Proving OCaml's type system?

Jacques Garrigue
Nagoya University

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 . . .

What are the guarantees?

Already proved

- Type soundness and principality of type inference for various subsets (by hand)
- Mechanical proof of type soundness for the core part: OCaml-Light project (without the relaxed value restriction)

Can we hope to formally prove properties of the whole type system?

- Some parts are not even fully formalized (*e.g.* subtyping)
- Recursive types seem to frighten everybody

What I have been doing

Playing around with Coq...

- Proved type soundness for structural polymorphism
  - accounts for polymorphic objects and variants, including equi-recursive types
  - proof is based on "Engineering formal metatheory"
- Formalized and proved various properties for subsets of subtyping
  - weak transitivity, correctness of inference for parameterized types with variance annotations and private abbreviations
  - soundness of inference with the above extended with recursive object and variant types

Synopsis

- What's in OCaml's type system
- Structural polymorphism
  - Reusing formal metatheory
  - Encoding a framework with modules
  - Adding a non syntax-directed rule
- OCaml's subtyping
  - What is true. . . and what is not!
  - Proving it.

Structural polymorphism

A typing framework for polymorphic variants and records

- sufficient to describe most types 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
	$\mid$	u	base type
	$\mid$	$T \to T$	function type
σ	$::=$	$T \mid \forall \bar{\alpha}.K \triangleright T$	polytypes
K	$::=$	$\emptyset \mid K, \alpha :: k$	kinding environment
k	$::=$	$\bullet \mid (C; R)$	kind
R	$::=$	$\{r(a, T), \ldots\}$	relation set

Type judgments contain both a type and a kinding environment.

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

Example: polymorphic variants

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

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

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

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

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

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

Constraint domain

A set of abstract constraints $\mathcal{C}$ with entailment $\models$

- $\bot \in \mathcal{C}$ such that $\forall C.\bot \models C$ and $C \models \bot$ decidable
- $\models$ reflexive and transitive
- for any C and C', $C \wedge C'$ is the weakest constraint entailing both C and C'

Observations $C \vdash p(a)$ (a a symbol) compatible with entailment

Relating predicates $r(a, T)$ with propagation rules of the form:

$$\forall x.(r(x, \alpha_1) \wedge r(x, \alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2)$$

Typed constants and δ-rules, which should satisfy subject reduction

Admissible substitution

$K \vdash \theta : K'$ (θ admissible), if for all $\alpha :: (C, R)$ in K, $\theta(\alpha)$ is a type variable α' and it satisfies the following properties.

1. $\alpha' :: (C', R') \in K'$ *keep kinding*
2. $C' \models C$ *entailment of constraints*
3. $\theta(R) \subseteq R'$ *keep types*

Every C in K' shall be valid, and R satisfy propagation.

Typing rules

Variable

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

Abstraction

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

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

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

Let

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

Constant

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

Engineering formal metatheory

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, Stephanie Weirich [POPL08]

Proving soundness for various type systems ($F_{\leq}$, ML, CoC) in Coq

Two main ideas to avoid renaming:

- Locally nameless definitions
- Co-finite quantification

Locally nameless definitions

- α-conversion is a pain
- de Bruijn indices in derivations not so nice

Idea: use de Bruijn indices only for bound variables in terms (or type schemes), and name free variables.

$$\frac{x \notin \mathsf{Dom}(E) \cup \mathsf{FV}(t) \quad E, x{:}S \vdash t^x : T}{E \vdash \lambda t : S \to T}$$

Co-finite quantification

- we need to change non-locally bound names

Idea: quantify bound names universally, using a co-finite exclusion set

$$\frac{\forall x \notin L \quad E, x{:}S \vdash t^x : T}{E \vdash \lambda t : S \to T}$$

Intuition: L should be a superset of $\mathsf{Dom}(E) \cup \mathsf{FV}(T)$, so that there is no conflict, but we can grow L as needed when transforming proofs.

