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

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?

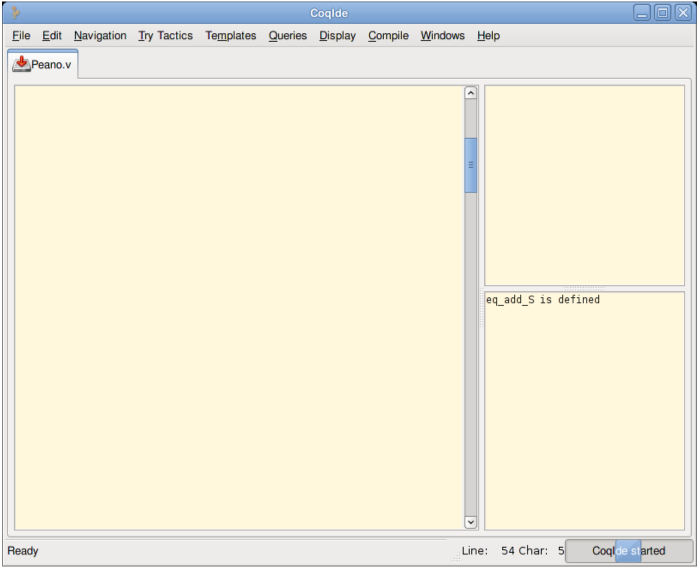
Incrementality in programming & proof languages

Q : Do you spend more time *writing* code or *editing* code?

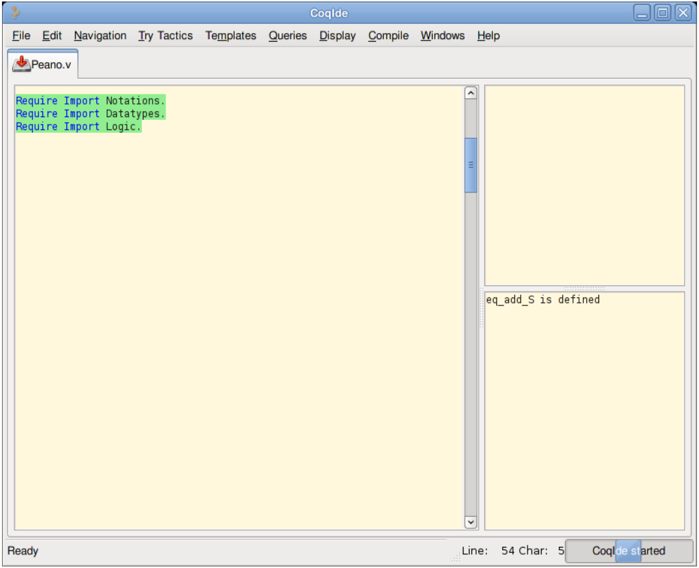
Today, we use:

- separate compilation
- dependency management
- version control on the scripts
- interactive toplevel with rollback (Coq)

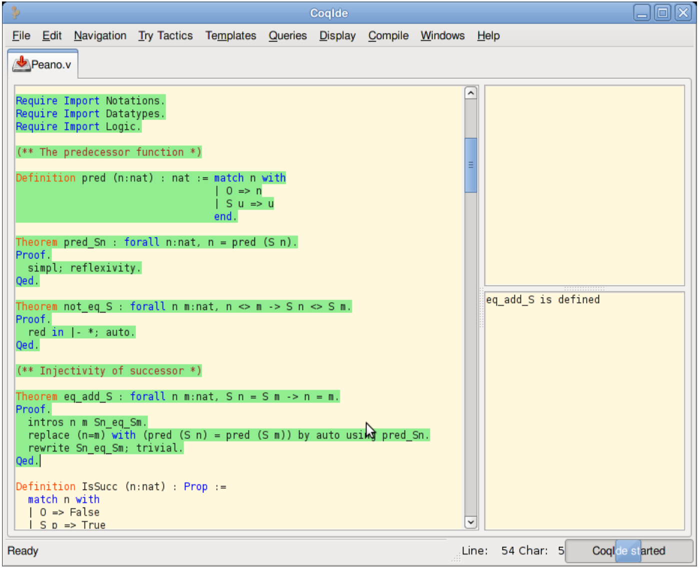
Incrementality in programming & proof languages



