CERTIFICATES FOR INCREMENTAL TYPE CHECKING

Thesis defense

Matthias Puech

Università di Bologna
Dottorato di Ricerca in Informatica, Ciclo xxvi

Université Paris Diderot
École Doctorale Sciences Mathématiques de Paris Centre

Andrea Asperti

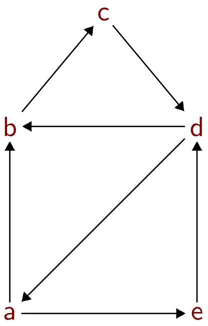
Hugo Herbelin & Yann Régis-Gianas

April 8, 2013

Outline

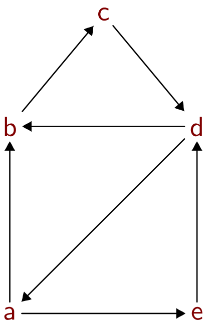
Quiz

Is there a cycle of size 2^N in this graph?



Quiz

Is there a cycle of size 2^N in this graph?



val `cycle` : `graph` → `bool`

The algorithm and its specification

Specification

$$\frac{A \to B}{A \leadsto_0 B}\ \textsc{Edge} \qquad \frac{A \leadsto_N B \qquad B \leadsto_N C}{A \leadsto_{\mathsf{s}(N)} C}\ \textsc{Trans}$$

Quiz

Given $\mathsf{a} \to \mathsf{b}$, $\mathsf{b} \to \mathsf{c}$, $\mathsf{c} \to \mathsf{d}$, $\mathsf{d} \to \mathsf{a}$, $\mathsf{d} \to \mathsf{b}$, $\mathsf{e} \to \mathsf{d}$, $\mathsf{a} \to \mathsf{e}$, are there A and N such that $A \leadsto_N A$ holds?

The algorithm and its specification

Specification

$$\textsc{Edge}\ \frac{A \to B}{A \rightsquigarrow_0 B} \qquad \textsc{Trans}\ \frac{A \rightsquigarrow_N B \qquad B \rightsquigarrow_N C}{A \rightsquigarrow_{\mathsf{s}(N)} C}$$

Quiz

Given $\mathsf{a} \to \mathsf{b}$, $\mathsf{b} \to \mathsf{c}$, $\mathsf{c} \to \mathsf{d}$, $\mathsf{d} \to \mathsf{a}$, $\mathsf{d} \to \mathsf{b}$, $\mathsf{e} \to \mathsf{d}$, $\mathsf{a} \to \mathsf{e}$, are there A and N such that $A \rightsquigarrow_N A$ holds? There are ($A = \mathsf{a}$, $N = 2$):

$$\dfrac{\dfrac{\dfrac{\overline{\mathsf{a} \to \mathsf{b}}}{\mathsf{a} \rightsquigarrow_0 \mathsf{b}}\ \textsc{Edge} \qquad \dfrac{\overline{\mathsf{b} \to \mathsf{c}}}{\mathsf{b} \rightsquigarrow_0 \mathsf{c}}\ \textsc{Edge}}{\mathsf{a} \rightsquigarrow_1 \mathsf{c}}\ \textsc{Trans} \qquad \dfrac{\dfrac{\overline{\mathsf{c} \to \mathsf{d}}}{\mathsf{c} \rightsquigarrow_0 \mathsf{d}}\ \textsc{Edge} \qquad \dfrac{\overline{\mathsf{d} \to \mathsf{a}}}{\mathsf{d} \rightsquigarrow_0 \mathsf{a}}\ \textsc{Edge}}{\mathsf{c} \rightsquigarrow_1 \mathsf{a}}\ \textsc{Trans}}{\mathsf{a} \rightsquigarrow_2 \mathsf{a}}\ \textsc{Trans}$$

Quiz

Is this true?

$$(p \supset q) \supset (p \vee r) \supset (q \vee r)$$

Quiz

Is this true?

