Gasp: an OCaml library for manipulating LF objects

Matthias Puech[1]

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, PiR2,
INRIA Paris-Rocquencourt, F-75205 Paris, France
Università di Bologna, Dipartimento di Informatica – Scienza e Ingegneria

Workshop on Formal Meta-Theory
LIX, March 6, 2013

[1] joint work with Yann Régis-Gianas

Outline

Outline

Motivations

Programming with proof certificates

Incremental type checking

Perspectives

Proof certificates

Witnesses of correctness of a computation,
independently verifiable by a small trusted program

- a specification of f, e.g. $\forall i.\, \exists o. f(i) = o$ and $\mathsf{P}(i, o)$
- an untrusted oracle computing $f(i) = \langle o, \pi \rangle$
- a trusted *kernel* deciding if π is a proof of $\mathsf{P}(i, o)$

$\rightsquigarrow$ reduces the *trusted base* of a computation

Examples

Trusted decision procedures

a.k.a *tactics* in interactive theorem provers (e.g. `Coq`)

- an untrusted procedure (e.g. `tauto`, `omega`)
- returns a proof term (e.g. in the CIC)
- verified at `Qed.` time by the *kernel* (De Bruijn criterion)

Examples

Certifying compilation

a.k.a *proof-carrying code* [Necula, 1997]

- source code that "doesn't go wrong"
- an untrusted compiler
- returns a proof that target code "doesn't go wrong" either
- verified by the client before executing

Examples

Safe type inference

a.k.a *type reconstruction* (e.g. Haskell→F_C)

- a (complex) type inference procedure

  val `infer`: `expr` → `bool`

Examples

Safe type inference

a.k.a *type reconstruction* (e.g. Haskell→F_C)

- a (complex) type inference procedure

  val `infer`: `expr` → `church`

- a (simpler) type *checking* procedure

  val `check` : `church` → `bool`

Examples

Safe type inference

a.k.a *type reconstruction* (e.g. Haskell→$\mathsf{F_C}$)

- a (simple) *declarative* type system

$$\frac{\vdash M : A' \qquad \vdash A' \leq A}{\vdash M : A}\ \textsc{Sub}$$

- a (complex) *syntax-directed*, equivalent version

$$\frac{\vdash M : A \to B \qquad \vdash N : A' \qquad \vdash A' \leq A}{\vdash M\,N : B}\ \textsc{AppSub}$$

Certifying software

certified
: program together with proof that it respects the specification on all input (Coq, Beluga...)

certifying
: black box, emits a proof certificate verifiable a posteriori, but not guaranteed to be correct

Certifying software

certified program together with proof that it respects the specification on all input (Coq, Beluga…)

certifying black box, emits a proof certificate verifiable a posteriori, but not guaranteed to be correct

Advantages of the certifying scheme

- same safety (but different quality of implementation)
- program source need not be revealed
- more lightweight (partial formalization)
  e.g. no verification of graph coloring, term indexing…

Representing syntax in LF [Harper et al., 1993]

LF is a universal *representation language*, for formal systems featuring hypothetical/parametrical reasoning
like HTML for structured documents

- systems (*resp.* derivations) encoded into *signatures* (*resp.* *objects*)
- dependently-typed λ-calculus ($\lambda\Pi$)
- *higher-order abstract syntax*

Representing syntax in LF [Harper et al., 1993]

Example (Encoding natural deductions)

$$\left(\begin{array}{c} \dfrac{\begin{array}{c}[\vdash A] \\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}\,\textsc{ImpI} \\[2em] \dfrac{\vdash A \supset B \qquad \vdash A}{\vdash B}\,\textsc{ImpE} \end{array} \right)^* = \left(\begin{array}{l} \mathsf{prop} : *. \\ \mathsf{p} : \mathsf{prop}.\ \mathsf{q} : \mathsf{prop}. \\ \mathsf{imp} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}. \\ \mathsf{pf} : \mathsf{prop} \to *. \\ \mathsf{ImpI} : \Pi A B : \mathsf{prop}. \\ \quad (\mathsf{pf}\ A \to \mathsf{pf}\ B) \to \mathsf{pf}\ (\mathsf{imp}\ A\ B). \\ \mathsf{ImpE} : \Pi A B : \mathsf{prop}. \\ \quad \mathsf{pf}\ (\mathsf{imp}\ A\ B) \to \mathsf{pf}\ A \to \mathsf{pf}\ B. \end{array} \right)$$

$$\left(\dfrac{\dfrac{[\vdash \mathsf{p}]}{\vdash \mathsf{q} \supset \mathsf{p}}\,\textsc{ImpI}}{\vdash \mathsf{p} \supset \mathsf{q} \supset \mathsf{p}}\,\textsc{ImpI} \right)^* = \left(\begin{array}{l} \mathsf{ImpI}\ \mathsf{p}\ (\mathsf{imp}\ \mathsf{q}\ \mathsf{p}) \\ \quad (\lambda x. \mathsf{ImpI}\ \mathsf{q}\ \mathsf{p}\ (\lambda y. x)) \end{array} \right)$$

Computing LF objects?

Question

How to write programs which *values* are LF *objects*?

Computing LF objects?

Question

How to write programs which *values* are LF *objects*?

In this talk...

An OCaml *library* to ease programming with proof certificates:

- general purpose functional PL
- large corpus of libraries
- only simply-typed (ADT)

Not a system, a facility to implement systems

Yet another
"last implementation of substitution & LF type-checking"

Outline

Outline

Gasp: an OCaml library to manipulate LF objects

An implementation of LF. . .

