Verifying type soundness in HOL for OCaml: the core language

Scott Owens and Gilles Peskine

University of Cambridge

Full-scale

Calculi and pragmatic additions

- Understand the pragmatics
- Ensure a working combination
- Program verification

The Goal

- Type soundness
- Accuracy

The Players

- Objective Caml
- HOL
- Ott

Outline

- OCaml specification
- Testing
- Experience

Ott asides

OCaml

- $\approx$ Core ML/Caml light
- No modules
- No objects

- Operational semantics: LTS
- Type system: declarative

OCaml: Types

$$
\begin{array}{lcl}
\mathit{typeconstr} & ::= & \mathit{typeconstr\_name} \\
 & | & \textbf{int} \mid \textbf{exn} \mid \textbf{list} \mid \textbf{option} \mid \textbf{ref} \\
 & | & \ldots
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{typexpr} & ::= & \alpha \\
 & | & \_ \\
 & | & \mathit{typexpr}_1 \longrightarrow \mathit{typexpr}_2 \\
 & | & \mathit{typexpr}_1 * \ldots. * \mathit{typexpr}_n \\
 & | & \mathit{typeconstr} \\
 & | & \mathit{typexpr}\ \mathit{typeconstr} \\
 & | & (\ \mathit{typexpr}_1\ ,\ \ldots\ ,\ \mathit{typexpr}_n\ )\ \mathit{typeconstr} \\
 & | & \ldots
\end{array}
$$

Ott Aside

Ott:

| typexpr1 * .... * typexprn :: :: tuple

Hol:

| TE_tuple of typexpr list

OCaml: Data

$$
\begin{array}{lcl}
\mathit{constr} & ::= & \mathit{constr\_name} \\
& | & \textbf{Match\_failure} \mid \textbf{None} \mid \textbf{Some} \\
& | & \ldots \\
\mathit{constant} & ::= & \mathit{int\_literal} \\
& | & \mathit{constr} \\
& | & \textbf{true} \mid \textbf{false} \mid [] \mid () \\
& | & \ldots \\
\mathit{unary\_prim} & ::= & \textbf{raise} \mid \textbf{ref} \mid \textbf{not} \mid ! \mid \sim- \\
\mathit{binary\_prim} & ::= & + \mid - \mid * \mid / \mid := \mid =
\end{array}
$$

OCaml: Patterns

$$
\begin{array}{lcl}
pattern & ::= & value\_name \\
& | & \_ \\
& | & pattern\ \textbf{as}\ value\_name \\
& | & pattern_1 :: pattern_2 \\
& | & constant \\
& | & pattern_1\ ,\ \ldots.\ ,\ pattern_n \\
& | & constr\ (\ pattern_1\ ,\ \ldots\ ,\ pattern_n\ ) \\
& | & constr\ \_ \\
& | & \{\ field_1\ =\ pattern_1\ ;\ \ldots\ ;\ field_n\ =\ pattern_n\ \} \\
& | & (pattern_1 \mid pattern_2)
\end{array}
$$

Ott Aside

```
| pattern as value_name  ::    :: alias
    (+ xs = xs(pattern) union value_name +)



| pattern1 , .... , patternn  ::    :: tuple
     (+ xs = xs(pattern1....patternn) +)
```

OCaml: Expressions

$$
\begin{array}{lcl}
expr & ::= & (\,\%\mathbf{prim}\ unary\_prim\,) \;\mid\; (\,\%\mathbf{prim}\ binary\_prim\,) \\
 & \mid & value\_name \\
 & \mid & constant \\
 & \mid & (\ expr\ :\ typexpr\ ) \\
 & \mid & expr_1\ ,\ ....\ ,\ expr_n \\
 & \mid & constr\,(\ expr_1\ ,\ ..\ ,\ expr_n\ ) \\
 & \mid & expr_1 :: expr_2 \\
 & \mid & \{\,field_1\ =\ expr_1\ ;\ \ldots\ ;\ field_n\ =\ expr_n\,\} \\
 & \mid & \{\,expr\ \mathbf{with}\ field_1\ =\ expr_1\ ;\ \ldots\ ;\ field_n\ =\ expr_n\,\} \\
 & \mid & expr\ .\ field \\
 & \mid & expr_1\ expr_2 \\
 & \mid & \ldots
\end{array}
$$