$$(p \supset q) \supset (p \vee r) \supset (q \vee r)$$

val `prove` : `prop` → `bool`

The statement and its proof

Specification

$$\frac{\vdash A \supset B \qquad \vdash A}{\vdash B}\ \textsc{ImpE} \qquad \frac{\vdash A}{\vdash A \lor B}\ \textsc{DisjI1} \qquad \frac{\vdash B}{\vdash A \lor B}\ \textsc{DisjI2}$$

$$\frac{\begin{matrix}[\vdash A]\\ \vdots \\ \vdash B\end{matrix}}{\vdash A \supset B}\ \textsc{ImpI} \qquad \frac{\vdash A \lor B \qquad \begin{matrix}[\vdash A]\\ \vdots \\ \vdash C\end{matrix} \qquad \begin{matrix}[\vdash B]\\ \vdots \\ \vdash C\end{matrix}}{\vdash C}\ \textsc{DisjE}$$

Quiz

Does $\vdash (\mathsf{p} \supset \mathsf{q}) \supset (\mathsf{p} \lor \mathsf{r}) \supset (\mathsf{q} \lor \mathsf{r})$ hold?

The statement and its proof

Specification

$$\textsc{ImpE}\ \dfrac{\vdash A \supset B \qquad \vdash A}{\vdash B} \qquad\qquad \textsc{DisjI1}\ \dfrac{\vdash A}{\vdash A \vee B} \qquad\qquad \textsc{DisjI2}\ \dfrac{\vdash B}{\vdash A \vee B}$$

$$\dfrac{\begin{array}{c}[\vdash A]\\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}\ \textsc{ImpI} \qquad\qquad \dfrac{\vdash A \vee B \qquad \begin{array}{c}[\vdash A]\\ \vdots \\ \vdash C\end{array} \qquad \begin{array}{c}[\vdash B]\\ \vdots \\ \vdash C\end{array}}{\vdash C}\ \textsc{DisjE}$$

Quiz

Does $\vdash (p \supset q) \supset (p \vee r) \supset (q \vee r)$ hold? It does:

$$\dfrac{\dfrac{\dfrac{[\vdash (p \vee r)] \qquad \dfrac{\dfrac{[\vdash (p \supset q)] \qquad [\vdash p]}{\vdash q}\ \textsc{ImpE}}{\vdash q \vee r}\ \textsc{DisjI1} \qquad \dfrac{[\vdash r]}{\vdash q \vee r}\ \textsc{DisjI2}}{\vdash q \vee r}\ \textsc{DisjE}}{\vdash (p \vee r) \supset q \vee r}\ \textsc{ImpI}}{\vdash (p \supset q) \supset (p \vee r) \supset q \vee r}\ \textsc{ImpI}$$

Declarative specifications, performative algorithms

A program can be arbitrarily more complex than its specification.
How can I be sure that a program respects its specification?

Declarative specifications, performative algorithms

A program can be arbitrarily more complex than its specification.
How can I be sure that a program respects its specification?

Dynamically

```
let cycle : graph → path option = ...
verify (cycle [A, B; A, C; B, C])
```

Declarative specifications, performative algorithms

A program can be arbitrarily more complex than its specification.
How can I be sure that a program respects its specification?

Dynamically

let `cycle` : `graph` → `path option` = ...
`verify` (`cycle` [A, B; A, C; B, C])

Statically

let `cycle` : `graph` → `bool` = ...
Theorem for all G, `cycle` G = **true** *iff* G has a 2^N-cycle

Declarative specifications, performative algorithms

A program can be arbitrarily more complex than its specification.
How can I be sure that a program respects its specification?

Dynamically

```
let cycle : graph → path option = ...
verify (cycle [A, B; A, C; B, C])
```

Statically

```
let cycle : graph → bool = ...
```

Theorem for all G, `cycle` G = **true** *iff* G has a 2^N-cycle

In both cases, external tools are required

Thesis

We can integrate a program and its specification
by developping programming tools,
and relying on proof theory

Thesis

We can integrate a program and its specification
by developping programming tools,
and relying on proof theory

Argued by two main contributions:

1. programming with proof certificates
2. incremental type checking

Outline

Proof certificates

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

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

Examples

Proof-carrying code [Necula, 1997], *theorem proving* [Asperti and Tassi, 2007], *safe type inference* [Vytiniotis, 2008]

Proof certificates

Question

How to write programs manipulating proofs?