Example with weakening

Usually weakening requires renaming if $x \in \mathsf{Dom}(E')$

$$\frac{x \notin \mathsf{Dom}(E) \cup \mathsf{FV}(t) \quad E, x{:}S \vdash t^x : T}{E \vdash \lambda t : S \to T} \quad \longrightarrow \quad \frac{y \notin \mathsf{Dom}(E, E') \cup \mathsf{FV}(t) \quad E, E', y{:}S \vdash t^y : T}{E, E' \vdash \lambda t : S \to T}$$

No renaming needed if we enlarge L !

$$\frac{\forall x \notin L \quad E, x{:}S \vdash t^x : T}{E \vdash \lambda t : S \to T} \quad \longrightarrow \quad \frac{\forall x \notin L \cup \mathsf{Dom}(E') \quad E, E', x{:}S \vdash t^x : T}{E, E' \vdash \lambda t : S \to T}$$

Co-finite quantification and ML let

The translation of ML's let is a bit more involved:

$$\frac{E \vdash t_1 : T_1 \quad \bar{\alpha} \cap \mathsf{FV}(E) = \emptyset \quad E, x{:}\forall \bar{\alpha}.T_1 \vdash t_2 : T}{E \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T}$$

becomes

$$\frac{\forall \bar{\alpha} \notin L_1 \quad E \vdash t_1 : T_1^{\bar{\alpha}} \quad \forall x \notin L_2 \quad E, x{:}\forall^{|\bar{\alpha}|} T_1 \vdash t_2^x : T}{E \vdash \mathsf{let}\ t_1\ \mathsf{in}\ t_2 : T}$$

The only condition on $\bar{\alpha}$ is the derivability of $E \vdash t_1 : T_1^{\bar{\alpha}}$

Example with weakening

Again, without co-finite quantification, one has to rename the $\bar{\alpha}$ if E grows, as they may be referred by new bindings. This is particularly stupid as the new bindings do not contribute to the derivation.

An alternative approach would be to explictly consider only relevant bindings.

$$\frac{E \vdash t_1 : T_1 \quad \bar{\alpha} \cap \mathsf{FV}(E|_{\mathsf{FV}(t_1)}) = \emptyset \quad E, x{:}\forall\bar{\alpha}.T_1 \vdash t_2 : T}{E \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T}$$

The co-finite approach, where the constraint on $\bar{\alpha}$ is left implicit, is much smarter.

Engineering formal metatheory

Proofs are extremely short.

- Thanks to clever automation of the notion of freshness used by co-finite quantification, maintaining the conditions is easy.
- Many simple lemmas are required, but they are about types and terms, not derivations.
- Renaming inside derivations is very rarely needed. Soundess of $F_{\leq}$ or ML doesn't involve it.
- It is claimed that renaming lemmas for derivations can be obtained from substitution lemmas if needed.

Typing rules (co-finite) A

Variable

$$\frac{K, K_0 \vdash \theta : K \quad \mathrm{Dom}(\theta) \subset B}{K; E, x : \forall B.K_0 \triangleright T \vdash x : \theta(T)}$$

Abstraction

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

Application

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

Variable

$$\frac{K, K_0^{\bar\alpha} \vdash \theta : K \quad \mathrm{Dom}(\theta) = \bar\alpha}{K; E, x : K_0 \triangleright T \vdash x : T^{\theta(\bar\alpha)}}$$

Abstraction

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

Application

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

Typing rules (co-finite) B

Generalize

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

Let

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

Constant

$$\frac{K_0 \vdash \theta : K \quad \mathsf{T const}(c) = K_0 \triangleright T}{K; E \vdash c : \theta(T)}$$

Generalize

$$\frac{\forall \bar{\alpha} \notin L \quad K, K_o^{\bar{\alpha}}; E \vdash e : T^{\bar{\alpha}}}{K; E \vdash e : K_0 \triangleright T}$$

Let

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

Constant

$$\frac{K_0^{\bar{\alpha}} \vdash \theta : K \quad \mathsf{T const}(c) = K_0 \triangleright T}{K; E \vdash c : T^{\theta(\bar{\alpha})}}$$

Differences with proof for ML

Modifications are massive

- from a total of 970 lines for Core ML, 259 were modified and 1290 were added (excluding the 653 lines for instance domain proofs)
- the main technical modification was converting iterated substitutions into simultaneous ones
- used signatures and functors to allow instanciating the framework
- allowed adding constants and delta-rules modularly

Following the original proof plan, framework proofs were straightforward. Instance domain proofs were harder.