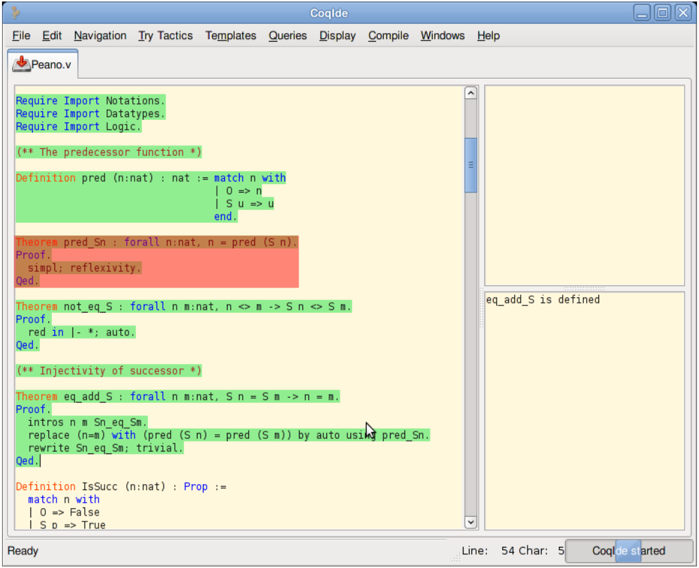
Incrementality in programming & proof languages



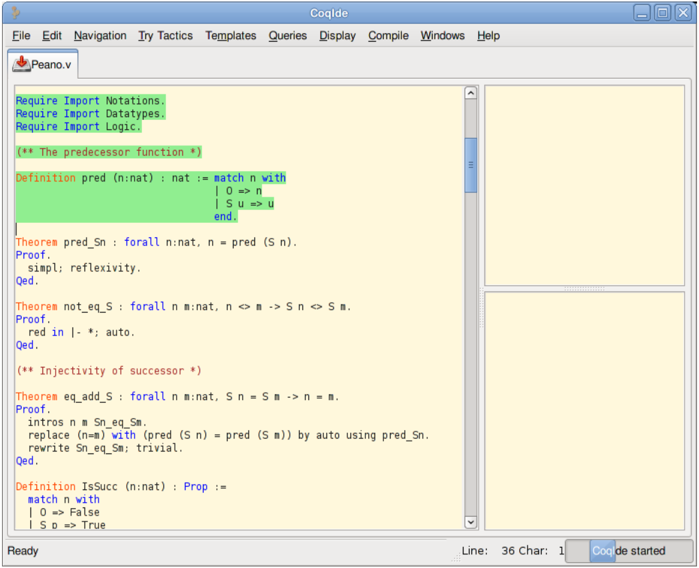
Incrementality in programming & proof languages



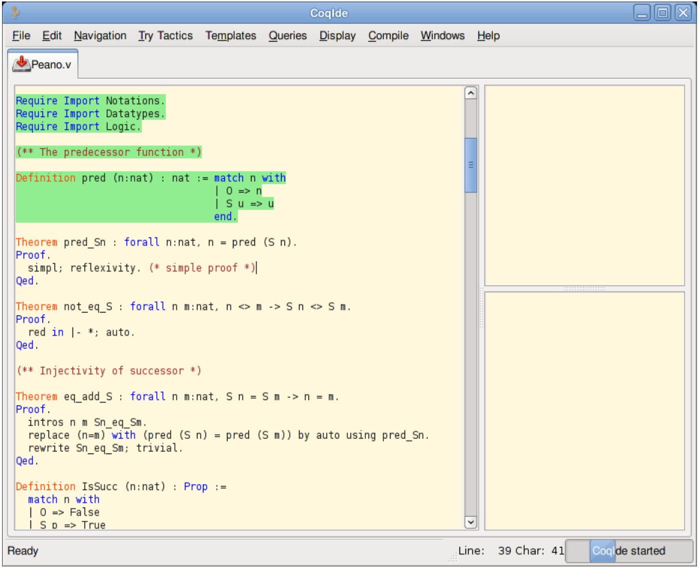
Incrementality in programming & proof languages



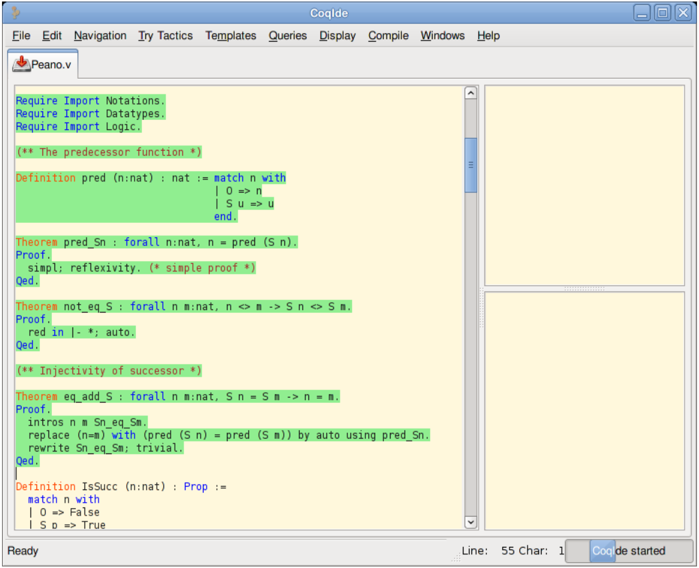
Incrementality in programming & proof languages



