Formal Semantics and Program Logics for a Fragment of OCaml

REMY SEASSAU, Inria, France
IRENE YOON, Inria, France
JEAN-MARIE MADIOT, Inria, France
FRANÇOIS POTTIER, Inria, France

This paper makes a first step towards a formal definition of OCaml and a foundational program verification environment for OCaml. We present a formal definition of OLang, a nontrivial sequential fragment of OCaml, which includes first-class functions, ordinary and extensible algebraic data types, pattern matching, references, exceptions, and effect handlers. We define the dynamic semantics of OLang as a monadic interpreter. This interpreter runs atop a custom monad where computations are internally represented as trees of operations and equipped with a small-step semantics. We define two program logics for OLang. A stateless Hoare Logic allows reasoning about so-called "pure" programs; an Iris-based Separation Logic allows reasoning about arbitrary programs. We present the construction of the two logics as well as some examples of their use.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**.

ACM Reference Format:
Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml. *Proc. ACM Program. Lang.* 9, ICFP, Article 240 (August 2025), 31 pages. https://doi.org/10.1145/3747509

1 Introduction

A formal definition can be a valuable foundation for a programming language. A mechanized semantics rules out the inaccuracies that usually appear in informal specifications and forms a bedrock for verified software. It can be used to test or verify an interpreter, a compiler, or a static analyzer; to prove the soundness of a type system; and to prove the soundness of a program logic.

Although writing down a complete formal description of a realistic programming language used to be a formidable task, we have entered an era where such an achievement is gradually becoming more commonplace. Several prominent low-level programming languages have been partially or fully formalized, including C [Norrish 1998; Ellison and Rosu 2012; Krebbers et al. 2014; Krebbers 2015] and its weak memory model [Lahav et al. 2017], JavaScript [Bodin et al. 2014; Gardner et al. 2015], and the intermediate languages WebAssembly [Watt 2021; Watt et al. 2021, 2023], MIR [Jung et al. 2018a, 2020], and LLVM IR [Zhao et al. 2012; Zakowski et al. 2021].

Among high-level programming languages, few have a formal semantics. The Definition of Standard ML [Milner et al. 1997] has been mechanized [Lee et al. 2007; Harper and Crary 2014; MacQueen et al. 2020], and the CakeML verified compiler [Kumar et al. 2014] accepts a fragment of Standard ML as its source language. Parts of Java have been mechanized [Klein and Nipkow 2006] and its weak memory model has been studied and formalized [Manson et al. 2005; Lochbihler 2012;

Authors' Contact Information: Remy Seassau, remy.seassau@inria.fr, Inria, Paris, France; Irene Yoon, inbox@ireneyoon.com, Inria, Paris, France; Jean-Marie Madiot, jean-marie.madiot@inria.fr, Inria, Paris, France; François Pottier, francois.pottier@inria.fr, Inria, Paris, France.

ACM 2475-1421/2025/8-ART240
https://doi.org/10.1145/3747509

Bender and Palsberg 2019]. However, other prominent high-level programming languages, such as Haskell, Scala, and OCaml, lack formal definitions. We believe that this lack must be remedied.

While small-step operational semantics [Plotkin 2004; Wright and Felleisen 1994], can describe the semantics of any programming language, it is considered heavy when used at a large scale [Bodin et al. 2019] and leads to semantic definitions that can be difficult to understand or maintain. The search for new semantic styles that are more elegant or more modular, therefore making formalization more manageable, is ongoing [Xia et al. 2020; Charguéraud et al. 2023; Frumin et al. 2024; Vistrup et al. 2025; Stepanenko et al. 2025].

In this paper, we focus on OCaml [Leroy et al. 2024], a descendant of Milner's ML [1978]. OCaml's main features include first-class functions, algebraic data types, pattern matching, dynamic memory allocation, mutable data, modules and functors [Leroy 2000], objects and classes [Rémy and Vouillon 1998], exceptions, delimited control effects [Sivaramakrishnan et al. 2021], concurrency, weak shared-memory [Sivaramakrishnan et al. 2020], and more. OCaml is widely used in academia, both in education and research, and has found a number of key industrial users [Leandersson 2022].

There are several OCaml compilers, which share a common front-end and differ in their back-ends. These include the OCaml bytecode and native code compilers [Leroy et al. 2024], the `flambda` and `flambda2` native code compilers, two OCaml-to-JavaScript compilers [Vouillon and Balat 2014; Monteiro 2025], and two OCaml-to-WebAssembly compilers [Andrès et al. 2023; Vouillon 2023]. Furthermore, the reference interpreter `Camlboot` [Courant et al. 2022] supports a subset of OCaml that is large enough to execute the OCaml compiler itself. It seems desirable for these diverse tools to agree on a common formal foundation.

The main contributions of this paper are as follows:

- Using Rocq (ex-Coq), we formalize the abstract syntax and dynamic semantics of OLang, a sequential fragment of OCaml. This fragment includes first-class functions, ordinary and extensible algebraic data types, pattern matching, references, exceptions, deep and shallow effect handlers, and nested modules (not functors). It has unspecified evaluation order.
- We implement a translator of OCaml into OLang. This translator consumes a typed OCaml AST, which is produced by the OCaml type-checker, and emits the corresponding OLang AST as a Rocq source file. This translator is simple, and must be trusted.
- We organize the semantics of OLang in two layers. The upper layer is a monadic interpreter; the lower layer is an original custom monad. The monad's combinators form the interface between the two layers. We choose this style because a monadic interpreter is easy to understand and review. In the lower layer, monadic computations are represented as trees, equipped with a small-step operational semantics.
- We define two program logics for OLang. A stateless Hoare Logic, Horus, allows reasoning about a class of *pure* programs, which cannot diverge, exploit mutable state, or perform control effects, but do have access to exceptions and non-determinism. A Separation Logic, Osiris, allows reasoning about arbitrary OLang programs, which may exhibit all kinds of effects. It is based on Iris [Jung et al. 2018b]. The two logics can interoperate: a Horus proof about a pure program fragment can be exploited inside an Osiris proof of a larger program.

Our work is carried out using Rocq; our results are machine-checked [The Osiris Project 2025]. Each reasoning rule in Horus and Osiris is a lemma. Furthermore, we prove the soundness of both program logics with respect to the dynamic semantics.

