Certifying OCaml type inference (and other type systems)

Jacques Garrigue

Nagoya University

http://www.math.nagoya-u.ac.jp/~garrigue/papers/

What's in OCaml's type system

- Core ML with relaxed value restriction
- Recursive types
- Polymorphic objects and variants
- Structural subtyping (with variance annotations)
- Modules and applicative functors
- Private types: private datatypes, rows and abbreviations
- Recursive modules . . .

The trouble and the solution(?)

- While most features were proved on paper, there is no overall specification for the whole type system
- For efficiency reasons the implementation is far from the the theory, making proving it very hard
- Actually, until 2008 there were many bug reports

A radical solution: a certified reference implementation

- The current implementation is not a good basis for certification
- One can check the expected behaviour with a (partial) reference implementation
- Relate explicitly the implementation and the theory, by developing it in Coq

What I have be proving in Coq

Over the last $2\frac{1}{2}$ years (on and off)

- Built a certified ML interpreter including structural polymorphism
- Proved type soundness and adequacy of evaluation
- Proved soundness and completeness (principality) of type inference
- Can handle recursive polymorphic object and variant types
- Both the type checker and interpreter can be extracted to OCaml and run
- Type soundness was based on *"Engineering formal metatheory"*

Related works

About core ML, there are already several proofs

- Type soundness was proved many times, and is included in *"Engineering formal metatheory"* [2008]
- For type inference, both Dubois et al. and Nipkow et al. proved algorithm W [1999]
- Their proofs rely on unification which was first proved by Paulson in LCF [1985]

However, there are very few proofs including recursive types

- Lee et al. proved type soundness for Standard ML [2007,2009]
- Owens et al. proved it for core OCaml [OCamlLight 2008]

Both of them do not handle type inference

The rest of this talkexplaining the proof

- What is structural polymorphism
- Definitions using co-finite quantification (450 lines)
- Type soundness (1150+650 lines)
- Automation and generic lemmas (1300 lines)
- Polymorphic objects and variants (800 lines)
- Unification (1800 lines)
- Type inference (3100 lines)
- Interpreter (3100 lines)
- Program extraction and examples of inference

Structural polymorphism

A typing framework for polymorphic variants and records.

- Faithful description of the core of OCaml.
- Polymorphism is described by local constraints.
- Constraints are kept in a recursive kinding environment.
- Constraints are abstract, and constraint domains with their δ-rules can be defined independently.

Types and kinds

Types are mixed with kinds in a mutually recursive way.

T	$::=$	α	type variable
	$\mid$	$T \to T$	function type
σ	$::=$	$\forall \bar{\alpha}.K \triangleright T$	polytypes
K	$::=$	$\emptyset \mid K, \alpha :: \kappa$	kinding environment
κ	$::=$	$\bullet \mid (C; R)$	kind
R	$::=$	$\{a : T, \ldots\}$	relation set

Extended type judgment and kind entailment:

$$K; E \vdash e : T$$

$$(C'; R') \models (C; R) \Leftrightarrow C' \models C \wedge R' \supset R$$

Example: polymorphic variants

Kinds have the form $(L, U; R)$, such that $L \subset U$.

$Number(5) \ : \ \alpha :: (\{Number\}, \mathcal{L}; \{Number : int\}) \triangleright \alpha$

$l_2 = [Number(5), Face(\texttt{"King"})]$
$l_2 \ : \ \alpha :: (\{Number, Face\}, \mathcal{L}; \{Number : int, Face : string\}) \triangleright \alpha \ list$

$length = \mathsf{function} \ Nil() \rightarrow 0 \mid Cons(a, l) \rightarrow 1 + length \ l$
$length \ : \ \alpha :: (\emptyset, \{Nil, Cons\}; \{Nil : unit, Cons : \beta \times \alpha\}) \triangleright \alpha \rightarrow int$

$length' = \mathsf{function} \ Nil() \rightarrow 0 \mid Cons(l) \rightarrow 1 + length \ l$
$length' \ : \ \alpha :: (\emptyset, \{Nil, Cons\}; \{Nil : unit, Cons : \alpha\}) \triangleright \alpha \rightarrow int$

$f \ l = length \ l + length2 \ l$
$f \ : \ \alpha :: (\emptyset, \{Nil, Cons\}; \{Nil : unit, Cons : \beta \times \alpha, Cons : \alpha\}) \triangleright \alpha \rightarrow int$

Kinding environment vs. μ-recursion

Kinding environment:

- recursive types as graphs rather than infinite trees
- easy to introduce polymorphism in nodes
- close to the implementation
- a good fit with locally nameless formalization?

