\]
Typing by annotating

**partial translation**: monadic LF → annotated monadic LF

\[(\Pi x : A \cdot B)[z/v] = \Pi x : A[z/v] \cdot B[z/v] \]

\[(\sigma \vdash a(l))[z/v] = \sigma[z/v] \vdash a(l[z/v]) \]

\[(\sigma[y = x(l) : a(m)])[z/v] = (\sigma[z/v])[y = x(l[z/v]) : a(m[z/v])] \]

\[(\sigma[y = z(l) : a(m)])[z/v] = \text{red}_\sigma^y(v, l) \]

\[\text{red}_\sigma^y(h : a(l), \cdot) = \sigma[y = h : a(l)] \]

\[\text{red}_\sigma^y(\lambda x : A \cdot t, (v, l)) = \text{red}_{\sigma \cup \rho}^y(w, l) \quad \text{if} \quad t[x/v] = (\rho \vdash w) \]

\[
\vdots
\]
Properties of the translation

Work in progress...

Theorem (Soundness)

\[ \text{if } \Gamma \vdash t : A \text{ then } \vdash \Gamma^* \rightarrow \Gamma \text{ and } (\cdot \vdash \cdot), \Gamma \vdash A^* \rightarrow A \text{ and } (\cdot \vdash \cdot), \Gamma \vdash t^* : A \rightarrow t \]

Definition (Checkout)

Let \( \cdot^- \) be the back-translation function of a repository into an LF term.

Theorem (Completeness)

\[ \text{if } (\cdot \vdash \cdot), \Gamma \vdash t^* : A \rightarrow t \text{ then } \Gamma^- \vdash t^- : A^- \]

Theorem (Substitution)

\[ \text{If } \mathcal{R}, \Gamma \vdash u : B \rightarrow (\sigma \vdash y : B) \text{ and } \Gamma^-[x : B] \Delta^- \vdash t : A \text{ then } (\sigma \vdash y : B), \Gamma\Delta\{x/y\} \vdash t\{x/y\} : B\{x/y\} \rightarrow \mathcal{R}' \]
Example

Signature

\[ A \quad B \quad C \quad D : \ast \]

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

Terms

\[ t_1 = a(\lambda x \cdot b(c(x)), c(d)) \]
\[ \mathcal{R}_1 = [v = c(d) : C] \]
\[ [u = a(\lambda x : D \cdot [w = c(x) : C][w' = b(w) : B] \vdash w' : B, v) : A] \]
\[ \vdash u : A \]

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

Just add to the signature $\Sigma$:

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

\textbf{Commit $t$}

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

then

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

is the new repository
Further work

- metatheory of annotated monadic LF
- from terms to derivations (ti)
- diff on terms
- mimick other operations from VCS (Merge)
Further work

- metatheory of annotated monadic LF
- from terms to derivations (ti)
- diff on terms
- mimick other operations from VCS (Merge)

Thank you!