

A Sound Semantics for OCaml_{light}

Scott Owens

University of Cambridge

Abstract. Few programming languages have a mathematically rigorous definition or metatheory—in part because they are perceived as too large and complex to work with. This paper demonstrates the feasibility of such undertakings: we formalize a substantial portion of the semantics of Objective Caml’s core language (which had not previously been given a formal semantics), and we develop a mechanized type soundness proof in HOL. We also develop an executable version of the operational semantics, verify that it coincides with our semantic definition, and use it to test conformance between the semantics and the OCaml implementation. We intend our semantics to be a suitable substrate for the verification of OCaml programs.

1 Mechanizing Metatheory

Researchers in programming languages and program verification routinely develop their ideas in the context of core calculi and idealized models. The advantage of the core calculus approach comes from the efficacy of pencil-and-paper mathematics, both for specification and proof; however, these techniques do not scale well. Usable programming languages contain numerous constructs that are designed for practical utility rather than mathematical elegance, and their presence makes the proofs too long and tedious to check reliably by hand. Furthermore, the specifications themselves are subject to errors [1, 2]. Formal verification offers a better path: using a proof assistant to formalize an unambiguous semantics and to mechanize high-assurance proofs.

In this paper, we present a formal verification methodology that successfully scales to a programming language that includes a large complement of pragmatic constructs. Although our formal proofs are more detailed than informal pencil-and-paper proofs, they follow the same structure, use the same mathematical techniques, and the proof assistant ensures that they are correct. We did not have to invent new specification or reasoning techniques to succeed, but we did exercise care in the details of our formalization.

We demonstrate the methodology on a substantial fragment of Objective Caml [3, Chapter 1] that we call OCaml_{light} (since its feature set is roughly comparable to that of Caml Light). As a guiding design principle, we try to ensure that OCaml_{light} could form a substrate for applying program verification techniques to a significant subset of real OCaml programs.

We design and formalize the OCaml_{light} type system and operational semantics using Ott [4], a tool for expressing such specifications, and we use the HOL-4 proof assistant [5] to prove a type soundness theorem.¹ We also formalize a deterministic, exe-

¹ We verify all of our proofs in HOL-4, but our techniques apply in most systems based on higher-order logic (HOL), including Isabelle/HOL and Coq.

cutable version of the operational semantics, and prove that the two correspond. OCaml is not defined formally, so our semantics is a post-hoc attempt to detail the behavior of the language—both as described in the manual and as observed on the implementation. To build confidence that the semantics corresponds to the implemented language, we run test programs on both. Testing is a crucial component in the acceptance of post hoc semantics—we are not associated with the OCaml developers, and the test suite provides tangible evidence that our semantics is accurate.

Our development has taken under six man-months from start to finish, and based on our experience, we believe there is no reason why many well-established programming languages cannot be formally specified and given mechanized metatheories. To summarize our contribution, we demonstrate the feasibility of mathematical specification and mechanized reasoning for a useful fragment of a real-world programming language by:

- creating and formally specifying an operational semantics and type system for OCaml_{light} (Sect. 2). The semantics themselves are a novel contribution; no prior attempt to give OCaml a formal semantics appears in the literature;
- using HOL to mechanize a type soundness proof for OCaml_{light} (Sect. 3), the first of this scale in HOL; and
- demonstrating the reliability of our operational semantics with testing, enabled by the creation and verification of an executable version of the semantics (Sect. 4).

The remainder of this section explains our methodology and related work, and Sect. 5 presents observations and statistics on the mechanization. All of the formalizations, proofs, and tests are available at <http://www.cl.cam.ac.uk/~so294/ocaml>. The specification is written in Ott source, and is accompanied by HOL-4 and L^AT_EX versions that are generated automatically by Ott. Ott can also generate Isabelle and Coq versions.²

Methodology. Our formalization and proofs use standard techniques, all of which are supported by HOL. The abstract syntax of OCaml_{light} is represented by mutually recursive algebraic datatypes, and the type system (which is purely syntactic and non-algorithmic) is expressed with inductively defined relations. The operational semantics is small-step, and is expressed as an inductively defined labeled transition relation over source programs (the labels carry store actions that allow us to avoid explicitly threading the store through the rules). Value and type identifiers are not treated up to α -equivalence, because our semantics never reduces under a value variable binder, and well-typed programs have no free variables. The semantics can reduce under a type variable binding (i.e., the bound expression of a `let`), requiring α -renaming in our proofs; we choose to represent type variables using de Bruijn indices. In the proof we make extensive use of rule induction and structural induction principles. We also rely upon the ability to create functions using well-founded recursion [6], and upon an automated first-order prover [7].

Our OCaml_{light} formalization follows the formal syntactic specification (BNF) in the OCaml manual [3, Chapter 6] as closely as is feasible. This close connection with

² Thanks to Tom Ridge and Gilles Peskine for developing the respective translations.

the source language helps make the semantic rules directly accessible to semanticists who are familiar with OCaml, and we believe that direct accessibility is particularly advantageous in the context of program verification, especially by symbolic execution techniques [8–10]. A consequence of this choice is that our soundness proof has to deal directly with complex source-level constructs (e.g., n-tuples, n-way recursive lets, and full OCaml-style datatypes and pattern matching) that do not usually appear in lambda calculi. This complexity adds up to a relatively large language definition, roughly comparable, in terms of the number of constructs and semantic rules, to full-scale languages. Our methodology is relatively unsophisticated, but it handles this scaling. Our language does not involve the most semantically intricate features that appear in some programming languages, such as Standard ML’s module system, so our methodology remains untested for these situations.