Simultaneous substitution

While opening local variables in a type scheme was already a simultaneous operation, global variable substitution was incremental.

```
Lemma typ_subst_open : forall (X:var) (U T:typ) (Ts:list typ),
  type U ->
  typ_subst X U (typ_open T Ts) =
   typ_open (typ_subst X U T) (List.map (typ_subst X U) Ts).

Lemma typ_subst_open : forall (S:env typ) (T:typ) (Ts:list typ),
  env_prop type S ->
  typ_subst S (typ_open T Ts) =
   typ_open (typ_subst S T) (List.map (typ_subst S) Ts).
```

The change was easy, using the environment type, but dozens of lemmas required modification.

Kinding proofs

```
Definition well_subst (K K':env kind) (S:env typ) :=
  forall (Z:var) (k:kind), binds Z k K ->
    well_kinded K' (kind_subst S k) (typ_subst S (typ_fvar Z)).

Lemma well_kinded_subst: forall S K K' k T,
  well_subst K K' S ->
  well_kinded K k T ->
  well_kinded K' (kind_subst S k) (typ_subst S T).

Lemma well_subst_fresh : forall K K' K'' S Ys L1 M,
  well_subst (K & K' & K'') (K & map (kind_subst S) K'') S ->
  fresh (L1 ∪ dom S ∪ dom (K & K'')) (length K0) Ys ->
  well_subst (K & K' & K'' & kinds_open_vars K0 Ys)
    (K & map (kind_subst S) (K'' & kinds_open_vars K0 Ys)) S.
```

well_kinded K' k (typ_fvar Z) =
exists k', binds Z k' K' ∧ entails k' k

Constraint domain and constants

```
Module Type CstrIntf.
  Parameter cstr : Set.
  Parameter valid : cstr -> Prop.
  Parameter entails : cstr -> cstr -> Prop.
  Parameter entails_refl : forall c, entails c c.
  Parameter entails_trans : forall c1 c2 c3,
    entails c1 c2 -> entails c2 c3 -> entails c1 c3.
  Parameter unique : cstr -> var -> Prop.
End CstrIntf.

Module Type CstIntf.
  Parameter const : Set.
  Parameter arity : const -> nat.
End CstIntf.
```

Modularity and delta-rules

```
Module MkDefs(Cstr:CstrIntf)(Const:CstIntf).
  ...
  Module Type DeltaIntf.
    Parameter type : Const.const -> sch.
    Parameter rule : nat -> trm -> trm -> Prop.
    Parameter term : forall n t1 t2 tl,
      rule n t1 t2 -> list_for_n term n tl ->
      term (trm_inst t1 tl) / term (trm_inst t2 tl).
  End DeltaIntf.
  Module MkJudge(Delta:DeltaIntf).
    Inductive typing : kenv -> env -> trm -> typ -> Prop := ...
    Inductive value : nat -> trm -> Prop := ...
    Inductive red : trm -> trm -> Prop := ...
  End MkJudge.
End MkDef.
```

Soundness proof

```
Module MkSound(Cstr:CstrIntf)(Const:CstIntf).
 Module Infra := MkInfra(Cstr)(Const).
 Import Infra Defs.
 Module Mk2(Delta:DeltaIntf).
  Module JudgInfra := MkJudgInfra(Delta).
  Import JudgInfra Judge.
  Module Type SndHypIntf.
   Parameter const_closed : forall c, sch_fv (Delta.type c) = {}.
   Parameter delta_typed : forall n t1 t2 tl K E T, ...
   Parameter const_arity_ok : forall c vl K T, ...
   Parameter delta_arity : forall n t1 t2, ...
  End SndHypIntf.
  Module Mk3(SH:SndHypIntf).
   ...
```

