Certificates for incremental type checking

Matthias Puech[1,2] Yann Régis-Gianas[2]

[1]Dept. of Computer Science, University of Bologna

[2]Univ. Paris Diderot, Sorbonne Paris Cité, PPS, CNRS, πr^2, INRIA

June 20, 2012

PPS – Groupe de travail théorie des types et réalisabilité

Introduction
- How to make a type checker incremental?
- How to trust your type checker?

Using Gasp
- As a programmer
- The LF notation for derivations
- As a type system designer

The design of Gasp
- Data structures
- Typed evaluation algorithm

Problem 1: How to make a type checker incremental?

Certificates for incremental type-checking

Observations

- Program elaboration is more and more an *interaction* between the programmer and the type-checker
- The richer the type system is, the more expensive type-checking gets

Example

- type inference (e.g. Haskell, unification)
- dependent types (conversion, esp. reflection)
- very large term

Certificates for incremental type-checking

Observations

- Program elaboration is more and more an *interaction* between the programmer and the type-checker
- The richer the type system is, the more expensive type-checking gets

Example

- type inference (e.g. Haskell, unification)
- dependent types (conversion, esp. reflection)
- very large term

…but is called repeatedly with almost the same input

Certificates for incremental type-checking

emacs@soupirail.inria.fr

File Edit Options Buffers Tools TypeRex Help

```
      kfprintf (print_log tag) formatter "@]@,"
    end else ikfprintf ignore formatter

  let close tag = if active tag then
      stack := pop tag !stack;
      pp_close_box formatter ()

end

module Topcatch = struct

  open Format

  exception Unhandled

  let stk = ref ([] : (formatter -> exn -> unit) list)

  let register f = stk := f :: !stk

  let print fmt e =
    let rec aux = function
      | [] -> Format.pp_print_newline fmt (); raise e
      | f :: stk -> try f fmt e with Unhandled -> aux stk in
    aux !stk

  let catch fct arg =
    try
      fct arg
    with x ->
      pp_print_newline Debug.formatter ();
      flush stdout;
      eprintf "@[Uncaught exception:@ @[%a@]@]@." print x;
      raise Unhandled

  let _ =
    register begin fun fmt -> function
      | e  -> raise e
    end

end
-U:---  util.ml        77% (207,18)  Git:master  (TypeRex Trim pair Abbrev)----
<M-Scroll_Lock> is undefined
```

Certificates for incremental type-checking

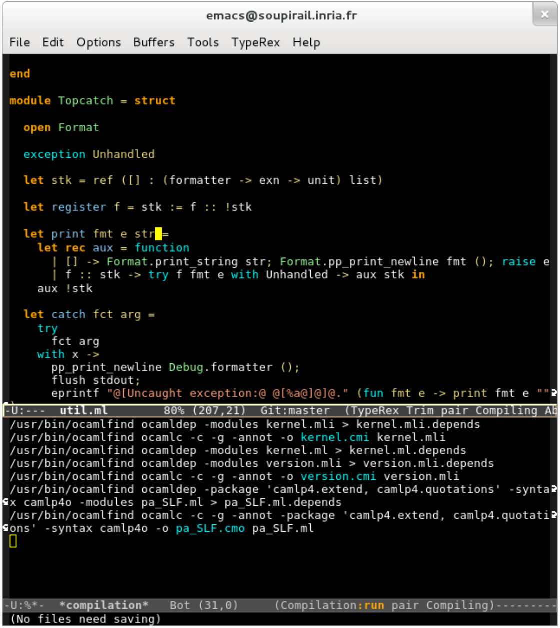
emacs@soupirail.inria.fr

File Edit Options Buffers Tools TypeRex Help

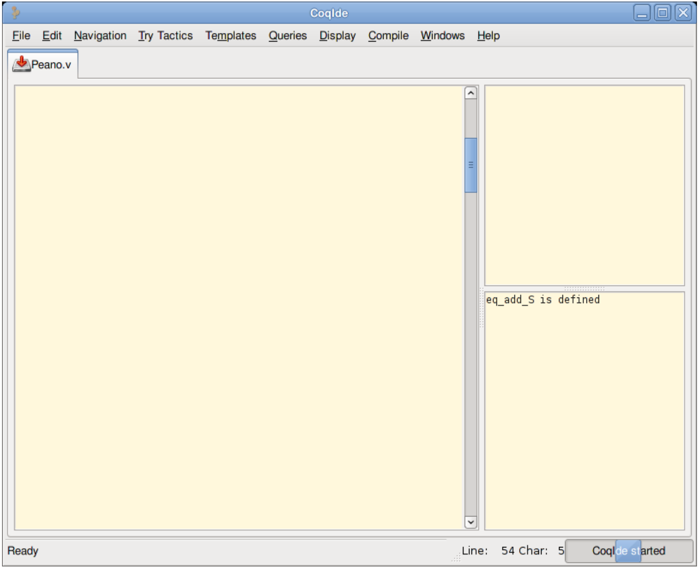
```
      kfprintf (print_log tag) formatter "@]@,"
    end else ikfprintf ignore formatter

  let close tag = if active tag then
      stack := pop tag !stack;
      pp_close_box formatter ()

end

module Topcatch = struct

  open Format

  exception Unhandled

  let stk = ref ([] : (formatter -> exn -> unit) list)

  let register f = stk := f :: !stk

  let print fmt e str =
    let rec aux = function
      | [] -> Format.print_string str; Format.pp_print_newline fmt (); raise e
      | f :: stk -> try f fmt e with Unhandled -> aux stk in
    aux !stk

  let catch fct arg =
    try
      fct arg
    with x ->
      pp_print_newline Debug.formatter ();
      flush stdout;
      eprintf "@[Uncaught exception:@ @[%a@]@]@." print x;
      raise Unhandled

  let _ =
    register begin fun fmt -> function
      | e  -> raise e
    end

end
```