Although no single feature of OLang is new, its combination of features is fairly complex. In particular, Osiris is the first program logic that supports OCaml's combination of exceptions and effect handlers (§2). In fact, the definition of semantics and program logics for delimited control effects is still the subject of current research [Stepanenko et al. 2025].

The paper begins with a discussion of our main design choices (§2). Then, we present our formal semantics, starting with the monadic interpreter (§3), and continuing with the monad (§4, §5). We discuss how our semantics might be validated (§6). We move on to a presentation of Horus (§7) and Osiris (§8). The paper ends with discussions of related work (§9) and future work (§10).

2 Architecture and Design Choices

A project of this scale raises many design questions. Our semantics must be elegant and practical: we want it to be easy to understand, test, and maintain. Furthermore, it must serve as a foundation for our program logics: that is, we want to establish that our reasoning rules are sound with respect to this semantics. In the following, we review some of our main design choices. They contribute to answering three main questions:

- What style should be used in the definition of the semantics?
- What view of the programming language should be offered to the user?
- What style of reasoning should be offered to the user?

Semantic style: hybrid monadic/operational semantics. Our semantics is a modular composition of two layers, each of which uses a distinct style. The top layer is a monadic interpreter. It is easy to understand and lends itself well to execution, either inside Rocq or via extraction. This layer is *denotational*: the interpreter is defined by induction on the syntax of the program. The bottom layer provides a construction of the monad that the interpreter relies upon. It supports a large collection of effects (Figure 2). In this layer, a computation is represented as a tree whose nodes include final outcomes, observable events, parallel compositions, and control delimiters. This layer is *operational*: the behavior of a computation is given by a small-step reduction relation. We strive for simplicity. Whereas related work has used co-inductive trees [Xia et al. 2020] and guarded recursive trees [Frumin et al. 2024; Stepanenko et al. 2025], we use just finite trees. Whereas many authors have placed emphasis on defining the meaning of each effect in isolation [Xia et al. 2020; Yoon et al. 2022; Frumin et al. 2024; Vistrup et al. 2025], we provide a monolithic construction.

Semantic style: environment-based semantics. The literature on type systems [Wright and Felleisen 1994] and program logics [Jung et al. 2018b, §6.1] often uses substitution-based semantics, where certain reduction steps involve replacing variables with values. In an environment-based semantics, instead, an explicit map of variables to values is maintained. Our monadic interpreter uses an environment-based style because it is more natural, more efficient, and seems better suited to the task of describing OCaml, where (due to **open** and **include**) name resolution is nontrivial.

Language view: untyped, yet high-level. The semantics of OLang is untyped. There is an inductive type of all values, *val*. Every value carries a tag. This tag is inspected by dynamic checks that can cause runtime failures, also known as *crashes*. For example, providing a Boolean value as an argument to an integer addition operation causes a crash. Thus, our semantics offers a high-level view of values: a value is not just a sequence of bits in memory; it is a finite tree. This approach is standard: it is that of untyped λ-calculus. It offers two benefits. First, it lets us assign a meaning to all programs, not just well-typed programs. This lets us support certain uses of unsafe type casts. Second, it removes the need to define OCaml's type system, which would be a formidable task.

Our translator of OCaml into OLang is intended to be as simple as possible. This matters, as it is unverified and must be trusted. Yet, some ambiguities in OCaml's syntax create a potential difficulty. For example, in OCaml, two distinct algebraic data types can have data constructors named `A`. In OLang, this ambiguity does not exist. To eliminate this ambiguity while keeping our translator simple, we let the OCaml type-checker perform type-based disambiguation. Thus, although OLang is untyped, our translator expects OCaml code that has been accepted by the OCaml type-checker.

Language view: unspecified evaluation order. OCaml has unspecified evaluation order: in many constructs, such as an application of a function to n arguments or the construction of a tuple with $n + 1$ fields, the order in which the $n + 1$ subexpressions are evaluated is unspecified. It is not necessarily left-to-right or right-to-left; it can be an arbitrary permutation. The order that is chosen in practice can vary across compilers[1] and can be difficult for the user to predict. Furthermore, some compiler optimizations, such as tail modulo cons [Allain et al. 2025], exploit the opportunity of choosing an evaluation order.

To account for this feature, our semantics of OLang must be non-deterministic: it must permit all of the permutations that the OCaml manual allows. In fact, by making our semantics even more relaxed than required by the manual, we are able to simplify it. We allow *parallel evaluation* of the subexpressions. This lets us view an n-ary function application as a nest of binary function applications, while still allowing the $n + 1$ subexpressions to be evaluated in an arbitrary order.

This decision implies that some programs that have only one possible result according to the OCaml manual can in our semantics have several possible results. An example is `let r = ref 0 in (incr r, incr r); !r`, where both subexpressions of the pair increment the reference `r`. Because the two subexpressions `incr r` run in parallel, both might read 0 from `r` and write 1 into `r`. The final result can be 1 or 2.

With program verification in mind, this over-approximation seems acceptable, for two reasons. First, adopting a stricter semantics, which involves non-deterministic choices but does not allow parallel evaluation, would not allow us to offer simpler reasoning rules. To substantiate this claim, we refer the reader to the treatments of non-interleaved function calls in C by Krebbers [2014] and by Frumin et al. [2019], which are interesting but complex, as they involve shared resource invariants. Second, assuming that the user who verifies a program has control over the source code, it is easy to use an explicit sequence in places where this helps verify the code.

In an application to compiler verification, this over-approximation may be more problematic. Although adopting a non-deterministic semantics for the source language offers more freedom to the compiler, a user of a compiler likely does not expect the above example program to return 1, which is an allowed output in our non-deterministic semantics. Perhaps, in the context of verifying a compiler whose source language is OLang, one would prefer to adopt a stricter semantics of OLang. One would separately prove that the two semantics are related.

Reasoning style: source-level reasoning. We name our monad *micro* because it is an intermediate language, or “microcode”, into which OLang code is expanded. In principle, our interpreter can be used as a compiler: by applying the interpreter to an OLang AST and by letting Rocq perform partial evaluation, one obtains a *micro* AST. This is the first Futamura projection [1999a; 1999b]. Taking this idea seriously, one might wish to expand OLang code into *micro* code and let the user perform program verification at this level, by applying the reasoning rules of Figures 7 and 10. Thus, one would save the work of building program logics for OLang. We experimented with this idea, but found it impractical: unless Rocq's reduction strategy is carefully controlled, the size of the goal explodes. Furthermore, we do not truly wish for the end user to work at the *micro* level. We prefer to develop program logics for OLang and to let the user work at the source level.

