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 πr^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

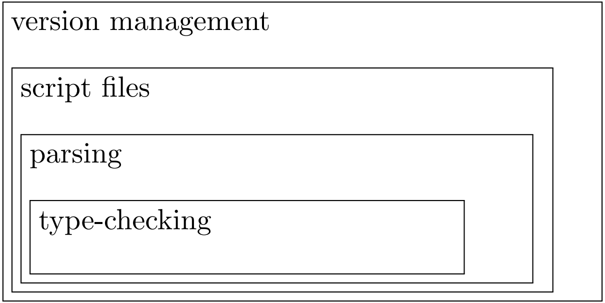
Menu

The big picture
- Incremental type-checking
- Why not memoization?

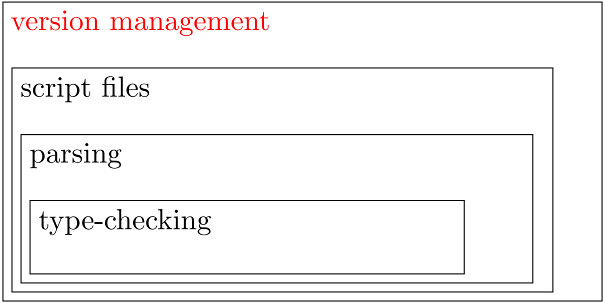
Our approach
- Two-passes type-checking
- The data-oriented way