-U:**- util.ml   77% (207,21) Git:master (TypeRex Trim pair Abbrev)----

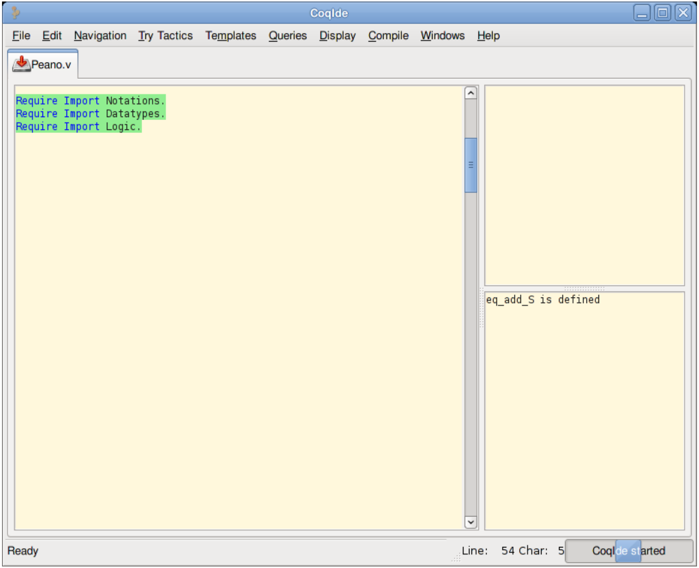
Certificates for incremental type-checking



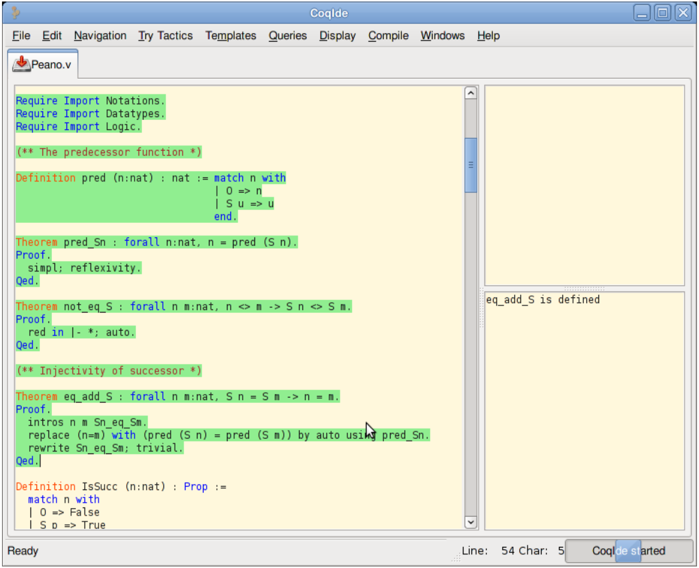
Certificates for incremental type-checking



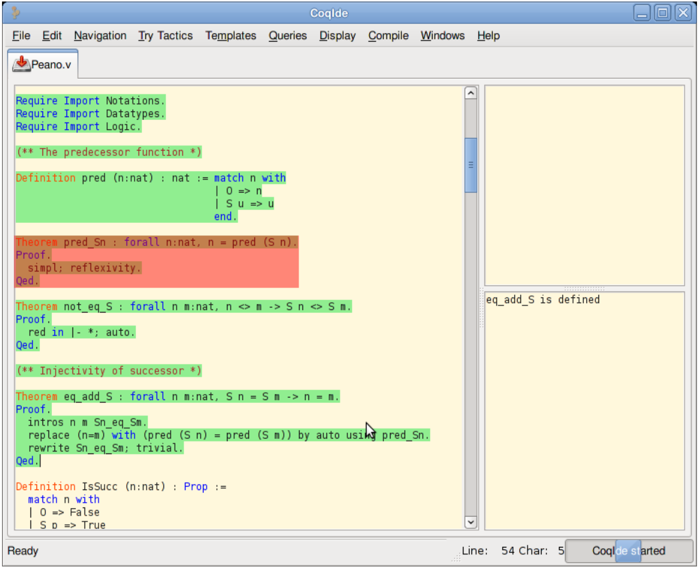
Certificates for incremental type-checking



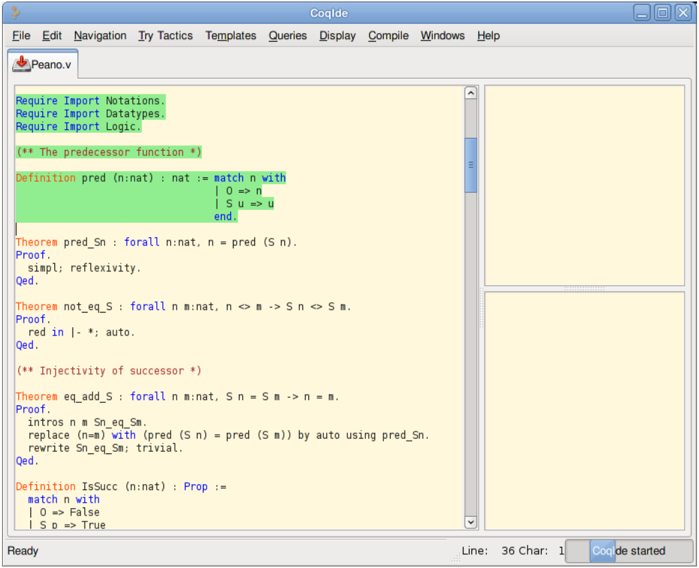
Certificates for incremental type-checking



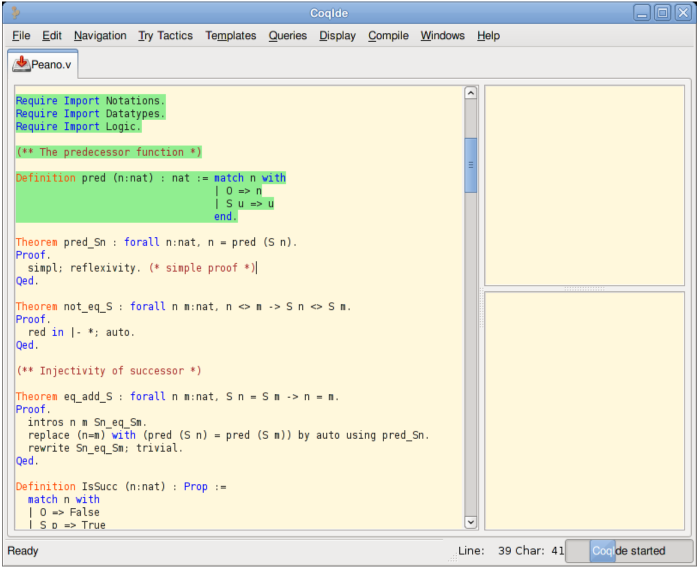
Certificates for incremental type-checking



