A Sound Semantics for OCaml$_{light}$

Scott Owens

OCaml$_{light}$ (ESOP '08) is a formal semantics for a substantial subset of the Objective Caml core language, suitable for writing and verifying real programs. It includes:

- definitions
  - variant data types (e.g., `type t = I of int | C of char`),
  - record types (e.g., `type t = {f : int; g : bool}`),
  - parametric type constructors (e.g., `type 'a t = C of 'a`),
  - type abbreviations (e.g., `type 'a t = 'a * int`),
  - mutually recursive combinations of the above (excepting abbreviations),
  - exceptions, and values;
- expressions for type annotations, sequencing, and primitive values (functions, lists, tuples, and records);
- `with` (record update), `if`, `while`, `for`, `assert`, `try`, and `raise` expressions;
- let-based polymorphism with an SML-style value restriction;
- mutually-recursive function definitions via `let rec`;
- pattern matching, with nested patterns, as patterns, and "or" (|)patterns;
- mutable references with `ref`, `!`, and `:=`;
- polymorphic equality (the Objective Caml = operator);
- 31-bit word semantics for `ints` (using an existing HOL library); and
- IEEE-754 semantics for `floats` (using an existing HOL library).

OCaml$_{light}$ key points

- Faithful to Objective Caml (very nearly)
- Type soundness proof mechanized in HOL
- Operational semantics validated on test programs
- Written in Ott
- Small-step operational semantics
  - Inductively defined relations
  - SOS-style
  - Labelled transitions for state
  - Substution-based
- Type system
  - Inductively defined relations
  - Syntactic
  - Declarative (non-algorithmic)

Proof Effort

A typical type soundness proof

- ≈ 7–8 man-months (including testing)
- Specification: 3.2K lines of HOL (from 4.0K Ott)
  - 143 constructors in 42 datatypes
  - 310 rules in 46 relations
- Proof: 9.5K lines of HOL
  - 17 files
  - 653 lemmas
  - 48 definitions

Challenges:

- Finding the right lemma using
  - my memory
  - lemma naming conventions
  - documentation (for library lemmas)
  - term matching on the theorem database
- Nested inductions
  - $expr = \ldots \mid Expr\_tuple$ **of** $expr$ **list** $\mid \ldots$
  - "Pointwise" reasoning vs. using separate lemmas
- Proof assistant generated names

Non-challenges:

- Managing De Bruijn indices

Testing

Executable small-step semantics

(Type soundness is insufficient)

- 145 tests
- Full coverage
- 540 line HOL definition of the executable semantics
- 1K line HOL proof of equivalence between declatative and executable semantics

$red : expr \to result$
type $result =$
$\quad Stuck$
$\mid Step$ **of** $expr$
$\mid StepAlloc$ **of** $(expr \to expr) * expr$
$\mid StepLookup$ **of** $(expr \to expr) * location$
$\mid StepAssign$ **of** $expr * location * expr$

```
match [] with
 x::y -> 1
 | [] -> 2
EXPECT
2

TESTSTUCK
x
END
```

Related Work

Mechanized metatheory for real-world languages

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

Representing Binding (straightforward)

User type variables	`'a`	concrete	`let f (x : 'a) : 'a = x && true;;`	`f : bool -> bool`

Value names	`x`	fully concrete (only need closed substs)	`let v = function x -> x;;`
			`let x = 1;;`
			`let w = v 9;;`
				`let x = 1;;`
				`let w = (function x -> x) 9;;`

Let-bound type variables		De Bruijn	`let f x =`
			` ... (let g x = ... in ...)  ...`
			`in`
			`let h x = ... f ...`
			`in`
			`...`	

Type names	`int`	fully concrete + no shadowing (following OCaml)	`type t = { f : int };;`
			`let v = { f = 1 };;`
			
			`type t = { g : bool };;`
			`let _ = v.g;;`
				`type t = { f : int };;`
				`type t = { g : bool };;`
				`let _ = { f = 1 }.g;;`

Constructor names
Field names
	`None`	fully concrete + no shadowing (a slight restriction)	`type t = C of int;;`
			`let v = C 1;;`
			
			`type u = C of bool;;`
			`let _ = v;;`
				`type t = C of int;;`
				`type u = C of bool;;`
				`let _ = C 1;;`