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

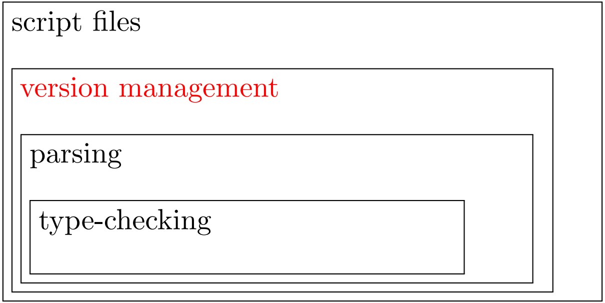
The big picture



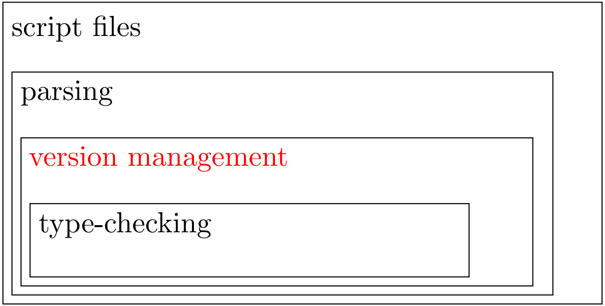
The big picture



The big picture

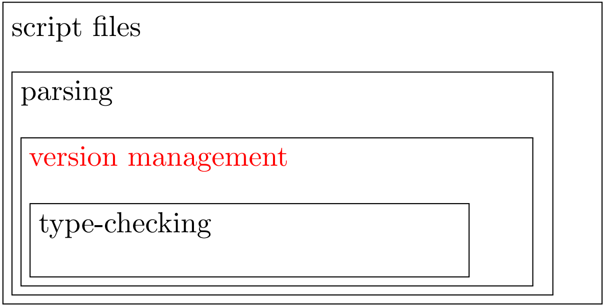


The big picture



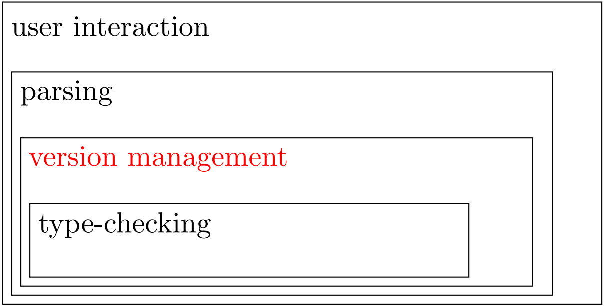
- AST representation

The big picture



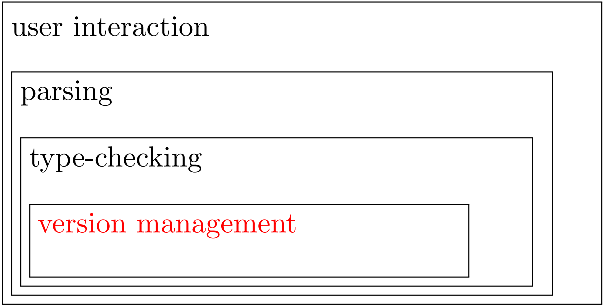
- AST representation

The big picture



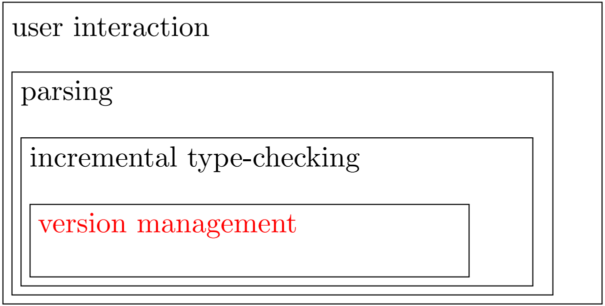
- AST representation

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

```
val check : env → term → types → 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 $\mathcal{R} : \mathit{repository}$ and a $\delta : \mathit{delta}$ and

$$\mathsf{apply} : \mathit{repository} \rightarrow \mathit{delta} \rightarrow \mathit{derivation} ,$$

an incremental type-checker

$$\mathsf{tc} : \mathit{repository} \rightarrow \mathit{delta} \rightarrow \mathit{bool}$$

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

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

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 $\mathcal{R} : \mathit{repository}$ and a $\delta : \mathit{delta}$ and

$$\mathsf{apply} : \mathit{repository} \to \mathit{delta} \to \mathit{derivation}\ ,$$

an incremental type-checker

$$\mathsf{tc} : \mathit{repository} \to \mathit{delta} \to \mathit{repository\ option}$$

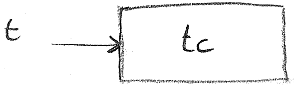
decides if $\mathsf{apply}(\delta, \mathcal{R})$ is well-typed in $O(|\delta|)$.
(and not $O(|\mathsf{apply}(\delta, \mathcal{R})|)$)

A logical framework for incremental type-checking

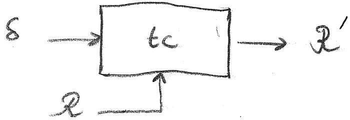
Goal Type-check a large derivation taking advantage of the knowledge from type-checking previous versions

Idea Remember all derivations!

from



to



Memoization maybe?

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

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

Memoization maybe?

```
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
- + $repository = \mathsf{table}$, $delta = \mathsf{t}$

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
- \+ $repository = \mathsf{table}$, $delta = \mathsf{t}$
- − 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?

$$\frac{J \in \Gamma}{\Gamma \vdash J \ \mathsf{wf} \Rightarrow \Gamma} \qquad \frac{\Gamma_1 \vdash J_1 \ \mathsf{wf} \Rightarrow \Gamma_2 \qquad \dots \qquad \Gamma_{n-1}[J_{n-1}] \vdash J_n \ \mathsf{wf} \Rightarrow \Gamma_n}{\Gamma_1 \vdash J \ \mathsf{wf} \Rightarrow \Gamma_n[J_n][J]}$$

Memoization maybe?

Syntactically

- $+$ lightweight, efficient implementation
- $+$ $\mathit{repository} = \mathsf{table}$, $\mathit{delta} = \mathsf{t}$
- $-$ 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?

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

- $-$ 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?

$$\frac{J \in \Gamma}{\Gamma \vdash J \ \mathsf{wf} \Rightarrow \Gamma} \qquad \frac{\Gamma_1 \vdash J_1 \ \mathsf{wf} \Rightarrow \Gamma_2 \qquad \dots \qquad \Gamma_{n-1}[J_{n-1}] \vdash J_n \ \mathsf{wf} \Rightarrow \Gamma_n}{\Gamma_1 \vdash J \ \mathsf{wf} \Rightarrow \Gamma_n[J_n][J]}$$

− imperative (introduces a dissymmetry)

Mixes two goals: *derivation synthesis* & *object reuse*

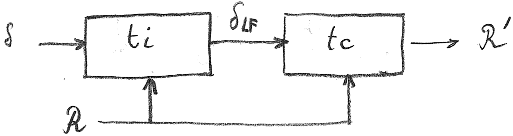
Menu

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



ti = type inference = derivation delta synthesis

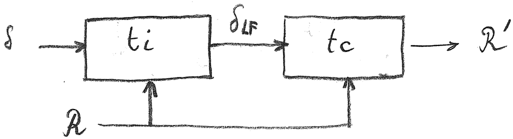
tc = type checking = derivation delta checking

δ = program delta

δ_{LF} = derivation delta

$\mathcal{R}$ = repository of derivations

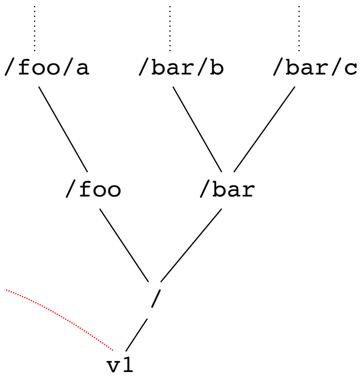
Two-passes type-checking



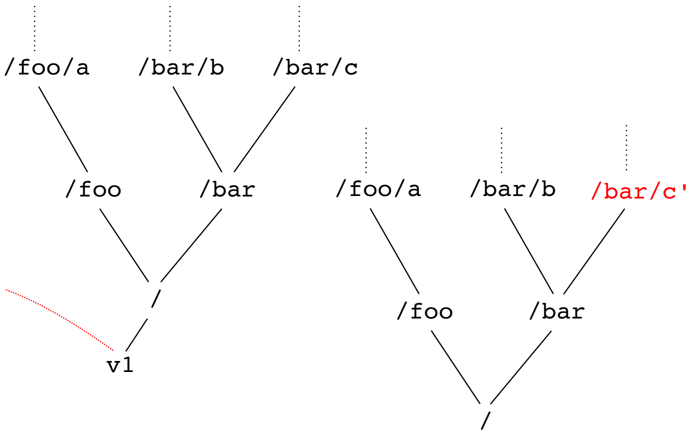
$$\begin{aligned} \mathsf{ti} &= \text{type inference} = \text{derivation delta synthesis} \\ \mathsf{tc} &= \text{type checking} = \text{derivation delta checking} \\ \delta &= \text{program delta} \\ \delta_{LF} &= \text{derivation delta} \\ \mathcal{R} &= \text{repository of derivations} \end{aligned}$$

Shift of trust: ti (complex, ad-hoc algorithm) $\rightarrow$ tc (simple, generic kernel)