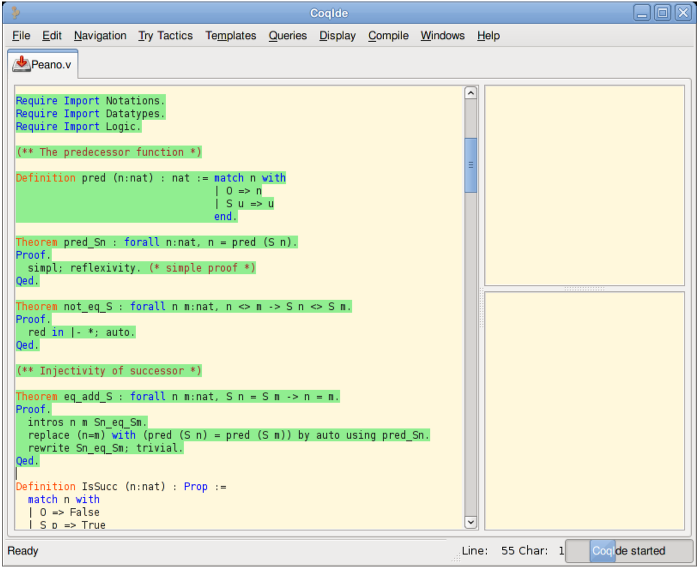
Certificates for incremental type-checking



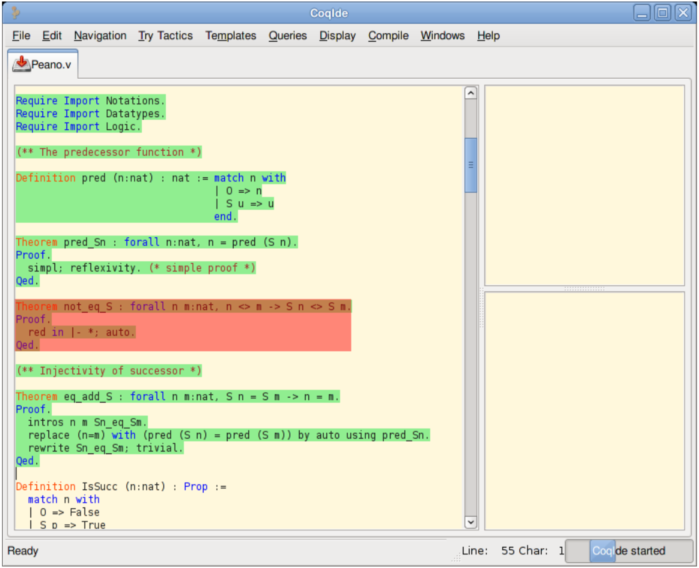
Certificates for incremental type-checking



Certificates for incremental type-checking



Certificates for incremental type-checking



Certificates for incremental type checking

Problem

How to take advantage of the knowledge from previous type-checks?

- Reuse already-computed results
- Recheck only the changed part of a program and its *impact*

Problem 2: How to trust your type checker?

A compiler designer's job

System Z

$$\downarrow$$

$$\uparrow \ : \ \mathsf{env} \to \mathsf{tm} \to \mathsf{tp\ option}$$

set of declarative inference rules $\to$ decision algorithm

A compiler designer's job

System Z

$$\downarrow$$

$\uparrow \ : \ \mathsf{env} \to \mathsf{tm} \to \mathsf{tp\ option}$

set of declarative inference rules → decision algorithm

- non trivial (inference, conversion. . .)
- critical

Example: System $\mathsf{T}_{<:}$

Syntax

$$
\begin{array}{rcl}
M & ::= & \mathsf{o} \mid \mathsf{s}(M) \mid M\,M \mid \lambda x.\ M \mid \mathsf{rec}(M, N, xy.\ P) \\
A & ::= & \mathsf{nat} \mid \mathsf{even} \mid \mathsf{odd} \mid A \to A
\end{array}
$$

Typing rules

$$
\frac{\vdash M : \mathsf{nat} \qquad \vdash N : A \qquad \begin{array}{c}[\vdash x : \mathsf{nat}] \quad [\vdash y : A] \\ \vdots \\ \vdash P : A\end{array}}{\vdash \mathsf{rec}(M, N, xy.\ P) : A}
$$

$$
\frac{\vdash M : A \qquad \vdash A \leq B}{\vdash M : B}
$$

Example: System $\mathsf{T}_{<:}$

Syntax

$$
\begin{aligned}
M &::= \mathsf{o} \mid \mathsf{s}(M) \mid M\,M \mid \lambda x.\ M \mid \mathsf{rec}(M, N, xy.\ P) \\
A &::= \mathsf{nat} \mid \mathsf{even} \mid \mathsf{odd} \mid A \to A
\end{aligned}
$$

Typing rules

$$
\frac{\vdash M : \mathsf{nat} \qquad \vdash N : A \qquad \begin{array}{c} [\vdash x : \mathsf{nat}] \quad [\vdash y : A] \\ \vdots \\ \vdash P : A \end{array}}{\vdash \mathsf{rec}(M, N, xy.\ P) : A}
$$

$$
\frac{\vdash M : A \qquad \vdash A \leq B}{\vdash M : B}
$$

Not syntax directed!

Example: System $\mathsf{T}_{<:}$

Typing algorithm

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

Example: System $\mathsf{T}_{<:}$

Typing algorithm

$$\frac{\Gamma \vdash M : A \to B \qquad \dfrac{\Gamma \vdash N : A' \qquad \Gamma \vdash A' \leq A}{\Gamma \vdash N : A}}{\Gamma \vdash M\ N : B}$$

Example: System $\mathsf{T}_{<:}$

Typing algorithm

$$\frac{\Gamma \vdash M : \mathsf{nat} \qquad \Gamma \vdash N : A \qquad \Gamma, x : \mathsf{nat}, y : A \vdash P : A}{\vdash \mathsf{rec}(M, N, xy.\ P) : A}$$

Example: System $\mathsf{T}_{<:}$

Typing algorithm

$$\frac{\Gamma \vdash M : T_M \qquad \Gamma \vdash T_M \leq \mathsf{nat} \qquad \Gamma \vdash N : T_N \quad \Gamma, x : \mathsf{nat}, y : T_N \vdash P : T_P \quad \Gamma, x : \mathsf{nat}, y : T_N \sqcap T_P \vdash P : T_N \sqcap T_P}{\Gamma \vdash \mathsf{rec}(M, N, xy.\ P) : T_N \sqcap T_P}$$

Example: System $\mathsf{T}_{<:}$

Typing algorithm

$$\frac{\Gamma \vdash M : T_M \qquad \Gamma \vdash T_M \leq \mathsf{nat} \qquad \Gamma \vdash N : T_N \quad \Gamma, x : \mathsf{nat}, y : T_N \vdash P : T_P \quad \Gamma, x : \mathsf{nat}, y : T_N \sqcap T_P \vdash P : T_N \sqcap T_P}{\Gamma \vdash \mathsf{rec}(M, N, xy.\ P) : T_N \sqcap T_P}$$

- Far from the declarative system
- Hard to prove

How to trust your typing algorithm?