"Mechanized metatheory of SML" [Lee et al. 2007] uses μ-recursive types.

Typing rules

Variable

$$\frac{K, K_0 \vdash \theta : K \qquad \mathsf{dom}(\theta) \subset B}{K; E, x : \forall B.K_0 \triangleright T \vdash x : \theta(T)}$$

Generalize

$$\frac{K; E \vdash e : T \qquad B \cap \mathsf{FV}_K(E) = \emptyset}{K|_{\overline{B}}; E \vdash e : \forall B.K|_B \triangleright T}$$

Abstraction

$$\frac{K; E, x : T \vdash e : T'}{K; E \vdash \mathsf{fun}\ x \to e : T \to T'}$$

Let

$$\frac{K; E \vdash e_1 : \sigma \qquad K; E, x : \sigma \vdash e_2 : T}{K; E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T}$$

Application

$$\frac{K; E \vdash e_1 : T \to T' \quad K; E \vdash e_2 : T}{K; E \vdash e_1\ e_2 : T'}$$

Constant

$$\frac{K_0 \vdash \theta : K \qquad \mathbf{type}(c) = K_0 \triangleright T}{K; E \vdash c : \theta(T)}$$

$K_0 \vdash \theta : K$ iff $\alpha :: \kappa \in K_0$ implies $\theta(\alpha) :: \kappa' \in K$ and $\kappa' \models \theta(\kappa)$

Engineering formal metatheory

Aydemir, Charguéraud, Pierce, Pollack, Weirich [POPL08]

Soundness for various type systems ($F_{\leq}$, ML, CoC).

Two main ideas to avoid renaming:

- Locally nameless definitions
  Use de-bruijn indices inside terms and types,
  but named variables for environments.
- Co-finite quantification
  Variables local to a branch are quantified universally.
  This allows reuse of derivations in different contexts.

Formalization is not always intuitive, but streamlines proofs of type soundness.

Typing rules (co-finite)

Variable

$$\frac{K \vdash \bar{T} :: \bar{\kappa}^{\bar{T}}}{K; E, x : \bar{\kappa} \triangleright T_1 \vdash x : T_1^{\bar{T}}}$$

Abstraction

$$\frac{\forall x \notin L \quad K; E, x : T \vdash e^x : T'}{K; E \vdash \lambda e : T \to T'}$$

Application

$$\frac{K; E \vdash e_1 : T \to T' \quad K; E \vdash e_2 : T}{K; E \vdash e_1\ e_2 : T'}$$

Generalize

$$\frac{\forall \bar{\alpha} \notin L \quad K, \bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}}; E \vdash e : T}{K; E \vdash e : \bar{\kappa} \triangleright T}$$

Let

$$\frac{K; E \vdash e_1 : \sigma \quad \forall x \notin L \quad K; E, x : \sigma \vdash e_2^x : T}{K; E \vdash \mathsf{let}\ e_1\ \mathsf{in}\ e_2 : T}$$

Constant

$$\frac{K \vdash \bar{T} :: \bar{\kappa}^{\bar{T}} \quad \mathsf{Tconst}(c) = \bar{\kappa} \triangleright T_1}{K; E \vdash c : T_1^{\bar{T}}}$$

$K \vdash \alpha :: \kappa$ when $\alpha :: \kappa' \in K$ and $\kappa' \models \kappa$

$K \vdash T :: \bullet$ always

Soundness results

Started from *Engineering formal metatheory* ML proof,
with many modifications to accomodate mutual recursion.
No renaming needed for soundness!

```
Lemma preservation : ∀ K E e e' T,
  K ; E |= e ~: T →
  e --> e' →
  K ; E |= e' ~: T.

Lemma progress : ∀ K e T,
  K ; empty |= e ~: T →
  value e ∨ exists e', e --> e'.

Lemma value_irreducible : ∀ e e',
  value e → ~(e --> e').
```

Extent of changes

Need simultaneous substitutions rather than iterated.