Related Work. There has been extensive work on formalizing language semantics (e.g., POPLmark [11] solutions), and Java, C, and Standard ML (SML) have all been subject to large-scale developments in proof assistants, with varying degrees of success [12–19]. Type soundness has been proved for large subsets of Java [12, 15, 18], with a similar methodology to ours; however, the formalized versions of Java are significantly simpler than OCaml_{light}: for example, they lack parametric polymorphism and pattern matching. Norrish’s formalization of C [16] contains some metatheoretic results, but none of them approaches the scale of the OCaml_{light} type soundness proof.

The only prior mechanized type soundness proof for a realistic ML-like language (for SML) [13] follows a methodology that differs significantly from ours. Instead of formalizing SML directly, it uses an internal language (IL) into which SML, including its module system, can be elaborated ([13] proved type soundness only for IL, leaving a formalized elaboration to future work; this work has now been completed [20]). IL is essentially a heavily streamlined version of SML designed to yield an elegant and tractable mechanized soundness proof. In particular, IL does not directly support pattern matching, implicit polymorphism, or n-ary constructs; they are all compiled away by the elaboration. Unlike the IL soundness proof, we do not formalize a generative module system, but our system directly handles some features, such as datatypes and polymorphism, which are handled only by the module system in IL. However, OCaml’s semantics differs from SML’s in many details, some of which make our job easier (e.g., OCaml does not support local type or exception definitions, equality types, or overloading). Extending OCaml_{light} to add a full SML-like module system would likely require direct proof assistant support for reasoning about binding, or some amount of elaboration, or both.

Another important difference between the SML proof and ours is the setting of the mechanization; the SML proof was carried out in the Twelf proof assistant [21] which differs significantly from HOL:

- HOL is a classical, impredicative logic (with a model in ZFC set theory) whereas Twelf is constructive and predicative.
- Twelf supports higher-order abstract syntax (binding for the programming language being modeled is represented using the binding of the Twelf logic itself) which alleviates the burden of binding-related reasoning in proofs. HOL’s logic is not suited for higher-order abstract syntax representations of programming languages.

- Twelf’s proof system only supports $\forall\exists$ -theorems (i.e., no universal quantification is allowed under an existential quantifier). Syntactic soundness theorems fit into this restriction, but other theorems or program verifications might not (proofs by logical relations are a standard example). HOL faces no such restriction.
- HOL uses a powerful tactic-based proof system that allows common proof steps to be automated using SML programs (such as the first-order logic proof search mentioned above). However, learning to effectively use the system is a non-trivial task, and reading existing proofs can be difficult. Twelf uses a more accessible declarative proof style, but lacks this powerful automation.

Prior SML proof efforts tried to closely follow the mathematical specification of *The Definition of Standard ML* [22], but failed, in part because of its big-step operational semantics, in part because of its bugs, and in part because the proof assistant technology of the time was lacking [1, 2, 14, 17, 19]. The Metis [7] and TFL [6] packages, upon which we rely, did not exist at the time.

2 OCaml_{light}

To a crude approximation, our OCaml_{light} semantics is a core ML, excluding only modules and objects. In detail, it covers the following features, which form a complete language for writing programs without undue burden:

- definitions
 - variant data types (e.g., **type** $t = I$ of **int** | C of **char**),
 - record types (e.g., **type** $t = \{ f : \mathbf{int} ; g : \mathbf{bool} \}$),
 - parametric type constructors (e.g., **type** $'a$ $t = C$ of $'a$),
 - recursive and mutually recursive combinations of the above,
 - exceptions,
 - 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 OCaml **=** operator);
- 31-bit word semantics for **ints** (using an existing HOL library);
- IEEE-754 semantics for **floats** (using an existing HOL library).

OCaml_{light} overspecifies evaluation ordering relative to the OCaml manual (which makes no guarantees), generally going right-to-left in agreement with our observations of the OCaml bytecode implementation. On the above features it differs from the OCaml implementation in only the following four minor ways (the first three discrepancies could be repaired with a lightweight source-to-source elaboration):

```

typeconstr, tc ::= typeconstr_name | int | exn | list | option | ref | ...
typeexpr ::=  $\alpha$  | _ | typeexpr1 → typeexpr2 | typeexpr1 * ... * typeexprn | typeconstr
           | typeexpr typeconstr | ( typeexpr1, ..., typeexprn ) typeconstr | ...
constr ::= constr_name | Match_failure | None | Some | ...
constant ::= int_literal | constr | true | false | [] | () | ...
unary_prim ::= raise | ref | not | ! | ~-
binary_prim ::= = | + | - | * | / | :=
pattern ::= value_name | constant | { field1 = pattern1; ...; fieldn = patternn }
          | _ | pattern as value_name | ( pattern1 | pattern2 ) | ...
expr ::= (%prim unary_prim) | value_name | constant | ( expr : typeexpr )
        | expr1, ..., exprn | constr ( expr1, .., exprn ) | expr1 :: expr2
        | { field1 = expr1; ...; fieldn = exprn } | expr . field | expr1 expr2
        | { expr with field1 = expr1; ...; fieldn = exprn } | while expr1 do expr2 done
        | let pattern = expr in expr | let rec letrec_bindings in expr
        | try expr with pattern_matching | location | ...
pattern_matching, pat_mat ::= pattern1 → expr1 | ... | patternn → exprn
letrec_bindings, lrbs ::= letrec_binding1 and ... and letrec_bindingn
letrec_binding ::= value_name = function pattern_matching
definition ::= let let_binding | let rec lrbs | type_definition | exception_definition

```

Fig. 1. Grammar (excerpt)

- OCaml’s record expression evaluation ordering is right-to-left in the order of the labels from the record’s type definition, and in OCaml_{light} the ordering is right-to-left based on the record expression only;
- the behavior of partially applied, curried functions with non-exhaustive pattern matches can differ when the pattern matching fails at run time;
- the OCaml_{light} type system rejects programs with duplicate data constructor or record field name definitions; and
- the OCaml_{light} type system enforces a value restriction on let-based polymorphism, and rejects programs that require OCaml’s greater permissiveness.