Option 1

Prove equivalence:

$$\uparrow \Gamma\ M = \mathsf{Some}\ A \quad \text{iff} \quad \vdash M : A$$

- + the safest
- − tedious proof
- − non modular

How to trust your typing algorithm?

Option 2

Return a System $\mathsf{T}_{<:}$ derivation:

$$\uparrow \;:\; \mathsf{env} \to \mathsf{tm} \to \mathsf{tp} \times \mathsf{deriv}$$

Checked a posteriori:

$$\boldsymbol{kernel} \;:\; \mathsf{env} \to \mathsf{deriv} \to \mathsf{bool}$$

- − only *certifying* (not certified)
- + lightweight
- + evident witness of well-typing (PCC, . . .)

How to trust your typing algorithm?

Option 2

Return a System $\mathsf{T}_{<:}$ derivation:

$$\uparrow \;:\; \mathsf{env} \to \mathsf{tm} \to \mathsf{tp} \times \mathsf{deriv}$$

Checked a posteriori:

$$\boldsymbol{kernel} \;:\; \mathsf{env} \to \mathsf{deriv} \to \mathsf{bool}$$

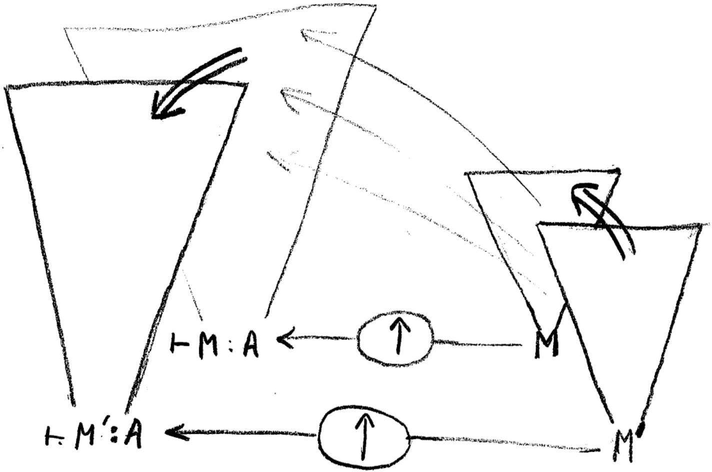
- − only *certifying* (not certified)
- \+ lightweight
- \+ evident witness of well-typing (PCC, . . .)

. . . but there is more

Observation

Let $\mathcal{D} = \uparrow M$.
Let M' be a slightly modified M.
Then $\mathcal{D}' = \uparrow M'$ is a slightly modified $\mathcal{D}$.

Observation

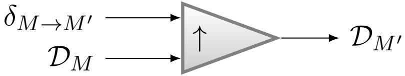


Back to Problem 1

Problem

How to take advantage of the knowledge from previous type-checks?

- Reuse pieces of a computed derivation $\mathcal{D}$
- Check only the changed part (the *delta*) of a program M



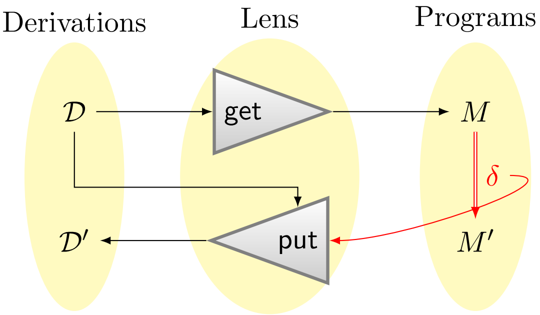
Requirements

- $\dfrac{\mathcal{D}_{M'}}{\vdash M' : A}$ iff $\uparrow(\mathcal{D}_M, \delta_{M \to M'}) = \mathcal{D}_{M'}$
- $\uparrow(\mathcal{D}_M, \delta_{M \to M'})$ computes $\mathcal{D}_{M'}$ in less than $O(|M'|)$

  (ideally $O(|\delta_{M \to M'}|)$)

Back to Problem 1

Bidirectional incremental updates



- $\mathsf{get}(\mathcal{D})$ projects derivation $\mathcal{D}$ to a program M
- $\mathsf{put}(\mathcal{D}, \delta)$ checks δ against $\mathcal{D}$ and returns $\mathcal{D}'$
  - the incremental type-checker
  - change-based approach
  - justification for each change ($\mathcal{D}'$)

Examples

initial term	$\textbf{let}\ f\,x = x + 1\ \textbf{in}\ f\ 3\ /\ 2$

Examples

initial term	**let** $f\,x = x + 1$ **in** $f\ 3\ /\ 2$
easy interleave	**let** $f\,x = 2 * (x + 1)$ **in** $f\ 3\ /\ 2$

Examples

initial term	$\mathbf{let}\ f\ x = x + 1\ \mathbf{in}\ f\ 3\ /\ 2$
easy interleave	$\mathbf{let}\ f\ x = 2\ *\ (x + 1)\ \mathbf{in}\ f\ 3\ /\ 2$
env interleave	$\mathbf{let}\ f\ x = (\mathbf{let}\ y = \mathrm{true}\ \mathbf{in}\ x + 1)\ \mathbf{in}\ f\ 3\ /\ 2$

Examples

initial term	$\textbf{let}\ f\ x = x + 1\ \textbf{in}\ f\ 3\ /\ 2$
easy interleave	$\textbf{let}\ f\ x = 2 * (x + 1)\ \textbf{in}\ f\ 3\ /\ 2$
env interleave	$\textbf{let}\ f\ x = (\textbf{let}\ y = \text{true}\ \textbf{in}\ x + 1)\ \textbf{in}\ f\ 3\ /\ 2$
type change	$\textbf{let}\ f\ x = x > 1\ \textbf{in}\ \boxed{f\ 3\ /\ 2}$

In this talk...

The message

Generating certificates of well-typing allows type checking incrementality by sharing pieces of derivations

The difficulty

Proofs are *higher-order objects* (binders, substitution property)

- What delta language?
- What data structure for derivations?
- What language to write synthesis algorithm?