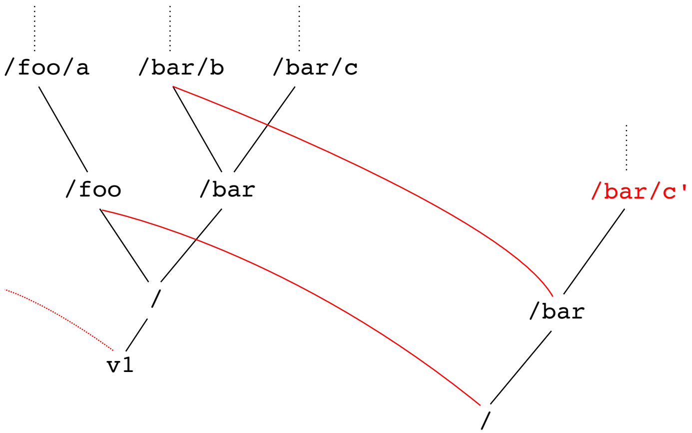
A popular storage model for directories



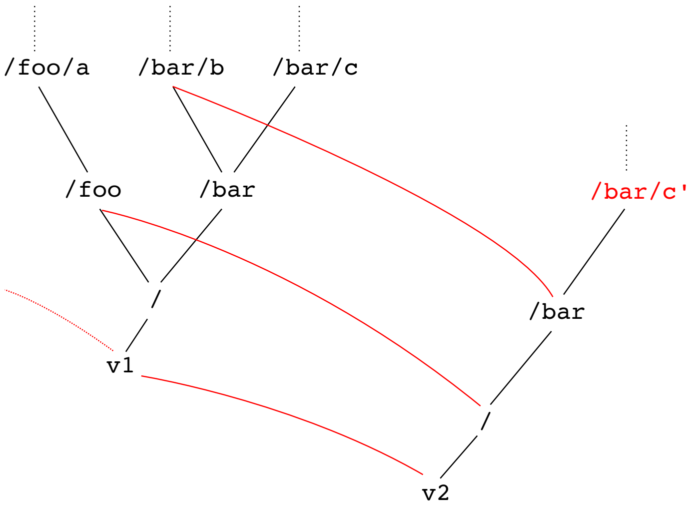
A popular storage model for directories



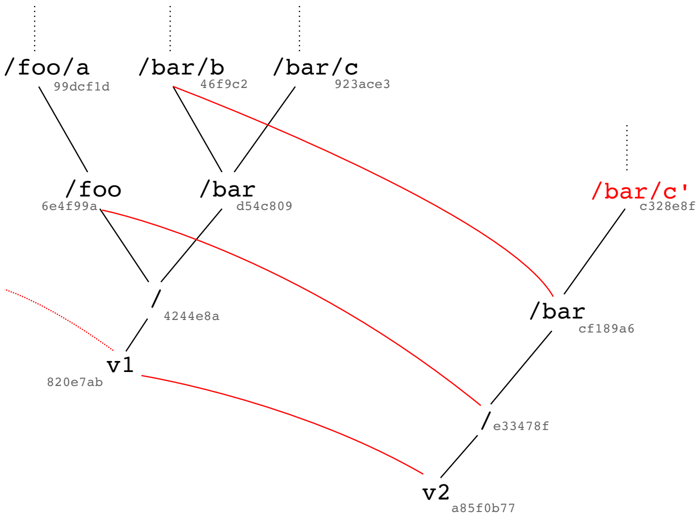
A popular storage model for directories



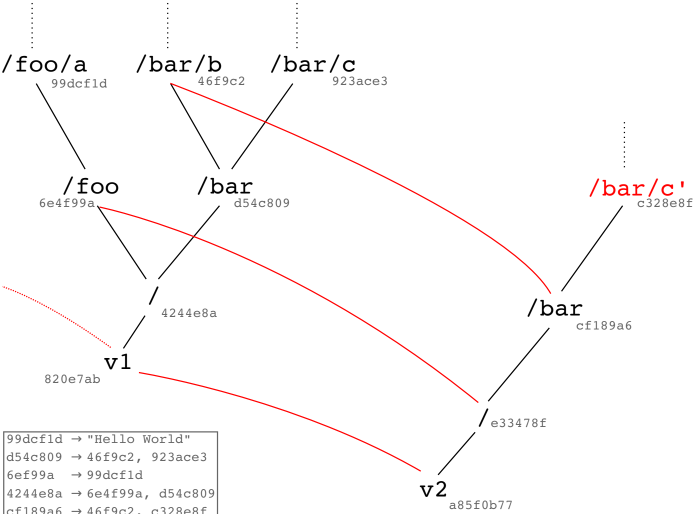
A popular storage model for directories



A popular storage model for directories

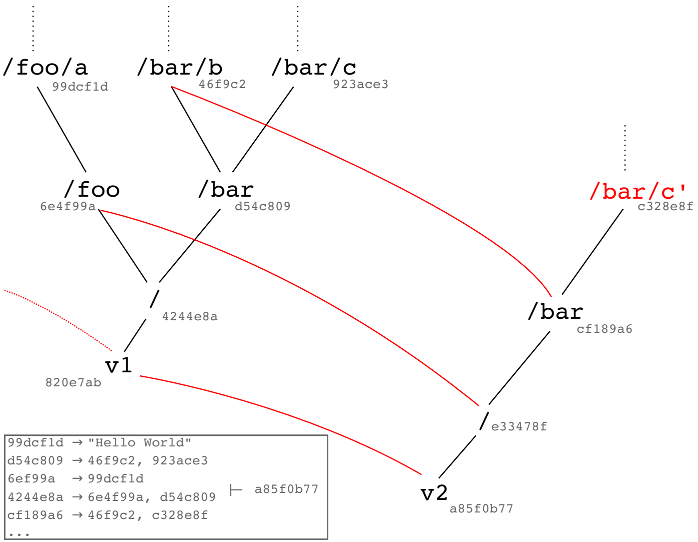


A popular storage model for directories




```
99dcf1d  → "Hello World"
d54c809  → 46f9c2, 923ace3
6ef99a   → 99dcf1d
4244e8a  → 6e4f99a, d54c809
cf189a6  → 46f9c2, c328e8f
...
```

A popular storage model for directories



A popular storage model for directories

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

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

Operations

`commit` δ
- 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 (Δ, x):