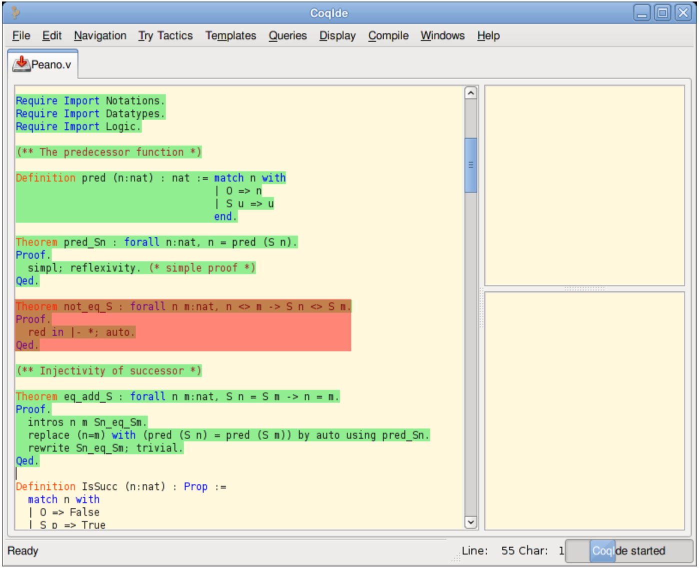
Incrementality in programming & proof languages



Incrementality in programming & proof languages



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

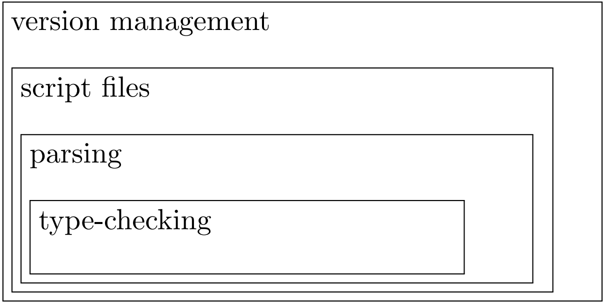
The big picture
- Incremental type-checking
- Why not memoization?

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

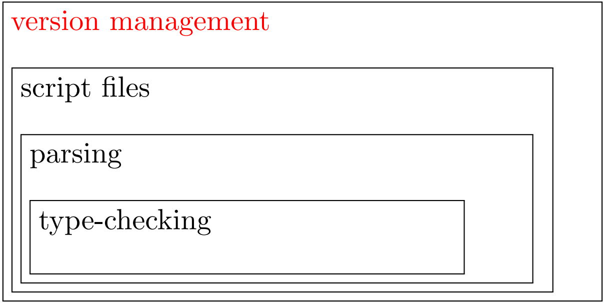
A metalanguage of repository
- The LF logical framework
- Monadic LF
- Committing to MLF