In this talk...

The artifact

Gasp: a *language-independent* backend to develop certifying, incremental type checkers

syntax
typing rules
checker (untrusted)

⇝

incremental checker
tactic writing language
version control...

The open question

What else can we do with it?

Introduction

- How to make a type checker incremental?
- How to trust your type checker?

Using Gasp

- As a programmer
- The LF notation for derivations
- As a type system designer

The design of Gasp

- Data structures
- Typed evaluation algorithm

Introduction

- How to make a type checker incremental?
- How to trust your type checker?

Using Gasp

- As a programmer
- The LF notation for derivations
- As a type system designer

The design of Gasp

- Data structures
- Typed evaluation algorithm

Example

```
Gasp 0.1

#
```

Example

```
Gasp 0.1

#
```

$\uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)))$

Example

```
Gasp 0.1

#   ↑(rec(s(o), s(o), xy. s(x)))
```

$$\mathcal{D}_1 \;:\; \vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd} = \dfrac{\dfrac{}{\vdash \mathsf{o} : \mathsf{even}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd}}$$

Example

```
Gasp 0.1
```

`#` $\uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)))$

$$\mathcal{D}_1 \ : \ \vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd} = \frac{\overline{\vdash \mathsf{o} : \mathsf{even}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd}}$$

$$\mathcal{D}_2[\vdash x : \mathsf{nat}] \ : \ \vdash \mathsf{s}(x) : \mathsf{nat} = \frac{[\vdash x : \mathsf{nat}]}{\vdash \mathsf{s}(x) : \mathsf{nat}}$$

Example

```
Gasp 0.1
```

$$\# \quad \uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)))$$

$$\mathcal{D}_1 \;:\; \vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd} = \frac{\overline{\vdash \mathsf{o} : \mathsf{even}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd}}$$

$$\mathcal{D}_2[\vdash x : \mathsf{nat}] \;:\; \vdash \mathsf{s}(x) : \mathsf{nat} = \frac{[\vdash x : \mathsf{nat}]}{\vdash \mathsf{s}(x) : \mathsf{nat}}$$

$$\mathcal{D}_3 \;:\; \vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat} = \frac{\mathcal{D}_1 \qquad \overline{\vdash \mathsf{odd} \leq \mathsf{nat}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat}}$$

Example

```
Gasp 0.1
```

$$\# \quad \uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)))$$

$$\mathcal{D}_1 \;:\; \vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd} = \cfrac{\overline{\vdash \mathsf{o} : \mathsf{even}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd}}$$

$$\mathcal{D}_2[\vdash x : \mathsf{nat}] \;:\; \vdash \mathsf{s}(x) : \mathsf{nat} = \cfrac{[\vdash x : \mathsf{nat}]}{\vdash \mathsf{s}(x) : \mathsf{nat}}$$

$$\mathcal{D}_3 \;:\; \vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat} = \cfrac{\mathcal{D}_1 \qquad \overline{\vdash \mathsf{odd} \leq \mathsf{nat}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat}}$$

$$\boxed{\mathcal{D}_4} \;:\; \vdash \mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)) : \mathsf{nat} = \cfrac{\mathcal{D}_1 \qquad \mathcal{D}_3 \qquad \cfrac{[\vdash x : \mathsf{nat}]}{\mathcal{D}_2}}{\vdash \mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)) : \mathsf{na}}$$

Example

```
Gasp 0.1
```

$\uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)))$

$$\mathcal{D}_1 \ :\ \vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd} = \frac{\overline{\vdash \mathsf{o} : \mathsf{even}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd}}$$

$$\mathcal{D}_2[\vdash x : \mathsf{nat}] \ :\ \vdash \mathsf{s}(x) : \mathsf{nat} = \frac{[\vdash x : \mathsf{nat}]}{\vdash \mathsf{s}(x) : \mathsf{nat}}$$

$$\mathcal{D}_3 \ :\ \vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat} = \frac{\mathcal{D}_1 \qquad \overline{\vdash \mathsf{odd} \leq \mathsf{nat}}}{\vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat}}$$

$$\boxed{\mathcal{D}_4} \ :\ \vdash \mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)) : \mathsf{nat} = \frac{\mathcal{D}_1 \qquad \mathcal{D}_3 \qquad \dfrac{[\vdash x : \mathsf{nat}]}{\mathcal{D}_2}}{\vdash \mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)) : \mathsf{na}}$$

Functions

$\uparrow M$: derivation generator

Example

`#` $\uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{s}(\mathsf{o})), \mathsf{s}(\mathsf{o}), x\,y.\ \mathsf{s}(x)))$

Functions

$\uparrow M$: derivation generator

Example

$\uparrow(\mathsf{rec}(\mathsf{s}(\mathsf{s}(\mathsf{o})), \mathsf{s}(\mathsf{o}), x\,y.\ \mathsf{s}(x)))$

Functions
$\uparrow M$: derivation generator

Example

$\uparrow(\mathsf{rec}(\mathsf{s}(\mathcal{D}_1), \mathcal{D}_3, xy.\ \mathsf{s}(x)))$

Functions

$\uparrow M$: derivation generator

Example

$\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, xy.\ \mathsf{s}(x)))$

Functions

$\uparrow M$: derivation generator

$\downarrow\mathcal{D}$: coercion from derivation to the program it types

Example

$\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, xy.\downarrow\mathcal{D}_2))$

Functions

$\uparrow M$: derivation generator

$\downarrow\mathcal{D}$: coercion from derivation to the program it types

Example

\# $\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, xy.\downarrow\mathcal{D}_2[\uparrow x]))$

Functions

$\uparrow M$: derivation generator

$\downarrow\mathcal{D}$: coercion from derivation to the program it types

Example

$\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, xy.\ \downarrow\mathcal{D}_2[\uparrow x]))$

$$\begin{aligned} &\ldots \text{ (all of the above, plus:)} \\ \mathcal{D}_5 &: \vdash \mathsf{s}(\mathsf{s}(\mathsf{o})) : \mathsf{nat} = \ldots \\ \boxed{\mathcal{D}_6} &: \vdash \mathsf{rec}(\mathsf{s}(\mathsf{s}(\mathsf{o})), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)) : \mathsf{nat} = \ldots \end{aligned}$$