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

Invariants

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

A popular storage model for directories

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

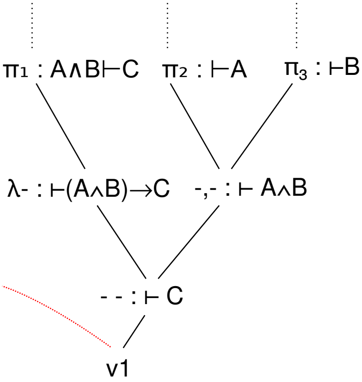
$$\Delta : x \mapsto (\mathsf{Commit}\ (x \times y) \mid \mathsf{Tree}\ \vec{x} \mid \mathsf{Blob}\ \mathit{string})$$

Invariants

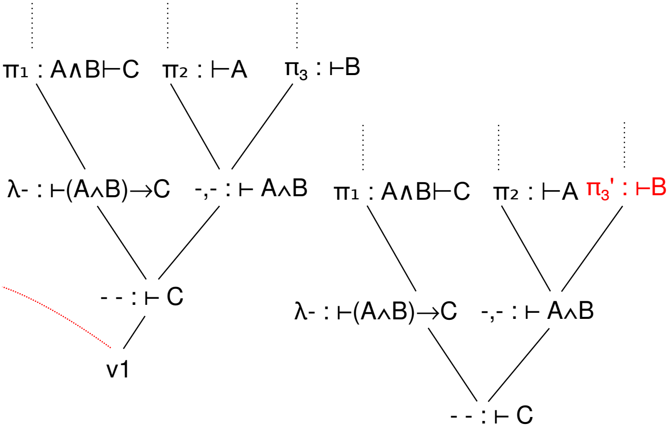
- Δ forms a DAG
- if $(x, \mathsf{Commit}\ (y, z)) \in \Delta$ then
  - $(y, \mathsf{Tree}\ t) \in \Delta$
  - $(z, \mathsf{Commit}\ (t, v)) \in \Delta$
- if $(x, \mathsf{Tree}(\vec{y})) \in \Delta$ then
  for all y_i, either $(y_i, \mathsf{Tree}(\vec{z}))$ or $(y_i, \mathsf{Blob}(s)) \in \Delta$

Let's do the same with *proofs*

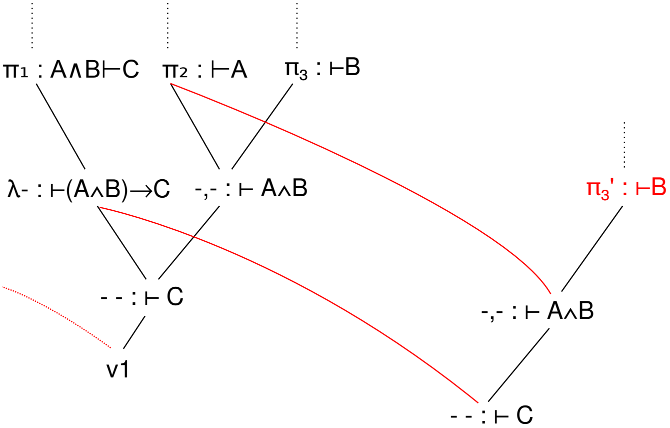
A *typed* repository of proofs



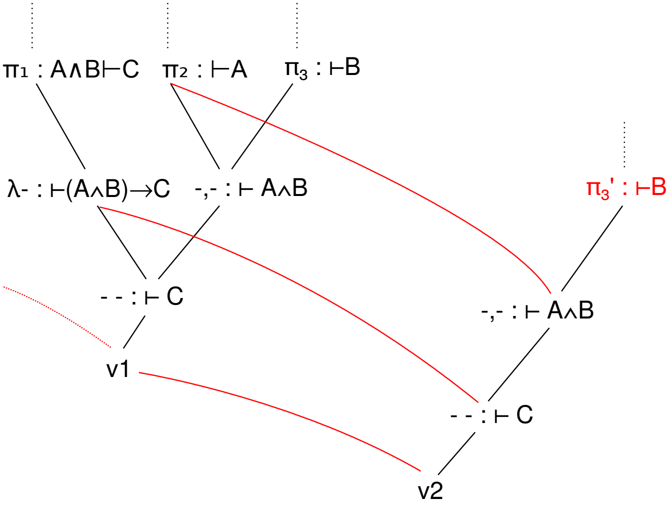
A *typed* repository of proofs



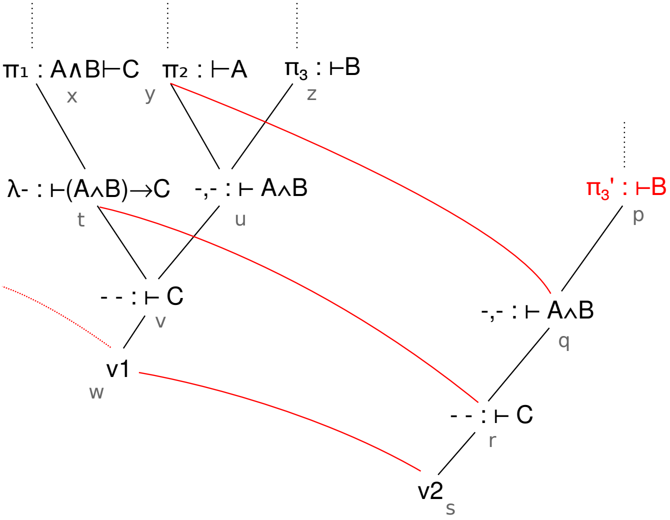
A *typed* repository of proofs



A *typed* repository of proofs



A *typed* repository of proofs



A *typed* repository of proofs

$$
\begin{aligned}
x &= \ldots \quad : A \wedge B \vdash C \\
y &= \ldots \quad : \vdash A \\
z &= \ldots \quad : \vdash B \\
t &= \lambda a : A \wedge B \cdot x : \vdash A \wedge B \to C \\
u &= (y, z) : \vdash A \wedge B \\
v &= t\ u : \vdash C \\
w &= \textsf{Commit}(v, w1) : \textsf{Version}
\end{aligned}
$$