It is notably hard to write *correct* certificate-issuing programs with a *general-purpose* programming language:

1. how to represent proofs?
2. how to manipulate hypotheses? (binders)
3. how to compute on hypothetical proofs? (free variables)

1. A data structure for proofs?

Question

How to *represent* proofs?

Example

```
type vertex = string
type edge = vertex × vertex
type path =
  | Edge of edge
  | Trans of path × path

let check l : path → bool = …
```

1. A data structure for proofs?

Question

How to *represent* proofs?

Example

```
type prop = Atom of string | Imp of prop × prop
type pf =
  | DisjI1 of prop × pf | DisjI2 of prop × pf
  | ImpE of pf  × pf
  | ImpI of string × pf
  | DisjE of pf × string × pf × string × pf
let check l : pf → bool = ...
```

1. LF: a universal notation for hypothetical proofs

- many formal systems and logics
- a common *hypothetical reasoning* core
  e.g. logics with dischargeable hypotheses, programming languages with variable binders

1. LF: a universal notation for hypothetical proofs

- many formal systems and logics
- a common *hypothetical reasoning* core
  e.g. logics with dischargeable hypotheses, programming languages with variable binders

Proposition

Use LF [Harper et al., 1993] as the *values* manipulated

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

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

1. LF: a universal notation for hypothetical proofs

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*?

Related work

Twelf [Pfenning and Schürmann, 1999], Beluga [Pientka and Dunfield, 2010], Delphin [Poswolsky and Schürmann, 2008]

Computing LF objects?

Question

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

Related work

Twelf [Pfenning and Schürmann, 1999], Beluga [Pientka and Dunfield, 2010], Delphin [Poswolsky and Schürmann, 2008]

Proposition

An OCaml library to ease programming with proof certificates:

- ✓ general purpose functional programming language
- ✓ large corpus of libraries
- ✗ only simply-typed (ADT)

OCaml + LF = Gasp

http://www.cs.unibo.it/~puech/

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
  e.g. $(\text{«}^{sign}$ prop : $*$. imp : prop $\rightarrow$ prop $\rightarrow$ prop. » : `sign`)
  e.g. **fun** (`x`:`obj`) $\rightarrow$ (« imp "`x`" "`x`" » : `obj`)

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
  e.g. («^{sign} prop : ∗. imp : prop → prop → prop. » : `sign`)
  e.g. **fun** (`x`:`obj`) → (« imp "`x`" "`x`" » : `obj`)

…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

Gasp: an OCaml library to manipulate LF objects

Example

```
# let s = s ++ «sign
  tryProveIdentity : Πx : prop. pf x = "fun x → match x with
      | « imp "y" "z" » → « Impl "y" "z" (λx.x) »".
» ;;
s : sign = «sign ... »
```

Gasp: an OCaml library to manipulate LF objects

Example

```
# let s = s ++ «sign
  tryProveIdentity : Πx : prop. pf x = "fun x → match x with
      | « imp "y" "z" » → « Impl "y" "z" (λx.x) »".
» ;;
s : sign = «sign ... »

# eval s « tryProveIdentity (imp p p) » ;;
- : obj = « Impl p p (λx.x) »
```

Gasp: an OCaml library to manipulate LF objects

Example

```
# let s = s ++ «sign
  tryProveIdentity : Πx : prop. pf x = "fun x → match x with
      | « imp "y" "z" » → « Impl "y" "z" (λx.x) »".
» ;;
s : sign = «sign ... »

# eval s « tryProveIdentity (imp p p) » ;;
- : obj = « Impl p p (λx.x) »

# eval s « tryProveIdentity (imp p (imp q p)) » ;;
Exception: Type_error (« λx.x », « pf p → pf (imp q p) »)
```

2. Manipulating values with binders?

Consider the signature of the λ-calculus

```
# let s = «sign tm : ∗.
   app : tm → tm → tm.   lam : (tm → tm) → tm. » ;;
```

and function

```
# let eta_expand m = « lam λx. (app "m" x) » ;;
```

2. Manipulating values with binders?

Consider the signature of the λ-calculus

let s = «^{sign} tm : ∗.
  app : tm → tm → tm.   lam : (tm → tm) → tm. » ;;

and function

let eta_expand m = « lam λx. (app "m" x) » ;;

What is the value of this expression?

