A Certified Interpreter for M
Structural Polymorphism

Jacques Garrigue
Nagoya University
`http://www.math.nagoya-u.ac.jp/~garrig`

What's in OCaml's type sy

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

What are the guarantee

Proved before (by many)

– Type soundness and principality of type inferenc
subsets (by hand).

– Mechanical proof of type soundness for the core
OCaml-Light project (without the relaxed value

What I have done in Coq over the last 2 years

– A certified interpreter for ML with structural po

– Includes type soundness and principality of infer

– Covers polymorphic objects and variants, with r

– Mechanization is based on *"Engineering formal*

Structural polymorphis

A typing framework for polymorphic variant

- Faithful description of the core of OCa
- Polymorphism is described by local cons
- Constraints are kept in a recursive kindi environment.
- Constraints are abstract, and constraint their δ-rules can be defined independent

Types and kinds

Types are mixed with kinds in a mutually r

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 environm
κ	$::=$	$\bullet \mid (C; R)$	kind
R	$::=$	$\{a : T, \ldots\}$	relation set

Type judgments contain both a type and a

environment.

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

Example: polymorphic vari

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

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

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

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

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

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

Typing rules

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

Generalize
$$\frac{K; E \vdash e : T \qquad B \cap}{K|_{\overline{B}}; E \vdash e : \forall B.K}$$

Abstraction
$$\frac{K; E, x : T \vdash e : T'}{K; E \vdash \text{fun } x \rightarrow e : T \rightarrow T'}$$

Let
$$\frac{K; E \vdash e_1 : \sigma \qquad K;}{K; E \vdash \text{let } x = e_1}$$

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

Constant
$$\frac{K_0 \vdash \theta : K \qquad \textbf{type}}{K; E \vdash c : \theta(T)}$$

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

Engineering formal metath

Aydemir, Charguéraud, Pierce, Pollack, Weirich

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 univer
  This allows reuse of derivations in different cont

Formalization is not always intuitive, but streamline
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

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

Let

$$\frac{K; E \vdash e_1 : \sigma \quad K;}{K; E \vdash \text{let } e_1 \text{ in } e_2}$$

Constant

$$\frac{K \vdash \bar{T} :: \bar{\kappa}^{\bar{T}} \quad \text{Tco}}{K; E \vdash c : T_1^{\bar{T}}}$$

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

$K \vdash T :: \bullet$ always

Soundness results

Started from *Engineering formal metatheory* ML pr
with many modifications to accomodate mutual rec
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 iterate

As a consequence, freshness of sequences of variabl
insufficient, and we need disjointness conditions $(L_1$

Also added a framework for constants and δ-rules.

Overall size just doubled, with no significant jump i

This does not include:

Additions to the metatheory, with tactics for fin
disjointness, etc... (1300 lines)

Domain proofs, for concrete constraints and co

Constraint domain proo

Instantiation of the framework to a constraint doma
following "dialog". This was done for the domain o
variants and records.

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

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

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

Adding a non-structural r

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 Ki

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

– Formalizes the intuition that kinds not appearin T are not relevant to the typing judgment.

– Good for modularity.

– Not derivable in the original type system, as all 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 deriva
domain proofs become much harder (inversion no lo

One would like to prove the following lemma:

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

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

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

Afterwards, I realized that I only needed canonicizat
which is only 100 lines, as it does not require renan

Type inference

Type inference is done in the usual ML way

- W-like algorithm relying on type unificat
- All functions return both a normalized s
  and an updated kinding environment.
- Statements of inductive theorems becor
  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 environmen
algorithm much more complicated.

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

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

Type inference

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

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

The practical type inference alorithm is very comple
subtleties of **generalize**.

Both soundness and principality require renaming. S
generalize renames type variables twice!

More than 3000 lines of proof, with lots of lemmas
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 $\bar{\kappa}' = \mathbf{map}\ (\lambda\_.\bullet)\ \bar{\alpha}'$ i
  $\langle (K|_A, K'|_L), [\bar{\alpha}\bar{\alpha}'](\bar{\kappa}\bar{\kappa}' \triangleright T)\rangle$

$\mathbf{typinf}(K, E, \mathsf{let}\ e_1\ \mathsf{in}\ e_2, T, \theta, L) =$
  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 $\langle K'', \sigma\rangle = \mathbf{generalize}(\theta'(K'), \theta'(E), \theta'(T), \theta'(\mathsf{do}$
    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 infere

Soundness

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

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

Principality

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

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

Interpreter

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

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

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' = cl
```

Impact of locally nameless and

Since local and global variables are distinct, many c
be duplicated, and we need lemmas to connect the

- This is particularly painful for kinding environme
  recursive.
- Yet having to handle explicitly names of bound
  would probably be even more painful.

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

- principality only requires term variable renaming
- soundness requires both term and type variable
  surprising since we build a co-finite proof from a

Dependent types in valu

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

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

I used dependent types in values in one other place
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 :
  kcstr : Cstr.cstr;
  kvalid : Cstr.vali
  krel  : list (Cstr
  kcoherent : cohere
```

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

What's wrong?

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

- The `value` predicate is complex, as it handles co

  It might have been better to define constants a
  constructors from the start.

  This would require writing the induction princip
  already done for closures.

- Using functions to represent algorithms is dirty.

  In some cases, adding input-output inductive re
  but in general it does not change the proof size

- Still wondering...

Using the algorithm

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

```
(* This example is equivalent to the ocaml term [f
# typinf1 (Coq_trm_cst (Const.Coq_tag (Variables.v
- : (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), Ni
 Coq_typ_arrow (Coq_typ_fvar 1, Coq_typ_fvar 2))
```

A more complete examp

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

Conclusion

- Formalized completely structural polymorphism.
- Proved not only type soundness, but also sound
  principality of inference, and correctness of eval
  an abstract machine.
- First step towards a certified reference impleme
  OCaml. Next step might be type constructors a
  value restriction.
- The techniques in *Engineering formal metatheo*
  but had to redo the automation.
- Extractable proof scripts at

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