OCaml: Expressions

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \textbf{while}\ \mathit{expr}_1\ \textbf{do}\ \mathit{expr}_2\ \textbf{done} \\
& | & \textbf{for}\ x\ =\ \mathit{expr}_1\ [\textbf{down}]\textbf{to}\ \mathit{expr}_2\ \textbf{do}\ \mathit{expr}_3\ \textbf{done} \\
& | & \textbf{let}\ \mathit{pattern}\ =\ \mathit{expr}\ \textbf{in}\ \mathit{expr} \\
& | & \textbf{let rec}\ \mathit{letrec\_bindings}\ \textbf{in}\ \mathit{expr} \\
& | & \textbf{try}\ \mathit{expr}\ \textbf{with}\ \mathit{pattern\_matching} \\
& | & \mathit{location} \\
& | & \ldots
\end{array}
$$

$$\mathit{pattern\_matching} ::= \mathit{pattern}_1 \rightarrow \mathit{expr}_1 \mid \ldots \mid \mathit{pattern}_{\mathrm{n}} \rightarrow \mathit{expr}_{\mathrm{n}}$$

$$\mathit{letrec\_bindings} ::= \mathit{letrec\_binding}_1\ \textbf{and} \ldots \textbf{and}\ \mathit{letrec\_binding}_n$$

$$\mathit{letrec\_binding} ::= \mathit{value\_name} = \textbf{function}\ \mathit{pattern\_matching}$$

Ott Aside

```
| let rec letrec_bindings in expr ::  :: letrec
   (+ bind xs(letrec_bindings) in
           letrec_bindings +)
   (+ bind xs(letrec_bindings) in expr +)
```

OCaml: Definitions

$$
\begin{array}{lcl}
\textit{definition} & ::= & \textbf{let}\ \textit{let\_binding} \\
& | & \textbf{let rec}\ \textit{letrec\_bindings} \\
& | & \textit{type\_definition} \\
& | & \textit{exception\_definition}
\end{array}
$$

$$
\textit{type\_definition} \ ::= \ \textbf{type}\ \textit{typedef}_1\ \textbf{and}\ ..\ \textbf{and}\ \textit{typedef}_n
$$

$$
\textit{typedef} \ ::= \ \textit{type\_params}\ \textit{typeconstr\_name} \ = \ \textit{type\_information}
$$

$$
\begin{array}{lcl}
\textit{type\_information} & ::= & \textit{constr\_decl}_1 \,|\, \ldots \,|\, \textit{constr\_decl}_n \\
& | & \{\, \textit{field\_decl}_1 \,;\, \ldots \,;\, \textit{field\_decl}_n \,\}
\end{array}
$$

OCaml: Evaluation in Context

$$\frac{\vdash\ e_1 \xrightarrow{L} e_1'}{\vdash\ e_1\ v_0 \xrightarrow{L} e_1'\ v_0}$$

```
JR_expr (Expr_apply expr1 expr2) a1 a2 =
(∃ e1'.
   (a2 = Expr_apply e1' expr2) ∧
   is_value_of_expr expr2 ∧
   JR_expr expr1 a1 e1') ∨
...
```

OCaml: Evaluation in Context

$$\frac{\vdash\ e_1 \xrightarrow{L} e_1'}{\vdash\ e_1\ v_0 \xrightarrow{L} e_1'\ v_0}$$

OCaml: Store

$$\frac{}{\vdash \ \mathbf{ref}\ v \xrightarrow{\mathbf{ref}\ v = l} l}$$

$$\frac{}{\vdash !\, l \xrightarrow{!\, l = v} v}$$

OCaml: Primitives

```
let (+) = (-) in
  2 + 1
```

```
let f x = x 4 in
  f ((+) 1)
```

OCaml: Primitives

```
let (+) = (%uprim -) in
  2 + 1
```

```
let f x = x 4 in
  f ((%uprim +) 1)
```

OCaml: Currying

```
let f = function 1 -> function _ -> 0;;
f 2;;
```

```
let f = function 1 -> function _ -> 0
               | _ -> function _ ->
                         raise Match_failure;;
f 2;;
```

OCaml: Binding

```
type t = C of int;;
let v = C 1;;
type t = D of bool;;
let _ = match v with D (b) -> (b && false);;
```

OCaml: Binding

```
type t1 = C of int;;
let v = C 1;;
type t2 = C of bool;;
let _ = v;;
```

OCaml: Binding

```
let v1 = function x -> x;;
let x = 1;;
let v2 = v1 9;;
```

OCaml: Environments

$$
\begin{array}{lll}
E & ::= & \textbf{empty} \\
 & | & E, EB \\
EB & ::= & \textbf{TV} \\
 & | & value\_name \; : \; typescheme \\
 & | & constr\_name \; \textbf{of} \; typeconstr \\
 & | & constr\_name \; \textbf{of} \; \forall \, type\_params, \, (\, typexprs \,) \; : \; typeconstr \\
 & | & field\_name \; : \; \forall \, type\_params, \\
 & & \qquad typeconstr\_name \; \rightarrow \; typexpr \\
 & | & typeconstr\_name \; : \; kind \\
 & | & typeconstr\_name \; : \; kind \\
 & & \qquad \{\, field\_name_1 \, ; \, \ldots \, ; \, field\_name_n \,\} \\
 & | & location \; : \; typexpr
\end{array}
$$