- a type of LF objects `obj`
- a type of signatures `sign`
- a concrete syntax with quotations and anti-quotations

Gasp: an OCaml library to manipulate LF objects

An implementation of LF…

- a type of LF objects `obj`
- a type of signatures `sign`
- a concrete syntax with quotations and anti-quotations

…where signatures can declare *functions symbols* f

- their code is untyped OCaml code (type `obj`)
  can use e.g. pattern-matching, exceptions, partiality…
- their LF types act as a specifications
  dynamically checked at run time

(Anti-)Quotations

supported by CamlP4:

- a parser for OCaml expressions
  `parsee` : `string` → Ocaml.expr
- a parser for LF objects and signatures
  `parseq` : `string` → OCaml.expr *(* of type obj *)*
- `parsee` replaces strings `s` between « ... » by `parseq s` (quotations)
- `parseq` replaces strings `s` between "..." by `parsee s` (antiquotations)

then use `parsee` to parse main program (`ocamlc -pp`)

(Anti-)Quotations

Examples

- $(\text{«}^{sign}\ \mathsf{prop} : *.\ \ \mathsf{imp} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}.\ \text{»}\ : \mathtt{sign})$

(Anti-)Quotations

Examples

- $(\text{«}^{sign}\ \mathsf{prop} : *.\ \ \mathsf{imp} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}.\ \text{»}\ : \mathtt{sign})$
  $\rightsquigarrow$ [("prop", KType);
  ("imp", Arr (Atom ("prop", [])) (Arr ...))]

(Anti-)Quotations

Examples

- $(\ll^{sign}\ \mathsf{prop} : *.\ \ \mathsf{imp} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}.\ \gg\ : \mathtt{sign})$
  $\rightsquigarrow\ \ [(\texttt{"prop"}, \mathsf{KType});$
  $(\texttt{"imp"}, \mathsf{Arr}\ (\mathsf{Atom}\ (\texttt{"prop"}, []))\ (\mathsf{Arr}\ \ldots))]$
- $\textbf{let}\ \mathtt{f} : \mathtt{obj} \to \mathtt{obj} = \textbf{fun}\ \mathtt{x} \to \ll \mathsf{imp}\ \text{"}\mathtt{x}\text{"}\ \text{"}\mathtt{x}\text{"} \gg$

(Anti-)Quotations

Examples

- $(\text{«}^{\mathit{sign}}\ \mathsf{prop} : *.\ \ \mathsf{imp} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}.\ \text{»}\ : \mathtt{sign})$
  $\rightsquigarrow\quad [(\texttt{"prop"}, \mathsf{KType});$
  $\quad (\texttt{"imp"}, \mathsf{Arr}\ (\mathsf{Atom}\ (\texttt{"prop"}, []))\ (\mathsf{Arr}\ \ldots))]$
- $\mathbf{let}\ \mathtt{f} : \mathtt{obj} \to \mathtt{obj} = \mathbf{fun}\ \mathtt{x} \to \text{«}\ \mathsf{imp}\ \text{"}\mathtt{x}\text{"}\ \text{"}\mathtt{x}\text{"}\ \text{»}$
  $\rightsquigarrow\quad \mathbf{let}\ \mathtt{f} : \mathtt{obj} \to \mathtt{obj} = \mathbf{fun}\ \mathtt{x} \to \mathsf{Atom}\ (\texttt{"imp"}, [\mathtt{x}, \mathtt{x}])$

(Anti-)Quotations

Examples

- $(\text{«}^{sign}$ prop : $*$. imp : prop $\to$ prop $\to$ prop. » : `sign`)
  $\leadsto$ [("`prop`", KType);
  ("`imp`", Arr (Atom ("`prop`", [])) (Arr ...))]
- **let** `f` : `obj` $\to$ `obj` = **fun** `x` $\to$ « imp "`x`" "`x`" »
  $\leadsto$ **let** `f` : `obj` $\to$ `obj` = **fun** `x` $\to$ Atom ("`imp`", [`x`, `x`])
- «conj ("`f` « disj p q »") ("`f` « p »") »

(Anti-)Quotations

Examples

- («^{sign} prop : ∗. imp : prop → prop → prop. » : `sign`)
  ⇝ [("`prop`", KType);
  ("`imp`", Arr (Atom ("`prop`", [])) (Arr ...))]
- **let** `f` : `obj` → `obj` = **fun** `x` → « imp "`x`" "`x`" »
  ⇝ **let** `f` : `obj` → `obj` = **fun** `x` → Atom ("`imp`", [`x`, `x`])
- «conj ("`f` « disj p q »") ("`f` « p »") »
  ⇝ Atom ("`conj`", [
  `f` (Atom ("`disj`", [Atom ("`p`", []), Atom ("`q`", [])]),
  `f` (Atom "`p`", [])
  ])

Declared functions

$$\Sigma \ ::= \ \cdot \ \big| \ \Sigma, \mathtt{c} : A \ \big| \ \Sigma, \mathtt{a} : K \ \big| \ \Sigma, f : A = \text{“}T\text{”}$$

… where T is an OCaml expression of type $|A|$:

$$|P| = \mathtt{obj}$$
$$|\Pi x : A. B| = \mathtt{obj} \to |B|$$

… and objects can refer to them:

$$H \ ::= \ x \ \big| \ \mathtt{c} \ \big| \ f$$

Specification A is checked at run time just before/after executing code T.