Reasoning style: two program logics. We propose two program logics for OLang. Horus can verify *pure* programs, which must terminate and cannot use mutable state or control effects; Osiris can verify arbitrary programs. Horus is much simpler than Osiris, as it is a stateless Hoare Logic, whereas Osiris is an Iris-based Separation Logic. We believe that the user will be happy to work with Horus where possible and that Horus offers a gentler learning curve. Furthermore, Horus

[1]This unresolved issue offers an example: https://github.com/ocaml/ocaml/issues/13641

$$\begin{array}{lcl}
\mathit{val} & := & \mathit{VInt}\ (i : \mathit{int}) \\
 & & \mathit{VTuple}\ (\mathit{vs} : \mathit{list}\ \mathit{val}) \\
 & & \mathit{VData}\ (c : \mathit{string})\ (\mathit{vs} : \mathit{list}\ \mathit{val}) \\
 & & \mathit{VXData}\ (\ell : \mathit{loc})\ (\mathit{vs} : \mathit{list}\ \mathit{val}) \\
 & & \mathit{VLoc}\ (\ell : \mathit{loc}) \mid \mathit{VCont}\ (\ell : \mathit{loc}) \\
 & & \mathit{VClo}\ (\eta : \mathit{env})\ (a : \mathit{anonfun})
\end{array}$$

$$\begin{array}{lcl}
\mathit{loc} & := & \mathbb{Z} \\
\mathit{env} & := & \mathit{list}\ (\mathit{var} \times \mathit{val}) \\
\mathit{anonfun} & := & \mathit{AnonFun}\ (x : \mathit{var})\ (e : \mathit{expr}) \\
\mathit{exn} & := & \mathit{val} \\
\mathit{eff} & := & \mathit{val}
\end{array}$$

Fig. 1. OLang's type of values

can prove termination, whereas Osiris cannot: following most of the Iris literature, Osiris imposes just partial correctness, because this makes verifying concurrent programs easier. Finally, Horus helps us study certain problems (such as the treatment of pattern matching) in a simpler setting.

3 A Monadic Interpreter

Our semantics takes the form of a *monadic interpreter* [Liang et al. 1995] for OLang. This interpreter is implemented in Rocq, a pure and total programming language. It uses the *micro* monad to represent computational effects that cannot be expressed in Rocq. They include divergence, fatal failure (crashes), non-fatal failure (exceptions), state, parallelism, nondeterminism, and delimited control. The *micro* monad offers a fixed collection of primitive effectful operations, or *combinators*, which the interpreter exploits.

In this section, we offer a gradual exposition of the interpreter. At the same time, as we go, we present the combinators that the interpreter needs. For reference, these combinators are listed in Figure 2; they form the public API of the *micro* monad. In the next sections (§4, §5), we explain how the *micro* monad is defined and equipped with a small-step operational semantics.

3.1 Syntax

The syntax of OLang involves several syntactic categories, including expressions, patterns, module expressions, structure items, and many more. In this paper, for the sake of brevity, we put emphasis mainly on expressions. We use a deep embedding [Gibbons and Wu 2014]: that is, we represent OLang's syntax in Rocq via several inductive types, such as *expr*, the type of expressions (whose definition is not shown). The syntax of OLang closely resembles the surface syntax of OCaml, so it is easy to transform OCaml code into (Rocq definitions of) OLang abstract syntax. We provide a translator for this purpose. In our syntax, variables are represented as strings.

3.2 Values and Environments

The result of interpreting an expression is a *value*. We represent values in Rocq as an inductive type *val*, whose definition appears in Figure 1. Because OLang is untyped, this type represents all kinds of values, including machine integers (*VInt*), tuples (*VTuple*), inhabitants of algebraic data types (*VData*) and extensible algebraic data types (*VXData*), addresses of heap-allocated memory blocks (*VLoc*) and of heap-allocated continuations (*VCont*), closures (*VClo*), and more.

In *VData*, a data constructor is identified by its name, a string. In *VXData*, it is identified by a memory location. A mapping of names (of constructors of extensible algebraic data types) to memory locations is maintained as part of the interpreter's environment.

Exceptions and effects carry a first-class value, which we refer to as the "payload". Therefore, we define the types *exn* and *eff* as synonyms for *val*.

Our semantics is environment-based. An environment η is a finite map of variables to values: we represent it as an association list. Because a closure (*VClo*) contains an environment, the types *val* and *env* are mutually inductive.

Inductive $outcome_2\ (A\, E : Type)\ : Type := Ret_2\ (a : A) \mid Throw_2\ (e : E)$
Inductive $outcome_3\ (A\, E : Type)\ : Type := Ret_3\ (a : A) \mid Throw_3\ (e : E) \mid Perform_3\ (v : eff)\ (\ell : loc)$

$micro$	:	$Type \longrightarrow Type \longrightarrow Type$	
ret	:	$A \longrightarrow micro\ A\ E$	
$throw$	:	$E \longrightarrow micro\ A\ E$	
try_2	:	$micro\ B\ E' \longrightarrow (outcome_2\ B\ E' \longrightarrow micro\ A\ E) \longrightarrow micro\ A\ E$	
$bind$	:	$micro\ B\ E \longrightarrow (B \longrightarrow micro\ A\ E) \longrightarrow micro\ A\ E$	— derived from try_2
$crash$	:	$micro\ A\ E$	
$please\_eval$	:	$env \longrightarrow expr \longrightarrow micro\ val\ exn$	
$alloc$	:	$val \longrightarrow micro\ loc\ exn$	
$load$	:	$loc \longrightarrow micro\ val\ exn$	
$store$	:	$loc \longrightarrow val \longrightarrow micro\ val\ exn$	
par	:	$micro\ A_1\ E \longrightarrow micro\ A_2\ E \longrightarrow micro\ (A_1 \times A_2)\ E$	
$handle$	:	$micro\ val\ exn \longrightarrow (outcome_3\ val\ exn \longrightarrow micro\ A\ E) \longrightarrow micro\ A\ E$	
$perform$	:	$eff \longrightarrow micro\ val\ exn$	
$resume$	:	$loc \longrightarrow outcome_2\ val\ exn \longrightarrow micro\ val\ exn$	
$wrap$	:	$loc \longrightarrow env \longrightarrow handler \longrightarrow micro\ loc\ exn$	