Menu

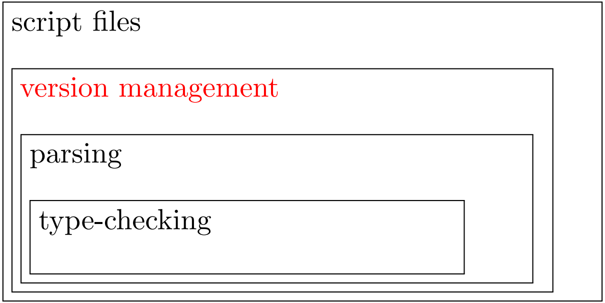
The big picture



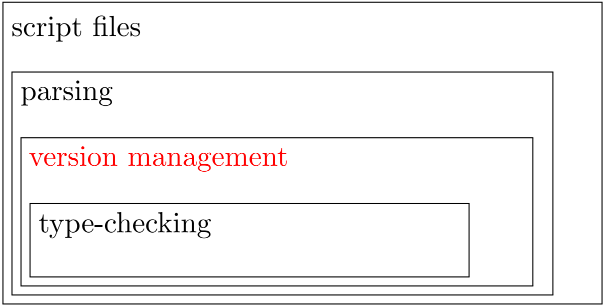
The big picture



The big picture

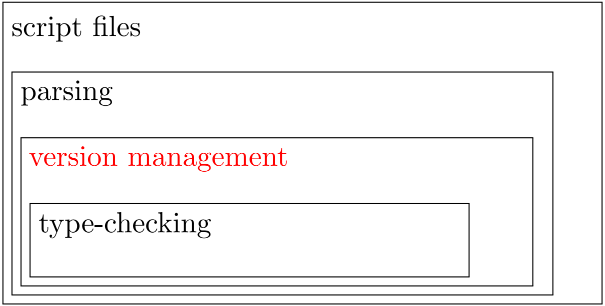


The big picture



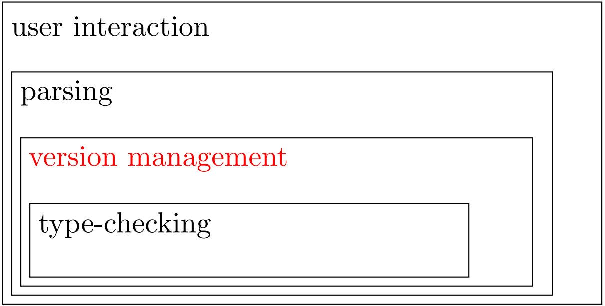
- AST representation

The big picture



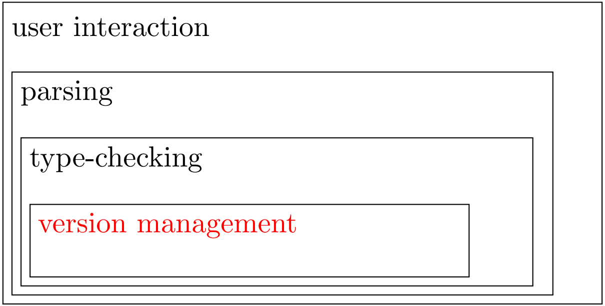
- AST representation

The big picture



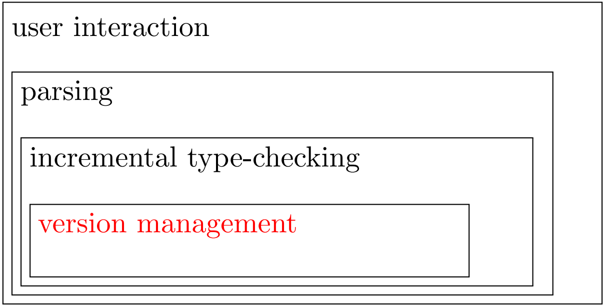
- AST representation

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

$$\textbf{val}\ \mathsf{check} : \mathsf{env} \to \mathsf{term} \to \mathsf{types} \to \mathsf{bool}$$

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

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

$$\to_i \dfrac{\to_i \dfrac{\to_i \dfrac{\to_e \dfrac{\dfrac{}{A \to B, B \to C, A \vdash B \to C}\,Ax \qquad \dfrac{\dfrac{}{A \to B, B \to C, A \vdash A \to B}\,Ax \qquad \dfrac{}{A \to B, B \to C, A \vdash A}\,Ax}{A \to B, B \to C, A \vdash B}\,\to_e}{A \to B, B \to C, A \vdash C}}{A \to B, B \to C \vdash A \to C}}{A \to B \vdash (B \to C) \to A \to C}}{\vdash (A \to B) \to (B \to C) \to A \to C}$$

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

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

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

an incremental type-checker

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

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

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

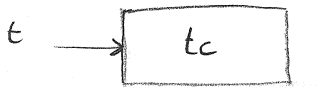
decides if $\mathsf{apply}(\delta, \mathcal{R})$ is well-typed in $O(|\delta|)$.
(and not $O(|\mathsf{apply}(\delta, \mathcal{R})|)$)

A logical framework for incremental type-checking

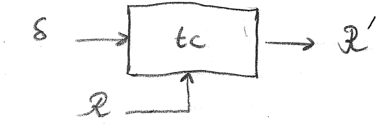
Goal Type-check a large derivation taking advantage of the knowledge from type-checking previous versions

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
+ $repository = \mathsf{table}$, $delta = \mathsf{t}$
– syntactic comparison (no quotient on judgements)
  What if I want *e.g.* weakening or permutation to be taken into account?

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?

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

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?

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

- − 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?