A *typed* repository of proofs

$$
\begin{aligned}
x &= \dots \;\; : A \wedge B \vdash C \\
y &= \dots \;\; : \vdash A \\
z &= \dots \;\; : \vdash B \\
t &= \lambda a : A \wedge B \cdot x : \vdash A \wedge B \to C \\
u &= (y, z) : \vdash A \wedge B \\
v &= t\ u : \vdash C \\
w &= \mathsf{Commit}(v, w1) : \mathsf{Version} \qquad , \quad w
\end{aligned}
$$

A *typed* repository of proofs

$$
\begin{aligned}
x &= \ldots \;: A \wedge B \vdash C \\
y &= \ldots \;: \vdash A \\
z &= \ldots \;: \vdash B \\
t &= \lambda a : A \wedge B \cdot x : \vdash A \wedge B \to C \\
u &= (y, z) : \vdash A \wedge B \\
v &= t\ u : \vdash C \\
w &= \mathsf{Commit}(v, w1) : \mathsf{Version} \\
p &= \ldots \;: \vdash B \\
q &= (y, p) : \vdash A \wedge B \\
r &= t\ q : \vdash C \\
s &= \mathsf{Commit}(r, w) : \mathsf{Version}
\end{aligned}
$$

A *typed* repository of proofs

$$
\begin{aligned}
x &= \dots \ : A \wedge B \vdash C \\
y &= \dots \ : \vdash A \\
z &= \dots \ : \vdash B \\
t &= \lambda a : A \wedge B \cdot x : \vdash A \wedge B \to C \\
u &= (y, z) : \vdash A \wedge B \\
v &= t\ u : \vdash C \\
w &= \mathsf{Commit}(v, w1) : \mathsf{Version} \\
p &= \dots \ : \vdash B \\
q &= (y, p) : \vdash A \wedge B \\
r &= t\ q : \vdash C \\
s &= \mathsf{Commit}(r, w) : \mathsf{Version} \qquad , \quad s
\end{aligned}
$$

A data-oriented notion of delta

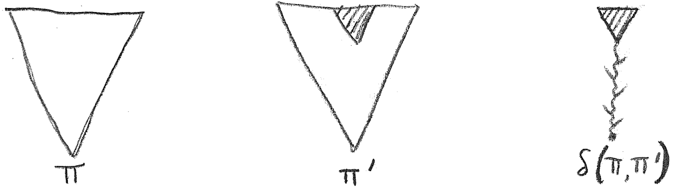
- A delta is a term t with *variables* x, y, defined in the repository

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*

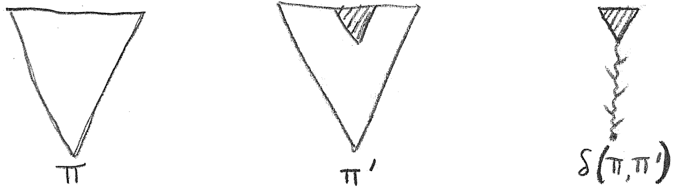
A data-oriented notion of delta

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



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



Invariants

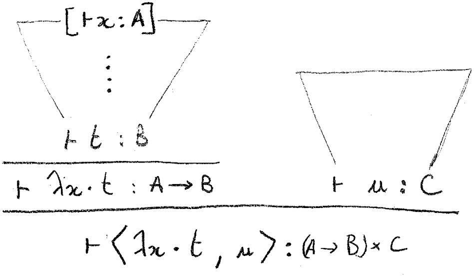
- $\mathcal{R}$ forms a DAG
- Annotations are valid *wrt.* proof rules

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

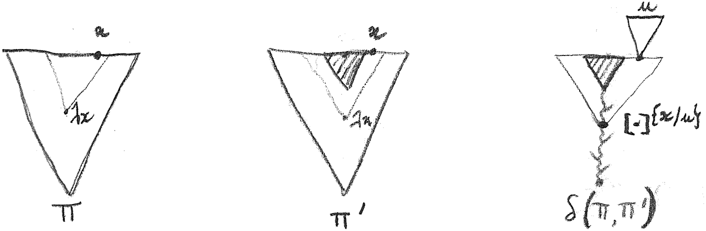
Solutions

- "first-orderize" your logic (de Bruijn indices, Γ 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 $[t]^{u}_{y.n}$ to jump over lambdas in the repository

Towards a metalanguage of proof repository

Repository language

1. name all proof steps
2. annote 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. $\lambda\Pi$) 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. $\lambda\Pi$) 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

- $$\dfrac{\begin{array}{c}[x : A] \\ \vdots \\ t : B\end{array}}{\lambda x \cdot t : A \to B} \quad \rightsquigarrow \quad \begin{array}{ll}\mathsf{is\text{-}lam} : & \Pi A, B : \mathsf{ty} \cdot \Pi t : \mathsf{tm} \to \mathsf{tm} \cdot \\ & (\Pi x : \mathsf{tm} \cdot \mathsf{is}\ x\ A \to \mathsf{is}\ (t\ x)\ B) \to \\ & \mathsf{is}\ (\mathsf{lam}\ A\ (\lambda x \cdot t\ x))(\mathsf{arr}\ A\ B)\end{array}$$
- $$\dfrac{[x : \mathbb{N}]}{\lambda x \cdot x : \mathbb{N} \to \mathbb{N}} \quad \rightsquigarrow \quad \begin{array}{l}\mathsf{is\text{-}lam}\ \mathsf{nat}\ \mathsf{nat}\ (\lambda x \cdot x)\ (\lambda yz \cdot z) \\ \quad : \mathsf{is}\ (\mathsf{lam}\ \mathsf{nat}\ (\lambda x \cdot x))\ (\mathsf{arr}\ \mathsf{nat}\ \mathsf{nat})\end{array}$$