Fig. 2. The *micro* monad: public interface

3.3 Structure of the Monadic Interpreter

The interpreter is composed of several mutually recursive functions. There is typically one function for each syntactic category of OLang along with a number of auxiliary functions. In this paper, we are mainly interested in the function $eval\_expr : env \longrightarrow expr \longrightarrow micro\ val\ exn$, which forms the heart of the monadic interpreter. We write *eval* as a short-hand for *eval_expr*. The meta-level expression $eval\ \eta\ e$ evaluates the OLang expression e under the environment η. Its type is *micro val exn*. This means that it is a monadic computation that can produce a normal result of type *val* (an OLang value) or an abnormal result of type *exn* (an OLang exception). It can also exhibit a range of effectful behaviors, including crashing, diverging, and more; we discuss these later on. *eval* is defined by induction on its second argument, an abstract syntax tree e. In the following subsections (§3.4–§3.9), we present several fragments of its definition. This illustrates how the combinators of the *micro* monad are used by the interpreter.

3.4 Integer Arithmetic / Return, Bind, Crash

We use OLang's integer arithmetic expressions to illustrate the most basic combinators of the *micro* monad, whose full list appears in Figure 2. The computation $ret\, a$ returns the result a. The sequential composition of two computations, $bind\, m_1\ (\lambda x.\, m_2)$, is also written $x \longleftarrow m_1;\ m_2$. The combinator *crash* can be understood as a fatal failure or as undefined behavior; it is a bad event that must be avoided. The following code fragment (left) shows how integer literals and unary negation are evaluated. It uses two auxiliary functions *val_as_int* and *as_int* (right).

```
Fixpoint eval η e :=
  match e with
  | EInt i ⇒
      ret (VInt (int.repr i))
  | EIntNeg e ⇒
      i ← as_int (eval η e) ;
      ret (VInt (int.neg i))
  | ...
```

```
Definition val_as_int (v : val) : micro int exn :=
  match v with
  | VInt i ⇒ ret i
  | _      ⇒ crash
  end.

Definition as_int (m : micro val exn) : micro int exn :=
  v ← m ; val_as_int v.
```

An integer literal expression *EInt* i carries an unbounded integer i, whose type is $\mathbb{Z}$.[2] We convert i to a machine integer via *int.repr*, convert it to an integer value via *VInt*, then return it. An integer negation expression *EIntNeg* e carries a subexpression e. We first evaluate e via a recursive call to *eval*. Then, using *as_int*, we check that the resulting value is an integer value. If *as_int* is applied to an integer value *VInt* i, then it returns the machine integer i; otherwise, it crashes.

3.5 Algebraic Data Types / Parallel Composition

OLang supports user-defined algebraic data types. In *EData* c *es*, the data constructor c is applied to the expressions *es*. The order of evaluation of these expressions is unspecified. Similarly, in the construction of a tuple, evaluation order is unspecified. To model this, we rely on the binary parallel composition combinator *par* (Figure 2). Here are the relevant cases in the definition of *eval* (left):

```
| EData c es ⇒
    vs ← evals η es ;
    ret (VData c vs)
| ETuple es ⇒
    vs ← evals η es ;
    ret (VTuple vs)
```

```
Fixpoint evals η (es : list expr) : micro (list val) exn :=
  match es with
  | [] ⇒ ret []
  | e :: es ⇒
      '(v, vs) ← par (eval η e) (evals η es) ;
      ret (v :: vs) end.
```

In the auxiliary function *evals* (right), *par* is used to evaluate one expression e and the remaining expressions *es* in parallel. This yields a pair of a value v and a list of values vs. The computation *par* m_1 m_2 lets m_1 and m_2 run in parallel and produces a pair of their results. It is nondeterministic: the effects of m_1 and m_2 can take place in an arbitrary order and can be interleaved.

3.6 First-Class Functions / Divergence

In OLang, all functions are unary; function application is binary. We represent a function `fun x -> e` as the expression *EAnonFun* (*AnonFun* x e). Evaluating it produces a closure *VClo* η (*AnonFun* x e) where the current environment η is captured.

```
| EAnonFun a ⇒
    ret (VClo η a)
| EApp e1 e2 ⇒
    '(v1, v2) ← par (eval η e1) (eval η e2);
    call v1 v2
```

```
Definition call v1 v2 : micro val exn :=
  match v1 with
  | VClo η (AnonFun x e) ⇒
      please_eval ((x, v2) :: η) e
  | _ ⇒ crash end.
```

In a function application, *par* is again used to allow unspecified evaluation order. After evaluating the function e_1 and the argument e_2, we invoke the auxiliary function *call*. This function first checks that v_1 is a closure; then, it executes the function body e, in the closure's environment, extended with a binding of the formal parameter x to the actual argument v_2. For this purpose, instead of *eval*, we use the combinator *please_eval* (Figure 2), whose type is the same as that of *eval*. Our host language, Rocq, allows writing terminating functions only; a plain recursive call would be rejected. *please_eval* can be understood as a request for a potentially dangerous recursive call (one that could cause divergence), as opposed to a native recursive call. This idea is due to McBride [2015], who showed that "general recursive definitions can be represented in the free monad".

OLang also supports (mutually) recursive functions: the syntax of expressions includes *ELetRec*, and the syntax of values includes recursive closures (*VCloRec*). In the paper, they are omitted.

[2] In OCaml, machine integers are signed and have a fixed bit width w. The value of w is unspecified, and thus our semantics is parameterized by it. The manual explicitly states that w can be 31, 32, or 63, but does not rule out other values. We assume $w \geq 31$. In Rocq, we write *int* for the type of signed integers of bit width w, which lie in the semi-open interval $[-2^{w-1}, 2^{w-1})$. We write *int.repr* for the projection of $\mathbb{Z}$ into *int*. Our Rocq library *int*, which is borrowed from CompCert, defines the usual operations on machine integers.

3.7 State / Alloc, Load, Store

OLang has mutable references. To define their semantics, we rely on three combinators offered by the *micro* monad, namely *alloc*, *load*, and *store* (Figure 2). Thus, support for dynamic memory allocation and mutable state is built into the monad.