« lam λx. "eta_expand « x »" »

2. Manipulating values with binders?

Consider the signature of the λ-calculus

let s = «^{sign} tm : ∗.
 app : tm → tm → tm. lam : (tm → tm) → tm. » ;;

and function

let eta_expand m = « lam λx. (app "m" x) » ;;

What is the value of this expression?

« lam λx. (lam λx. app x x) »

2. Manipulating values with binders?

Consider the signature of the λ-calculus

$\#$ **let** s = «^{sign} tm : $*$.
  app : tm $\rightarrow$ tm $\rightarrow$ tm.   lam : (tm $\rightarrow$ tm) $\rightarrow$ tm. » ;;

and function

$\#$ **let** eta_expand m = « lam λx. (app "m" x) » ;;

What is the value of this expression?

« lam λx. (lam λx. app x x) »

$\rightsquigarrow$ *Variable capture during substitution*

Related work
FreshML [Shinwell et al., 2003], Cαml [Pottier, 2007], [McBride and McKinna, 2004], [Pouillard and Pottier, 2010]

2. Safe renaming & substitution in Gasp

let s = «^{sign}
tm : ∗. app : tm → tm → tm. lam : (tm → tm) → tm.
etaExpand : tm → tm = "**fun** m → « lam (λ*x*. app "m" *x*) »". » ;;

2. Safe renaming & substitution in Gasp

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

# eval s « lam (λx. etaExpand x) » ;;
```

2. Safe renaming & substitution in Gasp

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

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

2. Safe renaming & substitution in Gasp

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

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

# let s = s ++ «sign
  vl : ∗.   vlam : (tm → tm) → vl.
  interpret : tm → vl = "fun m → match m with
    | « lam "m" » → « vlam "m" »
    | « app "m" "n" » →
        let « vlam "p" » = « interpret "m" » in
        « interpret ("p" "n") »". » ;;
```

2. Safe renaming & substitution in Gasp

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

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

# let s = s ++ «sign
  vl : ∗.   vlam : (tm → tm) → vl.
  interpret : tm → vl = "fun m → match m with
    | « lam "m" » → « vlam "m" »
    | « app "m" "n" » →
        let « vlam "p" » = « interpret "m" » in
        « interpret ("p" "n") »". » ;;

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

2. Safe renaming & substitution in Gasp

Contribution

✓ Substitution and α-renaming are handled by the tool, thanks to an original *locally named* representation of concrete LF objects:

- free variables are *numbered*
  protected against capture
- bound variables are *named*
  written by the user

Example

«lam ($\lambda x.$ *etaExpand* x)»

2. Safe renaming & substitution in Gasp

Contribution

✓ Substitution and α-renaming are handled by the tool, thanks to an original *locally named* representation of concrete LF objects:

- free variables are *numbered*
  protected against capture
- bound variables are *named*
  written by the user

Example

«lam *(λx. etaExpand #1)*»

2. Safe renaming & substitution in Gasp

Contribution

✓ Substitution and α-renaming are handled by the tool, thanks to an original *locally named* representation of concrete LF objects:

- free variables are *numbered*
  protected against capture
- bound variables are *named*
  written by the user

Example

«lam $(\lambda x.$ lam $(\lambda x.$ app $\#2\ x))$»

2. Safe renaming & substitution in Gasp

Contribution

✓ Substitution and α-renaming are handled by the tool, thanks to an original *locally named* representation of concrete LF objects:

- free variables are *numbered*
  protected against capture
- bound variables are *named*
  written by the user

Example

«lam $(\lambda x.$ lam $(\lambda x'.$ app $x\ x'))$»

3. Computing on objects with free variables?

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

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

3. Computing on objects with free variables?

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 "f" » → « s (size "f") » ".
```

3. Computing on objects with free variables?

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 "f" » → « s (size "f") » ".
```

⇝ *Ill-typed:* f *has type* tm → tm

Related work

[Miller and Palamidessi, 1999], Contextual Modal Type Theory [Nanevski et al., 2008], Abella [Gacek et al., 2011]