Functions

$\uparrow M$: derivation generator

$\downarrow\mathcal{D}$: coercion from derivation to the program it types

Example

$\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, xy.\ \downarrow\mathcal{D}_2[\uparrow x]))$

$$\begin{aligned} &\ldots \text{ (all of the above, plus:)} \\ \mathcal{D}_5 &: \vdash \mathsf{s}(\mathsf{s}(\mathsf{o})) : \mathsf{nat} = \ldots \\ \boxed{\mathcal{D}_6} &: \vdash \mathsf{rec}(\mathsf{s}(\mathsf{s}(\mathsf{o})), \mathsf{s}(\mathsf{o}), xy.\ \mathsf{s}(x)) : \mathsf{nat} = \ldots \end{aligned}$$

$\uparrow(\mathsf{rec}(\downarrow\mathcal{D}_5, \downarrow\mathcal{D}_3, xy.\ \downarrow\mathcal{D}_2[\mathcal{D}_2[\uparrow x]]))$

Functions

$\uparrow M$: derivation generator

$\downarrow\mathcal{D}$: coercion from derivation to the program it types

Methodology

- user inputs commands made of terms (programs), functions ($\uparrow$, $\downarrow$) and *contextual metavariables* $\mathcal{D}_i$
- to each function $A \rightarrow B$ there is an "inverse" $B \rightarrow A$ (put output back into input)
- system evaluates functions to value (derivations)
- checks value (kernel)
- extracts (from context) and names all subterms to a map (repository) for future reuse: *slicing*

What notation for derivations?

Preamble

- First-order *vs.* higher-order notations

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad vs. \qquad \begin{array}{c} [\vdash A] \\ \vdots \\ \dfrac{\vdash B}{\vdash A \to B} \end{array}$$

Explicit structural rules | Handled by the notation

What notation for derivations?

Preamble

- First-order *vs.* higher-order notations

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad vs. \qquad \begin{array}{c}[\vdash A] \\ \vdots \\ \dfrac{\vdash B}{\vdash A \to B}\end{array}$$

Explicit structural rules — Handled by the notation

- Local *vs.* global verification

$$\frac{\dfrac{\mathcal{D}_1}{\vdash A \to B \to C} \qquad \dfrac{\mathcal{D}_2}{\vdash A}}{\vdash B \to C} \qquad vs. \qquad M\ N$$

Can locally verify rule — Need M and N

What notation for derivations?

The LF notation

is a higher-order, local notation for derivations (and terms).
Comes with a small verification algorithm (typing)

Adequacy

in LF, a	is a	example
atomic type constant	syntactical category	$\mathsf{tm} : *$
family of types constant	judgement	$\mathsf{is} : \mathsf{tm} \to \mathsf{tp} \to *$
object constant	constructor or rule	$\mathsf{lam} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}$
applied object constant	rule application	
well-typed object	well-formed derivation	

Examples

- $\mathsf{is\_lam} : \Pi A{,}B : \mathsf{ty}.\ \Pi t : \mathsf{tm} \to \mathsf{tm}.$
  $(\Pi x : \mathsf{tm}.\ \mathsf{is}\ x\ A \to \mathsf{is}\ (t\ x)\ B) \to \mathsf{is}\ (\mathsf{lam}\ A\ (\lambda x.\ t\ x))(\mathsf{arr}\ A\ B)$
- $\mathsf{is\_lam}\ \mathsf{nat}\ \mathsf{nat}\ (\lambda x.\ x)\ (\lambda x\ h.\ \mathcal{D}[x, h])\ :$
  $\mathsf{is}\ (\mathsf{lam}\ \lambda x.\ {\downarrow}\mathcal{D})\ (\mathsf{arr}\ \mathsf{nat}\ \mathsf{nat})$

What notation for derivations?

Syntax

K	$::=$	$\Pi x : A.\ K \mid *$	Kind
A	$::=$	$\Pi x : A.\ A \mid P$	Type family
P	$::=$	$\mathsf{a}\ S$	Atomic type
M	$::=$	$\lambda x.\ M \mid F$	Canonical object
F	$::=$	$H\ S$	Atomic object
H	$::=$	$x \mid \mathsf{c}$	Head
S	$::=$	$\cdot \mid M\ S$	Spine

- The F are the *values* we want to manipulate.

What notation for derivations?

Syntax

$$
\begin{array}{rcll}
K & ::= & \Pi x : A.\ K \mid * & \text{Kind} \\
A & ::= & \Pi x : A.\ A \mid P & \text{Type family} \\
P & ::= & \mathsf{a}\ S & \text{Atomic type} \\
M & ::= & \lambda x.\ M \mid F & \text{Canonical object} \\
F & ::= & H\ S & \text{Atomic object} \\
H & ::= & x \mid \mathsf{c} & \text{Head} \\
S & ::= & \cdot \mid M\ S & \text{Spine}
\end{array}
$$

- The F are the *values* we want to manipulate.

. . . what are the computations?

How to write the unsafe type checker?

The computation language CL:

- an unsafe language to manipulate LF objects
- but with runtime check: each input & output of functions must be well-typed

Syntax

$$
\begin{array}{llll}
T & ::= & \lambda x.\, T \mid U & \text{Term} \\
U & ::= & F \mid \textbf{case } U \textbf{ in } \Gamma \textbf{ of } C & \text{Atomic term} \\
C & ::= & \cdot \mid C \mid P \to U & \text{Branches} \\
P & ::= & H\ x\ \ldots\ x & \text{Pattern}
\end{array}
$$

Example

$$\uparrow \; : \; \Pi M : \mathsf{tm}.\ \Sigma A : \mathsf{tp}.\ (\vdash M : A) =$$

Example

$$\uparrow \; : \; \Pi M : \mathsf{tm}.\ \Sigma A : \mathsf{tp}.\ (\vdash M : A) =$$
$$\lambda M.\ \textbf{case}\ \ M\ \ \textbf{of}$$

Example

$$
\begin{array}{l}
\uparrow\ :\ \Pi M : \mathsf{tm}.\ \Sigma A : \mathsf{tp}.\ (\vdash M : A) = \\
\lambda M.\ \mathbf{case}\ \ M\ \ \mathbf{of} \\
\quad \mid \mathsf{o} \rightarrow \langle \mathsf{even}, \dfrac{}{\vdash \mathsf{o} : \mathsf{even}} \rangle \\
\quad \mid \mathsf{s}(M) \rightarrow \mathbf{case}\ \uparrow M\ \ \mathbf{of} \\
\qquad \mid \langle \mathsf{even}, \mathcal{D} \rangle \rightarrow \langle \mathsf{odd}, \dfrac{\mathcal{D}}{\vdash \mathsf{s}\ M\ : \mathsf{odd}} \rangle \\
\qquad \mid \langle \mathsf{odd}, \mathcal{D} \rangle \rightarrow \langle \mathsf{even}, \dfrac{\mathcal{D}}{\vdash \mathsf{s}\ M\ : \mathsf{even}} \rangle \\
\qquad \mid \langle \mathsf{nat}, \mathcal{D} \rangle \rightarrow \langle \mathsf{nat}, \dfrac{\mathcal{D}}{\vdash \mathsf{s}\ M\ : \mathsf{nat}} \rangle
\end{array}
$$