3.8 Exceptions / Throw

In OCaml, an exception is raised using the primitive construct "**`raise`** `e`". Then, it propagates up to the nearest exception handler, which can either handle it (that is, catch it) or let it propagate further. An exception handler takes the form "**`match`** `e` **`with`** `bs`" where the list of branches `bs` can contain value-handling branches "`p -> e`" and exception-handling branches "**`exception`** `p -> e`" where `p` is a pattern.[3]

To interpret "**`raise`** `e`", we use the combinator *throw* (Figure 2).

```
| ERaise e ⇒
    v ← eval η e ; throw v
```

In this code fragment, the value v has type *val*. We have defined the type *exn* as a synonym for *val*, so *throw* v has type *micro val exn*, as required for this code fragment to be well-typed. In OCaml, the static type system requires all exceptions to have type `exn`, a predefined extensible algebraic data type.[4] This guarantees that all exception-raising sites and all exception handlers agree on a common type. In a dynamic semantics, though, there is no need for such a restriction. Therefore, in the above code fragment, no dynamic tag check is applied to the value v.

If the construct "**`match`** `e` **`with`** `bs`" could handle just normal and exceptional outcomes then we would interpret it using the monadic combinator try_2 (Figure 2), a generalization of *bind*. In the sequential composition $try_2\ m\ f$, the computation f expects a parameter of type $outcome_2\ B\ E'$, a sum type that can represent both normal and exceptional outcomes. However, the **`match`** construct is more powerful than this: in addition to normal and exceptional outcomes, it can handle delimited control effects. We defer an explanation of it to §3.9.

3.9 Delimited Control Effects and Handlers / Perform, Handle, Resume, Install

Let us briefly review OCaml's control effects and effect handlers [Sivaramakrishnan et al. 2021] before presenting the manner in which our interpreter supports these features.

Overview. The OCaml expression **`perform`** `e` performs an effect. To a certain extent, this is analogous to raising an exception via **`raise`** `e`: indeed, both constructs interrupt the normal flow of computation and transfer control to a handler. Yet, from the point of view of the context that surrounds them, the expressions **`raise`** `e` and **`perform`** `e` behave differently: whereas **`raise`** `e` always raises an exception, **`perform`** `e` can appear to return a value, to raise an exception, or to never terminate. The choice between these alternatives is up to the handler. Indeed, an effect handler receives a *continuation* `k`, which can be thought of as "the computation that has been suspended by **`perform`** `e`", or "the context that surrounds **`perform`** `e` and awaits its outcome". If this continuation is *continued* then **`perform`** `e` appears to return a value; if it is *discontinued* then **`perform`** `e` appears to raise an exception. More precisely, if **`continue`** `k v` is executed then **`perform`** `e` appears to return the value `v`; if **`discontinue`** `k v` is executed then **`perform`** `e` appears to raise the exception `v`. In either case, we say that the continuation `k` is *resumed*.

Effect handlers come in two flavors. A *shallow handler* monitors a computation until one effect is performed; it handles this effect, then disappears. A *deep handler* monitors a computation until this

[3]The syntax of patterns is not shown in the paper.
[4]Allocating a new exception name via "**`exception`** `E` **`of`** `int`" is sugar for "**`type`** `exn += E` **`of`** `int`".

computation terminates; it successively handles all of the effects that this computation performs. In OCaml's surface syntax, as of OCaml 5.3, a deep effect handler takes the form "**`match`** `e` **`with`** `bs`" where the list of branches `bs` contains at least one effect-handling branch **`effect`** `p, k -> e'`. This branch is entered if the value that was passed to **`perform`** matches the pattern `p`. OCaml offers no surface syntax for shallow handlers; instead, they are accessed via the library `Effect.Shallow`. In this paper, we discuss deep handlers only, as they seem more common and somewhat easier to use. Our semantics and reasoning rules do support shallow handlers. Our translator does not yet support them, and we do not yet have examples of verified code that uses them.

Performing an effect. Our interpretation of **`perform`** `e` appears in the following fragment of the definition of *eval*. It uses the monadic combinator *perform* (Figure 2). This combinator is meant to interact with the combinator *handle*, which is discussed later on in this section.

```
| EPerform e ⇒                          | EContinue e1 e2 ⇒
    v ← eval η e ;                          '(l, v) ← par (as_cont (eval η e1)) (eval η e2) ;
    perform v                               resume l (Ret2 v)
| EMatch e bs ⇒                         | EDiscontinue e1 e2 ⇒
    handle (eval η e)                       '(l, v) ← par (as_cont (eval η e1)) (eval η e2) ;
      (wrap_eval_branches η bs)             resume l (Throw2 v)
```

The *micro* monad offers just a bare-bones effect handling facility. A handler that is installed via *handle* is *shallow*: it handles at most one effect, then vanishes. Furthermore, it is *catch-all*: it always handles an effect that it observes; it never allows this effect to be propagated up to the next handler. Thus, in the definition of *eval* and of its auxiliary functions, we must explicitly implement (A) the self-replicating behavior of deep handlers and (B) the propagation of an effect from a handler that is unable to handle this effect up to the next handler.

Resuming a continuation. Our interpretations of **`continue`** `e1 e2` and **`discontinue`** `e1 e2` also appear in the above fragment of *eval*. They expect `e1` to produce a value of the form *VCont* ℓ, where ℓ is a *stored continuation*, that is, a heap address where a continuation is stored. This dynamic check is carried out by the auxiliary function *as_cont* (not shown). Both rely on the combinator *resume* (Figure 2), whose arguments are a heap address where a continuation is stored and an outcome with which to resume this continuation. This outcome has type $\mathit{outcome}_2\ \mathit{val}\ \mathit{exn}$. In **`continue`**, the continuation is resumed with a normal outcome $\mathit{Ret}_2\ v$; in **`discontinue`**, it is resumed with an exceptional outcome $\mathit{Throw}_2\ v$.

Handling effects. Our interpretation of "**`match`** `e` **`with`** `bs`" also appears in the above fragment. To interpret this construct, we evaluate the expression `e` in the scope of an effect handler. To install this handler and to delimit its scope, we use *handle* (Figure 2). *handle m h* runs the computation *m* and lets the handler *h* inspect its outcome, which can be one of three events: normal termination, exceptional termination, or an effect. The sum type $\mathit{outcome}_3$ (Figure 2) describes these three cases. In the event where an effect takes place, the outcome $\mathit{Perform}_3\ v\ \ell$ carries the effect's payload $v : \mathit{eff}$ and the stored continuation $\ell : \mathit{loc}$. By convention, when a handler *h* is invoked, the continuation has been captured and stored in the heap already; the handler receives its address.