Gasp's interface

```
module Gasp = struct
  type obj
  type sign
  ...
  val empty : sign
  val (++) : sign → sign → sign
  val eval : ?env:env → sign → obj → obj
end
```

Examples

```
# let s = «sign
  tm : ∗.   app : tm → tm → tm.   lam : (tm → tm) → tm.
  eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;
```

Examples

```
# let s = «sign
  tm : *.   app : tm → tm → tm.   lam : (tm → tm) → tm.
  eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;

# eval s « lam (λx. eta x) » ;;
- : obj = « lam (λx. lam (λx'. app x x')) »
```

Examples

```
# let s = «sign
  tm : ∗.   app : tm → tm → tm.   lam : (tm → tm) → tm.
  eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;

# eval s « lam (λx. eta x) » ;;
- : obj = « lam (λx. lam (λx'. app x x')) »

# let s = s ++ «sign
  weak : tm → tm = " function
    | « lam "m" » → « lam "m" »
    | « app "m" "n" » →
        let « lam "p" » = « weak "m" » in
        « weak ("p" "n") »".
» ;;
```

Examples

```
# let s = «sign
  tm : *.   app : tm → tm → tm.   lam : (tm → tm) → tm.
  eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;

# eval s « lam (λx. eta x) » ;;
- : obj = « lam (λx. lam (λx'. app x x')) »

# let s = s ++ «sign
  weak : tm → tm = " function
    | « lam "m" » → « lam "m" »
    | « app "m" "n" » →
        let « lam "p" » = « weak "m" » in
        « weak ("p" "n") »".
» ;;

# eval s « weak (app (lam (λx.x)) (lam (λx.x))) » ;;
- : obj = « lam (λx.x) »
```

Examples

```
# let s = s ++ «sign
  tp : *.
  nat : tp.
  arr : tp → tp → tp.
  is : tm → tp → *.
  App : ΠM N : tm. ΠA B : tp.
     is M (arr A B) → is N A → is (app M N) B.
  Lam : ΠM : tm → tm. ΠA B : tp. ΠB : tp.
     (Πx : tm. is x A → is (M x) B) → is (lam A λu.M u) (arr A B).
  inf : tm → *.
  ex : ΠM : tm. ΠA : tp. is M A → inf M.

  infer : ΠM : tm. inf M = " ... ".
» ;;
```

Examples

```
# eval s « infer (lam nat λx.x) » ;;
- : obj = « ex (lam λx.x) (arr nat nat)
              (Lam (λx.x) nat nat (λx. λh.h)) »
```

Examples

```
# eval s « infer (lam nat λx.x) » ;;
- : obj = « ex (lam λx.x) (arr nat nat)
                (Lam (λx.x) nat nat (λx. λh.h)) »

# eval s « infer (lam nat λx. app x x) » ;;
Exception: Failure "non-functional application"
                                    (* my term is ill-typed *)
```

Examples

```
# eval s « infer (lam nat λx.x) » ;;
- : obj = « ex (lam λx.x) (arr nat nat)
                  (Lam (λx.x) nat nat (λx.λh.h)) »

# eval s « infer (lam nat λx.app x x) » ;;
Exception: Failure "non-functional application"
                                   (* my term is ill-typed *)

# eval s « infer (lam nat λx....) » ;;
Exception: Type_error(« λx.x »)
                                   (* there's a bug in my type checker *)
```

Computing on open objects?

Consider the *size* function $|\cdot|$ defined recursively on λ-terms:

$$
\begin{aligned}
|x| &= 0 \\
|\lambda x. M| &= |M| + 1 \\
|M\,N| &= |M| + |N| + 1
\end{aligned}
$$

Computing on open objects?

Consider the *size* function $|\cdot|$ defined recursively on λ-terms:

$$
\begin{aligned}
|x| &= 0 \\
|\lambda x. M| &= |M| + 1 \\
|M\,N| &= |M| + |N| + 1
\end{aligned}
$$

Can we code it as follows?

```
size : tm → nat = "fun m → match m with
  | « x » → « o »
  | « app "m" "n" » → « s (plus (size "m") (size "n")) »
  | « lam λx. "m" » → « s (size "m") » ".
```

Computing on open objects?

Consider the *size* function $|\cdot|$ defined recursively on λ-terms:

$$\begin{aligned} |x| &= 0 \\ |\lambda x. M| &= |M| + 1 \\ |M\,N| &= |M| + |N| + 1 \end{aligned}$$

Can we code it as follows?

```
size : tm → nat = "fun m → match m with
  | « x » → « o »
  | « app "m" "n" » → « s (plus (size "m") (size "n")) »
  | « lam λx. "m" » → « s (size "m") » ".
```

- m *has a free variable*
- *I have access to the name of variables*

Computing on open objects?

Example

Contextual types to the rescue. In Beluga*:*

```
rec size : {g:ctx} [g. tm] → [. nat] =
mlam g ⇒ fn t ⇒ case t of
| [g. #p ..] ⇒ [. o]
| [g. lam λx. T .. x] ⇒
   let [. N] = size [g, x:tm] [g, x. T .. x ] in [. s N]
| [g. app (T ..) (U ..)] ⇒
   let [. N] = plus (size [g] [g. T ..]) (size [g] [g. U ..]) in [. s N] ;
```

Environment of terms is tracked and checked throughout the term.
No encoding directly in OCaml

Inspiration

Traditional type-checking algorithm

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A . M : A \to B}\ \textsc{Lam} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}\ \textsc{App}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\mathit{infer} : \mathsf{env} \to \mathsf{tm} \to \mathsf{tp}$$