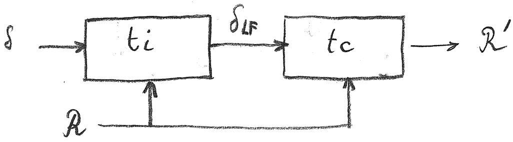
$$\frac{J \in \Gamma}{\Gamma \vdash J \ \mathsf{wf} \Rightarrow \Gamma} \qquad \frac{\Gamma_1 \vdash J_1 \ \mathsf{wf} \Rightarrow \Gamma_2 \quad \dots \quad \Gamma_{n-1}[J_{n-1}] \vdash J_n \ \mathsf{wf} \Rightarrow \Gamma_n}{\Gamma_1 \vdash J \ \mathsf{wf} \Rightarrow \Gamma_n[J_n][J]}$$

- − imperative (introduces a dissymmetry)

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

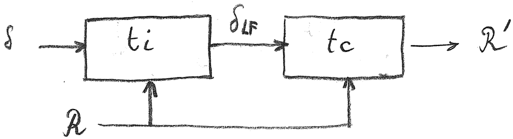
Menu

Two-passes type-checking



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

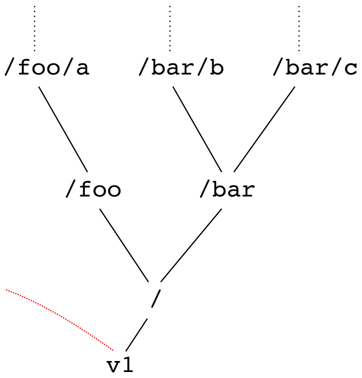
Two-passes type-checking



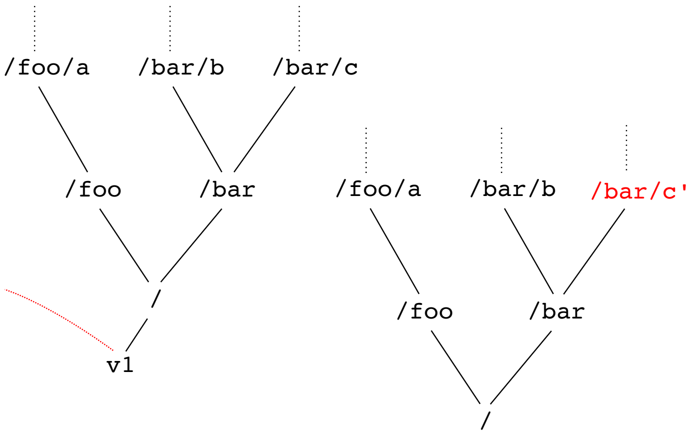
$$
\begin{aligned}
\mathsf{ti} &= \text{type inference} = \text{derivation delta synthesis} \\
\mathsf{tc} &= \text{type checking} = \text{derivation delta checking} \\
\delta &= \text{program delta} \\
\delta_{LF} &= \text{derivation delta} \\
\mathcal{R} &= \text{repository of derivations}
\end{aligned}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