Our handler, $\lambda o.\ \mathit{wrap\_eval\_branches}\ \eta\ \mathit{bs}\ o$, is defined in two lines (Figure 3). First, via the auxiliary function *wrap_outcome* (not shown), the outcome `o` is wrapped in a copy of the effect handler **`match`** `...` **`with`** `bs`. Then, it is passed on to the function *eval_branches*, which successively tests whether each branch in the list `bs` is able to deal with this outcome. These two steps are discussed in the next two paragraphs.

```
Fixpoint eval_branches η o bs : micro val exn :=
  match bs with
  | Branch cp e :: bs ⇒
      try2 (eval_cpat η η cp o) (λ o ⇒
        match o with
        | Ret2 η'  ⇒ eval η' e
        | Throw2 () ⇒ eval_branches η o bs
        end)
  | [] ⇒
      match o with
      | Ret3 _   ⇒ crash
      | Throw3 v ⇒ throw v
      | Perform3 v l ⇒
          try2 (perform v) (λ o ⇒ resume l o)
      end
  end.
```

```
Fixpoint wrap_eval_branches η bs o :=
  o ← wrap_outcome η bs o ;
  eval_branches η o bs.

Fixpoint eval_cpat η δ cp o
: micro env unit :=
  match cp, o with
  | CVal p, Ret3 v ⇒
      eval_pat η δ p v
  | CExc p, Throw3 v ⇒
      eval_pat η δ p v
  | CEff pe pk, Perform3 e k ⇒
      δ ← eval_pat η δ pe e ;
      eval_pat η δ pk (VCont k)
  | _, _ ⇒ throw ()
  end.
```

Fig. 3. Case analysis on outcomes

Wrapping a continuation in a handler. In the case where the outcome o is an effect ($Perform_3$), which carries a stored continuation, *wrap_outcome* wraps this continuation in a copy of the effect handler **match** ... **with** bs. This serves two purposes at once: first, this is needed to obtain the self-replicating behavior of a deep handler (A); second, this is required in the event that this effect is not handled by this handler and must be propagated upwards (B). To wrap the continuation in a handler, *wrap_outcome* uses the monadic combinator *wrap* (Figure 2), which returns a new stored continuation. In the other two cases (Ret_3, $Throw_3$), *wrap_outcome* acts as an identity function.

Case analysis on outcomes. The function *eval_branches* (Figure 3) performs case analysis on an outcome. Its code can be summed up as follows: try each branch in the list bs until either a branch applies to this outcome or the end of the list is reached. If a branch applies, execute this branch. If no branch applies, propagate this outcome.

A branch takes the form *Branch cp e*, where *cp* is a *computation pattern*. The abstract syntax of computation patterns includes forms that match a normal result (*CVal*), an exceptional result (*CExc*), and an effect (*CEff*). The function *eval_cpat* (Figure 3) determines whether an outcome matches a computation pattern. It relies on the meta-level expression *eval_pat* $\eta\ \delta\ p\ v$, which matches the value v against the pattern p. These functions return an extended environment if pattern matching succeeds, throw a metal-level exception (*throw* ()) if pattern matching fails, and crash if the pattern and the value have incompatible tags: this occurs, for example, if p is a tuple pattern and v is an integer value.

When *eval_branches* runs out of branches, it behaves as follows. If o is a normal outcome (Ret_3), then a crash occurs. Indeed, we want a non-exhaustive case analysis to be considered an undesirable behavior. If o is an exceptional ($Throw_3$) or effectful ($Perform_3$) outcome, then it is propagated. Technically, it is converted back to a monadic computation, whose behavior can then be observed by the next enclosing handler. An exceptional outcome is converted to a computation via *throw*; an effectful outcome is converted via *perform*. In the latter case, whereas ℓ is a stored continuation (a memory location), $\lambda o.\ resume\ \ell\ o$ is a semantic continuation (a function), which forms a suitable argument for try_2. Thus, $try_2\ (perform\ v)\ (\lambda o.\ resume\ \ell\ o)$ performs an effect with payload v and continuation ℓ.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Inductive } \mathit{micro}\ (A : \mathit{Type})\ (E : \mathit{Type}) : \mathit{Type} :=} \\
\mid \mathit{Ret} & : & A \to \mathit{micro}\ A\ E \\
\mid \mathit{Throw} & : & E \to \mathit{micro}\ A\ E \\
\mid \mathit{Crash} & : & \mathit{micro}\ A\ E \\
\mid \mathit{Stop} & : & \mathit{code}\ X\ Y\ E' \to X \to (\mathit{outcome}_2\ Y\ E' \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E \\
\mid \mathit{Par} & : & \mathit{micro}\ A_1\ E' \to \mathit{micro}\ A_2\ E' \to (\mathit{outcome}_2\ (A_1 \times A_2)\ E' \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E \\
\mid \mathit{Handle} & : & \mathit{micro}\ \mathit{val}\ \mathit{exn} \to (\mathit{outcome}_3\ \mathit{val}\ \mathit{exn} \to \mathit{micro}\ A\ E) \to \mathit{micro}\ A\ E
\end{array}
$$

Fig. 4. The *micro* monad: definition

4 The Micro Monad

The *micro* monad offers an abstract type of computations along with its fundamental combinators, *ret* and *bind*. The remaining combinators (Figure 2) offer access to various computational effects, including exceptions, crashes, divergence, state, structured parallelism, non-deterministic choice, and delimited control.

Under the hood, computations are represented as trees, where leaves (*Ret*) represent results and internal nodes (*Stop*) represent observable events, or *system calls*. This representation is inspired by a long line of previous work on the free monad [Swierstra 2008, §6], the freer monad [Kiselyov and Ishii 2015], and interaction trees [Xia et al. 2020]. *Stop* carries a continuation, a meta-level function. One can think of this continuation as the computation that remains to be carried out once this system call has produced a result. One can also think of it as a family of subtrees, indexed with results. Because it is convenient to also have a variant of *Stop* that does not carry a continuation, we write *stop c v* for *Stop c v* inject_2, where inject_2 is the trivial continuation.[5] *bind* is defined as a meta-level function on trees.

The constructor *Stop* carries a *code*, which can be viewed as the name of a system call, as well as the argument of this system call. Although in previous work the type of codes is usually a parameter of the monad, we work with a fixed type of codes, that is, with a specific set of system calls, which provide support for just the effects that we need.