3. *Environment-free* computation in Gasp

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" (size⁰ o))) » ".
```

3. *Environment-free* computation in Gasp

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" (size⁰ o))) » ".
```

We add *contraction* to the reduction

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

3. *Environment-free* computation in Gasp

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" (size⁰ o))) » ".
```

We add *contraction* to the reduction

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

Contribution

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

- ✓ to each n-ary function f, n function inverses $f^0, f^1, \ldots\ f^n$
- ✓ the adequate reduction: $f \circ f^0 = \mathrm{id}$

Gasp's *typed evaluation*

How to evaluate an object containing function symbols?

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

Gasp's *typed evaluation*

How to evaluate an object containing function symbols?

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

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

Gasp's *typed evaluation*

How to identify failures as soon as possible?

- checking the certificate *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)) »

Gasp's *typed evaluation*

How to identify failures as soon as possible?

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

⇝ *typed evaluation*: typing and evaluation are the same process
(dynamic typing?)

Gasp's *typed evaluation*

How to identify failures as soon as possible?

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

⇝ *typed evaluation*: typing and evaluation are the same process
(dynamic typing?)

Contribution

✓ An evaluation algorithm `eval` performing "on-the-fly" typing of open LF objects:

  ✓ issued certificates are guaranteed to be correct
  ✓ errors are signaled *where* the certificate is ill-typed

Two case studies for Gasp

Automated proof search

An automated theorem prover based on LJT [Dyckhoff, 1992] returning NJ proof certificates:

```
# let s = «^sign ... prove : ΠA : prop. proof A = ... » ;;
# eval s « prove (imp p (imp q p)) » ;;
- : obj = « Impl p (imp q p) (λx. Impl p q λx'.x) »
# eval s « prove (imp p q) » ;;
Exception: Not_provable
```

Two case studies for Gasp

Safe type checking

A type checker for System $\mathsf{T}_{<:}$ returning a typing derivation:

```
# let s = « … infer : Πm : expr. {a : tp | is m a} = … » ;;
# eval s « infer (lam λx. s x) » ;;
- : obj = « ⟨arr nat nat, IsLam (lam λx. s x) … ⟩ »
```

Two case studies for Gasp

Safe type checking

A type checker for System $\mathsf{T}_{<:}$ returning a typing derivation:

$\#$ **let** `s` $=$ « … *infer* : $\Pi m : \mathsf{expr}.\ \{\mathsf{a} : \mathsf{tp} \mid \mathsf{is}\ m\ a\} = \ldots$ » ;;
$\#$ `eval s` « *infer* ($\mathsf{lam}\ \lambda x.\, \mathsf{s}\ x$) » ;;
- : `obj` $=$ « $\langle \mathsf{arr\ nat\ nat}, \mathsf{IsLam}\ (\mathsf{lam}\ \lambda x.\, \mathsf{s}\ x) \ldots \rangle$ »

- ✓ lightweight development (partial formalization)
- ✓ reuse off-the-shelf libraries (e.g. term indexing)

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

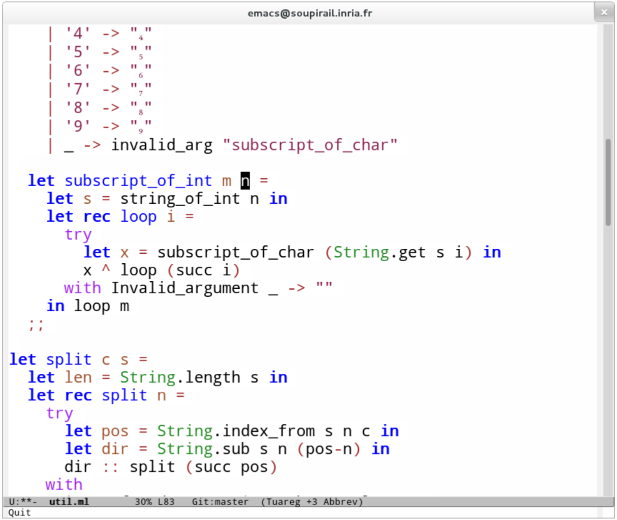
```
    | '4' -> "₄"
    | '5' -> "₅"
    | '6' -> "₆"
    | '7' -> "₇"
    | '8' -> "₈"
    | '9' -> "₉"
    | _ -> invalid_arg "subscript_of_char"

  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* $|apply(M, \delta)|$.