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

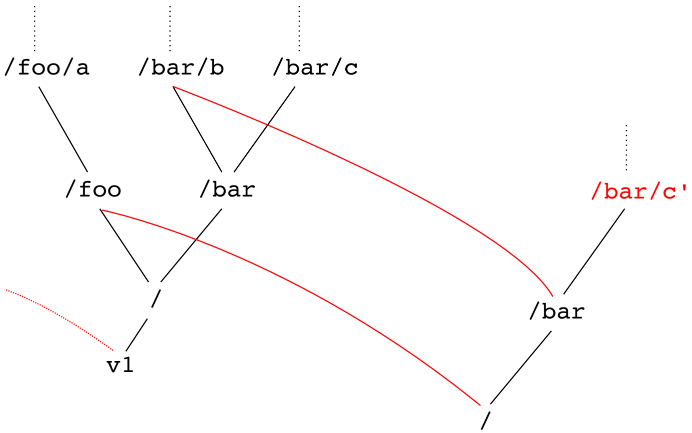
A popular storage model for directories



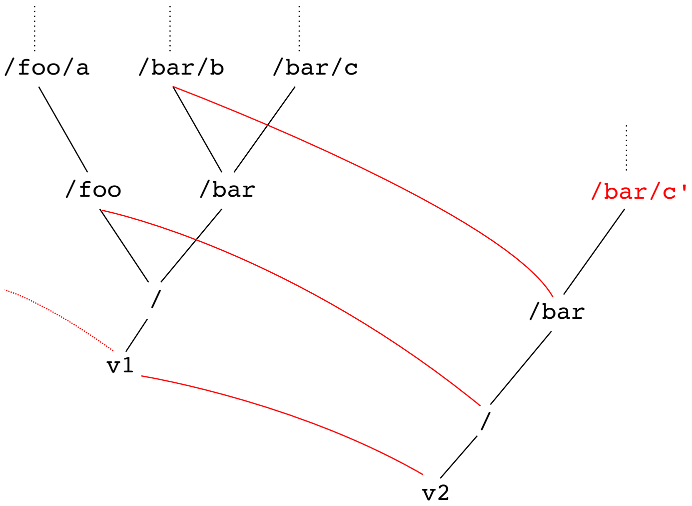
A popular storage model for directories



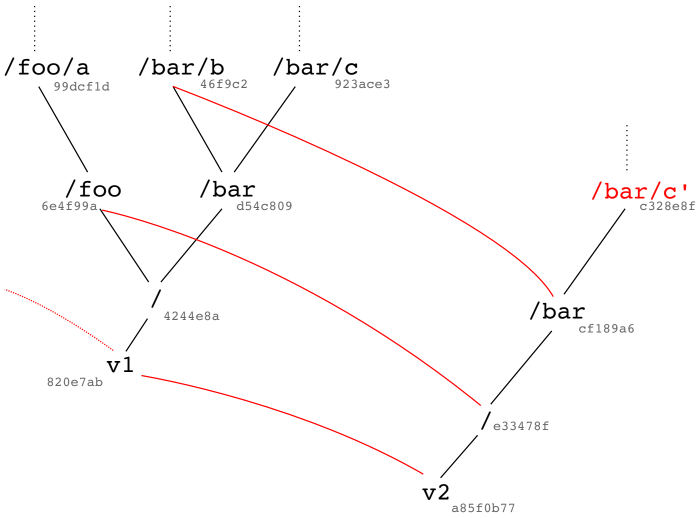
A popular storage model for directories



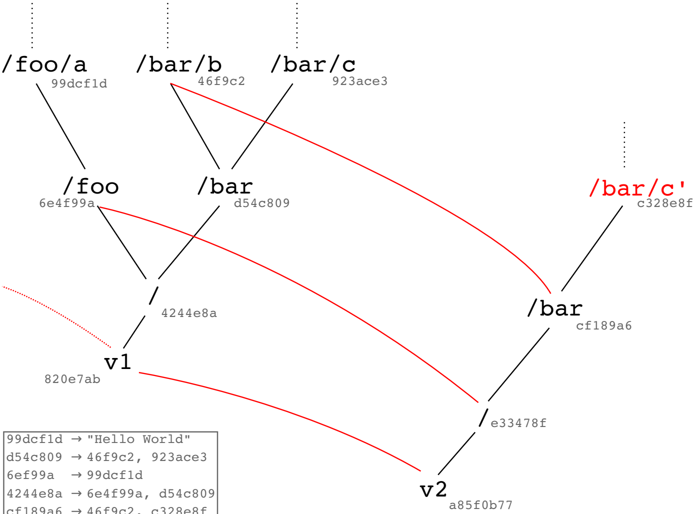
A popular storage model for directories



A popular storage model for directories

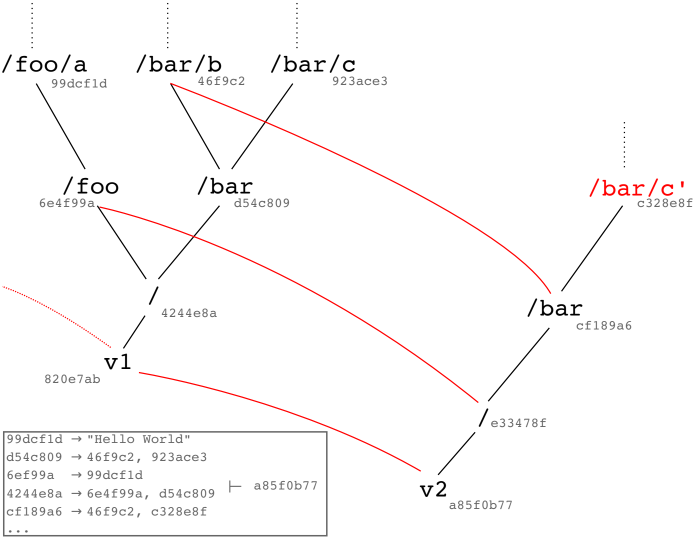


A popular storage model for directories




```
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):

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

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):

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

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):

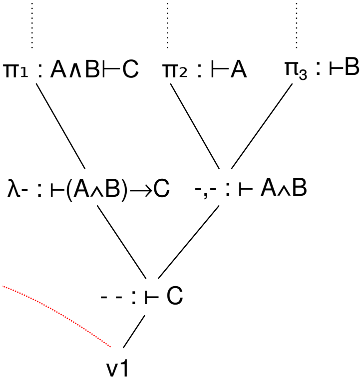
$$\Delta : x \mapsto (\mathsf{Commit}\ (x \times y) \mid \mathsf{Tree}\ \vec{x} \mid \mathsf{Blob}\ string)$$

Invariants

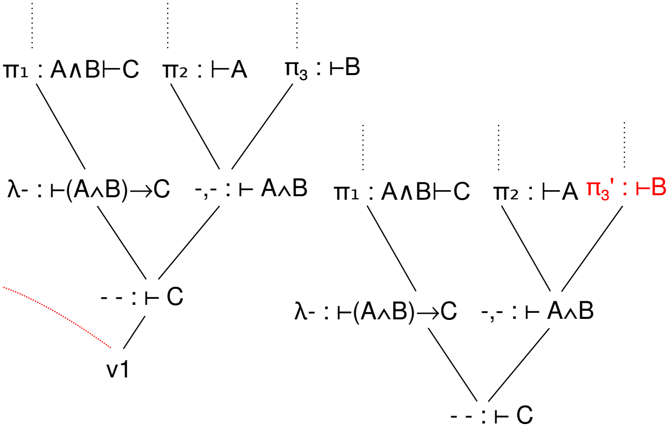
- Δ forms a DAG
- if $(x, \mathsf{Commit}\ (y, z)) \in \Delta$ then
  - $(y, \mathsf{Tree}\ t) \in \Delta$
  - $(z, \mathsf{Commit}\ (t, v)) \in \Delta$
- if $(x, \mathsf{Tree}(\vec{y})) \in \Delta$ then
  for all y_i, either $(y_i, \mathsf{Tree}(\vec{z}))$ or $(y_i, \mathsf{Blob}(s)) \in \Delta$

Let's do the same with *proofs*

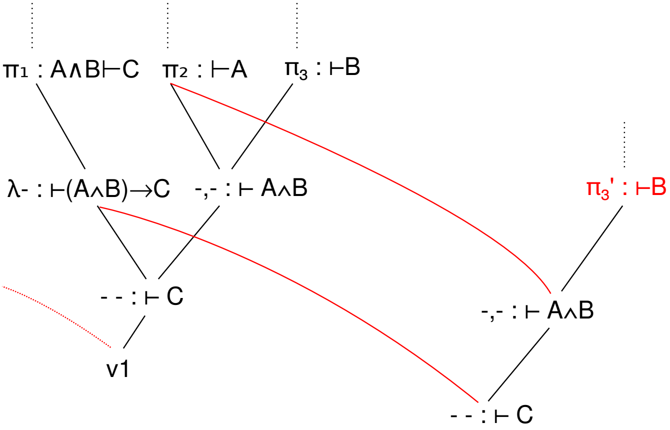
A *typed* repository of proofs



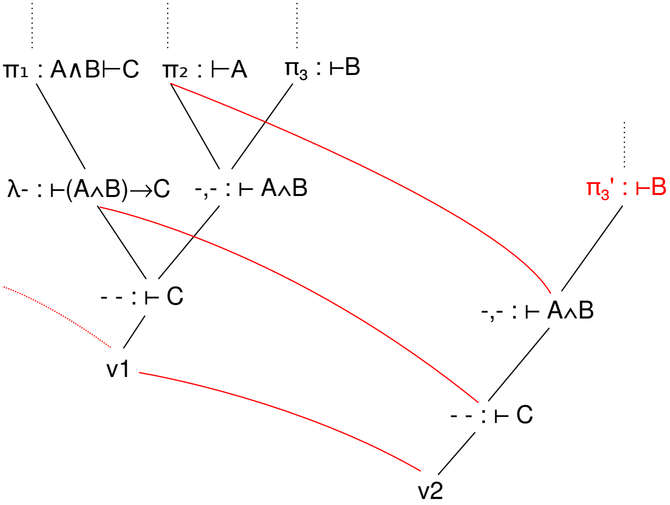
A *typed* repository of proofs



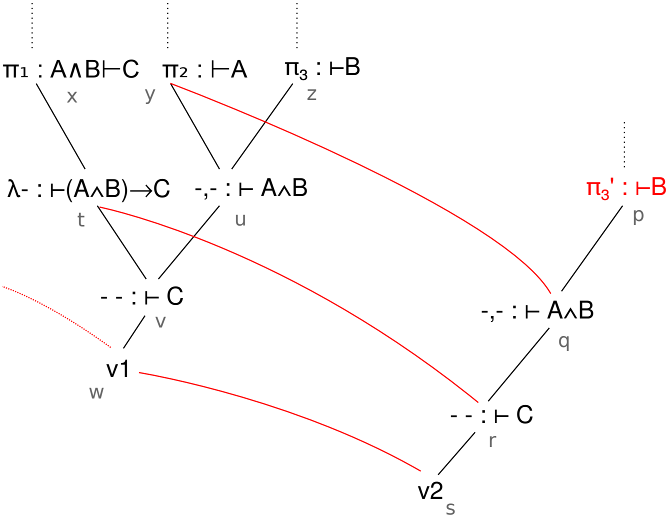
A *typed* repository of proofs



A *typed* repository of proofs



A *typed* repository of proofs



A *typed* repository of proofs

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

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

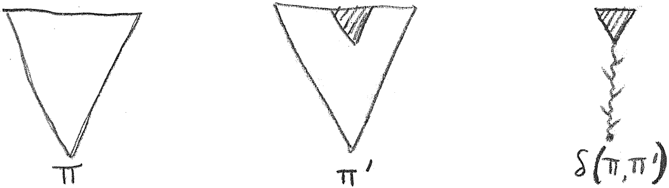
$$
\begin{aligned}