In the future we would like to add support for type abbreviations, pattern matching guards (**when**), mutable records, arrays, and modules. We expect only the last of these to require significant changes to the formalization, and perhaps a lightweight elaboration, because of the more complex type theory underlying OCaml modules.

Although the grammar is too large to show here in its entirety (it has 251 productions), the (Ott generated) excerpt in Fig. 1 demonstrates its general flavor. In a few cases, we treat a source-level construct as syntactic sugar; for example **fun** is locally translated into **function**. Ott turns “ $x_1 \dots x_n$ ” in grammars and rules into corresponding list-based HOL code.

2.1 Operational Semantics

The (small-step) operational semantics of OCaml_{light} is phrased as a relation on definitions, programs, and stores; Fig. 2 gives an overview of the main relations and a few of

$L ::= \epsilon \mid \mathbf{ref} \text{ value} = \text{location} \mid ! \text{location} = \text{value} \mid \text{location} := \text{value}$
 $\text{program} ::= \text{definitions} \mid (\% \mathbf{prim \ raise}) \text{ expr}$

$\vdash \langle \text{definitions}, \text{program}, \text{store} \rangle \longrightarrow \langle \text{definitions}', \text{program}', \text{store}' \rangle \quad \vdash \text{expr} \xrightarrow{L} \text{expr}'$
 $\vdash \langle \text{definitions}, \text{program} \rangle \xrightarrow{L} \langle \text{definitions}', \text{program}' \rangle \quad \vdash \text{store} \xrightarrow{L} \text{store}'$
 $\vdash \text{expr matches pattern} \triangleright \{\{ \text{subst}_x \}\} \quad \vdash \text{expr matches pattern}$
 $\vdash v \text{ with pattern_matching} \longrightarrow \text{expr}$

$$\frac{\vdash e_1 \xrightarrow{L} e'_1}{\vdash e_1 v_0 \xrightarrow{L} e'_1 v_0} (1) \quad \frac{}{\vdash \mathbf{ref} v \xrightarrow{\mathbf{ref} v = l} l} (2)$$

$$\frac{\vdash v \text{ matches pat} \triangleright \{\{ x_1 \leftarrow v_1, \dots, x_m \leftarrow v_m \}\}}{\vdash v \text{ with pat} \rightarrow e \mid \text{pat}_1 \rightarrow e_1 \mid \dots \mid \text{pat}_n \rightarrow e_n \longrightarrow \{\{ x_1 \leftarrow v_1, \dots, x_m \leftarrow v_m \}\} e} (3)$$

$$\frac{\vdash v_1 \text{ matches pat}_1 \triangleright \{\{ \text{subst}_{x_1} \}\} \quad \dots \quad \vdash v_n \text{ matches pat}_n \triangleright \{\{ \text{subst}_{x_n} \}\}}{\vdash (v_1, \dots, v_n) \text{ matches } (\text{pat}_1, \dots, \text{pat}_n) \triangleright \{\{ \text{subst}_{x_1} @ \dots @ \text{subst}_{x_n} \}\}} (4)$$

$$\frac{\neg(v \text{ matches pat}_1) \quad \vdash v \text{ matches pat}_2 \triangleright \{\{ x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n \}\}}{\vdash v \text{ matches pat}_1 \mid \text{pat}_2 \triangleright \{\{ x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n \}\}} (5) \quad \frac{\vdash v \text{ matches pat}_2}{\vdash v \text{ matches pat}_1 \mid \text{pat}_2} (6)$$

$$\frac{\text{lrbs} = (x_1 = \mathbf{function} \text{ pat_mat}_1 \text{ and } \dots \text{ and } x_n = \mathbf{function} \text{ pat_mat}_n)}{\mathbf{recfun}(\text{lrbs}, \text{pat_mat}) \triangleright} (7)$$

$$\{\{ x_1 \leftarrow \mathbf{let} \mathbf{rec} \text{ lrbs in } x_1, \dots, x_n \leftarrow \mathbf{let} \mathbf{rec} \text{ lrbs in } x_n \}\} (\mathbf{function} \text{ pat_mat})$$

$$\text{lrbs} = (x_1 = \mathbf{function} \text{ pat_mat}_1 \text{ and } \dots \text{ and } x_n = \mathbf{function} \text{ pat_mat}_n)$$

$$\mathbf{recfun}(\text{lrbs}, \text{pat_mat}_1) \triangleright e_1 \quad \dots \quad \mathbf{recfun}(\text{lrbs}, \text{pat_mat}_n) \triangleright e_n$$

$$\vdash \mathbf{let} \mathbf{rec} \text{ lrbs in } e \longrightarrow \{\{ x_1 \leftarrow e_1, \dots, x_n \leftarrow e_n \}\} e$$

$$\frac{}{\vdash \mathbf{try} (\% \mathbf{prim \ raise}) v \text{ with pattern_matching} \longrightarrow \mathbf{match} v \text{ with pattern_matching} \mid _ \rightarrow ((\% \mathbf{prim \ raise}) v)} (9)$$

$$\vdash \text{store} \xrightarrow{L} \text{store}'$$

$$\frac{\vdash \langle \text{definitions_value}, \text{program} \rangle \xrightarrow{L} \langle \text{definitions}, \text{program}' \rangle}{\vdash \langle \text{definitions_value}, \text{program}, \text{store} \rangle \longrightarrow \langle \text{definitions}, \text{program}', \text{store}' \rangle} (10)$$

