Verification of OCaml typing proofs

Pierrick Couderc

OCamlPro

U2IS, ENSTA ParisTech

Feb. 21, 2017

Typedtrees as Typing Proofs

Motivations:

- Correctness of OCaml typing
- Detect regressions of OCaml typing

Idea:

- Check Typedtree as typing proofs
- Write a type checker as a type system, with soundness proof

Typedtrees

Typedtree node: $\langle$ e : τ $\rangle$

- Expression: e
- Inferred type: τ
- ~~Environment used for inference: Γ_{inf}~~

Avoid reproducing possible bugs from the OCaml compiler. Compiler is designed for inference.

$\Rightarrow$ Use only data structures

Specification as explicit as possible

Classical ML specification:

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Specification as explicit as possible

Classical ML specification:

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Without sharing metavariables:

$$\text{App}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash \tau_1 \prec \tau_2' \to \tau' \qquad \Gamma \vdash \tau_2' \equiv \tau_2 \qquad \Gamma \vdash \tau' \equiv \tau\end{array}}{\Gamma \vdash e_1\ e_2 : \tau}$$

Replacing unification

Unification: Mainly designed for Inference, with

- Equivalence of types
- Instantiation of type schemes
- Matching on the form of types
- Generation of equations with GADTs
- Checking the scope of type constructors

Equivalence

$$\Gamma, \Phi \vdash \tau_1 \equiv \tau_2$$

Structural type equality with expansion of abbreviations.

Instantiation

$$\Gamma, \Phi, \theta \vdash \tau \leq \sigma \Rightarrow \theta'$$

- θ: Substitution from type variables to types

For example:

$$\Gamma, \Phi, \theta \vdash \mathit{int} \to \mathit{int} \leq (\forall\alpha.)\ \alpha \to \alpha \Rightarrow \theta \oplus [\alpha \mapsto \mathit{int}]$$

Instantiation

$$\text{Var}\ \frac{\mathit{let}\ \sigma_x = \Gamma.\mathit{Values}(x) \qquad \Gamma, \Phi, \theta_{\mathit{nongen}}(\Gamma, x) \vdash \texttt{int} \rightarrow \texttt{int} \leq \sigma_x \Rightarrow \theta}{\Gamma, \Phi \vdash \langle x : \texttt{int} \rightarrow \texttt{int} \rangle}$$

with

$\Gamma.\mathit{Values}(x) = \alpha \rightarrow \alpha$

$\theta_{\mathit{nongen}}(\Gamma, x) = \{\beta \mapsto \beta \mid \beta \in \mathit{fv}(\Gamma - \mathrm{x})\}$

Needed due to the lack of explicit type schemes.

Instantiation: checking rules

$$\text{Inst-Var-Unbound}\ \frac{\alpha \notin \mathit{dom}(\theta)}{\Gamma, \theta \vdash\ \tau \leq\ \alpha \Rightarrow \theta \oplus [\alpha \rightarrow \tau]}$$

$$\text{Inst-Var-Bound}\ \frac{\begin{array}{c}\alpha \in \mathit{dom}(\theta) \\ \mathit{let}\ \tau_\alpha = \theta(\alpha) \\ \Gamma, \theta \vdash \tau_\alpha \equiv \tau\end{array}}{\Gamma, \theta \vdash\ \tau \leq\ \alpha \Rightarrow \theta}$$

$$\text{Inst-Fun}\ \frac{\begin{array}{c}\Gamma, \theta \vdash \tau_1 \leq \tau_1' \Rightarrow \theta_1 \\ \Gamma, \theta_1 \vdash \tau_2 \leq \tau_2' \Rightarrow \theta_2\end{array}}{\Gamma, \theta \vdash \tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2' \Rightarrow \theta_2}$$

