

Certificates for incremental type checking

Matthias Puech^{1,2} Yann Régis-Gianas²

¹Dept. of Computer Science, University of Bologna

²Univ. 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

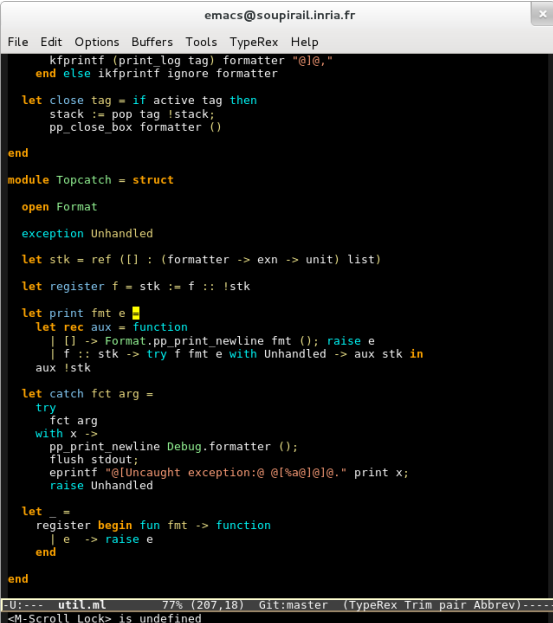
- Program elaboration is more and more an *interaction* between the programmer and the type-checker
- The richer the type system is, the more expensive type-checking gets

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

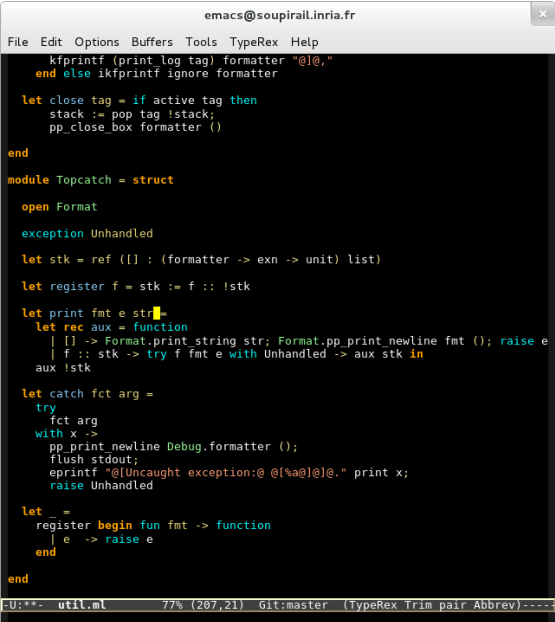
  let catch fct arg =
    try
      fct arg
    with x ->
      pp_print_newline Debug.formatter ();
      flush stdout;
      eprintf "[Uncaught exception:@ [%a@!@].]" print x;
      raise Unhandled