Fig. 2. Operational semantics (excerpt)

the 137 rules to illustrate interesting aspects of OCaml_{light} and the formalization (*value* and *v* indicate the Ott-enforced value grammar for *expr*).

Contexts. We specify evaluation-in-context with a collection of congruence rules (e.g., Rule 1). Alternative evaluation-context based approaches (in which the context information is encapsulated into a grammar-with-a-hole and hole filling function) can provide a more compact formalization, but initial experimentation showed them to be less convenient for HOL manipulation. HOL's automation works well with congruence rules directly, whereas the process of interpreting the hole filling on a data object (the evaluation context itself) introduced significantly more overhead.

Rule 1 incorporates right-to-left evaluation ordering for OCaml_{light}. Dropping the ordering requirement would require only trivial changes to the reduction rules and soundness proof, but looking ahead to testing, or even program verification via symbolic execution [8–10], we believe the benefits of a deterministic semantics outweigh the drawbacks of slightly over-specifying the language.

Store. The reduction rules are annotated with labels (L in Fig 2). Whenever the store must be consulted or updated (e.g., Rule 2), the label records the relevant information, both input from and output to the store. Thus, the store does not need to be threaded through the semantics, a formal version of the informal “state convention” of *The Definition of Standard ML*.³ Rule 10 correlates the program’s reduction with a reduction in the store. We anticipate using the labels to conveniently add other features, including I/O and concurrency.

Value Binding. All variable binding in OCaml_{light} is expressed through pattern matching, and the pattern matching rules implement binding with (parallel) substitutions (Rules 3 and 4).⁴

Rule 5 for matching the right side of an “or” pattern must check that the left pattern does not match. HOL’s rule induction package forbids a recursive call to the pattern matching relation under the requisite negation because such constructions are not inductive in general. We use a previously defined relation that simply checks pattern matching without building the substitution. This relation is acceptable to HOL because it does not need to discriminate between the cases where the first or second pattern match (Rule 6).

Recursive bindings use a substitution that replaces the bound variables with the entire recursive binding construct (Rules 7 and 8). This is one place where working directly on OCaml source complicates the specification and metatheory; the rule for a single function recursive `let` would be significantly simpler.

Primitives. We denote primitive operators with a special symbol `%prim` to avoid confusing primitives (e.g., `(%prim +)`), which the semantics must directly interpret, with variables (e.g., `+`), which can be rebound in the source program. Furthermore, it distinguishes partially applied binary primitives (e.g., `(%prim +)0`), which are values, from other applications, which are not. Evaluation starts with a substitution that replaces identifiers in the initial environment with the corresponding `%prim` values (e.g., substituting `(%prim +)` for `+`).

Exceptions. For each congruence rule, there is a corresponding exception rule that discards the immediate context. When an exception reaches a `try` expression it is matched against the `with` portion (Rule 9). If it reaches the top level of a program, the rest of the program is discarded, and no further evaluation occurs.

Curried functions. The semantics of OCaml_{light} differs slightly from OCaml in its treatment of partial pattern matches in curried functions. We reduce function applications one-at-a-time whereas the OCaml implementation does not reduce a curried

³ The evaluation context approach can also avoid store threading.

⁴ We write `cons` as `,` and `append` as `@`.

function until all of its arguments are available. Thus, the following program raises a pattern matching exception on our semantics, but not in the implementation.

```
let f = function 1 → function _ → 0;; let _ = f 2;;
```

Specifying the implemented behavior would entail using the same elaborative pre-pass for detecting multiple-argument functions as the OCaml compiler. We do not believe this to be worth the effort given that the departure is small, and it is only observable in the presence of non-exhaustively matched patterns (whose existence the compiler can detect) which furthermore fail at run time.

2.2 Type System

Figure 3 gives an overview of our type system for OCaml_{light}, along with a few of its 173 rules. The type system is mostly syntax directed, but non-algorithmic, due to declarative handling of polymorphism and recursion.

Environments. The E productions describe the environments used by the type system. A binding, EB , can be one of the following, in order: a de Bruijn type variable binding (**TV**), a value binding; a constant data constructor; a parameterized data constructor; a variant type constructor; a record’s field name; a record type constructor; or a store location. If our type system checked pattern matches for exhaustiveness, variant type constructors would have to keep a constructor list similar to a record type constructor’s field list. Location bindings are introduced only by the top-level store, and type constructor, field, and value constructor bindings are introduced only by top-level definitions. Type variable and value bindings are introduced by `let` expressions, with unbounded nesting.