x &= \ldots \ : A \wedge B \vdash C \\
y &= \ldots \ : \vdash A \\
z &= \ldots \ : \vdash B \\
t &= \lambda a : A \wedge B \cdot x : \vdash A \wedge B \to C \\
u &= (y, z) : \vdash A \wedge B \\
v &= t\ u : \vdash C \\
w &= \mathsf{Commit}(v, w1) : \mathsf{Version} \\
p &= \ldots \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

A *typed* repository of proofs

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

A data-oriented notion of delta

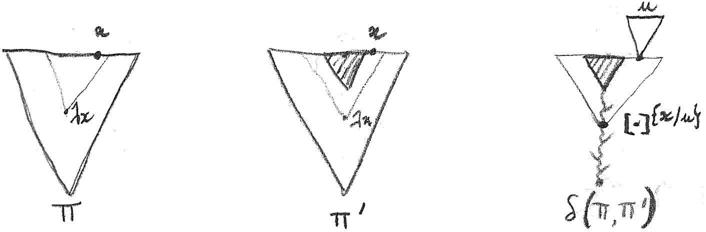
The first-order case



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

A data-oriented notion of delta

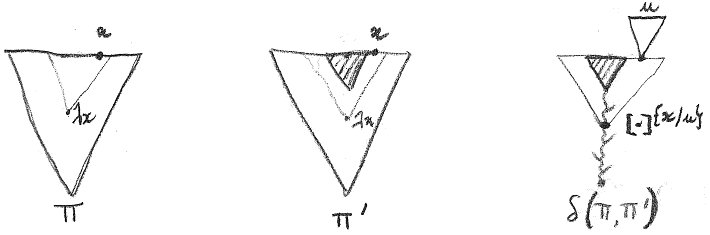
The binder case



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

Towards a metalanguage of proof repository

Repository language

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

Delta language

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

$\leadsto$ **Need extra-logical features!**

Menu

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

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

Judgements

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

The delta language

Syntax

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

Judgements

- $\mathcal{R}, \Gamma \vdash_\Sigma K \Rightarrow \mathcal{R}$
- $\mathcal{R}, \Gamma \vdash_\Sigma A : K \Rightarrow \mathcal{R}$
- $\mathcal{R}, \Gamma \vdash_\Sigma t : A \Rightarrow \mathcal{R}$
- $\vdash \Sigma$

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 $\mathcal{R}$)".
- $\mathcal{R}, \Gamma \vdash_\Sigma [t]^{\{x/u\}}_{y.n} \Rightarrow \mathcal{R}'$ means
  "Variable y has the form $\_(v_1 \ldots v_{n-1}(\lambda x \cdot \mathcal{R}'') \ldots)$ in $\mathcal{R}$. Make all variables in $\mathcal{R}''$ in scope for t, taking u for x. t will produce $\mathcal{R}'$"

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

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

Naming of proof steps

Positionality inefficiency

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

Naming of proof steps

Positionality inefficiency

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

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

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

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

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

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

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

Monadic LF

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

Monadic LF

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

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

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

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

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)

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

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:

$$\textsc{Var}\ \frac{\Gamma(x) = A \qquad \text{or} \qquad \sigma(x) : A}{(\sigma \vdash \_ : \_), \Gamma \vdash_\Sigma x : A \Rightarrow (\sigma \vdash x : A)}$$

$$\textsc{Box}\ \frac{\sigma(x).i = \lambda y : B \cdot (\rho \vdash H'') \qquad (\sigma \vdash H), \Gamma \vdash u : B \Rightarrow (\theta \vdash H') \qquad (\rho \cup \theta[y = H'] \vdash H''), \Gamma \vdash t : A \Rightarrow \mathcal{R}}{(\sigma \vdash H), \Gamma \vdash [t]_{x.i}^{\{y/u\}} : A \Rightarrow \mathcal{R}}$$

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

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

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

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