As a consequence, freshness of sequences of variables ($\bar{\alpha} \notin L$) is insufficient, and we need disjointness conditions ($L_1 \cap L_2 = \emptyset$).

Also added a framework for constants and δ-rules.

Overall size just doubled, with no significant jump in complexity.

This does not include:

Additions to the metatheory, with tactics for finite set inclusion, disjointness, etc... (1300 lines)

Domain proofs, for concrete constraints and constants. (800)

Constraint domain proofs

Instantiation of the framework to a constraint domain results in the following "dialog". This was done for the domain of polymorphic variants and records.

```
Module Cstr.   (* Define constraints *)    End Cstr.
Module Const.  (* Constants and arities *) End Const.
Module Sound1 := MkSound(Cstr)(Const).
Import Sound1 Infra Defs.

Module Delta.  (* Constant types and delta-rules *) End Delta.
Module Sound2 := Mk2(Delta).
Import Sound2 JudgInfra Judge.

Module SndHyp. (* Domain proofs *) End SndHyp.
Module Soundness := Mk3(SndHyp).
```

Adding a non-structural rule

Kind GC

$$\frac{\mathsf{FV}_K(E,T) \cap \mathsf{dom}(K') = \emptyset \qquad K, K'; E \vdash e : T}{K; E \vdash e : T}$$

cofinite Kind GC

$$\frac{\forall \bar{\alpha} \notin L \qquad K, \bar{\alpha} :: \bar{\kappa}^{\bar{\alpha}}; E \vdash_{GC} e : T}{K; E \vdash_{GC} e : T}$$

– Formalizes the intuition that kinds not appearing in either E or T are not relevant to the typing judgment.

– Good for modularity.

– Not derivable in the original type system, as all kinds used in a derivation must be in K from the beginning.

– Again, the co-finite version is implicit.

Working with Kind GC

Framework proofs are still easy (induction on derivations), but domain proofs become much harder (inversion no longer works).

One would like to prove the following lemma:

$$K; E \vdash_{GC} e : T \Rightarrow \exists K',\ K, K'; E \vdash e : T$$

I got completely stuck in the co-finite system, as co-finite quantification in Generalize does not commute with Kind GC.

I could finally prove it in more than 1300 lines, including renaming lemmas for both terms and types.

Afterwards, I realized that I only needed canonicization of proofs, which is only 100 lines, as it does not require renaming.

Type inference

Type inference is done in the usual ML way:

- W-like algorithm relying on type unification.
- All functions return both a normalized substitution and an updated kinding environment.
- Statements of inductive theorems become much more complex.
- Simpler statements as corrolary.
- Renaming lemmas are needed.

Unification

Formal proofs in LCF by Paulson as early as 1985.

Here we also need to handle the kinding environment, making the algorithm much more complicated.

Rather than θ is more general than θ' ($\exists\theta_1,\ \theta' = \theta_1 \circ \theta$) used the simpler θ' extends θ ($\theta' \circ \theta = \theta'$). They are equivalent when θ is idempotent.

900 lines for definitions and soundness, thanks to an induction lemma exploiting symmetries. 1000 more lines for completeness, with a large part for termination.

Type inference

For core ML, **W**'s correctness was proved about 10 years ago, both in Isabelle and Coq.

The original paper on type inference structural polymorphism contained only proofs about unification.

The practical type inference alorithm is very complex, due to subtleties of **generalize**.

Both soundness and principality require renaming. Soundness of **generalize** renames type variables twice!

More than 3000 lines of proof, with lots of lemmas about free variables.

Type inference (let case)

$\mathbf{generalize}(K, E, T, L) =$
  let $A = \mathsf{FV}_K(E)$ and $B = \mathsf{FV}_K(T)$ in
  let $K' = K|_{\overline{A}}$ in let $\bar\alpha :: \bar\kappa = K'|_B$ in
  let $\{\bar\alpha'\} = B \setminus (A \cup \{\bar\alpha\})$ in let $\bar\kappa' = \mathbf{map}\ (\lambda\_.\bullet)\ \bar\alpha'$ in
  $\langle (K|_A, K'|_L), [\bar\alpha\bar\alpha'](\bar\kappa\bar\kappa' \triangleright T)\rangle$