Soundness results

```
Lemma preservation : forall K E t t' T,
  K ; E |= t ~: T ->
  t --> t' ->
  K ; E |= t' ~: T.

Lemma progress := forall K t T,
  K ; empty |= t ~: T ->
    value t
  \/ exists t', t --> t'.

Lemma value_irreducible : forall n t t',
  value n t -> ~(t --> t').
```

Constraint domain proofs

```
Module Cstr. ... End Cstr.
Module Const. ... End Const.
Module Sound1 := MkSound(Cstr)(Const).
Import Sound1 Infra Defs.

Module Delta. ... End Delta.
Module Sound2 := Mk2(Delta).
Import Sound2 JudgInfra Judge.

Module SndHyp.
  ... ... ...
End SndHyp.
Module Soundness := Mk3(SndHyp).
```

Structural polymorphism

Proved soundess of structural polymorphism, including the constraint domain for polymorphic variants and records.

- reusing proofs and tactics was a tremendous help
- recursive types were not a problem at all!
- using functors for constructing the framework did work, but it is heavy

Adding a non-structural rule

Kind GC

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

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

– This cannot be proved in the original type system, as all kinds used in a derivation must be in K from the beginning

Co-finite non-structural Kind GC

We need to make it co-finite too.

$$\text{Kind GC}\quad \frac{\forall \bar{\alpha} \notin L \quad K, K_0^{\bar{\alpha}}; E \vdash_{GC} e : T}{K; E \vdash_{GC} e : T}$$

- framework proofs are still easy
- domain proofs become much harder:
  inversion no longer works directly
- a real nightmare...

Looking for an inversion lemma

We would like to prove the following lemma:

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

It is true in the original (non co-finite) type system.

It cannot be proved in the co-finite system, as co-finite quantification in Gen does not commute with Kind GC.

This discrepancy is very counter-intuitive. . .

Canonical derivations

I could solve the problem months later, by showing that in canonical derivations, Kind GC can be restricted to appear either at the end or just above Let.

- Knowing this we can relax the proof requirement for δ-rules, by assuming they only apply to canonical derivations not ending with Kind GC.
- We can then directly reuse domain proofs for the original system.
- Proving the canonization lemma requires a renaming lemma for term variables (to make quantifiers commute). It was actually easier to prove it from scratch than to attempt to reuse term substitution (as suggested in the paper).

OCaml's subtyping

- mainly structural (objects and variants)
- variance annotations to allow subtyping under type constructors
- recent additions: private abbreviations and monomorphisation of polymorphic methods
- all subtyping is explicit
- as types may contain type variables, subtyping is done in two phases, avoiding interference with type inference
  - structural comparison of types
  - unification of types that didn't match a subtyping rule
- no formal specification! (and easy to get wrong)

A formalization

Subtyping assumes a set E of type equations.

$$E \vdash T_1 \leq T_2$$

Subsumption can be defined as follows.

$$\text{Subtype}\quad \frac{E \vdash e : \sigma(T_1) \quad E \vdash T_1 \leq T_2 \quad \sigma \models E}{E \vdash (e : T_1 :> T_2) : \sigma(T_2)}$$

where $\sigma \models E$ means that σ is an unifier of E.

Basic subtyping (with private abbreviations)

Equal

$$\frac{T_1 = T_2 \in E}{E \vdash T_1 \leq T_2}$$

Tconstr

$$\frac{\forall i \in P(t)\ E \vdash T_i \leq T'_i \quad \forall j \in N(t)\ E \vdash T'_j \leq T_j \qquad \forall k \in I(t)\ T_k = T'_k \in E}{E \vdash (T_1, \ldots, T_n)\, t \leq (T'_1, \ldots, T'_n)\, t}$$

Private

$$\frac{E \vdash T[T_1 \ldots T_n / \alpha_1 \ldots \alpha_n] \leq T' \quad \texttt{type}\ (\alpha_1, \ldots, \alpha_m)\, t = \texttt{private}\ T}{E \vdash (T_1, \ldots, T_n)\, t \leq T'}$$

Recursive polymorphic variants

We need two extra environments, K for kinds and S for memoizing subtyping between kinded variables.