  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

```
emacs@soupirail.inria.fr
File Edit Options Buffers Tools TypeRex Help

end

module Topcatch = struct

  open Format

  exception Unhandled

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

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

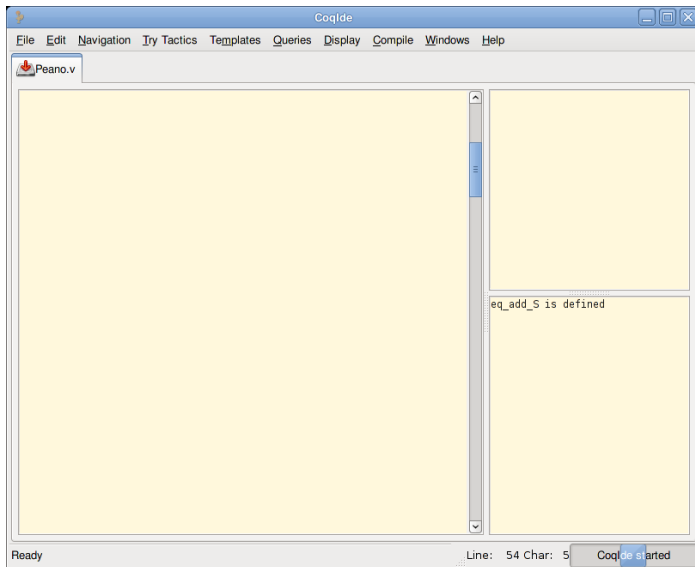
  let print fmt e str =
    let rec aux = function
      | [] -> Format.print_string str; Format.pp_print_newline fmt (); raise e
      | f :: stk -> try f fmt e with Unhandled -> aux stk in
      aux !stk

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

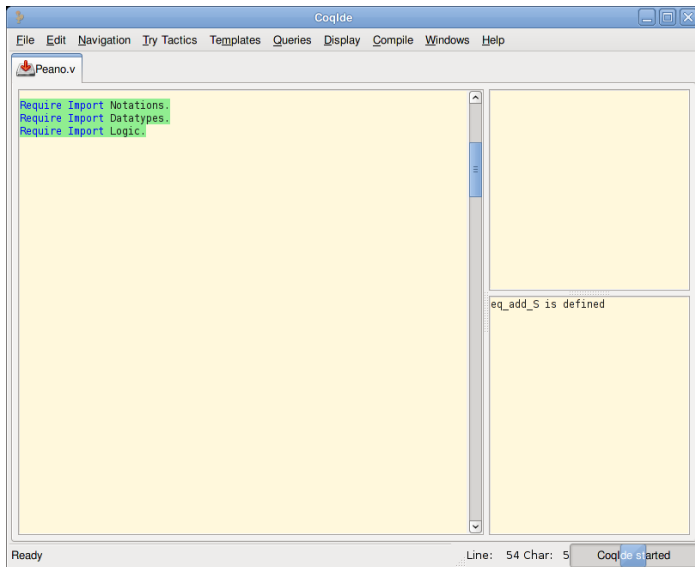
U:--- util.ml 80% (207,21) Git:master (TypeRex Trim pair Compiling At
/usr/bin/ocamlfind ocamldep -modules kernel.mli > kernel.mli.depends
/usr/bin/ocamlfind ocamlc -c -g -annot -o kernel.cmi kernel.mli
/usr/bin/ocamlfind ocamldep -modules kernel.ml > kernel.ml.depends
/usr/bin/ocamlfind ocamldep -modules version.mli > version.mli.depends
/usr/bin/ocamlfind ocamlc -c -g -annot -o version.cmi version.mli
/usr/bin/ocamlfind ocamldep -package 'camlp4.extend, camlp4.quotations' -syntax
camlp4o -modules pa_SLF.ml > pa_SLF.ml.depends
/usr/bin/ocamlfind ocamlc -c -g -annot -package 'camlp4.extend, camlp4.quotati
ons' -syntax camlp4o -o pa_SLF.cmo pa_SLF.ml
]

-U:%- *compilation* Bot (31,0) (Compilation:run pair Compiling)-----
(No files need saving)
```

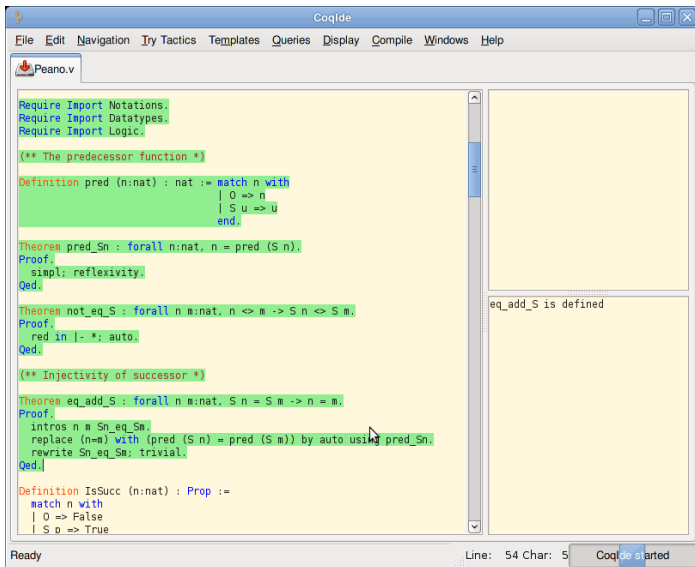

Certificates for incremental **type-checking**



Certificates for incremental **type-checking**



Certificates for incremental type-checking



CoqIDE

File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help

Peano.v

```
Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)
Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity.
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *. auto.
Qed.

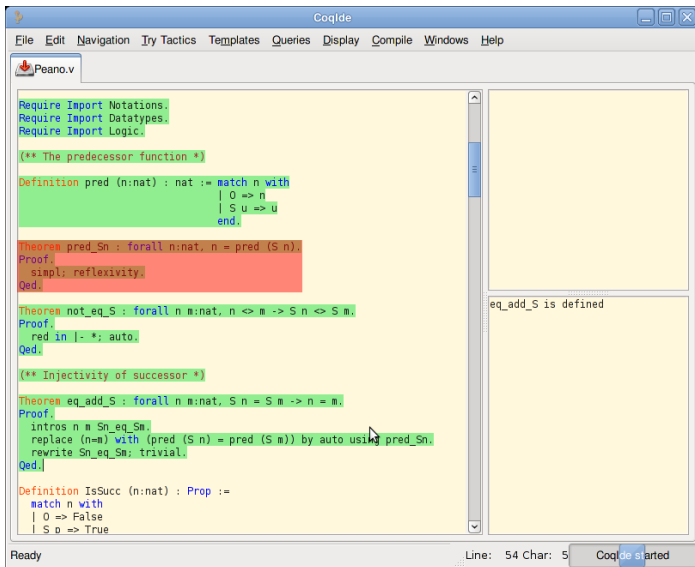
(** Injectivity of successor *)
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S o => True
```