In an `ok` environment E , all type constructor and type variable references are bound by a prior EB . Also, E contains no duplicate bindings for a location, type constructor, value constructor, or field name. The type constructor restriction (which is enforced in OCaml) is necessary for type soundness; for example, the following program, which gets stuck, would otherwise typecheck since v has type t which, at the field access site, is specified to have a field g :

```
type t = { f : int };; let v = { f = 1 };; type t = { g : bool };; let _ = v.g
```

The constructor and field restrictions do not preclude programs that get stuck, but they are necessary for type preservation (and hence for our soundness proof). In the following example, once v is substituted, the enclosing binding of C is different, and $C\ 1$ can no longer be typed.

```
type t = C of int;; let v = C 1;; type u = C of bool;; let _ = v
```

The *absence* of a similar restriction on value name repetition is also necessary for type preservation, because we do not treat value names up to α -equivalence. The following example does not start with a repeated, nested binding, but immediately after v is substituted, x is duplicated in the environment.

```
let v = function x → x;; let x = 1;; let w = v 9
```

$$\begin{aligned}
E &::= \mathbf{empty} \mid E, EB \\
EB &::= \mathbf{TV} \mid \text{value_name} : \text{typescheme} \mid \text{constr_name} \mathbf{of} \text{typeconstr} \\
&\quad \mid \text{constr_name} \mathbf{of} \forall \text{type_params}, \text{typeexprs} : \text{typeconstr} \mid \text{typeconstr_name} : \text{kind} \\
&\quad \mid \text{field_name} : \forall \text{type_params}, \text{typeconstr_name} \rightarrow \text{typeexpr} \\
&\quad \mid \text{typeconstr_name} : \text{kind} \{ \text{field_name}_1; \dots; \text{field_name}_n \} \mid \text{location} : \text{typeexpr} \\
\sigma^T &::= \{ \alpha_1 \leftarrow \text{typeexpr}_1, \dots, \alpha_n \leftarrow \text{typeexpr}_n \} \\
E \vdash \mathbf{ok} & \qquad \qquad \qquad E \vdash \text{typeexpr} : \text{kind} \\
\sigma^T \& E \vdash \text{pattern} : \text{typeexpr} \triangleright E' & \quad \sigma^T \& E \vdash \text{expr} : \text{typeexpr} \\
E \vdash \text{definition} : E' & \quad E \vdash \text{program} : E' \\
E \vdash \text{store} : E' & \quad E \vdash \langle \text{program}, \text{store} \rangle \\
\mathbf{shift} 0 \mid \sigma^T \& E, \mathbf{TV} \vdash \text{pat} = \text{nexp} \triangleright x_1 : t_1, \dots, x_n : t_n & \\
\sigma^T \& E @ x_1 : \forall t_1, \dots, x_n : \forall t_n \vdash e : t & \\
\hline
\sigma^T \& E \vdash \mathbf{let} \text{pat} = \text{nexp} \mathbf{in} e : t & \quad (11) \\
E \vdash \text{value_name} \triangleright \text{value_name} : \text{ts} & \quad E \vdash \forall t' : \mathbf{Type} \\
E \vdash t \leq \text{ts} & \quad E \vdash t_1 : \mathbf{Type} \dots E \vdash t_n : \mathbf{Type} \\
\hline
E \vdash \text{value_name} : t & \quad \{ \{ t_1, \dots, t_n \} t' \triangleright t'' \} \\
\hline
\sigma^T \& E \vdash e : t & \quad \sigma^T \& E \vdash e : t \\
E \vdash t \leq \sigma^T \text{src_}t & \quad E \vdash \text{field_name} : t \rightarrow t' \\
\hline
\sigma^T \& E \vdash (e : \text{src_}t) : t & \quad \sigma^T \& E \vdash e.\text{field_name} : t' \quad (14) \quad (15) \\
E @ E' \vdash \text{store} : E' & \quad E \vdash \text{store} : E' \\
E @ E' \vdash \text{program} : E'' & \quad \{ \} \& E \vdash v : t \\
\hline
E \vdash \langle \text{program}, \text{store} \rangle & \quad E \vdash \text{store}, l \mapsto v : E', (l : t) \quad (16) \quad (17) \\
E \vdash \text{field_name} \triangleright \text{field_name} : \forall (\alpha_1, \dots, \alpha_m), \text{typeconstr_name} \rightarrow t & \\
E \vdash (t'_1, \dots, t'_m) \text{typeconstr_name} \rightarrow t'' \leq & \\
\forall (\alpha_1, \dots, \alpha_m), (\alpha_1, \dots, \alpha_m) \text{typeconstr_name} \rightarrow t & \\
\hline
E \vdash \text{field_name} : (t'_1, \dots, t'_m) \text{typeconstr_name} \rightarrow t'' & \quad (18) \\
\sigma^T \& E, \mathbf{TV} \vdash \text{pat} = \text{nexp} \triangleright (x_1 : t'_1), \dots, (x_k : t'_k) & \\
\hline
E \vdash \mathbf{let} \text{pat} = \text{nexp} : (x_1 : \forall t'_1), \dots, (x_k : \forall t'_k) & \quad (19)
\end{aligned}$$

Fig. 3. Type system (excerpt)

Polymorphism. A **let** expression with a non-expansive binding introduces a type variable binding into E . The type of the binding can refer to this variable as a well-formed, but opaque, type. When the binding's type is added to E to check the **let** body, it is first generalized into a type scheme (Rule 11). Where the body refers to the binding, the type scheme is instantiated to a type that is valid at the use point (Rules 12 and 13).⁵

One of the more subtle aspects of OCaml_{light} is the treatment of type variables that appear in explicit annotations. These are scoped by top level definitions, and stand in for arbitrary types. Thus, the top-level **let** definition rule (19) creates a substitution σ^T

⁵ Each **TV** actually introduces an infinite collection of type variables, indexed by numbers, because a **let** expression can introduce any number of type variables.

that supplies the types for these variables to Rule 14. Substituting `bool` for $'a$, f in the following program has type `bool` \rightarrow `bool`, which agrees with OCaml.⁶

$$\text{let } f(x : 'a) : 'a = x \&\& \text{true}$$

Stores. Since the store can contain cyclic structures, it is type checked in a context that includes E' , the types of the locations in the store (Rule 16). Values in the store cannot have type variables in their enclosed type annotations (hence the empty substitution appears in the Rule 17), because such variables would have escaped their scope at a top-level definition. Thus, before placing a value into the store, the operational semantics replaces all of its type variables with the wildcard type variable $_$.