Inspiration

Environment-free algorithm [Geuvers et al., 2010, Boespflug, 2011]

$$\textsc{EFLam}\quad \frac{\vdash M[x/\mathit{infer}^0\ A] : B}{\vdash \lambda x : A. M : A \to B}$$

$$\textsc{EFApp}\quad \frac{\vdash M : A \to B \qquad \vdash N : A}{\vdash M\,N : B}$$

$$\textsc{EFAnnot}\quad \frac{}{\vdash \mathit{infer}^0\ A : A}$$

$$\mathit{infer} : \mathsf{tm} \to \mathsf{tp}$$
$$\mathit{infer}^0 : \mathsf{tp} \to \mathsf{tm}$$

"substitute variables by their computed type"

The *environment-free* style

Proposition

Consider *size* only called on closed objects (no variable case), introduce *function inverse* $size^0$: nat → tm feeding *output* back to *input*

$size$: tm → nat = "**fun** m → **match** m **with**
  | « app "m" "n" » → « s (*plus* (*size* "m") (*size* "n")) »
  | « lam "f" » → « s (*size* ("f" ($size^0$ o))) » ".

Add *contraction* to the reduction

$$size\ (size^0\ M) = M$$

The *environment-free* style

Generalization

The *environment-free* style: during computation, objects are closed by the expected result on their variables

- ✓ to each function $f : \mathsf{A} \to \mathsf{B}$, a function inverse $f^{0} : \mathsf{B} \to \mathsf{A}$
- ✓ the adequate reduction: $f \circ f^{0} = \mathrm{id}$
- ✓ all patterns are of the form $P ::= \text{“}\mathtt{x}\text{”} \mid \mathsf{c}\; P \ldots P$

The *environment-free* style

Example

Simple types inference, à la Church

```
# let s = s ++ «sign
  tp : ∗. nat : tp. arr : tp → tp → tp.

  infer : tm → tp = " function
    | « app "m" "n" » →
     match eval « infer "m" » with
        | « arr "a" "b" » when a = eval « infer "n" » → b
        | _ → failwith "non-functional application"
    | « lam "a" "f" » → « infer ("f" (infer⁰ "a")) »
» ;;
```

The *environment-free* style

Example

Naïve full evaluation

```
# let s = s ++ «sign
  full : tm → tm = " function
    | « app "m" "n" » → match eval « full "m" » with
        | « lam "f" » → « full ("f" "n") »
        | « app "m" "n'" » → « app (app "m" "n") "n'" »
    | « lam "f" » → « lam λx. ("f" (full⁰ x)) » ".
» ;;
```

n-ary inverses

Generalization to n-ary functions $f : A_0 \to \ldots \to A_n \to A$

- n inverses $f^0 : A \to A_0, \ldots, f^n : A \to A_n$
- reduction rule: $f\ (f^0\ M)\ \ldots\ (f^n\ M) = M$

Example

```
# let s = s ++ «sign
 equals : tm → tm → bool = " fun m n → match m, n with
   | « app "m1" "m2" », « app "n1" "n2" » →
       « and (equals "m1" "n1") (equals "m2" "n2") »
   | « lam "f" », « lam "g" » →
       « equals ("f" (equals^0 true)) ("g" (equals^1 true)) » ".
» ;;
```

Dependent inverses

Problem

What is the inverse of $\mathit{infer} : \Pi x : \mathsf{tm}.\ \mathsf{inf}\ x$?

Dependent inverses

Problem

What is the inverse of $\mathit{infer} : \Pi x : \mathsf{tm}.\ \mathsf{inf}\ x$?

Proposition

$$\mathit{infer}^{0} : \qquad \mathsf{inf}\ x \to \mathsf{tm}$$

Dependent inverses

Problem

What is the inverse of $\mathit{infer} : \Pi x : \mathsf{tm}.\ \mathsf{inf}\ x$?

Proposition

Generalization: abstract by dependent arguments

$$\mathit{infer}^{0} : \Pi x : \mathsf{tm}.\ \mathsf{inf}\ x \to \mathsf{tm}$$

Dependent inverses

Example

$\mathit{infer} : \Pi M : \mathsf{tm}.\ \mathsf{inf}\ M =$ " **fun** `m` → **match** `m` **with**

  | « app "m" "n" » →

    let « ex "_" (arr "a" "b") "d1" » = `eval` « *infer* "m" » **in**

    let « ex "_" "a'" "d2" » = `eval` « *infer* "n" » **in**

    « ex (app "m" "n") "b"

        (App "m" "n" "a" "b" "d1" "d2") »

  | « lam "a" "m" » →

    let « ex "_" "b" "d" » = `eval ~env:«`^{env} x : tm; h : is x "a" »

      « *infer* ("m" (infer^0 x (ex x "a" h))) » **in**

    « ex (lam "a" "m") (arr "a" "b")

        (Lam "m" "a" "b" ($\lambda x.\lambda h.$"d")) »

Outline

Safe renaming & substitution in Gasp

With bare OCaml functions

```
# let eta m = « lam λx. (app "m" x) » ;;
eta : obj → obj = <fun>
```

Safe renaming & substitution in Gasp

With bare OCaml functions

```
# let eta m = « lam λx. (app "m" x) » ;;
eta : obj → obj = <fun>

# « lam (λx. "eta « x »") » ;;
```

Safe renaming & substitution in Gasp

With bare OCaml functions

```
# let eta m = « lam λx. (app "m" x) » ;;
eta : obj → obj = <fun>
```

```
# « lam (λx. "eta « x »") » ;;
- : obj = « lam (λx. lam (λx. app x x)) »   (* x captured *)
```

Safe renaming & substitution in Gasp

With bare OCaml functions

```
# let eta m = « lam λx. (app "m" x) » ;;
eta : obj → obj = <fun>
```

```
# « lam (λx. "eta « x »") » ;;
- : obj = « lam (λx. lam (λx. app x x)) »   (* x captured *)
```