eq_add_S is defined

Ready Line: 54 Char: 5 CoqIDE started

Certificates for incremental type-checking



CoqIDE

File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help

Peano.v

```
Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)
Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity.
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *. auto.
Qed.

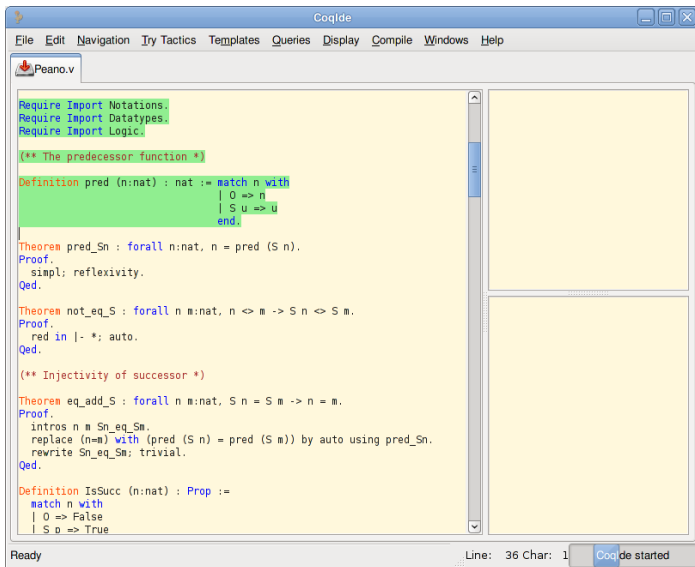
(** Injectivity of successor *)
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S o => True
```

eq_add_S is defined

Ready Line: 54 Char: 5 CoqIDE started

Certificates for incremental type-checking



```
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help

Peano.v

Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)

Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity.
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *. auto.
Qed.

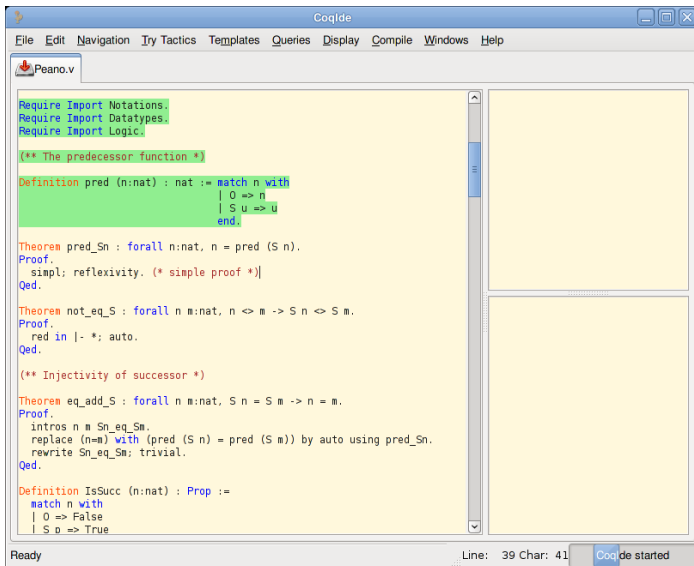
(** Injectivity of successor *)

Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S o => True
```