Records and Variants. Rules 15 and 18 show how expressions and patterns consult the environment for the types of the fields and constructors.

3 Type Soundness

The type soundness theorem (Theorem 1) ensures that a well-typed program does not get stuck. Our mechanized proof follows the standard methodology of preservation and progress lemmas. These lemmas are proved at three levels: for expressions, for definitions and for top level definition/program/store tuples. They rely on a typical collection of other main lemmas: substitution, weakening, strengthening, type substitution, and validity. The number and size of these lemmas prevents a full presentation here, so we instead highlight several aspects.

Theorem 1. If both $\vdash \langle \epsilon, \text{program}, \text{store} \rangle \longrightarrow^* \langle \text{definitions}', \text{program}', \text{store}' \rangle$ and $\epsilon \vdash \langle \text{program}, \text{store} \rangle$ then either $(\text{program}' = \epsilon)$, or $(\exists \text{value}. \text{program}' = (\% \text{prim raise}) \text{value})$, or $\exists \text{definitions}'' \text{ program}'' \text{ store}''$.
 $\vdash \langle \text{definitions}', \text{program}', \text{store}' \rangle \longrightarrow \langle \text{definitions}'', \text{program}'', \text{store}'' \rangle$.

Type Variable Binding. We use a de Bruijn index encoding of type variables. Shift operations appear in the rules of the type system 19 times, 16 of which occur to generalize a type into a type scheme from a non-polymorphic binder. Since the type variables that appear in source language type annotations are handled with substitutions (Sect. 2.2), the de Bruijn encoding does not require changing the source language at all: indices and shifts exist only in the type rules and the soundness proof (but not the operational semantics). Thus, the de Bruijn encoding does not substantially complicate OCaml_{light}.

Unlike value variables, type variables must not be repeated in the context. Otherwise, a type scheme generalization in one `let` expression might capture a type variable introduced by a different `let` expression. The prohibition on repetition is exactly the reason that α -renaming is required in the soundness proof. In the following example, the first evaluation step substitutes the middle `let` underneath the type variable binding introduced by the rightmost `let`.

⁶ The corresponding program in SML is not well typed.

$\text{let } x = (\text{function } _ \rightarrow \text{let } y = \text{function } w \rightarrow w \text{ in } y) \text{ in let } z = x \text{ in } z$

In the proof, this situation corresponds to the polymorphic **let** case of the weakening lemma. Given a **let** expression and typing derivation in $E_1 @ E_2$, we must show that it has a type in $E_1, 'a @ E_2$. In the given derivation, the **let** extends the environment with a type variable that might be $'a$. If so, a new derivation cannot be formed without renaming one of the type variables.

We investigated two proof approaches that do not use an α -aware representation for type variables. The first approach begins by showing an equivariance property: that the result of consistently renaming the type variables in a typing derivation remains a typing derivation. Equivariance is then used in the weakening proof to rename the type variable added by the **let**. The drawback of this approach is that weakening cannot be proved with a rule induction because the renamed derivation does not match up with the induction hypothesis. Instead the induction must be on derivation heights, which requires a significant amount of additional work to formalize in HOL. The second approach combines the statements of equivariance and weakening into a single lemma, which restores the ability to do rule induction.⁷ Ultimately, we concluded that both approaches required significantly more work to mechanize than the de Bruijn representation, where the equivariance result is intrinsic to the representation. Since the de Bruijn representation was easy to work with in the proof, we did not consider other equivariant representations.

Lemma 1 states expression weakening for a single de Bruijn type variable.

Lemma 1. If $(\sigma^T \& E_2 @ E_1 \vdash \text{expr} : \text{typeexpr})$ then
 $\sigma^T \uparrow_{\text{num_tv}(E_1)}^1 \& E_2, \mathbf{TV} @ E_1 \uparrow_0^1 \vdash \text{expr} : \text{typeexpr} \uparrow_{\text{num_tv}(E_1)}^1$.

Labels. The preservation and progress lemmas are split into pieces according to the transition labels. Lemma 2 states that a well typed program can take a step with some label, and Lemma 3 allows the label to be altered so that it works for a given store. The $\vdash \text{store} \xrightarrow{L} \text{store}'$ relation updates a store according to a label.

Lemma 2. Suppose that E has no value bindings. If $\sigma^T \& E \vdash \text{expr} : \text{typeexpr}$ then either expr is a value, or $(\exists \text{value}. \text{expr} = (\% \text{prim raise}) \text{value})$, or $(\exists L \text{expr}'. \vdash \text{expr} \xrightarrow{L} \text{expr}')$.

Lemma 3. Suppose that all of the locations in expr are bound to values in store . If $\vdash \text{expr} \xrightarrow{L} \text{expr}'$ then $\exists L' \text{expr}' \text{store}'. \vdash \text{expr} \xrightarrow{L'} \text{expr}' \wedge \vdash \text{store} \xrightarrow{L'} \text{store}'$.