With Gasp functions

```
# let s = s ++ «sign
   eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;
s : sign = ...
```

Safe renaming & substitution in Gasp

With bare OCaml functions

```
# let eta m = « lam λx. (app "m" x) » ;;
eta : obj → obj = <fun>

# « lam (λx. "eta « x »") » ;;
- : obj = « lam (λx. lam (λx. app x x)) »   (* x captured *)
```

With Gasp functions

```
# let s = s ++ «sign
   eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;
s : sign = ...

# eval s « lam (λx. eta x) » ;;
```

Safe renaming & substitution in Gasp

With bare OCaml functions

```
# let eta m = « lam λx. (app "m" x) » ;;
eta : obj → obj = <fun>

# « lam (λx. "eta « x »") » ;;
- : obj = « lam (λx. lam (λx. app x x)) »   (* x captured *)
```

With Gasp functions

```
# let s = s ++ «sign
    eta : tm → tm = "fun m → « lam (λx. app "m" x) »".
» ;;
s : sign = ...

# eval s « lam (λx. eta x) » ;;
- : obj = « lam (λx. lam (λx'. app x x')) »   (* x protected *)
```

Two-level, *locally named* term representation

Abstract level

Abstract objects `aobj` are represented by standard *canonical*, *spine-form* λ-terms

$$
\begin{aligned}
M &::= \lambda x.M \;\big|\; H(S) \\
H &::= \mathsf{c} \;\big|\; f \;\big|\; f^{n} \;\big|\; \#n \\
S &::= \cdot \;\big|\; M, S
\end{aligned}
$$

- variables are *numbered* (De Bruijn indices)
- no *free* variables
- abstract objects are *type-checked*

Two-level, *locally named* term representation

Concrete level

Concrete objects `obj` are represented by usual (non-canonical) λ-terms with two kinds of variables:

$$T \ ::= \ \mathtt{id} \ \big| \ \#n \ \big| \ \lambda\mathtt{x}.\,T \ \big| \ T\ T$$

- free variables n are *numbered* (protected against capture)
- bound variables/constants *id* are *named* (written by the user)
- concrete objects are *parsed* and *computed* by functions

Two-level, *locally named* term representation

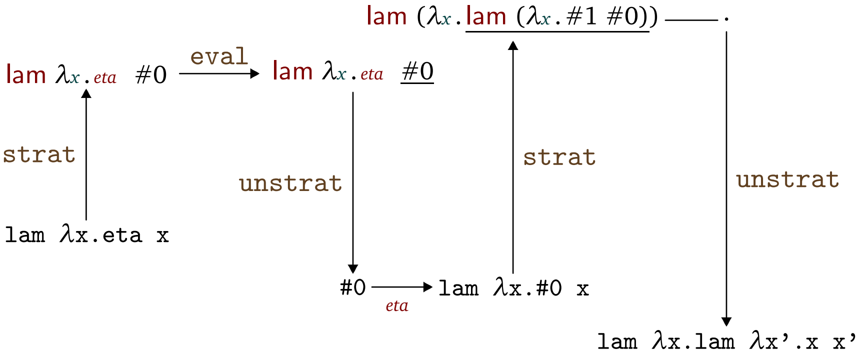
(Un)Stratification

$$\begin{tikzcd} \text{Abstract objects } \texttt{aobj} \arrow[d, bend left=60, "\texttt{unstrat}"] \\ \text{Concrete objects } \texttt{obj} \arrow[u, bend left=60, "\texttt{strat}"] \end{tikzcd}$$

- `strat` : `sign` → `env` → `obj` → `aobj`
  - distinguishes constants/functions/variables
  - normalizes object
    (hereditary substitutions, e.g. *eval* ("`p`" "`n`"))
- `unstrat` : `env` → `aobj` → `obj`
  - distinguishes free/bound variables
  - freshens names

Two-level, *locally named* term representation

Example



Gasp's *typed evaluation*

Evaluation

How to evaluate an object containing function symbols?

- full evaluation, e.g. «$\mathsf{lam}\ (\lambda x.\,eta\ x)$»
- call-by-value (*contraction*), e.g. « $size\ (id\ (size^{0}\ \mathsf{o}))$ »
- weak evaluation first, e.g. « $f\ (\mathsf{lam}\ \lambda x.f\ x)$»

Gasp's *typed evaluation*

Evaluation

How to evaluate an object containing function symbols?

- full evaluation, e.g. «$\mathsf{lam}\ (\lambda x.\, \mathit{eta}\ x)$»
- call-by-value (*contraction*), e.g. « $\mathit{size}\ (\mathit{id}\ (\mathit{size}^0\ \mathsf{o}))$ »
- weak evaluation first, e.g. « $f\ (\mathsf{lam}\ \lambda x. f\ x)$»

⇝ *"full evaluation by iterated symbolic weak evaluation"*, a.k.a *normalization-by-evaluation*
(adapted from Grégoire and Leroy [2002])

Gasp's *typed evaluation*

Typing

When to type-check the LF objects?

- checking the objects *a posteriori* is not precise enough
  e.g. $\mid$ « app "m" "n" » $\rightarrow$ « z (*plus* (*size* "m") (*size* "n")) »
- errors must be detected early (during prototyping)
  # `eval` « *size* (lam $\lambda x.$ app x (app x x))» $\rightsquigarrow$ « z (z (z o)) »

We want to identify failures as soon as possible

Gasp's *typed evaluation*

Typing

When to type-check the LF objects?