Ready Line: 36 Char: 1 CoqIDE started

Certificates for incremental type-checking



```
Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)
Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity. (* simple proof *)
Qed.

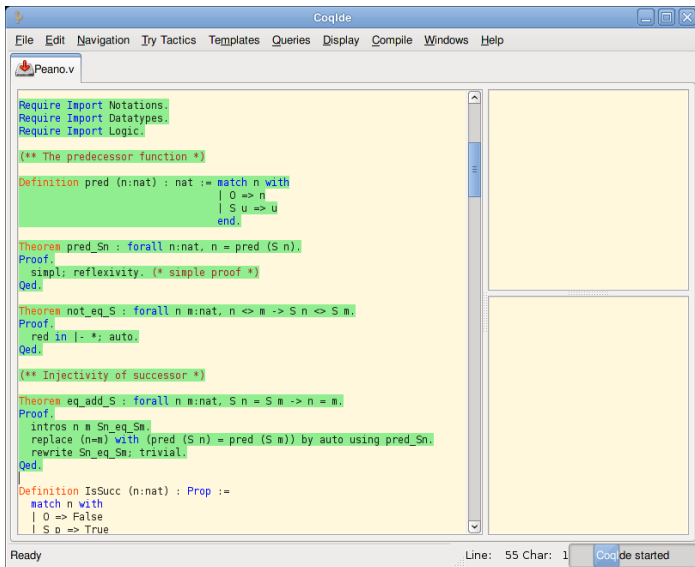
Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *. auto.
Qed.

(** Injectivity of successor *)
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S o => True
```

Ready Line: 39 Char: 41 CoqIDE started

Certificates for incremental type-checking



The screenshot shows the CoqIDE window with a file named 'Peano.v' open. The editor contains a Coq script for defining natural numbers and their properties. The script includes imports for Notations, Datatypes, and Logic. It defines a predecessor function 'pred' and proves several theorems: 'pred_Sn' (predecessor of a successor is the original number), 'not_eq_S' (if a number is not equal to a successor, then its predecessor is not equal to the predecessor of the successor), and 'eq_add_S' (if a number is equal to a successor, then its predecessor is equal to the predecessor of the successor). Finally, it defines 'IsSucc' as a proposition indicating if a number is a successor.

```
Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)
Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity. (* simple proof *)
Qed.

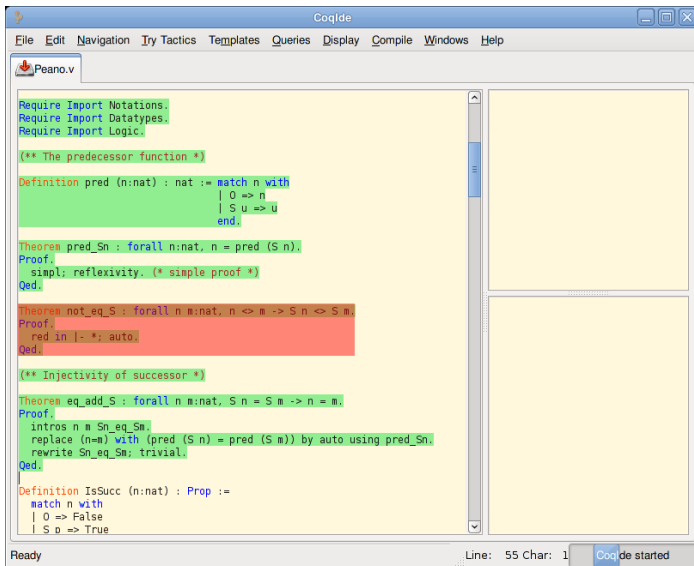
Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *. auto.
Qed.

(** Injectivity of successor *)
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S o => True
```

At the bottom of the window, the status bar shows 'Ready', 'Line: 55 Char: 1', and a button that says 'CoqIDE started'.

Certificates for incremental type-checking



The screenshot shows the CoqIDE window with a file named 'Peano.v' open. The editor contains Coq code for defining natural numbers and proving basic properties. The code is color-coded: blue for imports, green for comments and theorems, red for definitions and proofs, and orange for the final definition. The status bar at the bottom indicates 'Ready', 'Line: 55 Char: 1', and 'Coqde started'.

```
Require Import Notations.
Require Import Datatypes.
Require Import Logic.

(** The predecessor function *)
Definition pred (n:nat) : nat := match n with
| 0 => n
| S u => u
end.

Theorem pred_Sn : forall n:nat, n = pred (S n).
Proof.
  simpl; reflexivity. (* simple proof *)
Qed.

Theorem not_eq_S : forall n m:nat, n <> m -> S n <> S m.
Proof.
  red in |- *. auto.
Qed.

(** Injectivity of successor *)
Theorem eq_add_S : forall n m:nat, S n = S m -> n = m.
Proof.
  intros n m Sn_eq_Sm.
  replace (n=m) with (pred (S n) = pred (S m)) by auto using pred_Sn.
  rewrite Sn_eq_Sm; trivial.
Qed.

Definition IsSucc (n:nat) : Prop :=
  match n with
  | 0 => False
  | S o => True
```


Certificates for **incremental** type checking

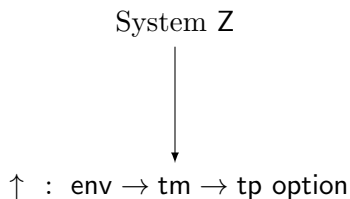
Problem

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*

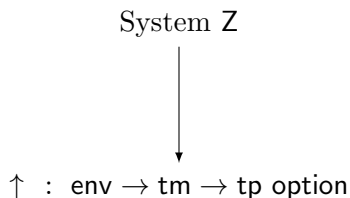
Problem 2: How to trust your type checker?