Incremental type checking

Question

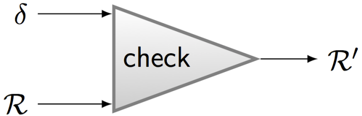
How can we make type checking *incremental*?

Definition

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

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

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

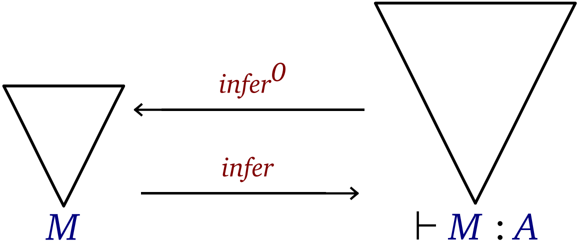
$$
\overline{\vdash (\lambda f\,x. f\,x)\,(\lambda x. \mathbf{s}(x))\,(\mathbf{s}(\mathbf{o})) : \mathsf{nat}}
$$

- it contains all intermediate type information

Incrementality by derivation reuse

Proposition

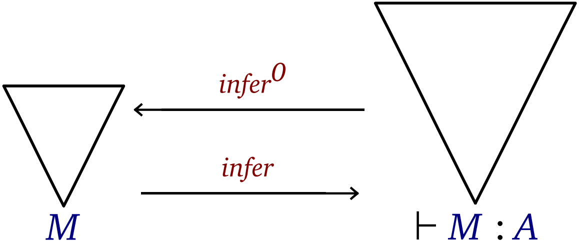
A certifying type checker in Gasp computes pieces of derivations



Incrementality by derivation reuse

Proposition

A certifying type checker in Gasp computes pieces of derivations



We need a way to

- address any *subderivation* $\mathscr{D}_i$
- 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

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

Example

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

Example

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

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

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

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

Example

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

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

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

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

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

Example

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

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

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

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

Example

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

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

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

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

Example

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

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

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

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

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

Example

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

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

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

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

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

Example

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

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

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

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

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

Example

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

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

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

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

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

Contributions

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

Outline

Contributions

- ✓ Gasp, a library for manipulating LF proof certificates
- ✓ support for *environment-free* style thanks to *function inverses*
- ✓ its extension to proof reuse enabling *incremental type checking*

Contributions

- ✓ Gasp, a library for manipulating LF proof certificates
- ✓ support for *environment-free* style thanks to *function inverses*
- ✓ its extension to proof reuse enabling *incremental type checking*

Other contributions:

- ✓ *inter-deriving* sequent calculi from natural deduction, using off-the-shelf *program transformations*:

$$\text{type checker for N.D.} \xrightarrow{\text{compilation}} \text{type checker for S.C.}$$

- ✓ an original metatheory of *spine-form* LF

Perspectives

- 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

Thank you

Backup slides

Certifying software

a.k.a *Proof-Carrying Code* [Necula, 1997]

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

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

Certifying software

a.k.a *Proof-Carrying Code* [Necula, 1997]

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

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…

A. Asperti and E. Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. *Lecture Notes in Computer Science*, 4573:146, 2007.

R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Inf. Comput.*, 209(1):48–73, 2011.

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.

C. McBride and J. McKinna. Functional pearl: i am not a number–i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM, 2004.

D. Miller and C. Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys (CSUR)*, 31(3es):11, 1999.

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.

F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, pages 202–206, 1999.

B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). *Automated Reasoning*, pages 15–21, 2010.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 93–107, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78738-0, 978-3-540-78738-9. URL `http://dl.acm.org/citation.cfm?id=1792878.1792887`.

François Pottier. Static name control for freshml. In *LICS*, pages 356–365, 2007.

Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *ICFP*, pages 217–228, 2010.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. Freshml: programming with binders made simple. In *ICFP*, pages 263–274, 2003.

D. Vytiniotis. *Practical type inference for first-class polymorphism*. PhD thesis, University of Pennsylvania, 2008.