- checking the objects *a posteriori* is not precise enough
  e.g. $\mid$ « app "m" "n" » → « z (*plus* (*size* "m") (*size* "n")) »
- errors must be detected early (during prototyping)
  \# `eval` « *size* (lam λx. app x (app x x))» ⇝ « z (z (z o)) »

We want to identify failures as soon as possible

⇝ *typed evaluation*: typing and evaluation are the same process `eval` ("on-the-fly", dynamic typing?)

- ✓ inputs and outputs of functions are type-checked before and after execution
- ✓ issued certificates are guaranteed to be correct
- ✓ errors are signaled *where* the certificate is ill-typed

Gasp's *typed evaluation*

Overview

Judgments:

- $\Gamma \vdash M : A \downarrow M'$: weak typed evaluation
- $\Gamma \vdash M : A \Downarrow M'$: full typed evaluation
- $\Gamma \vdash M : A \uparrow M'$: readback

$$\frac{f : A = \text{“}T\text{”} \in \Sigma \qquad \Gamma; A \vdash S \downarrow f^0(S), \ldots, f^n(S) : P}{\Gamma \vdash f(S) \downarrow \pi_n(A) \star S : P}\ \textsc{FEvalInv}$$

$$\frac{\begin{array}{ll} f : A = \text{“}T\text{”} \in \Sigma & S' \neq f^0(S_0), \ldots, f^n(S_n) \\ \Gamma; A \vdash S \downarrow S' : P & \Gamma \vdash T \star S' \downarrow F : P \end{array}}{\Gamma \vdash f(S) \downarrow F : P}\ \textsc{FEval}$$

Outline

Interaction in typed program elaboration

Observations

- typed program elaboration an *interaction*

$$\text{programmer} \longleftrightarrow \text{type checker}$$

- the richer the type system is, the more expensive type checking gets (e.g. Haskell, Agda)
- typing is a *batch process* (part of compilation)
- yet, it is fed repeatedly with similar input (versions)

Interaction in typed program elaboration

Example

emacs@soupirail.inria.fr

```
    | '4' -> "₄"
    | '5' -> "₅"
    | '6' -> "₆"
    | '7' -> "₇"
    | '8' -> "₈"
    | '9' -> "₉"
    | _ -> invalid_arg "subscript_of_char"

  let subscript_of_int n =
    let s = string_of_int n in
    let rec loop i =
      try
        let x = subscript_of_char (String.get s i) in
        x ^ loop (succ i)
      with Invalid_argument _ -> ""
    in loop 0
  ;;

let split c s =
  let len = String.length s in
  let rec split n =
    try
      let pos = String.index_from s n c in
      let dir = String.sub s n (pos-n) in
      dir :: split (succ pos)
    with
U:---  util.ml      30% L83   Git:master  (Tuareg +3 Abbrev)
+,-,0 for further adjustment:
```

Interaction in typed program elaboration

Example

emacs@soupirail.inria.fr

```
    | '4' -> "₄"
    | '5' -> "₅"
    | '6' -> "₆"
    | '7' -> "₇"
    | '8' -> "₈"
    | '9' -> "₉"
    | _ -> invalid_arg "subscript_of_char"

  let subscript_of_int m n =
    let s = string_of_int n in
    let rec loop i =
      try
        let x = subscript_of_char (String.get s i) in
        x ^ loop (succ i)
      with Invalid_argument _ -> ""
    in loop m
  ;;

let split c s =
  let len = String.length s in
  let rec split n =
    try
      let pos = String.index_from s n c in
      let dir = String.sub s n (pos-n) in
      dir :: split (succ pos)
    with
U:**-  util.ml      30% L83   Git:master  (Tuareg +3 Abbrev)
Quit
```

Interaction in typed program elaboration

Example

Incremental type checking

Question

How can we make type checking *incremental*?

Definition

Given a list of well-typed programs M_0, M_1,… M and the representation of a change δ, decide whether $apply(M, \delta)$ is well-typed in less than $O(|apply(M, \delta)|)$.

Incremental type checking

Question

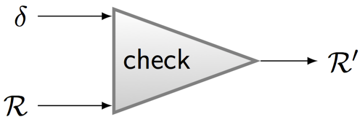
How can we make type checking *incremental*?

Definition

Given a list of well-typed programs M_0, M_1, ... M and the representation of a change δ, decide whether $apply(M, \delta)$ is well-typed in less than $O(|apply(M, \delta)|)$.

Hint

- save intermediate type information between runs (*context*)
- use this information in changes



Incrementality by derivation reuse

Proposition

The witness of type checking is a derivation: use it as context

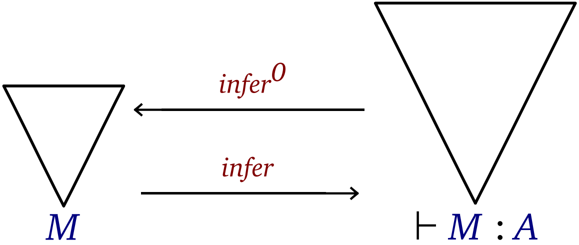
$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{[\vdash f : \mathsf{nat} \to \mathsf{nat}] \quad [\vdash x : \mathsf{nat}]}{\vdash f\,x : \mathsf{nat}}
      }{\vdash \lambda x.f\,x : \mathsf{nat} \to \mathsf{nat}}
    }{\vdash \lambda f\,x.f\,x : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}}
    \quad
    \dfrac{
      \dfrac{[\vdash x : \mathsf{nat}]}{\vdash \mathbf{s}(x) : \mathsf{nat}}
    }{\vdash \lambda x.\,\mathbf{s}(x) : \mathsf{nat} \to \mathsf{nat}}
  }{\vdash (\lambda f\,x.f\,x)\,(\lambda x.\,\mathbf{s}(x)) : \mathsf{nat}}
  \quad
  \dfrac{\dfrac{}{\vdash \mathbf{o} : \mathsf{nat}}}{\vdash \mathbf{s}(\mathbf{o}) : \mathsf{nat}}
}{\vdash (\lambda f\,x.f\,x)\,(\lambda x.\,\mathbf{s}(x))\,(\mathbf{s}(\mathbf{o})) : \mathsf{nat}}
$$