A compiler designer's job



set of declarative inference rules \rightarrow decision algorithm

A compiler designer's job



set of declarative inference rules \rightarrow decision algorithm

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

Example: System $T_{<}$:

Syntax

$$M ::= o \mid s(M) \mid MM \mid \lambda x. M \mid \text{rec}(M, N, xy. P)$$
$$A ::= \text{nat} \mid \text{even} \mid \text{odd} \mid A \rightarrow A$$

Typing rules

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

Example: System $T_{<}$:

Syntax

$$M ::= o \mid s(M) \mid MM \mid \lambda x. M \mid \text{rec}(M, N, xy. P)$$
$$A ::= \text{nat} \mid \text{even} \mid \text{odd} \mid A \rightarrow A$$

Typing rules

$$\frac{\begin{array}{ccc} \vdash M : \text{nat} & \vdash N : A & \begin{array}{c} [\vdash x : \text{nat}] \quad [\vdash y : A] \\ \vdots \\ \vdash P : A \end{array} \\ \hline \vdash \text{rec}(M, N, xy. P) : A \end{array}}{\frac{\vdash M : A \quad \vdash A \leq B}{\vdash M : B}}$$

Not syntax directed!

Example: System $T_{<}$:

Typing algorithm

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

Example: System $T_{<}$:

Typing algorithm

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \frac{\Gamma \vdash N : A' \quad \Gamma \vdash A' \leq A}{\Gamma \vdash N : A}}{\Gamma \vdash M N : B}$$

Example: System $T_{<}$:

Typing algorithm

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

Example: System $T_{<}$:

Typing algorithm

$$\frac{\begin{array}{c} \Gamma \vdash M : T_M \quad \Gamma \vdash T_M \leq \text{nat} \quad \Gamma \vdash N : T_N \\ \Gamma, x : \text{nat}, y : T_N \vdash P : T_P \\ \Gamma, x : \text{nat}, y : T_N \sqcap T_P \vdash P : T_N \sqcap T_P \end{array}}{\Gamma \vdash \text{rec}(M, N, xy. P) : T_N \sqcap T_P}$$

Example: System $T_{<}$:

Typing algorithm

$$\frac{\begin{array}{c} \Gamma \vdash M : T_M \quad \Gamma \vdash T_M \leq \text{nat} \quad \Gamma \vdash N : T_N \\ \Gamma, x : \text{nat}, y : T_N \vdash P : T_P \\ \Gamma, x : \text{nat}, y : T_N \sqcap T_P \vdash P : T_N \sqcap T_P \end{array}}{\Gamma \vdash \text{rec}(M, N, xy. P) : T_N \sqcap T_P}$$

- Far from the declarative system
- Hard to prove

How to trust your typing algorithm?

Option 1

Prove equivalence:

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

- + the safest
- tedious proof
- non modular

How to trust your typing algorithm?

Option 2

Return a System $T_{<}$: derivation:

$$\uparrow : \text{env} \rightarrow \text{tm} \rightarrow \text{tp} \times \text{deriv}$$

Checked a posteriori:

$$\textit{kernel} : \text{env} \rightarrow \text{deriv} \rightarrow \text{bool}$$

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

How to trust your typing algorithm?

Option 2

Return a System $T_{<}$: derivation:

$$\uparrow : \text{env} \rightarrow \text{tm} \rightarrow \text{tp} \times \text{deriv}$$

Checked a posteriori:

$$\textit{kernel} : \text{env} \rightarrow \text{deriv} \rightarrow \text{bool}$$

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

...but there is more

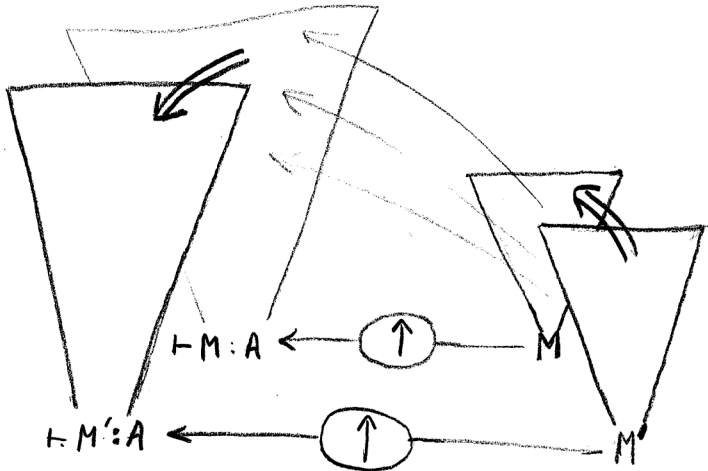
Observation

Let $\mathcal{D} = \uparrow M$.