The statement of preservation (Lemma 4) relies on two relations for checking labels. The first $\sigma^T \& E \vdash L$ ensures that the parts of L that are input into the expression reduction (e.g., the value of a location dereference) are well formed and well typed. The second $\sigma^T \& E \vdash L \triangleright E'$ ensures that the output parts of L (e.g., the location being dereferenced) are well formed and well typed. It also gives the environment bindings created by the output. Thus, the input relation appears in the preservation statement's assumptions and the output in its conclusions.

⁷ Thanks to Tom Ridge for this observation.

Lemma 4. Suppose that E has no value bindings.

If $\vdash \text{expr} \xrightarrow{L} \text{expr}'$ and $\sigma^T \& E \vdash \text{expr} : \text{typeexpr}$ and $\sigma^T \& E \vdash L$ then
 $\exists E'. \sigma^T \& E \vdash L \triangleright E' \wedge \sigma^T \& E @ E' \vdash \text{expr}' : \text{typeexpr}$.

The top level preservation theorem for program/store tuples relies on Lemma 4 (lifted to definition sequences) and two other lemmas. The first states that updating a well typed store with a well typed (by the expression output label typing relation) label gives a well typed store. The second states that if a well typed store is updated with a label then the label is well typed (by the expression input label). These parts of the proof are not particularly difficult: the key insight is to split the label checking between two relations.

Store Typing. Not only can the store contain cyclic references as mentioned in Sect. 2.2, but it can also contain constructed and record values. To ensure preservation, the store must be typed in an environment that has bindings for any type definitions that might be needed. Thus, the operational semantics builds a separate record of type definitions as it encounters them. Intermediate computation steps are type checked by converting the type definitions into an environment, then checking the store in this environment, and finally checking the unevaluated definitions with both the type definitions' and the store's environments.

4 Testing and Determinism

Because creating a large semantics is an error-prone activity [1, 2], we run a test suite of programs on the semantics to build confidence in its accuracy. Crucially, we can transfer this confidence to others by showing them the test suite. To our knowledge, the testing of full-language-scale semantics has been previously carried out only for the Scheme semantics in the PLT Redex term rewriting system [23] and in ACL2 for a Java Virtual Machine [9, 24] (symbolic execution is part of the standard ACL2 methodology, and it has often been applied to test full-scale hardware formalizations). Although both the type system and operational semantics should be tested, our focus here is on the operational semantics, leaving the type system for future work.

Although a mechanized type soundness proof rules out certain kinds of errors, it falls far short of ensuring that the semantics accurately models the intended language. In fact, we discovered an error while preparing the executable semantics, before testing even began. While proving Lemma 5 we discovered that we had omitted the negated premise of the “or” pattern matching rule (Sect. 2.1). This mistake allowed the semantics to non-deterministically return incorrect results in some cases, while remaining perfectly type sound.

Another class of specification mistakes arises from pattern-matching rules that do not find a match when they should. These mistakes do not cause type soundness violations because a `Match_failure` exception is raised when no patterns match. For example, after finishing the type soundness proof, we realized that the rule for record patterns was incorrectly requiring the field lists to be in the same order for both the pattern and value. However, the pattern matcher can be tested for this sort of bug.

Evaluation Function. The first step toward testing the semantics is to create an executable version of it. Although the relational formulation of the semantics could in principle be executed as a logic program (as Twelf does), we chose to create a functional version for execution. This is in part because HOL-4 does not currently support relational execution, but can evaluate functions (and can generate ML code from functions for greater speed). Additionally, since the semantics is deterministic, we would like to prove that fact. We prove determinacy and support testing with the same technique: we create a functional version of the semantics and prove it equal to the relational version.

We use the ability to do functional programming in HOL’s logic to implement the single-step execution function using common patterns of functional abstraction to reduce the amount of redundant code and make the definition tractable for a language the size of OCaml_{light}. Because HOL is a logic of total functions only, we must prove that the functions always terminate. The proof is not difficult, but the mutually recursive helper functions used to abstract out common patterns make it complicated enough that HOL-4 does not prove termination automatically.

To maintain the correspondence with the labeled transition relation, the store is not an input to the expression reduction function (`red`); instead, the result type for `red` includes cases for interactions with the store. For example, the result for a store lookup reduction indicates the location of the reference, and it supplies a function that is applied to the value in the location to determine the result. Lemma 5 correlates the relational and functional expression semantics using `interp_result`, which applies the information in a label to a reduction result.

Lemma 5. $\vdash \text{expr} \xrightarrow{L} \text{expr}'$ iff `interp_result(red(expr), L) = Some expr'`.

Unlike the expression semantics, the top-level semantics is not deterministic: the location allocator can use any unallocated location for the new reference. Thus, the `top_red` function is parameterized over an allocation function that, for a given store, returns the next location to use.

Theorem 2. Say that *alloc* is good if `alloc(store)` never results in a location already mapped in *store*.

$\vdash \langle \text{definitions}, \text{program}, \text{store} \rangle \longrightarrow \langle \text{definitions}', \text{program}', \text{store}' \rangle$ iff
 $\exists \text{alloc}. \text{alloc is good} \wedge$
 $\text{top_red}(\text{alloc}, (\text{definitions}, \text{store}, \text{program})) = \text{Some}(\text{definitions}', \text{store}', \text{program}')$.