- it contains all intermediate type information
- it is *compositional*

Incrementality by derivation reuse

Proposition

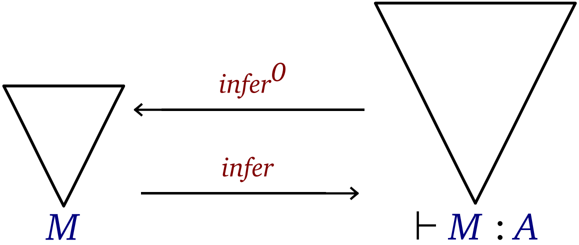
A certifying type checker in Gasp computes pieces of derivations



Incrementality by derivation reuse

Proposition

A certifying type checker in Gasp computes pieces of derivations



We need a way to

- address any *subderivation*
- reuse them in *programs* M using inverses

Naming and sharing LF objects

Contribution

- a conservative extension of LF based on *Contextual Modal Type Theory* [Nanevski et al., 2008] where objects are *sliced* in a *context* Δ of *metavariables* X
- every well-typed applicative subterm gets a metavariable *name* and can be reused by *instantiation*

Naming and sharing LF objects

Contribution

- a conservative extension of LF based on *Contextual Modal Type Theory* [Nanevski et al., 2008] where objects are *sliced* in a *context* Δ of *metavariables* X
- every well-typed applicative subterm gets a metavariable *name* and can be reused by *instantiation*

Example

The object $\mathsf{lam}\,(\lambda x.\,\mathsf{lam}\,(\lambda y.\,\mathsf{app}\,(x, \mathsf{app}\,(x,y))))$
is sliced into X
in the context

$$\Delta = \left(\begin{array}{l} X : \mathsf{tm} = \mathsf{lam}\,(\lambda x.\,Y[x/x]) \\ Y[x : \mathsf{tm}] : \mathsf{tm} = \mathsf{lam}\,(\lambda y.\,Z[x/x, y/\mathsf{app}\,(x,y)]) \\ Z[x : \mathsf{tm}, y : \mathsf{tm}] : \mathsf{tm} = \mathsf{app}\,(x,y) \end{array}\right)$$

Example

$\#$ $\mathit{infer}\ ((\lambda f.\, \lambda x. f\ x)\ (\lambda y.\, \mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

Example

$\#$ $\mathit{infer}\ ((\lambda f.\, \lambda x. f\ x)\ (\lambda y.\, \mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

$$\dfrac{X}{\vdash \lambda f.\, \lambda x. f\ x : \mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{nat}} \qquad \dfrac{Y}{\vdash \lambda y.\, \mathsf{s}\ y : \mathsf{nat}{\to}\mathsf{nat}}$$

$$\dfrac{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}} \qquad \begin{array}{c}[\vdash y : \mathsf{nat}] \\ \dfrac{T}{\vdash \mathsf{s}\ y : \mathsf{nat}}\end{array}$$

Example

$\#$ *infer* $((\lambda f.\, \lambda x. f\ x)\ (\lambda y.\, \mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

$$\frac{X}{\vdash \lambda f.\, \lambda x. f\ x : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}} \qquad \frac{Y}{\vdash \lambda y.\, \mathsf{s}\ y : \mathsf{nat} \to \mathsf{nat}}$$

$$\frac{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}} \qquad \begin{array}{c}[\vdash y : \mathsf{nat}] \\ T \\ \vdash \mathsf{s}\ y : \mathsf{nat}\end{array}$$

$\#$ *infer* $((\lambda f.\, \lambda x. f\ x)\ (\lambda y.\, \mathsf{s}\ y)\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})))$

Example

$\#$ $\textit{infer}\ ((\lambda f.\, \lambda x. f\ x)\ (\lambda y. \mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

$$\frac{X}{\vdash \lambda f.\, \lambda x. f\ x : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}} \qquad \frac{Y}{\vdash \lambda y. \mathsf{s}\ y : \mathsf{nat} \to \mathsf{nat}}$$

$$[\vdash y : \mathsf{nat}]$$

$$\frac{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}} \qquad \frac{T}{\vdash \mathsf{s}\ y : \mathsf{nat}}$$

$\#$ $\textit{infer}\ (\ X\ Y\ (\mathsf{s}\ Z))$

Example

\# $\mathit{infer}\ ((\lambda f.\, \lambda x. f\ x)\ (\lambda y.\, \mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

$$\frac{X}{\vdash \lambda f.\, \lambda x. f\ x : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}} \qquad \frac{\begin{array}{c}[\vdash y : \mathsf{nat}] \\ T \\ \vdash \mathsf{s}\ y : \mathsf{nat}\end{array}}{\vdash \lambda y.\, \mathsf{s}\ y : \mathsf{nat} \to \mathsf{nat}}\ Y$$

$$\frac{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}}$$

\# $\mathit{infer}\ ((\mathit{infer}^0\ X)\ (\mathit{infer}^0\ Y)\ (\mathsf{s}\ (\mathit{infer}^0\ Z)))$