$\mathbf{typinf}(K, E, \mathsf{let}\ e_1\ \mathsf{in}\ e_2, T, \theta, L) =$
  let $\alpha = \mathbf{fresh}(L)$ in
  match $\mathbf{typinf}(K, E, e_1, \alpha, \theta, L \cup \{\alpha\})$ with
  | $\langle K', \theta', L'\rangle \Rightarrow$
    let $\langle K'', \sigma\rangle = \mathbf{generalize}(\theta'(K'), \theta'(E), \theta'(T), \theta'(\mathsf{dom}(K)))$ in
    let $x = \mathsf{fresh}(\mathsf{dom}(E) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2))$ in
    $\mathbf{typinf}(K'', (E, x : \sigma), e_2^x, T, \theta', L')$
  | $\langle\rangle \Rightarrow \langle\rangle$

Properties of type inference

Soundness

$$\mathbf{typinf}'(E, e) = \langle K, T\rangle \rightarrow \mathsf{FV}(E) = \emptyset \rightarrow K; E \vdash e : T$$

$$\mathbf{typinf}(K, E, e, T, \theta, L) = \langle K', \theta', L'\rangle \rightarrow$$
$$\mathsf{dom}(\theta) \cap \mathsf{dom}(K) = \emptyset \rightarrow \mathsf{FV}(\theta, K, E, T) \subset L \rightarrow$$
$$\theta'(K'); \theta'(E) \vdash e : \theta'(T) \wedge \theta' \sqsubseteq \theta \wedge K \vdash \theta' : \theta'(K') \wedge$$
$$\mathsf{dom}(\theta') \cap \mathsf{dom}(K') = \emptyset \wedge \mathsf{FV}(\theta', K', E) \cup L \subset L'$$

Principality

$$K; E \vdash e : T \rightarrow \mathsf{FV}(E) = \emptyset \rightarrow$$
$$\exists K'T', \mathbf{typinf}'(E, e) = \langle K', T'\rangle \wedge \exists\theta, T = \theta(T') \wedge K' \vdash \theta : K$$

$$K; E \vdash e : \theta(T) \rightarrow K \vdash \theta(E_1) \leq E \rightarrow \theta \sqsubseteq \theta_1 \rightarrow K_1 \vdash \theta : K \rightarrow$$
$$\mathsf{dom}(\theta_1) \cap \mathsf{dom}(K_1) = \emptyset \rightarrow \mathsf{dom}(\theta) \cup \mathsf{FV}(\theta_1, K_1, E_1, T) \subset L \rightarrow$$
$$\exists K'\theta'L',\ \mathbf{typinf}(K_1, E_1, e, T, \theta_1, L) = \langle K', \theta', L'\rangle \wedge$$
$$\exists\theta'',\ \theta\theta'' \sqsubseteq \theta' \wedge K' \vdash \theta\theta'' : K \wedge \mathsf{dom}(\theta'') \subset L' \setminus L$$

Interpreter

Defined a stack based abstract machine.
Since variables are de Bruijn indices, we can use terms as code.

```
Theorem eval_sound_rec :
  ∀ (h:nat) (fl:list frame) (benv args:list clos) K t T,
    closed_n (length benv) t ->
    K ; E |= stack2trm (app2trm (inst t benv) args) fl ~: T ->
    K ; E |= res2trm (eval fenv h benv args t fl) ~: T.

Theorem eval_complete : ∀ K t t' T,
  K ; E |= t ~: T ->
  clos_refl_trans_1n _ red t t' -> value t' ->
  ∃h : nat,∃cl : clos,
    eval fenv h [] [] t [] = Result 0 cl ∧ t' = clos2trm cl.
```

Impact of locally nameless and co-finite

Since local and global variables are distinct, many definitions must be duplicated, and we need lemmas to connect them.

- This is particularly painful for kinding environments, as they are recursive.
- Yet having to handle explicitly names of bound type variables would probably be even more painful.

Co-finite approach seems to be always a boon. Even for type inference, only few proofs use renaming lemmas:

- principality only requires term variable renaming once.
- soundness requires both term and type variables renaming, not surprising since we build a co-finite proof from a finite one.

Dependent types in values

They are used in the "engineering metatheory" framework only when generating fresh variables:

```
Lemma var_fresh : ∀ L : vars, { x : var | x ∉ L }.
```

I used dependent types in values in one other place: all kinds are valid and coherent by construction.