$$\text{Inst-Construct}\ \frac{\begin{array}{c}\Gamma, \theta \vdash \tau_0 \leq \tau_0' \Rightarrow \theta_0 \\ \forall i_{\geq 1}.\ \Gamma, \theta_{i-1} \vdash \tau_i \leq \tau_i' \Rightarrow \theta_i\end{array}}{\Gamma, \theta \vdash (\overline{\tau})\ \mathtt{t} \leq (\overline{\tau'})\ \mathtt{t} \Rightarrow \theta_n}$$

$$\text{Inst-Construct-Exp-Left}\ \frac{\begin{array}{c}\mathit{let}\ \tau = \mathit{expand}(\Gamma, \mathtt{t}, \overline{\tau}) \\ \Gamma, \theta \vdash \tau \leq \tau' \Rightarrow \theta'\end{array}}{\Gamma, \theta \vdash (\overline{\tau})\ \mathtt{t} \leq \tau' \Rightarrow \theta'}$$

$$\text{Inst-Construct-Exp-Right}\ \frac{\begin{array}{c}\mathit{let}\ \tau' = \mathit{expand}(\Gamma, \mathtt{t}', \overline{\tau'}) \\ \Gamma, \theta \vdash \tau \leq \tau' \Rightarrow \theta'\end{array}}{\Gamma, \theta \vdash \tau \leq (\overline{\tau'})\ \mathtt{t}' \Rightarrow \theta'}$$

Type matching

$$\Gamma, \Phi \vdash \tau \prec \tau'$$

- Check the term structure of type
- Introduce sub terms as meta-variables

For example

$$\text{App}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \\ \Gamma, \Phi \vdash \tau_1 \prec \tau_d \rightarrow \tau_{cd} \qquad \Gamma, \Phi \vdash \tau_2 \equiv \tau_d \qquad \Gamma, \Phi \vdash \tau_{cd} \equiv \tau\end{array}}{\Gamma, \Phi \vdash \langle e_1\ e_2 : \tau \rangle}$$

Generated equations for GADTs

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

In OCaml, registered as abbreviations in environment:

- Int branch: `type a = int`
- Bool branch: `type a = bool`

Generated equations for GADTs

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

Type equalities as equivalence classes, handled by union-find.

1. a is registered as *rigid variable*, then $\Phi(\mathsf{a}) = \{\mathsf{a}\}$.
2. On `Int` branch, $\Phi(\mathsf{a}) = \Phi(\texttt{int}) = \{\texttt{int, a}\}$.
3. On `Bool` branch, $\Phi(\mathsf{a}) = \Phi(\texttt{bool}) = \{\texttt{bool, a}\}$.

Checking the scope of type constructor

```
let l = ref [] (* level 1 *)

type t = A | B (* level 2 *)

(* Rejected, level 3 *)
let _ = l := [ A ]
```

- In OCaml: levels easily check scope escaping of types.
- Without levels: wellformedness checking

Unification: Wellformedness rules

$$\text{Wf-Var}\ \frac{\mathit{level}(\alpha) = \mathit{gen\_level}}{\Gamma \vdash\ \alpha\ \mathit{wf}} \qquad \text{Wf-Fun}\ \frac{\Gamma \vdash \tau_1\ \mathit{wf} \qquad \Gamma \vdash \tau_2\ \mathit{wf}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathit{wf}}$$

$$\text{Wf-Construct}\ \frac{\begin{array}{c}\mathit{let}\ \tau_{\mathit{param}_0}, .., \tau_{\mathit{param}_n} = \Gamma(\mathtt{t}) \\ \forall i. C \vdash \tau_i\ \mathit{wf} \qquad \Gamma, \varnothing \vdash (\overline{\tau})\ \mathtt{t} \leq (\overline{\tau_{\mathit{param}}})\ \mathtt{t} \Rightarrow \theta\end{array}}{\Gamma \vdash (\overline{\tau})\ \mathtt{t}\ \ \mathit{wf}}$$