Example

$$
\begin{array}{l}
\mid M\ N \to \\
\quad \textbf{let}\ \langle A_1 \to B, \mathcal{D}_1 \rangle = \uparrow M\ \ \textbf{in} \\
\quad \textbf{let}\ \langle A_2, \mathcal{D}_2 \rangle = \uparrow N\ \ \textbf{in} \\
\quad \textbf{let}\ \mathcal{D}_{\leq} = A_1 \leq A_2\ \ \textbf{in} \\
\quad \textbf{case}\ \ \mathcal{D}_{\leq}\ \ \textbf{of} \\
\qquad \mid \dfrac{}{\vdash A \leq A} \to \langle B, \dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash M\ N : B} \rangle \\
\qquad \mid \_ \to \langle B, \dfrac{\mathcal{D}_1 \qquad \dfrac{\mathcal{D}_2 \qquad \mathcal{D}_{\leq}}{\vdash N : A_1}}{\vdash M\ N : B} \rangle
\end{array}
$$

Functions

$\leq\ :\ \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ \vdash A \leq B = \ldots$

Example

$| \ \lambda x : A.\ M \to$

$| \ x \to$

Example

$$
\begin{array}{l}
\mid \lambda x : A.\ M \rightarrow \\
\quad \textbf{let}\ \langle B, \mathcal{D} \rangle = \\
\qquad \uparrow M\ \textbf{in}
\end{array}
$$

$$\langle A \rightarrow B, \frac{\mathcal{D}}{\vdash \lambda x.\ M : A \rightarrow B} \rangle$$

$$\mid x \rightarrow\ ???$$

Example

$$
\begin{array}{l}
\mid \lambda x : A.\ M \to \\
\quad \textbf{let}\ \langle B, \mathcal{D} \rangle = \\
\quad\quad \uparrow M[x / \downarrow \langle A, \mathcal{D}_x \rangle]\ \textbf{in} \\
\quad \langle A \to B, \dfrac{\begin{array}{c}[\mathcal{D}_x] \\ \mathcal{D}\end{array}}{\vdash \lambda x.\ M : A \to B} \rangle
\end{array}
$$

$$\sout{\mid x \to\ ???}$$

Note

$\uparrow\downarrow\langle A, \mathcal{D} \rangle = \langle A, \mathcal{D} \rangle$

Example

$$
\begin{array}{l}
\mid \lambda x : A.\ M \to \\
\quad \textbf{let}\ \langle B, \mathcal{D} \rangle\ \textbf{in}\ \mathcal{D}_x : (\vdash x : A) = \\
\qquad \uparrow M[x / \downarrow \langle A, \mathcal{D}_x \rangle]\ \textbf{in} \\
\quad \langle A \to B, \dfrac{\begin{array}{c}[\mathcal{D}_x] \\ \mathcal{D}\end{array}}{\vdash \lambda x.\ M : A \to B} \rangle
\end{array}
$$

~~$\mid x \to ???$~~

Note

$\uparrow\downarrow\langle A, \mathcal{D} \rangle = \langle A, \mathcal{D} \rangle$

Example

$$
\begin{array}{l}
\mid \mathsf{rec}(M, N, xy.\ P) \to \\
\quad \mathbf{let}\ \langle A_M, \mathcal{D}_M \rangle = {\uparrow} M \ \ \mathbf{in} \\
\quad \mathbf{let}\ \mathcal{D}_{A_M} = A_M \leq \mathsf{nat} \ \ \mathbf{in} \\
\quad \mathbf{let}\ \langle A_N, \mathcal{D}_N \rangle = {\uparrow} N \ \ \mathbf{in} \\
\quad \mathbf{let}\ \langle A_P, \mathcal{D}_P \rangle\ \mathbf{in}\ (\mathcal{D}_x : (\vdash x : \mathsf{nat}), \mathcal{D}_y : (\vdash y : A_N)) = \\
\qquad {\uparrow} P[x/{\downarrow}\langle \mathsf{nat}, \mathcal{D}_x \rangle, y/{\downarrow}\langle A_N, \mathcal{D}_y \rangle]\ \mathbf{in} \\
\quad \mathbf{let}\ \langle A, \langle \mathcal{D}_{A_N}, \mathcal{D}_{A_P} \rangle \rangle = A_N \sqcap A_P \ \ \mathbf{in} \\
\quad \mathbf{let}\ \langle \_, \mathcal{D}_P \rangle\ \mathbf{in}\ (\mathcal{D}_x : (\vdash x : \mathsf{nat}), \mathcal{D}_y : (\vdash y : A)) = \\
\qquad {\uparrow} P[x/{\downarrow}\langle \mathsf{nat}, \mathcal{D}_x \rangle, y/{\downarrow}\langle A, \mathcal{D}_y \rangle]\ \mathbf{in}
\end{array}
$$

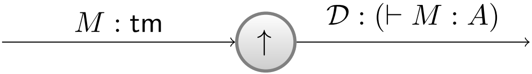
$$
\langle A, \dfrac{\dfrac{\mathcal{D}_M \qquad \mathcal{D}_{A_M}}{\vdash M : A} \qquad \dfrac{\mathcal{D}_N \qquad \mathcal{D}_{A_N}}{\vdash N : A} \qquad \dfrac{\begin{array}{c}[\mathcal{D}_x][\mathcal{D}_y] \\ \mathcal{D}_P\end{array} \qquad \mathcal{D}_{A_P}}{\vdash P : A}}{\vdash \mathsf{rec}(M, N, xy.\ P) : A} \rangle
$$

Functions

$\sqcap \ : \ \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ \Sigma C : \mathsf{tp}.\ (\vdash A \leq C) \times (\vdash B \leq C) = \ldots$

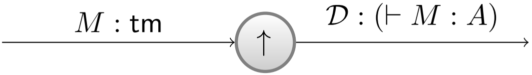
Discussion

- the "type" of a function is a kind of *contract*:



Discussion

- the "type" of a function is a kind of *contract*:



- "inverses" used to feed output back to input, same idea as *context-free* typing:

$$M \ ::= \ x \mid M\ M \mid \lambda x : A.\ M \mid \{x : A\}$$

$$\frac{\vdash M[x/\{x : A\}] : B}{\vdash \lambda x : A.\ M : A \to B} \qquad\qquad \frac{}{\vdash \{x : A\} : A}$$

Introduction
- How to make a type checker incremental?
- How to trust your type checker?