Variant

$$\frac{K(\alpha) = (L, U, R) \qquad K(\beta) = (L', U', R') \qquad U \subset L' \qquad \forall l \in U, \forall (l, T) \in R, \forall (l, T') \in R',\ K; E; S, \alpha \leq \beta \vdash T \leq T'}{K; E; S \vdash \alpha \leq \beta}$$

Recursion

$$\frac{\alpha \leq \beta \in S}{K; E; S \vdash \alpha \leq \beta}$$

Polymorphic methods

Type equations are prefixed by bijections of universal variables.

$$E ::= \emptyset \mid E, T_1 = T_2 \ldots \mid E, \bar{u} \leftrightarrow \bar{v}.E$$

Poly

$$\frac{E_1 \vdash T_1 \leq T_2 \quad \bar{u} \leftrightarrow \bar{v}.E_1 \in E}{E \vdash \forall \bar{u}.T_1 \leq \forall \bar{v}.T_2}$$

InstPoly

$$\frac{E \vdash T_1[\bar{\alpha}/\bar{u}] \leq T_2}{E \vdash \forall \bar{u}.T_1 \leq \forall .T_2}$$

Properties

Reflexivity For any type T, $T = T \vdash T \leq T$.

Trivial.

Transitivity If $E_1 \vdash T_1 \leq T_2$ and $E_2 \vdash T_2 \leq T_3$ and $\sigma \models E_1 \cup E_2$, then there is $\sigma \models E_3$ s.t. $E_3 \vdash T_1 \leq T_3$.

False! For instance, assuming `type t = private int`,

$\emptyset \vdash \mathsf{t} \leq \mathsf{int}$ and $\mathsf{int} = \alpha \vdash \mathsf{int} \leq \alpha$ and $\mathsf{t} = \alpha \vdash \mathsf{t} \leq \alpha$

but there is no σ s.t. both $\sigma \models \mathsf{int} = \alpha$ and $\sigma \models \mathsf{t} = \alpha$.

Properties

Weak transitivity If $E_1 \vdash T_1 \leq T_2$ and $E_2 \vdash T_2 \leq T_3$ and $\sigma \models E_1 \cup E_2$, and $\mathrm{Img}(\sigma) \subset Var$, then there is $\sigma \models E_3$ s.t. $E_3 \vdash T_1 \leq T_3$.

Monotony If $E \vdash T_1 \leq T_2$ and $\sigma' \circ \sigma \models E$, then there is E' s.t. $E' \vdash \sigma(T_1) \leq \sigma(T_2)$ and $\sigma' \models E'$.

Proved both for the base system including private abbreviations.

Properties of inference

Defined a algorithm `subinf`, producing E from T_1 an T_2.

Soundness If `subinf` T_1 $T_2 =$ Some E then $E \vdash T_1 \leq T_2$.

Proved for basic system, and adding recursive kinds.

Completeness If $E \vdash T_1 \leq T_2$ then `subinf` T_1 $T_2 =$ Some E, otherwise `subinf` T_1 $T_2 =$ None.

Proved only for basic system. Termination is a pain. . . .

Proofs

– used a simpler formalization, eliminating E

$$\sigma \vdash T_1 \leq T_2 \quad \Leftrightarrow \quad \exists E, \sigma \models E \wedge E \vdash T_1 \leq T_2$$

– made the relation symmetrical (adding $\sigma \vdash T_1 \geq T_2$), to allow induction.

– definitions are quite verbose (54 lines for subtyping)

– soundness is just tedious, but (weak) transitivity and completeness are not so easy.

Conclusion

– Trying to specify and prove OCaml's type system

– Interested not only in soundness, but in properties of inference too

– No hope to prove the current implementation, but could maybe create a reference implementation

– Thanks to Arthur for his nice proofs and libraries