To express exceptions and crashes, we add two more kinds of leaves, *Throw* and *Crash*. To express structured parallelism, we add a new constructor, *Par*, which carries two child computations and a continuation. To express delimited control, we add another constructor, *Handle*, which carries a computation and a handler.

In the remainder of this section, we briefly review the definition of the *micro* monad (§4.1) as well as the specific system calls that we find necessary (§4.2). Once these definitions are given, there still remains to assign a meaning, or a behavior, to each system call and to each of our ad hoc constructors, such as *Par* and *Handle*. We do so via a small-step reduction relation (§5).

4.1 Definition

An inductive type of computations. A mathematical object of type *micro A E* represents an effectful computation whose eventual outcome is either a result of type A or an exception of type E. The fact that an outcome is a sum type is visible in the type of the fundamental combinator try_2 (Figure 2). When two computations are sequentially composed, the second computation must be prepared to accept the outcome of the first computation, whose type is the sum type $\mathit{outcome}_2\ A\ E$.

The definition of the type *micro A E* appears in Figure 4. It is a variant of the freer monad: that is, it is an inductive type, whose constructors include *Ret* and *Stop*. The three arguments carried by *Stop* are a code (the name of the system call), an argument (the argument of the system call), and a continuation (what to do once the system call produces an outcome). In *Stop c v k*, the code *c*

[5] $\mathit{inject}_2 : \mathit{outcome}_2\ A\ E \to \mathit{micro}\ A\ E$ is defined by the equations $\mathit{inject}_2\ (\mathit{Ret}_2\ v) = \mathit{ret}\, v$ and $\mathit{inject}_2\ (\mathit{Throw}_2\ v) = \mathit{throw}\ v$.

determines the types of the argument v and of the outcome expected by the continuation k. Indeed, if c has type $code\ X\ Y\ E'$ then v has type X and k expects the system call to produce either a result of type Y or an exception of type E'.

Inert computations. Three constructors represent inert computations. Beyond $Ret\ v$, whose outcome is the result v, we have $Throw\ v$, a computation whose outcome is the exception v, and *Crash*, a computation that represents a fatal runtime failure. The combinators *ret*, *throw*, and *crash* are synonyms for *Ret*, *Throw*, and *Crash*.

Sequential composition. The sequential composition combinator try_2 is not a constructor: instead, it is defined by induction on its first argument. In $try_2\ m\ k$, the continuation k expects an outcome, that is, either a result or an exception. Crashes are propagated: $try_2\ (crash)\ k$ is *crash*.

The sequential composition combinator *bind* is obtained as a special case of try_2. In $bind\,m\,k$, the continuation k expects a result. The monadic laws are satisfied: in particular, $bind\,(ret\,v)\,k$ is $k\,v$. Exceptions and crashes are propagated: $bind\,(throw\ v)\,k$ is $throw\ v$ and $bind\,(crash)\,k$ is *crash*.

Parallel composition. The constructor *Par* offers structured parallelism, that is, the ability to run two computations in parallel and to wait for both of them to terminate. It carries two computations and a continuation, which is meant to be invoked once both computations have produced a result. The presence of this continuation is exploited in the definition of try_2. Nevertheless, by thinking in terms of the combinator *par* instead of the constructor *Par*, one can forget about this continuation. Indeed, $par\,m_1\,m_2$ is defined as $Par\,m_1\,m_2\ inject_2$, where $inject_2$ is the trivial continuation.[5]

Delimited control. The constructor *Handle* serves as a delimiter of control effects. It carries a computation m and a handler h: in short, the computation $Handle\ m\ h$ is the computation m running under the handler h. The combinator *handle* (Figure 2) is a synonym for *Handle*.

As indicated by the type of *Handle* in Figure 4, the handler h is a three-armed continuation: that is, it expects an outcome of type $outcome_3$ _ _.

We require the computation m to have type $micro\ val\ exn$, that is, to produce an OCaml value or an OCaml exception. Accordingly, the handler h expects an outcome of type $outcome_3\ val\ exn$. This convention guarantees that all continuations have the same type, therefore makes the heap homogeneous. This is visible in the definition of a memory block (§5).

The definitions of the types $outcome_3$ and *micro* are *not* mutually recursive. Indeed, $outcome_3$ is defined first (Figure 2); *micro* refers to it (Figure 4). The key reason why this is possible is that the second argument of the constructor $Perform_3$ is a *stored continuation*, that is, a memory location ℓ. If instead it was a continuation (a function) then its codomain would be $micro\ val\ exn$, so the types $outcome_3$ and *micro* would be mutually recursive. Furthermore, because $outcome_3$ appears in a negative position in the arguments of the constructor *Handle* (Figure 4), the definitions of the types $outcome_3$ and *micro* would be logically meaningless, and would be rejected by Rocq. In summary, an indirection through the heap lets us avoid a logical difficulty.

4.2 System Calls

For our purposes, it is acceptable to fix the definition of the type *code*, that is, to adopt a fixed, finite set of system calls. This definition appears in Figure 5. We now briefly review each system call, describe its argument and result types, and explain its intended semantics.

Divergence. The system call *CEval* is a request to evaluate an OCaml expression. Its argument is a pair of an environment η and an expression e. It returns a value or raises an exception. The combinator *please_eval* (Figure 2) is defined by $please\_eval\ \eta\ e = stop\,CEval\,(\eta, e)$.