Example

$\#\;\; \mathit{infer}\ ((\lambda f.\, \lambda x.\, f\ x)\ (\lambda y.\, \mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

$$\dfrac{X}{\vdash \lambda f.\, \lambda x.\, f\ x : \mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{nat}} \qquad \dfrac{Y}{\vdash \lambda y.\, \mathsf{s}\ y : \mathsf{nat}{\to}\mathsf{nat}}$$

$$\dfrac{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}} \qquad \begin{array}{c}[\vdash y : \mathsf{nat}] \\ T \\ \hline \vdash \mathsf{s}\ y : \mathsf{nat}\end{array}$$

$\#\;\; \mathit{infer}\ ((\mathit{infer}^0\ X)\ (\mathit{infer}^0\ Y)\ (\mathsf{s}\ (\mathit{infer}^0\ Z)))$

$\#\;\; \mathit{infer}\ ((\lambda f.\, \lambda x.\, f\ x)\ (\lambda y.\, \mathsf{s}\ (\mathsf{s}\ y))\ (\mathsf{s}\ \mathsf{o}))$

Example

$\#$ $\mathit{infer}\ ((\lambda f.\,\lambda x.f\ x)\ (\lambda y.\,\mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U\rangle$

$$\frac{X}{\vdash \lambda f.\,\lambda x.f\ x : \mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{nat}} \qquad \frac{Y}{\vdash \lambda y.\,\mathsf{s}\ y : \mathsf{nat}{\to}\mathsf{nat}}$$

$$\frac{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}} \qquad \frac{\begin{array}{c}[\vdash y : \mathsf{nat}]\\ T\end{array}}{\vdash \mathsf{s}\ y : \mathsf{nat}}$$

$\#$ $\mathit{infer}\ ((\mathit{infer}^0\ X)\ (\mathit{infer}^0\ Y)\ (\mathsf{s}\ (\mathit{infer}^0\ Z)))$

$\#$ $\mathit{infer}\ ((\mathit{infer}^0\ X)\ (\lambda y.\,\mathsf{s}\ (\mathit{infer}^0\ T))\ (\mathit{infer}^0\ Z))$

Example

$\mathit{infer}\ ((\lambda f.\,\lambda x.f\ x)\ (\lambda y.\,\mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \rightsquigarrow \langle \mathsf{nat}, U \rangle$

$$\begin{array}{cc} X & Y \\ \vdash \lambda f.\,\lambda x.f\ x : \mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{nat} & \vdash \lambda y.\,\mathsf{s}\ y : \mathsf{nat}{\to}\mathsf{nat} \end{array}$$

$$\begin{array}{cc} & [\vdash y : \mathsf{nat}] \\ Z & T \\ \vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat} & \vdash \mathsf{s}\ y : \mathsf{nat} \end{array}$$

$\mathit{infer}\ ((\mathit{infer}^0\ X)\ (\mathit{infer}^0\ Y)\ (\mathsf{s}\ (\mathit{infer}^0\ Z)))$

$\mathit{infer}\ ((\mathit{infer}^0\ X)\ (\lambda y.\,\mathsf{s}\ (\mathit{infer}^0\ T[h/\mathit{infer}\ y]))\ (\mathit{infer}^0\ Z))$

Example

$\#\ \mathit{infer}\ ((\lambda f.\,\lambda x.\,f\ x)\ (\lambda y.\,\mathsf{s}\ y)\ (\mathsf{s}\ \mathsf{o})) \leadsto \langle \mathsf{nat}, U \rangle$

$$\overset{X}{\vdash \lambda f.\,\lambda x.\,f\ x : \mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{nat}} \qquad \overset{Y}{\vdash \lambda y.\,\mathsf{s}\ y : \mathsf{nat}{\to}\mathsf{nat}}$$

$$\overset{Z}{\vdash \mathsf{s}\ \mathsf{o} : \mathsf{nat}} \qquad \begin{array}{c}[\vdash y : \mathsf{nat}]\\ T \\ \vdash \mathsf{s}\ y : \mathsf{nat}\end{array}$$

$\#\ \mathit{infer}\ ((\mathit{infer}^0\ X)\ (\mathit{infer}^0\ Y)\ (\mathsf{s}\ (\mathit{infer}^0\ Z)))$

$\#\ \mathit{infer}\ ((\mathit{infer}^0\ X)\ (\lambda y.\,\mathsf{s}\ (\mathit{infer}^0\ T[h/\mathit{infer}\ y]))\ (\mathit{infer}^0\ Z))$

Summary

- ✓ Gasp: certifying type checker $\longrightarrow$ incremental type checking
- ✓ sharing computation results by *function inverses*
- ✓ a safe approach: (shared) type derivation always available

Outline

Perspectives

- implement LF type reconstruction
- isolate higher-order term manipulation library
  put the *locally named* pattern into practice
- investigate typing of inverse functions
  and their relation with *NbE*
- front-end editor generating *deltas* ("*structured editor*")
  safe refactoring tools, typed version control
- LCF-style interactive theorem prover based on LF
  tactics as OCaml functions

M. Boespflug. *Conception d'un noyau de vérification de preuves pour le lambda-Pi-calcul modulo*. PhD thesis, École Polytechnique, Palaiseau, January 2011.

H. Geuvers, R. Krebbers, J. McKinna, and F. Wiedijk. Pure type systems without explicit contexts. *arXiv preprint arXiv:1009.2792*, 2010.

B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 235–246. ACM, 2002.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40 (1):143–184, 1993.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3): 23, 2008.

G.C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM, 1997.