Test Suite. Our test suite currently contains 145 test cases, each designed to test one or two language features. A test case comprises a program and its expected result, both in OCaml syntax. We convert both into abstract syntax trees using the parser and (slightly modified) AST printer from the OCaml implementation. We use HOL-4’s SML code generation on the reduction functions and test ASTs to execute the test cases. The test suite fully covers the reduction function definition.⁸

⁸ We check coverage using Standard ML of New Jersey’s coverage checking tool. This ensures that every application point is taken by the test suite.

5 Discussion

The primary challenge we encountered in our formalization and mechanization was the scale of OCaml_{light}. The HOL proof techniques we use are typical, and up to the task: rewriting with equational theorems, backward and forward chaining with implicational theorems (including induction principles), instantiating existentially quantified variables, case-splitting, and doing first-order proof search. However, the scale of the language provided a constant source of friction. For example, proof cases involving arbitrarily sized constructs, such as tuples and records, often require inductive lemmas whereas their fixed size counterparts, such as “and” expressions, only involve a case split.

As to the scale, the grammar has 251 productions in 55 non-terminals. Of these, 142 and 36 (respectively) are parts of the source language, and the rest are used in intermediate results and to support the type system and semantics (e.g., notation for substitution). The type system comprises 173 rules that define 28 relations, and the operational semantics comprises 137 rules that define 15 relations. These relations rely on 12 helper functions in addition to the substitution and free variable functions generated by Ott. In total, the specification comprises about 3700 lines of Ott source. The evaluation function is 540 lines of typical functional programming (although in HOL). The proof contains about 9000 lines of HOL broken into 561 stated lemmas and theorems. The entire development has taken the author approximately 6 months to create.

We want to be able to extend the OCaml_{light} specification easily. We believe that the specification achieves this goal thorough the combination of Ott, labeled transitions, and careful planning ahead. However, the HOL proof scripts are fragile with respect to specification changes, including changing variable names or rule ordering. We have taken some initial steps, in conjunction with Ott, to alleviate some of the ordering problems. However, for proofs over definitions as large as OCaml_{light}, acceptable flexibility will come only from significant advances in proof assistant technology for proof maintenance and extension.

6 Conclusion

We have formally specified a type system and operational semantics for a substantial fragment of OCaml. We validated them by proving their type soundness in HOL, and by testing the semantics on a thorough test suite. Throughout the process we have maintained a close connection between our formalization and OCaml source code. Our effort is the first of its complexity with this goal—a goal motivated by our view that the mechanized specification and proofs are not only a final product, but also a starting point for the verification of OCaml programs.

Acknowledgments. We thank Gilles Peskine for his collaboration on parts of the formalization and for sharing his expertise in the OCaml language, and Peter Sewell and Francesco Zappa Nardelli for their work on the initial formalization. We acknowledge the support of EPSRC grants GR/T11715/01 and EP/C510712/1.

References

1. Kahrs, S.: Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh (April 1993)
2. Rossberg, A.: Defects in the revised definition of Standard ML. Technical report, Saarland University, Saarbrücken, Germany (October 2001) Updated 2007/01/22.
3. Leroy, X.: The Objective Caml System. 3.10 edn. (2007) <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>.
4. Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. In: Proc. ICFP. (2007)
5. Norrish, M., Slind, K.: HOL-4. <http://hol.sourceforge.net/>.
6. Slind, K.: Reasoning about Terminating Functional Programs. PhD thesis, Institut für Informatik, Technische Universität München (1999)
7. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Proc. Design and Application of Strategies/Tactics in Higher Order Logics. (2003)
8. Compton, M.: Stenning's protocol implemented in UDP and verified in Isabelle. In: Proc. Australasian Symposium on Theory of Computing. (2005)
9. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In: Proc. Theorem Proving in Higher Order Logics, LNCS 3223. (2004)
10. Ridge, T.: Operational reasoning for concurrent Caml programs and weak memory models. In: Proc. Theorem Proving in Higher Order Logics, LNCS 4732. (2007)
11. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark Challenge. In: Proc. TPHOLS, LNCS 3603. (2005)
12. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. *Trans. on Prog. Lang. and Systems* **28**(4) (2006) 619–695
13. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of Standard ML. In: Proc. Principles of Programming Languages. (2007)
14. Maharaj, S., Gunter, E.L.: Studying the ML module system in HOL. In: Proc. Higher Order Logic Theorem Proving and Its Applications, LNCS 859. (1994)
15. Nipkow, T., van Oheimb, D.: *Javalight* is type-safe — definitely. In: POPL. (1998)
16. Norrish, M.: C Formalised in HOL. PhD thesis, University of Cambridge (1998)
17. Syme, D.: Reasoning with the formal definition of Standard ML in HOL. In: Proc. Higher Order Logic Theorem Proving and its Applications, LNCS 780. (1993)
18. Syme, D.: Proving Java type soundness. In: Formal Syntax and Semantics of Java, Springer-Verlag (1999) 83–118
19. VanInwegen, M.: The Machine-Assisted Proof of Programming Language Properties. PhD thesis, University of Pennsylvania (1996)
20. Harper, R.: personal correspondence (2007)
21. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. *Journal of Functional Programming* **17**(4–5) (2007) 613–673
22. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). The MIT Press (1997)
23. Matthews, J., Findler, R.B.: An operational semantics for Scheme. *Journal of Functional Programming* *To appear*.
24. Moore, J.S.: Symbolic simulation: An ACL2 approach. In: FMCAD, LNCS 1522. (1998)