Checking patterns

$$\frac{\dots \qquad \dots \qquad \dots}{\Gamma, \Phi, \mathcal{V} \vdash \langle p : \tau \rangle \Rightarrow \mathcal{V}', \mathcal{T}', \Phi'}$$

Generate three components:

- Variables: $\mathcal{V}$ (or $(\overline{v : \tau})$)
- Existential types: $\mathcal{T}$ (or $(\overline{\tau_\exists})$)
- Equalities: Φ

Checking patterns: Existential type constructor

```
type ex = Ex : 'a -> ex

let f x = match x with Ex v -> 〈 g 〈 v : a#0 〉 : unit 〉
```

a#0: generated at type inference

⇒ Retrieved by pattern checking

Patterns: Example

$$\text{Pat-Tuple}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash \tau \prec \tau_{p_0} * \ \ .. \ \ * \tau_{p_n} \qquad \Gamma, \Phi, \mathcal{V} \vdash \langle p_0 : \tau_0 \rangle \Rightarrow \mathcal{V}_0, \mathcal{T}_0, \Phi_0 \\ \forall i_{\geq 1}.\ \Gamma, \Phi_{i-1}, \mathcal{V}_{i-1} \vdash \langle p_i : \tau_i \rangle \Rightarrow \mathcal{V}_i, \mathcal{T}_i, \Phi_i \\ \forall i.\ \Gamma, \Phi_i \vdash \tau_{p_i} \equiv \tau_i\end{array}}{\Gamma, \Phi, \mathcal{V} \vdash \langle p_0,\ ..\ , p_n : \tau \rangle \Rightarrow \mathcal{V}_n, \mathcal{T}_0 \cup .. \cup \mathcal{T}_n, \Phi_n}$$

1. τ has tuple form.
2. Check sub-patterns.
3. Check equivalence of sub-patterns type against tuple components

Typechecking Typedtree

$$\text{Const}\ \frac{c : \tau}{\Gamma, \Phi \vdash \langle c : \tau \rangle} \qquad \text{Var}\ \frac{\Gamma, \Phi, \theta_{nongen}(\Gamma, x) \vdash \tau \leq \Gamma.\mathit{Values}(x) \Rightarrow \theta}{\Gamma, \Phi \vdash \langle x : \tau \rangle}$$

$$\text{Abs}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash \tau \prec \tau_d \to \tau_{cd} \qquad \forall i.\ \Gamma, \Phi, \varnothing \vdash \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ \forall i.\ \Gamma, \Phi \vdash \tau_d \equiv \tau_i \qquad \forall i.\ \mathit{let}\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}) \oplus_{\mathcal{T}} (\overline{\tau_{\exists_i}}) \\ \forall i.\ \Gamma_i, \Phi_i \vdash \langle e_i : \tau_i' \rangle \qquad \forall i.\ \Gamma, \Phi \vdash \tau_i'\ \mathit{wf} \qquad \forall i.\ \Gamma_I, \Phi_i \vdash \tau_{cd} \equiv \tau_i' \end{array}}{\Gamma, \Phi \vdash \langle \mathbf{function}\ \overline{\mid\ p \to e}\ : \tau \rangle}$$

$$\text{App}\ \frac{\begin{array}{c} \Gamma, \Phi \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma, \Phi \vdash \langle e_2 : \tau_2 \rangle \\ \Gamma, \Phi \vdash \tau_1 \prec \tau_d \to \tau_{cd} \qquad \Gamma, \Phi \vdash \tau_2 \equiv \tau_d \qquad \Gamma, \Phi \vdash \tau_{cd} \equiv \tau \end{array}}{\Gamma, \Phi \vdash \langle e_1\ e_2 : \tau \rangle}$$

$$\text{Construct}\ \frac{\begin{array}{c} \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \\ \mathit{let}\ (\tau_{arg_0}, \ ..\ , \tau_{arg_n}, \tau_{constr}) = \mathit{find\_constructor}(\Gamma, \Phi, T, \tau) \\ \Gamma, \Phi, \varnothing \vdash \tau_0 \leq \tau_{arg_0} \Rightarrow \theta_0 \qquad \forall i_{\geq 1}.\ \Gamma, \Phi, \theta_{i-1} \vdash \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \\ \mathit{let}\ \tau_{inst} = \theta_n(\tau_{constr}) \qquad \Gamma, \Phi \vdash \tau_{inst} \equiv \tau \end{array}}{\Gamma, \Phi \vdash \langle T(e_0,\ ..\ , e_n) : \tau \rangle}$$