Let M' be a slightly modified M .

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

Observation

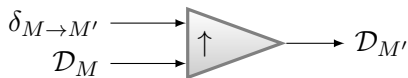


Back to Problem 1

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

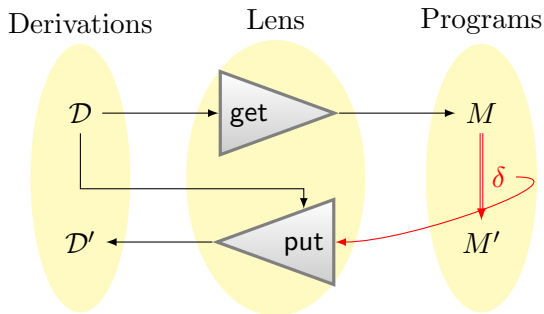


Requirements

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

Back to Problem 1

Bidirectional incremental updates



- $\text{get}(\mathcal{D})$ projects derivation \mathcal{D} to a program M
- $\text{put}(\mathcal{D}, \delta)$ 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		let $f\ x = x + 1$ 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 **let** $f\ x = x + 1$ **in** $f\ 3 / 2$

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

env interleave **let** $f\ x = (\text{let } y = \text{true in } x + 1)$ **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** $f\ 3 / 2$

env interleave **let** $f\ x = (\text{let } y = \text{true in } x + 1)$ **in** $f\ 3 / 2$

type change **let** $f\ x = x > 1$ **in** $f\ 3 / 2$

In this talk...

The message

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

The difficulty

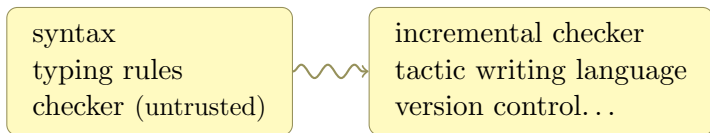
Proofs are *higher-order objects* (binders, substitution property)

- 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



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

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

Example

Gasp 0.1

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

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

Example

Gasp 0.1

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

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

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

Example

Gasp 0.1

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

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

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

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

Example

Gasp 0.1

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

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

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

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

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

Example

Gasp 0.1

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

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

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

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

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

Functions

$\uparrow M$: derivation generator

Example

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

Functions

$\uparrow M$: derivation generator

Example

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

Functions

$\uparrow M$: derivation generator

Example

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

Functions

$\uparrow M$: derivation generator

Example

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

Functions

$\uparrow M$: derivation generator

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

Example

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

Functions

$\uparrow M$: derivation generator

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

Example

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

Functions

$\uparrow M$: derivation generator

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

Example

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

... (all of the above, plus:)

$\mathcal{D}_5 : \vdash \text{s}(\text{s}(\text{o})) : \text{nat} = \dots$

$\boxed{\mathcal{D}_6} : \vdash \text{rec}(\text{s}(\text{s}(\text{o})), \text{s}(\text{o}), xy. \text{s}(x)) : \text{nat} = \dots$

Functions

$\uparrow M$: derivation generator

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

Example

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

... (all of the above, plus:)

$\mathcal{D}_5 : \vdash s(s(o)) : \text{nat} = \dots$

$\boxed{\mathcal{D}_6} : \vdash \text{rec}(s(s(o)), s(o), xy. s(x)) : \text{nat} = \dots$

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

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 $[\vdash A]$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \quad \text{vs.} \quad \frac{\vdots \quad \vdash B}{\vdash A \rightarrow B}$$

Explicit structural rules

Handled by the notation

What notation for derivations?

Preamble

- First-order *vs.* higher-order notations $[\vdash A]$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \quad \text{vs.} \quad \frac{\vdots}{\vdash A \rightarrow B}$$

Explicit structural rules

Handled by the notation

- Local *vs.* global verification

$$\frac{\frac{\mathcal{D}_1}{\vdash A \rightarrow B \rightarrow C} \quad \frac{\mathcal{D}_2}{\vdash A}}{\vdash B \rightarrow C} \quad \text{vs.} \quad M \ N$$

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	$\text{tm} : *$
family of types constant	judgement	$\text{is} : \text{tm} \rightarrow \text{tp} \rightarrow *$
object constant	constructor or rule	$\text{lam} : (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm}$
applied object constant	rule application	
well-typed object	well-formed derivation	

Examples

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

What notation for derivations?