Using Gasp
- As a programmer
- The LF notation for derivations
- As a type system designer

The design of Gasp
- Data structures
- Typed evaluation algorithm

Sliced LF

Syntax

K	$::=$	$\Pi x : A.\ K \mid *$	Kind
A	$::=$	$\Pi x : A.\ A \mid P$	Type family
P	$::=$	$\mathsf{a}\ S$	Atomic type
M	$::=$	$\lambda x.\ M \mid F$	Canonical object
F	$::=$	$H\ S \mid X[\sigma]$	Atomic object
H	$::=$	$x \mid \mathsf{c} \mid \boldsymbol{f}$	Head
S	$::=$	$\cdot \mid M\ S$	Spine
σ	$::=$	$\cdot \mid \sigma, x/M$	Parallel substitution

- The $X[\sigma]$ stand for open objects (CMTT).
- The σ close them.
- The $\boldsymbol{f}$ are computations to do

Data structures

Signature

An object language is defined by a *signature*:

$$\Sigma \ ::= \ \cdot \ \big| \ \Sigma, \mathsf{a} : K \ \big| \ \Sigma, \mathsf{c} : A \ \big| \ \Sigma, \boldsymbol{f} : A = T$$

Repository

A *repository* is the sliced representation of an atomic object (`evar_map`):

$$\mathcal{R} \ : \ (X \mapsto (\Gamma \vdash F : P)) \times X[\sigma]$$

We define $\mathsf{co}(\mathcal{R})$ the operation of stripping out all metavariables

Inverse functions

To each $\boldsymbol{f} : A = T \in \Sigma$, associate a family of $\boldsymbol{f}^n : A^{-n} = T^{-n}$
Project out the n-th argument of $\boldsymbol{f}$

Examples

- $\boldsymbol{infer} : \Pi M : \mathsf{tm}.\ \Sigma A : \mathsf{tp}.\ \mathsf{is}\ M\ A = T$
  $\boldsymbol{infer}^0 : \Pi\{M\} : \mathsf{tm}.\ (\Sigma A : \mathsf{tp}.\ \mathsf{is}\ M\ A) \to \mathsf{tm} = \lambda x.\ \lambda y.\ x$
- $\boldsymbol{equal} : \Pi M : \mathsf{tm}.\ \Pi N : \mathsf{tm}.\ \mathsf{eq}\ M\ N = T'$
  $\boldsymbol{equal}^0 : \Pi\{M\} : \mathsf{tm}.\ \Pi\{N\} : \mathsf{tm}.\ \mathsf{eq}\ M\ N \to \mathsf{tm} =$
  $\lambda m.\ \lambda n.\ \lambda h.\ m$
  $\boldsymbol{equal}^1 : \Pi\{M\} : \mathsf{tm}.\ \Pi\{N\} : \mathsf{tm}.\ \mathsf{eq}\ M\ N \to \mathsf{tm} =$
  $\lambda m.\ \lambda n.\ \lambda h.\ n$

Evaluation

- $\boldsymbol{infer}\ (\boldsymbol{infer}^0 \langle A, \mathcal{D} \rangle) = \langle A, \mathcal{D} \rangle$
- $\boldsymbol{equal}\ (\boldsymbol{equal}^0\ \mathcal{D})\ (\boldsymbol{equal}^1\ \mathcal{D}) = \mathcal{D}$

The typed evaluation algorithm

In [P. & R-G., CPP'12], we define $\mathsf{ci}_{\mathcal{R}}(F)$:

- evaluates functions $\boldsymbol{f}$ in F
- checks F, functions arguments and return (w.r.t. type of $\boldsymbol{f}$)
- slices values in $\mathcal{R}$
- returns the enlarged $\mathcal{R}'$

Evaluation strategy

- We want strong reduction
  Example $\mathsf{lam}\ \lambda x.\ \boldsymbol{f}\ (\mathsf{s}(x))$ not a value

Evaluation strategy

- We want strong reduction
  Example $\mathsf{lam}\ \lambda x.\ \boldsymbol{f}\ (\mathsf{s}(x))$ not a value
- But not call-by-value
  Example $\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, x\,y.\ \downarrow\mathcal{D}_2[\uparrow x]))$

Evaluation strategy

- We want strong reduction
  Example $\mathsf{lam}\ \lambda x.\ \boldsymbol{f}\ (\mathsf{s}(x))$ not a value
- But not call-by-value
  Example $\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, x\,y.\ \downarrow\mathcal{D}_2[\uparrow x]))$
- And not call-by-name either
  Example $\uparrow(\boldsymbol{id}\ (\downarrow\mathcal{D})) \neq \mathcal{D}$

Evaluation strategy

- We want strong reduction
  Example $\mathsf{lam}\ \lambda x.\ \boldsymbol{f}\ (\mathsf{s}(x))$ not a value
- But not call-by-value
  Example $\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, x\,y.\ \downarrow\mathcal{D}_2[\uparrow x]))$
- And not call-by-name either
  Example $\uparrow(\boldsymbol{id}\ (\downarrow\mathcal{D})) \neq \mathcal{D}$

Evaluation strategy

- We want strong reduction
  Example $\mathsf{lam}\ \lambda x.\ \boldsymbol{f}\ (\mathsf{s}(x))$ not a value
- But not call-by-value
  Example $\uparrow(\mathsf{rec}(\mathsf{s}(\downarrow\mathcal{D}_1), \downarrow\mathcal{D}_3, xy.\downarrow\mathcal{D}_2[\uparrow x]))$
- And not call-by-name either
  Example $\uparrow(\boldsymbol{id}\ (\downarrow\mathcal{D})) \neq \mathcal{D}$

Ugly solution

Strong call-by-name *except* in function $\boldsymbol{f}$ argument position
$\rightsquigarrow$ weak head call-by-name *except* $\boldsymbol{f}^n$

Demo