A logical framework for incremental type-checking

Syntax

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

Judgements

- $\Gamma \vdash_\Sigma K$
- $\Gamma \vdash_\Sigma A$
- $\Gamma \vdash_\Sigma t : A$
- $\Gamma, A \vdash_\Sigma l : A$
- $\vdash \Sigma$

$$
\frac{\Gamma \vdash t : A \qquad \Gamma, B[\![x/t]\!] \vdash l : B}{\Gamma, \Pi x : A \cdot B \vdash t, l : C}
$$

Remarks

- the spine-form, canonical flavor (β and η-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 $(\lambda x \cdot x)$ $(\lambda yz \cdot z)$

Naming of proof steps

Remark

In LF, proof step = term application spine

Example $\mathsf{is\text{-}lam\ nat\ nat}\ (\lambda x \cdot x)\ (\lambda yz \cdot z)$

Monadic Normal Form (MNF)

Program transformation, IR for FP compilers

Goal: sequentialize all computations by naming them (lets)

$$
\begin{array}{lcl} t & ::= & \lambda x \cdot t \mid t(l) \mid x \\ l & ::= & \cdot \mid t, l \end{array}
\quad \Longrightarrow \quad
\begin{array}{lcl} \underline{t} & ::= & \mathsf{ret}\ \underline{v} \mid \mathsf{let}\ x = \underline{v}(\underline{l})\ \mathsf{in}\ \underline{t} \mid \underline{v}(\underline{l}) \\ \underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} \\ \underline{v} & ::= & x \mid \lambda x \cdot \underline{t} \end{array}
$$

Examples

- $f(g(x)) \quad \notin \quad \mathsf{MNF}$
- $\lambda x \cdot f(g(\lambda y \cdot y, x)) \quad \Longrightarrow$
  $\mathsf{ret}\ (\lambda x \cdot \mathsf{let}\ a = g(\lambda y \cdot y, x)\ \mathsf{in}\ f(a))$

Naming of proof steps

Positionality inefficiency

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

$$
\begin{array}{l}
\mathsf{let}\ x = \ldots\ \mathsf{in} \\
\quad \mathsf{let}\ y = \ldots\ \mathsf{in} \\
\quad\quad \mathsf{let}\ z = \ldots\ \mathsf{in} \\
\quad\quad\quad \vdots \\
\quad\quad\quad\quad \underline{v}(\underline{l})
\end{array}
\quad \Longrightarrow \quad
\left(\begin{array}{l}
x = \ldots \\
y = \ldots \\
z = \ldots \\
\vdots
\end{array}\right) \vdash \underline{v}(\underline{l})
$$

Naming of proof steps

Positionality inefficiency

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

$$\begin{array}{l}\mathsf{let}\ x = \ldots\ \mathsf{in} \\ \quad \mathsf{let}\ y = \ldots\ \mathsf{in} \\ \quad\quad \mathsf{let}\ z = \ldots\ \mathsf{in} \\ \quad\quad\quad \vdots \\ \quad\quad\quad\quad \underline{v}(\underline{l})\end{array} \implies \begin{pmatrix} x = \ldots \\ y = \ldots \\ z = \ldots \\ \vdots \end{pmatrix} \vdash \underline{v}(\underline{l})$$

Non-positional monadic calculus

$$\begin{array}{rcl}\underline{t} & ::= & \mathsf{ret}\ \underline{v} \mid \mathsf{let}\ x = \underline{v}(\underline{l})\ \mathsf{in}\ \underline{t} \mid \underline{v}(\underline{l}) \\ \underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} \\ \underline{v} & ::= & x \mid \lambda x \cdot \underline{t}\end{array}$$

Naming of proof steps

Positionality inefficiency

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

$$
\begin{array}{l}
\mathsf{let}\ x = \ldots\ \mathsf{in} \\
\quad \mathsf{let}\ y = \ldots\ \mathsf{in} \\
\quad\quad \mathsf{let}\ z = \ldots\ \mathsf{in} \\
\quad\quad\quad \vdots \\
\quad\quad\quad\quad \underline{v}(\underline{l})
\end{array}
\quad\Longrightarrow\quad
\left(\begin{array}{l}
x = \ldots \\
y = \ldots \\
z = \ldots \\
\vdots
\end{array}\right) \vdash \underline{v}(\underline{l})
$$

Non-positional monadic calculus

$$
\begin{array}{rcl}
\underline{t} & ::= & \mathsf{ret}\ \underline{v} \mid \underline{\sigma} \vdash \underline{v}(\underline{l}) \\
\underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} \\
\underline{v} & ::= & x \mid \lambda x \cdot \underline{t} \\
\underline{\sigma} & ::= & \cdot \mid \underline{\sigma}[x = \underline{v}(\underline{l})]
\end{array}
$$

Naming of proof steps

Positionality inefficiency

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

$$\begin{array}{l}\mathsf{let}\ x = \ldots\ \mathsf{in} \\ \quad \mathsf{let}\ y = \ldots\ \mathsf{in} \\ \quad\quad \mathsf{let}\ z = \ldots\ \mathsf{in} \\ \quad\quad\quad \vdots \\ \quad\quad\quad\quad \underline{v}(\underline{l})\end{array} \quad \Longrightarrow \quad \left(\begin{array}{l} x = \ldots \\ y = \ldots \\ z = \ldots \\ \vdots \end{array}\right) \vdash \underline{v}(\underline{l})$$