Syntax

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

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

What notation for derivations?

Syntax

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

- 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

$T ::= \lambda x. T \mid U$	Term
$U ::= F \mid \mathbf{case} \ U \ \mathbf{in} \ \Gamma \ \mathbf{of} \ C$	Atomic term
$C ::= \cdot \mid C \mid P \rightarrow U$	Branches
$P ::= H \ x \ \dots \ x$	Pattern

Example

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

Example

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

Example

$$\begin{aligned} \uparrow & : \Pi M : \text{tm}. \Sigma A : \text{tp}. (\vdash M : A) = \\ \lambda M. & \text{ case } M \text{ of} \\ & | o \rightarrow \langle \text{even}, \frac{}{\vdash o : \text{even}} \rangle \\ & | s(M) \rightarrow \text{case } \uparrow M \text{ of} \\ & \quad | \langle \text{even}, \mathcal{D} \rangle \rightarrow \langle \text{odd}, \frac{\mathcal{D}}{\vdash s M : \text{odd}} \rangle \\ & \quad | \langle \text{odd}, \mathcal{D} \rangle \rightarrow \langle \text{even}, \frac{\mathcal{D}}{\vdash s M : \text{even}} \rangle \\ & \quad | \langle \text{nat}, \mathcal{D} \rangle \rightarrow \langle \text{nat}, \frac{\mathcal{D}}{\vdash s M : \text{nat}} \rangle \end{aligned}$$

Example

| $M \ N \rightarrow$
 let $\langle A_1 \rightarrow B, \mathcal{D}_1 \rangle = \uparrow M$ **in**
 let $\langle A_2, \mathcal{D}_2 \rangle = \uparrow N$ **in**
 let $\mathcal{D}_{\leq} = A_1 \leq A_2$ **in**
 case \mathcal{D}_{\leq} **of**
 | $\frac{}{\vdash A \leq A} \rightarrow \langle B, \frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\vdash M \ N : B} \rangle$
 | $\frac{}{-} \rightarrow \langle B, \frac{\mathcal{D}_1 \quad \frac{\mathcal{D}_2 \quad \mathcal{D}_{\leq}}{\vdash N : A_1}}{\vdash M \ N : B} \rangle$

Functions

\leq : $\Pi A : \text{tp. } \Pi B : \text{tp. } \vdash A \leq B = \dots$

Example

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

| $x \rightarrow$

Example

| $\lambda x : A. M \rightarrow$
 let $\langle B, \mathcal{D} \rangle =$
 $\uparrow M$ **in**

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

| $x \rightarrow ???$

Example

$$\begin{array}{l} | \lambda x : A. M \rightarrow \\ \quad \mathbf{let} \langle B, \mathcal{D} \rangle = \\ \quad \quad \uparrow M[x / \downarrow \langle A, \mathcal{D}_x \rangle] \mathbf{in} \\ \quad \quad \quad [\mathcal{D}_x] \\ \quad \quad \quad \mathcal{D} \\ \langle A \rightarrow B, \frac{\quad}{\vdash \lambda x. M : A \rightarrow B} \rangle \end{array}$$

$$\vdash x \rightarrow ???$$

Note

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

Example

$$\begin{array}{l} | \lambda x : A. M \rightarrow \\ \text{let } \langle B, \mathcal{D} \rangle \text{ in } \mathcal{D}_x : (\vdash x : A) = \\ \quad \uparrow M[x / \downarrow \langle A, \mathcal{D}_x \rangle] \text{ in} \\ \quad \quad \quad [\mathcal{D}_x] \\ \quad \quad \quad \mathcal{D} \\ \langle A \rightarrow B, \frac{\quad}{\vdash \lambda x. M : A \rightarrow B} \rangle \end{array}$$