Typechecking Typedtree: Match case in detail

$$\text{Match}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash \langle e_s : \tau_s \rangle \qquad \forall i.\ \Gamma, \Phi, \Sigma \vdash \langle p_i : \tau_i \rangle \Rightarrow (\overline{v_i : \tau_i}), (\overline{\tau_{\exists_i}}), \Phi_i \\ \forall i.\ \mathit{let}\ \Gamma_i = \Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_i}) \oplus (\overline{\tau_{\exists_i}}) \qquad \forall i.\ \Gamma, \Phi_i \vdash \tau_s \equiv \tau_i \\ \forall i.\ \Gamma_i, \Phi_i \vdash \langle e_i : \tau_{e_i} \rangle \qquad \forall i.\ \Gamma, \Phi \vdash \tau_{e_i}\ \mathit{wf} \qquad \forall i.\ \Gamma_i, \Phi_i \vdash \tau \equiv \tau_{e_i}\end{array}}{\Gamma, \Phi \vdash \langle \texttt{match}\ e_s\ \texttt{with}\ \overline{\mid\ p \rightarrow e} : \tau \rangle}$$

Type system implementation

- Covers a large subset of OCaml 4.02: GADTs, polymorphic variants, modules.

  Not checked: Objects and classes, recursive modules, variance.
- Implementation in OCaml
  - $\sim$ **5 kLoC** purely functional (OCaml corresponding subset: $\sim$ **12.5 kLoC**)
  - Except data structures, no sharing of code with OCaml compiler library
- Some bugs checked from mantis reports: 6992 (GADT + contractivity in functor argument), 7222 (existential constructor out of scope)

Formalization and type soundness

Reimplemented in Coq, for type soundness.

- Locally nameless representation of variables with cofinite quantification (Engineering Formal Metatheory, [Aydemir, Charguéraud et al.])
  $\Rightarrow$ Common ground with previous works
  - Charguéraud: ML + Ref + Exceptions + ADTs
  - Garrigue: ML + Recursive types + Polymorphic Variants
- Explicit quantification of type scheme
  $\Rightarrow$ Easier formalization of instantiation
- Goal: ML + Exceptions + GADTs

Typedtree: Operational semantics

Small step semantics by substitution of bound variables.

For example:

```
⟨ let ⟨ f ⟨ x : 'a ⟩ : 'a -> 'a ⟩ = ⟨ x : 'a ⟩ in
⟨ ⟨ f : int -> int ⟩ ⟨ 0: int ⟩ : int ⟩ : int ⟩
```

reduces as

```
⟨ ⟨ fun ⟨ x : int ⟩ -> ⟨ x : int ⟩ : int -> int ⟩ ⟨ 0: int ⟩ : int ⟩
```

Conclusion

- Considering Typedtrees as typing proofs
- Sharing as less as possible of code with OCaml compiler
- Type checking discipline as formal specification of type system
- Implementation in OCaml:

  `github.com/OCamlpro/ocp-typechecker`

Conclusion

- Considering Typedtrees as typing proofs
- Sharing as less as possible of code with OCaml compiler
- Type checking discipline as formal specification of type system
- Implementation in OCaml:

  `github.com/OCamlpro/ocp-typechecker`

Ongoing work:

- Implementation and formalization in Coq
- Operational semantics of the Typedtree
- Type soundness of a subset of OCaml