- A bit more complexity in domain proofs.
- But a big win since this property is kept by substitution.

```
Record ckind : Set := Kind {
  kcstr : Cstr.cstr;
  kvalid : Cstr.valid kcstr;
  krel  : list (Cstr.attr×typ);
  kcoherent : coherent kcstr krel }.
```

Also attempted to use dependent types for schemes (enforcing that they are well-formed), but dropped them as it made proofs about the type inference algorithm more complex.

What could be improved?

Some proofs are still much bigger than expected: eval_complete, type inference, . . .

- The `value` predicate is complex, as it handles constant arity.
  It might have been better to define constants as n-ary constructors from the start.
  This would require writing the induction principles by hand; already done for closures.
- Using functions to represent algorithms is dirty.
  In some cases, adding input-output inductive relations helped, but in general it does not change the proof size significantly.

Could I switch to adaptive proof scripts [Chlipala]?

Using the algorithm

Once the framework is instantiated, one can extract the type inference algorithm to ocaml, and run it.

```
(* This example is equivalent to the ocaml term [fun x -> `A0 x] *)
# typinf1 (Coq_trm_cst (Const.Coq_tag (Variables.var_of_nat 0)));;
- : (var * kind) list * typ =
([(1, None);
  (2,
   Some
    {kind_cstr = {cstr_low = {0}; cstr_high = None};
     kind_rel = Cons (Pair (0, Coq_typ_fvar 1), Nil)})],
 Coq_typ_arrow (Coq_typ_fvar 1, Coq_typ_fvar 2))
```

A more complete example

```
# let rev_append =
  recf (abs (abs (abs
  (matches [0;1] [abs (bvar 1);
    abs(apps(bvar 3)[sub 1 (bvar 0);cons(sub 0 (bvar 0))(bvar 1)]);
    bvar 1])))) ;;
val rev_append : trm = ...
# typinf2 Nil rev_append;;
- : (var * kind) list * typ =                (* using pretty printer *)
([(10, <Ksum, {}, {0; 1}, {0 => tv 15; 1 => tv 34}>);
  (29, <Ksum, {1}, any, {1 => tv 26}>);
  (34, <Kprod, {1; 0}, any, {0 => tv 30; 1 => tv 10}>); (30, any);
  (26, <Kprod, {}, {0; 1}, {0 => tv 30; 1 => tv 29}>); (15, any)],
 tv 10 @> tv 29 @> tv 29)
```

Path Resolution for Recursive Modules

(work done with Keiko Nakata)

- Problem: deciding path equality for recursive modules.
- Undecidable for a combination of first order strongly applicative functors and nested modules, allowing arbitrary recursive paths.
- Decidable by restricting submodule access in arguments to finite depth.
- The proof is by very involved induction, so that we decided to prove some lemmas in Coq.
- By choosing to have all recursive paths start from the root, we could avoid handling binders, and the proof is much easier.

Path normalization

The following language describes **recursive signatures** for a language with first-order functors.

Access signatures	S	$::=$	$\{m_1 : S_1 \cdots m_n : S_n\}$
Expressions	e	$::=$	$\{m_1 = e_1 \cdots m_n = e_n\} \mid \lambda(x : S)e \mid p$
Paths	p	$::=$	$vp \mid rp$
Variable paths	vp	$::=$	$x \mid vp.m$
Rooted paths	rp	$::=$	$\epsilon \mid rp(p) \mid rp.m$

We want to prove that for any program the normalizability of paths is decidable.

Enforcing restrictions

Enforcing that functor arguments are correctly accessed is not easy.

$$m_1 = \lambda(x : \{\})\{m_2 = x \quad m_3 = \epsilon.m_1(x).m_2.m_4\} \quad m_5 = \{\}\}$$

This program is incorrect

$$\epsilon.m_1(x).m_2.m_4 \longrightarrow x.m_4 \longrightarrow \text{error}$$

but all ground paths exhibit no problem.