Non-positional monadic calculus

$$\begin{array}{rcl} \underline{t} & ::= & \mathsf{ret}\ \underline{v} \mid \underline{\sigma} \vdash \underline{v}(\underline{l}) \\ \underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} \\ \underline{v} & ::= & x \mid \lambda x \cdot \underline{t} \\ \underline{\sigma} & : & x \mapsto \underline{v}(\underline{l}) \end{array}$$

Monadic LF

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

Monadic LF

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

Monadic LF

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

Monadic LF

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

Definition

$$\cdot^* \quad : \quad \text{LF} \rightarrow \text{monadic LF}$$

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

Type annotation

Remark

In LF, judgement annotation = type annotation

Example

$\mathsf{is\text{-}lam}\ \mathsf{nat}\ \mathsf{nat}\ (\lambda x \cdot x)\ (\lambda yz \cdot z)$

$\quad : \mathsf{is}\ (\mathsf{lam}\ \mathsf{nat}\ (\lambda x \cdot x))\ (\mathsf{arr}\ \mathsf{nat}\ \mathsf{nat})$

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)

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

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)

$$
\begin{array}{rcl}
\underline{K} & ::= & \Pi x : \underline{A} \cdot \underline{K} \mid * \\
\underline{A} & ::= & \Pi x : \underline{A} \cdot \underline{A} \mid \underline{\sigma} \vdash a(\underline{l}) \\
\mathcal{R} & ::= & \underline{\sigma} \vdash \underline{v} : a(\underline{l}) \\
\underline{h} & ::= & x \mid a \\
\underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} \\
\underline{v} & ::= & c \mid x \mid \lambda x : \underline{A} \cdot \mathcal{R} \\
\underline{\sigma} & : & x \mapsto \underline{h}(\underline{l}) : a(\underline{l}) \\
\underline{\Sigma} & ::= & \cdot \mid \underline{\Sigma}[c : \underline{A}] \mid \underline{\Sigma}[a : \underline{K}]
\end{array}
$$

The repository language

Remark

In LF, judgement annotation = type annotation

Example

$\mathsf{is\text{-}lam\ nat\ nat}\ (\lambda x \cdot x)\ (\lambda yz \cdot z)$
$\quad : \mathsf{is}\ (\mathsf{lam\ nat}\ (\lambda x \cdot x))\ (\mathsf{arr\ nat\ nat})$

$$
\begin{array}{rcl}
\underline{K} & ::= & \Pi x : \underline{A} \cdot \underline{K} \mid * \\
\underline{A} & ::= & \Pi x : \underline{A} \cdot \underline{A} \mid \underline{\sigma} \vdash a(\underline{l}) \\
\mathcal{R} & ::= & \underline{\sigma} \vdash \underline{v} : a(\underline{l}) \qquad \longleftarrow \underline{\sigma} \text{ DAG, binds in } \underline{v} \text{ and } \underline{l} \\
\underline{h} & ::= & x \mid a \\
\underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} \\
\underline{v} & ::= & c \mid x \mid \lambda x : \underline{A} \cdot \mathcal{R} \\
\underline{\sigma} & : & x \mapsto \underline{h}(\underline{l}) : a(\underline{l}) \\
\underline{\Sigma} & ::= & \cdot \mid \underline{\Sigma}[c : \underline{A}] \mid \underline{\Sigma}[a : \underline{K}]
\end{array}
$$

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)

$$
\begin{array}{rcll}
\underline{K} & ::= & \Pi x : \underline{A} \cdot \underline{K} \mid * & \\
\underline{A} & ::= & \Pi x : \underline{A} \cdot \underline{A} \mid \underline{\sigma} \vdash a(\underline{l}) & \\
\mathcal{R} & ::= & \underline{\sigma} \vdash \underline{v} : a(\underline{l}) & \longleftarrow \underline{\sigma} \text{ DAG, binds in } \underline{v} \text{ and } \underline{l} \\
\underline{h} & ::= & x \mid a & \\
\underline{l} & ::= & \cdot \mid \underline{v}, \underline{l} & \\
\underline{v} & ::= & c \mid x \mid \lambda x : \underline{A} \cdot \mathcal{R} & \\
\underline{\sigma} & : & x \mapsto \underline{h}(\underline{l}) : a(\underline{l}) & \longleftarrow \text{named implementation} \\
\underline{\Sigma} & ::= & \cdot \mid \underline{\Sigma}[c : \underline{A}] \mid \underline{\Sigma}[a : \underline{K}] &
\end{array}
$$

The delta language

Syntax

$$\begin{aligned} K &::= \Pi x : A \cdot K \mid * \\ A &::= \Pi x : A \cdot A \mid a(l) \\ t &::= \lambda x \cdot t \mid x(l) \mid c(l) \mid [t]^{t}_{x.n} \\ l &::= \cdot \mid t, l \\ \Sigma &::= \cdot \mid \Sigma[c : A] \mid \Sigma[a : K] \end{aligned}$$

Judgements

- $\mathcal{R}, \underline{\Gamma} \vdash K \to \underline{K}$
- $\mathcal{R}, \underline{\Gamma} \vdash A \to \underline{A}$
- $\mathcal{R}, \underline{\Gamma} \vdash t : \underline{A} \to \underline{t}$
- $\mathcal{R}, \underline{\Gamma}, \underline{A} \vdash l \to \underline{l} : \underline{A}$
- $\vdash \Sigma \to \underline{\Sigma}$

Informally

- $\mathcal{R}, \Gamma \vdash_{\Sigma} x \Rightarrow \mathcal{R}$ means
  "I am what x stands for, in Γ or in $\mathcal{R}$ (and produce $\mathcal{R}$)".
- $\mathcal{R}, \Gamma \vdash_{\Sigma} [t]^{u}_{y.n} \Rightarrow \mathcal{R}'$ 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

$$\mathcal{R}, \underline{\Gamma} \vdash t : \underline{A} \to \underline{t}$$

What does it do?