$$\vdash x \rightarrow \text{???}$$

Note

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

Example

$| \text{rec}(M, N, xy. P) \rightarrow$
 $\text{let } \langle A_M, \mathcal{D}_M \rangle = \uparrow M \text{ in}$
 $\text{let } \mathcal{D}_{A_M} = A_M \leq \text{nat} \text{ in}$
 $\text{let } \langle A_N, \mathcal{D}_N \rangle = \uparrow N \text{ in}$
 $\text{let } \langle A_P, \mathcal{D}_P \rangle \text{ in } (\mathcal{D}_x : (\vdash x : \text{nat}), \mathcal{D}_y : (\vdash y : A_N)) =$
 $\uparrow P[x/\downarrow \langle \text{nat}, \mathcal{D}_x \rangle, y/\downarrow \langle A_N, \mathcal{D}_y \rangle] \text{ in}$
 $\text{let } \langle A, \langle \mathcal{D}_{A_N}, \mathcal{D}_{A_P} \rangle \rangle = A_N \sqcap A_P \text{ in}$
 $\text{let } \langle -, \mathcal{D}_P \rangle \text{ in } (\mathcal{D}_x : (\vdash x : \text{nat}), \mathcal{D}_y : (\vdash y : A)) =$
 $\uparrow P[x/\downarrow \langle \text{nat}, \mathcal{D}_x \rangle, y/\downarrow \langle A, \mathcal{D}_y \rangle] \text{ in}$

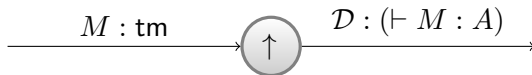
$$\begin{array}{c}
 \frac{\mathcal{D}_M \quad \mathcal{D}_{A_M}}{\vdash M : A} \quad \frac{\mathcal{D}_N \quad \mathcal{D}_{A_N}}{\vdash N : A} \quad \frac{[\mathcal{D}_x][\mathcal{D}_y] \quad \mathcal{D}_P \quad \mathcal{D}_{A_P}}{\vdash P : A} \\
 \langle A, \frac{\quad}{\vdash \text{rec}(M, N, xy.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

Functions

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

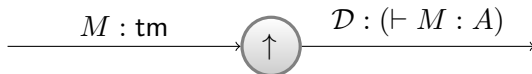
Discussion

- 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:

$$\begin{array}{c} M ::= x \mid M \mid M \mid \lambda x : A. M \mid \{x : A\} \\ \hline \frac{\vdash M[x/\{x : A\}] : B}{\vdash \lambda x : A. M : A \rightarrow B} \qquad \frac{}{\vdash \{x : A\} : A} \end{array}$$

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
$P ::= \mathbf{a} \ S$	Atomic type
$M ::= \lambda x. M \mid F$	Canonical object
$F ::= H \ S \mid \mathbf{X}[\sigma]$	Atomic object
$H ::= x \mid \mathbf{c} \mid \mathbf{f}$	Head
$S ::= \cdot \mid M \ S$	Spine
$\sigma ::= \cdot \mid \sigma, x/M$	Parallel substitution

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

Data structures

Signature

An object language is defined by a *signature*:

$$\Sigma ::= \cdot \mid \Sigma, \mathbf{a} : K \mid \Sigma, \mathbf{c} : A \mid \Sigma, \mathbf{f} : A = T$$

Repository

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

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

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

Inverse functions

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

Examples

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

Evaluation

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

The typed evaluation algorithm

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

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

Evaluation strategy

- We want strong reduction

Example $\text{lam } \lambda x. \textcolor{red}{f} (\textcolor{red}{s}(x))$ not a value

Evaluation strategy

- We want strong reduction

Example $\text{lam } \lambda x. \textcolor{red}{f} (\textcolor{red}{s}(x))$ not a value

- But not call-by-value

Example $\uparrow(\text{rec}(\textcolor{red}{s}(\downarrow \mathcal{D}_1), \downarrow \mathcal{D}_3, x y. \downarrow \mathcal{D}_2[\uparrow \textcolor{red}{x}])))$

Evaluation strategy

- We want strong reduction

Example $\text{lam } \lambda x. \textcolor{red}{f} (\textcolor{red}{s}(x))$ not a value

- But not call-by-value

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

- And not call-by-name either

Example $\uparrow(\textcolor{red}{id} (\downarrow \mathcal{D})) \neq \mathcal{D}$

Evaluation strategy

- We want strong reduction

Example $\text{lam } \lambda x. \textcolor{red}{f} (\textcolor{red}{s}(x))$ not a value

- But not call-by-value

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

- And not call-by-name either

Example $\uparrow(\textcolor{red}{id} (\downarrow \mathcal{D})) \neq \mathcal{D}$

Evaluation strategy

- We want strong reduction

Example $\text{lam } \lambda x. \textcolor{red}{f} (\textcolor{red}{s}(x))$ not a value

- But not call-by-value

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

- And not call-by-name either

Example $\uparrow(\textcolor{red}{id} (\downarrow \mathcal{D})) \neq \mathcal{D}$

Ugly solution

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

Conclusion

Demo