Safe path normalization : 3 different judgments.

$$\text{r-src}\frac{P \vdash p \mapsto (\theta, e) \quad P \vdash \theta \text{ safe} \quad e \text{ not a path}}{P \vdash p \downarrow p} \qquad \text{r-vp}\frac{ap \in \mathsf{sig}_P(x)}{P \vdash x.ap \downarrow x.ap}$$

$$\text{r-exp}\frac{P \vdash p \mapsto (\theta, p') \quad P \vdash \theta \text{ safe} \quad P \vdash \theta(p') \downarrow q}{P \vdash p \downarrow q}$$

$$\text{r-dot}\frac{P \vdash p \downarrow p' \quad P \vdash p'.m \downarrow q}{P \vdash p.m \downarrow q} \qquad \text{r-app}\frac{P \vdash p_1 \downarrow p_1' \quad P \vdash p_1'(p_2) \downarrow q}{P \vdash p_1(p_2) \downarrow q}$$

$$\text{s-rec}\frac{P \vdash p \downarrow q \quad P \vdash q.m_i : S_i \ (1 \le i \le n)}{P \vdash p : \{m_1 : S_1 \ \ldots \ m_n : S_n\}}$$

$$\text{s-subst}\frac{P \vdash p_i : \mathsf{sig}_P(x_i) \quad (1 \le i \le n)}{P \vdash [x_1 \mapsto p_1, \ldots, x_n \mapsto p_n] \text{ safe}}$$

Formally proved

Lemma 2 (substitution) *If* $P \vdash p : S$ *and* $P \vdash \theta$ *safe, then* $P \vdash \theta(p) : S$.

Proof by mutual induction on safe reduction, safety for a signature, and safe substitutions.

The formalization

- uses de Bruijn indices for functor arguments
- uses local environments rather than a global mapping to association signatures to arguments

```
Lemma substitution : ∀p S s e,
  safe p S →safe_subs s e → closed e p → safe (subst s p) S.
Proof. ...
  refine (safe_ind3
    (fun p1 p2 ⇒ closed e p1 →
              if rooted_path p2 then red (subst s p1) (subst s p2)
              else ∀S, safe p1 S → safe (subst s p1) S) _
    (fun s' e' ⇒ list_forall (closed e) s' →
              safe_subs (map (subst s) s') e') _ _ _ _ _ _ _ _ _ _);
    intros.
  (* 87 more lines to prove all the cases *)
Qed.
```

Note that the property to prove by induction on safe reduction depends on whether the form of the resulting path.

The whole formalization is 600 lines, using standard tactics, and was written in less than 1 week.

Conclusion

- Formalized completely structural polymorphism.
- Proved not only type soundness, but also soundness and principality of inference, and correctness of evaluation through an abstract machine.
- First step towards a certified reference implementation of OCaml. Next step might be type constructors and the relaxed value restriction.
- The techniques in *Engineering formal metatheory* proved useful, but had to redo the automation.
- Draft and extractable proof scripts at

  `http://www.math.nagoya-u.ac.jp/~garrigue/papers/`

Dependent types for termination

In Coq all functions must use structural induction.

- Straightforward approach
  - Add a dummy decreasing argument, raising error if 0.
  - Then prove that one can always build a sufficiently big value.
  - Works, but the value must be built in the extracted code too.
- Alternative approach
  - Work by induction on a proof of termination, given as extra argument to the function.
  - Proofs are discarded during extraction.
  - One must use dependent types heavily inside functions.

Dependent types for termination (2)

```
Fixpoint unify pairs K S (HS:is_subst S) (HK:ok K)
  (h:Accu S K pairs) {struct h} : option (kenv * subs) :=
  match pairs as pairs0
  return pairs = pairs0 -> option (kenv * subs) with
  | nil => fun _ => Some (K, S)
  | (T1, T2) :: pairs => fun eq =>
    match unify1_dep T1 T2 K S HS HK with
    | inright _ => None
    | inleft (exist (pairs', K', S')
                    (conj _ (conj HK' (conj HS' lt')))) =>
      unify (pairs' ++ pairs) K' S' HS' HK'
        (Acc_inv h _ (lt' _ _ eq))
    end
  end (refl_equal pairs).
```

Working with functions containing proofs

Two difficulties

- Evaluation may unroll huge proof terms
  - Need to change evaluation strategy:
    `lazy [unify]; fold unify` rather than `simpl`
  - Normalize proof terms with a lemma, using proof irrelevance
    `rewrite normalize_typinf`
- Case analysis must not break proof hypotheses
  - Wrap some computations in dependently typed functions
  - Proof premises obtained as result of these functions
  - This allows to break dependencies on actual values