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

Typing by annotating

partial translation : monadic LF $\to$ annotated monadic LF

VLam

$$\frac{\mathcal{R}, \underline{\Gamma}[x : \underline{A}] \vdash t : \underline{B} \to \underline{t}}{\mathcal{R}, \underline{\Gamma} \vdash \lambda x \cdot t : \Pi x : \underline{A} \cdot \underline{B} \to \lambda x : \underline{A} \cdot \underline{t}}$$

Typing by annotating

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

$$\textsc{HVar}\ \frac{\underline{\Gamma}(x) : \underline{A} \qquad \text{or} \qquad \underline{\sigma}(x) : \underline{A}}{(\underline{\sigma} \vdash \underline{v}), \underline{\Gamma} \vdash x \rightarrow \underline{A}}$$

Typing by annotating

partial translation : monadic LF → annotated monadic LF

LCons

$$\frac{\mathcal{R}, \underline{\Gamma} \vdash v : \underline{A} \to \underline{v} \qquad \mathcal{R}, \underline{\Gamma}, \underline{B}[\![x/\underline{v}]\!] \vdash l \to \underline{l} : a(\underline{l})}{\mathcal{R}, \underline{\Gamma}, \Pi x : \underline{A} \cdot \underline{B} \vdash v, l \to \underline{v}, \underline{l} : a(\underline{l})}$$

Typing by annotating

partial translation : monadic LF $\to$ annotated monadic LF

OBox

$$\frac{\mathcal{R}|_p = \lambda x : \underline{B} \cdot \mathcal{R}' \qquad \mathcal{R}, \underline{\Gamma} \vdash u : \underline{B} \to (\underline{\sigma} \vdash \underline{h} : a(\underline{l})) \qquad \mathcal{R}' \cup \underline{\sigma}[x = \underline{h} : a(\underline{l})], \underline{\Gamma} \vdash t : \underline{A} \to \underline{t}}{\mathcal{R}, \underline{\Gamma} \vdash [t]_p^u : \underline{A} \to \underline{t}}$$

Typing by annotating

partial translation : monadic LF → annotated monadic LF

$$(\Pi x : \underline{A} \cdot \underline{B})[\![z/\underline{v}]\!] = \Pi x : \underline{A}[\![z/\underline{v}]\!] \cdot \underline{B}[\![z/\underline{v}]\!]$$
$$(\underline{\sigma} \vdash a(\underline{l}))[\![z/\underline{v}]\!] = \underline{\sigma}[\![z/\underline{v}]\!] \vdash a(\underline{l}[\![z/\underline{v}]\!])$$

$$(\underline{\sigma}[y = x(\underline{l}) : a(\underline{m})])[\![z/\underline{v}]\!] = (\underline{\sigma}[\![z/\underline{v}]\!])[y = x(\underline{l}[\![z/\underline{v}]\!]) : a(\underline{m}[\![z/\underline{v}]\!])]$$
$$(\underline{\sigma}[y = z(\underline{l}) : a(\underline{m})])[\![z/\underline{v}]\!] = \mathsf{red}^y_{\underline{\sigma}}(\underline{v}, \underline{l})$$

$$\mathsf{red}^y_{\underline{\sigma}}(\underline{h} : a(\underline{l}), \cdot) = \underline{\sigma}[y = \underline{h} : a(\underline{l})]$$
$$\mathsf{red}^y_{\underline{\sigma}}(\lambda x : \underline{A} \cdot \underline{t}, (\underline{v}, \underline{l})) = \mathsf{red}^y_{\underline{\sigma} \cup \underline{\rho}}(\underline{w}, \underline{l}) \qquad \text{if} \quad \underline{t}[\![x/\underline{v}]\!] = (\underline{\rho} \vdash \underline{w})$$

$$\vdots$$

Properties of the translation

Work in progress…

Theorem (Soundness)

if $\Gamma \vdash t : A$ *then* $\vdash \Gamma^* \to \underline{\Gamma}$ *and* $(\cdot \vdash \_), \underline{\Gamma} \vdash A^* \to \underline{A}$ *and* $(\cdot \vdash \_), \underline{\Gamma} \vdash t^* : \underline{A} \to \underline{t}$

Definition (Checkout)

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

Theorem (Completeness)

if $(\cdot \vdash \_), \underline{\Gamma} \vdash t^* : \underline{A} \to \underline{t}$ *then* $\underline{\Gamma}^- \vdash \underline{t}^- : \underline{A}^-$

Theorem (Substitution)

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

Example

Signature

$$A\ B\ C\ D : *$$

$$a : (D \to B) \to C \to A \qquad\qquad b\ b' : C \to B$$

$$c : D \to C \qquad\qquad d : D$$

Terms

$$\begin{aligned}
t_1 &= a(\lambda x \cdot b(c(x)), c(d)) \\
\mathcal{R}_1 &= [v = c(d) : C] \\
&\quad [u = a(\lambda x : D \cdot [w = c(x) : C][w' = b(w) : B] \vdash w' : B, v) : A] \\
&\qquad \vdash u : A \\
t_2 &= a(\lambda y \cdot [b'(w)]^x_1 y) \\
\mathcal{R}_2 &= [v = c(d) : C] \\
&\quad [u = a(\lambda y : D \cdot [x = y][w = c(x) : C][w' = b(w) : B] \vdash w' : B, v) : A] \\
&\qquad \vdash u : A
\end{aligned}$$

Regaining version management

Just add to the signature Σ:

$$\begin{aligned} \mathsf{Version} &: * \\ \mathsf{Commit0} &: \mathsf{Version} \\ \mathsf{Commit} &: \Pi t : \mathsf{tm} \cdot \mathsf{is}(t, \mathsf{unit}) \to \mathsf{Version} \to \mathsf{Version} \end{aligned}$$

Commit t

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

then

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

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!