Ott Aside

```
environment , E :: Env_ ::=
      {{ hol (environment_binding list) }}
  | empty  ::   :: nil
    {{ hol [] }}
  | E , EB  ::   :: cons
    {{ hol ([[EB]]::[[E]]) }}
  | EB1 , .. , EBn  :: M :: list
    {{ hol (REVERSE [[EB1 .. EBn]]) }}
  | E1 @ .. @ En  :: M :: tree
    {{ hol (FLAT (REVERSE [[E1 .. En]])) }}
```

OCaml: Polymorphism

$$\frac{\begin{array}{l}\mathbf{shift}\ 0\ 1\ \sigma^T \,\&\, E, \mathbf{TV} \vdash \mathit{pat} = \mathit{nexp} \ \triangleright\ x_1 : t_1, .., x_n : t_n \\ \sigma^T \,\&\, E \,@\, x_1 : \forall\, t_1, .., x_n : \forall\, t_n \vdash e : t\end{array}}{\sigma^T \,\&\, E \vdash \mathbf{let}\ \mathit{pat} = \mathit{nexp}\ \mathbf{in}\ e : t}$$

$$\frac{\begin{array}{l}E \vdash \mathit{value\_name} \ \triangleright\ \mathit{value\_name} : \mathit{ts} \\ E \vdash t \leq \mathit{ts}\end{array}}{E \vdash \mathit{value\_name} : t}$$

Ott Aside

```
| value_name : typexpr :: M :: vntype
  {{ com value binding with no universal
         quantifier }}
  {{ ich (EB_vn [[value_name]]
      (TS_forall (shiftt 0 1 [[typexpr]]))) }}
```

OCaml: Type Annotations

Does it have a type?

```
let f (x : 'a) : 'a = (x : 'a) && true;;
```

OCaml: Type Annotations

```
let f (x : 'a) : 'a = (x : 'a) && true;;
```

$f : \mathbf{bool} \rightarrow \mathbf{bool}$

OCaml: Type Annotations

```
let f (x : 'a) : 'a = (x : 'a) && true;;
```

$$\frac{\sigma^T \,\&\, E \vdash e : t \qquad E \vdash t \leq \sigma^T\, src\_t}{\sigma^T \,\&\, E \vdash (\, e : src\_t \,) : t}$$

$$\frac{\sigma^T \,\&\, E, \mathbf{TV} \vdash pat = nexp \;\rhd\; (\, x_1 : t'_1 \,), .., (\, x_k : t'_k \,)}{E \vdash \mathbf{let}\, pat = nexp : (\, x_1 : \forall\, t'_1 \,), .., (\, x_k : \forall\, t'_k \,)}$$

Testing

Our approach:

- Deterministic, small-step reduction function
- FP in a theorem prover
- Objective Caml's parser

Other approaches:

- Logic programming
- Big step

Testing

$$\frac{\vdash\ e_1 \xrightarrow{L} e_1'}{\vdash\ e_1\ v_0 \xrightarrow{L} e_1'\ v_0}$$

```
JR_expr (Expr_apply expr1 expr2) a1 a2 =
( ∃ e1'.
    (a2 = Expr_apply e1' expr2) ∧
    is_value_of_expr expr2 ∧
    JR_expr expr1 a1 e1') ∨
...

red (Expr_apply expr1 expr2) =
  red_2 expr1 expr2 Expr_apply eval_apply
```

Statistics: Proof

- 6 man-months
- 9K HOL, 561 lemmas
- 3.7K Ott
- $\approx$ 50 pages typeset
- Grammar: 251 productions, 55 non-terminals (142, 63 source)
- Type system: 173 rules, 28 relations
- Op. Sem.: 137 rules, 15 relations, 12 helper functions

Statistics: Testing

- 540 lines HOL
- 145 tests
- Full coverage

Proof in HOL

Operations on a goal:

- Rewriting and simplification
- Backward and forward chaining
- Witness $\exists$
- Case split
- First-order proof search (Metis, 1191 times)

Specification:

- Algebraic datatypes
- Inductive relations
- Well-founded recursion

Related Work

- Standard ML
  - Lee, Crary, Harper (POPL 2007)
  - van Inwegen (1996)
  - Maharaj, Gunter (1994)
  - Syme (1993)
- Java
  - Java: Klein, Nipkow (TOPLAS 2006)
  - Syme (1999)
  - Nipkow, van Oheimb (POPL 1998)
- C
  - Norrish (1998)

Conclusion

Can work at full-scale

- Need good tools
- Binding is a minor concern

Type Soundness: Binding

```
let x = (function _ ->
            let y = function w -> w in
              y) in
let z = x in
  z
```

$$(E \,@\, E)\,,\alpha$$

$$(E\,,\alpha \,@\, E)\,,\alpha$$

Testing

```
eval_uprim Uprim_ref v = StepAlloc (\e. e) v


eval_bprim Bprim_assign v1 v2 =
    case v1 of
       Expr_location l ->
         StepAssign (Expr_constant CONST_unit)
                    l
                    v2
    || _ -> Stuck
```