State. The system calls *CAlloc*, *CLoad*, and *CStore* allocate, read, and write heap cells.

$$
\begin{array}{lcl}
\text{Inductive } \mathit{code} & : & \mathit{Type} \longrightarrow \mathit{Type} \longrightarrow \mathit{Type} \longrightarrow \mathit{Type} := \\
\mid \mathit{CEval} & : & \mathit{code}\ (\mathit{env} \times \mathit{expr})\ \mathit{val}\ \mathit{exn} \\
\mid \mathit{CAlloc} & : & \mathit{code}\ \mathit{val}\ \mathit{loc}\ \mathit{exn} \\
\mid \mathit{CLoad} & : & \mathit{code}\ \mathit{loc}\ \mathit{val}\ \mathit{exn} \\
\mid \mathit{CStore} & : & \mathit{code}\ (\mathit{loc} \times \mathit{val})\ \mathit{unit}\ \mathit{exn} \\
\mid \mathit{CPerf} & : & \mathit{code}\ \mathit{eff}\ \mathit{val}\ \mathit{exn} \\
\mid \mathit{CResume} & : & \mathit{code}\ (\mathit{loc} \times \mathit{outcome}_2\ \mathit{val}\ \mathit{exn})\ \mathit{val}\ \mathit{exn} \\
\mid \mathit{CWrap} & : & \mathit{code}\ (\mathit{loc} \times \mathit{env} \times \mathit{handler})\ \mathit{loc}\ \mathit{exn}
\end{array}
$$

Fig. 5. The *micro* monad: codes, also known as system calls

Delimited control. The system call *CPerf* is a request to perform a delimited control effect. Its argument is a value v. Its intended meaning is the same as that of the OCaml expression **`perform`** `v`. It can produce a value or an exception; this is determined when the continuation that is captured by this system call is later continued or discontinued. The combinator *perform* (Figure 2) is defined by $\mathit{perform}\ v = \mathit{stop}\ \mathit{CPerf}\ v$.

The system call *CResume* is a request to resume a continuation that has been previously captured and stored in the heap by *perform*. Its argument is a pair of a memory location ℓ and an outcome o. Its intended effect is to fetch the continuation at address ℓ and to resume it by applying it to o. If o has the form $\mathit{Ret}_2\ v$, then the continuation is *continued*; if o has the form $\mathit{Throw}_2\ v$, then the continuation is *discontinued*. The combinator *resume* (Figure 2) is defined by $\mathit{resume}\ \ell\ o = \mathit{stop}\ \mathit{CResume}\ (\ell, o)$.

The system call *CWrap* is a request to wrap a previously captured continuation in an effect handler, yielding a new continuation. Its argument is a triple (η, ℓ, bs), where bs is a *handler*, a list of branches. (This type is part of our abstract syntax of OLang.) Its intended effect is to allocate a new continuation which, once invoked, runs the existing continuation ℓ under the closed effect handler (η, bs). Its result is the address ℓ' of the new continuation. After this system call has returned, one can view ℓ as uniquely owned by ℓ'. The continuation ℓ must not be directly resumed; instead, it should be indirectly resumed by resuming the continuation ℓ'. The combinator *wrap* is defined by $\mathit{wrap}\ \ell\ \eta\ bs = \mathit{stop}\ \mathit{CWrap}\ (\eta, \ell, bs)$.

5 Small-Step Semantics for the Micro Monad

We now equip *micro* monad with a small-step operational semantics. This gives meaning to system calls (*Stop*) and to the monad's ad hoc constructors (*Par*, *Handle*).

The reduction rules act on *configurations* $m\ /\ \sigma$, that is, pairs of a computation m and a heap σ. A *heap*, or *store*, is a finite map of memory locations to memory blocks. The heap serves a dual purpose: it stores mutable memory cells (also known as *references*) and first-class continuations. Therefore, we define a *memory block* to be a value v, a continuation k, or the special mark $\not\ell$, which denotes a continuation that has been "shot" already.

In the previous sentence, k has type $\mathit{outcome}_2\ \mathit{val}\ \mathit{exn} \longrightarrow \mathit{micro}\ \mathit{val}\ \mathit{exn}$. Thus, all continuations have the same type. Furthermore, a continuation is represented as a meta-level function. This is a natural consequence of the structure of the *micro* monad. The continuation that is carried by the constructor *Stop* is a meta-level function. It is captured and stored in the heap when a control effect is performed.

A reduction step takes the form $m\ /\ \sigma \longrightarrow m'\ /\ \sigma'$. The reduction relation is inductively defined by the rules in Figure 6. We write ! as a short-hand for *Stop*.

Divergence. The first reduction rule states that the system call *CEval* with argument (η, e) and continuation k reduces in one step to the computation $\mathit{eval}\ \eta\ e$ followed with k. To better see this, recall that try_2 is the sequential composition operation of the monad. In particular, if k is the trivial

$$
\begin{array}{rcll}
\textbf{Divergence} & & & \\
!\,\mathit{CEval}\,(\eta, e)\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{eval}\;\eta\;e)\,k \,/\, \sigma & \\
\textbf{State} & & & \\
!\,\mathit{CAlloc}\,v\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{ret}\,\ell)\,k \,/\, (\ell, v) :: \sigma & \text{if } \ell \notin \mathit{dom}(\sigma) \\
!\,\mathit{CLoad}\,\ell\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{ret}\,v)\,k \,/\, \sigma & \text{if } \mathit{lookup}\;\sigma\;\ell = v \\
!\,\mathit{CLoad}\,\ell\,k \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma & \text{otherwise} \\
!\,\mathit{CStore}\,(\ell, v')\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{ret}\,())\,k \,/\, (\ell, v') :: \sigma & \text{if } \mathit{lookup}\;\sigma\;\ell = v \\
!\,\mathit{CStore}\,(\ell, v')\,k \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma & \text{otherwise} \\
\textbf{Parallelism} & & & \\
\mathit{Par}\,m_1\,m_2\,k \,/\, \sigma & \longrightarrow & \mathit{Par}\,m_1'\,m_2\,k \,/\, \sigma' & \text{if } m_1 \,/\, \sigma \longrightarrow m_1' \,/\, \sigma' \\
\mathit{Par}\,m_1\,m_2\,k \,/\, \sigma & \longrightarrow & \mathit{Par}\,m_1\,m_2'\,k \,/\, \sigma' & \text{if } m_2 \,/\, \sigma \longrightarrow m_2' \,/\, \sigma' \\
\mathit{Par}\,(\mathit{Ret}\;v_1)\,(\mathit{Ret}\;v_2)\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{ret}\,(v_1, v_2))\,k \,/\, \sigma & \\
\mathit{Par}\;\mathit{Crash}\,m_2\,k \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma & \\
\mathit{Par}\,m_1\;\mathit{Crash}\,k \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma & \\
\mathit{Par}\,(\mathit{Throw}\;v)\,m_2\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{throw}\;v)\,k \,/\, \sigma & \\
\mathit{Par}\,m_1\,(\mathit{Throw}\;v)\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{throw}\;v)\,k \,/\, \sigma & \\
\mathit{Par}\,(!\,\mathit{CPerf}\;v\,k)\,m_2\,k' \,/\, \sigma & \longrightarrow & !\,\mathit{CPerf}\;v\,(\lambda o.\,\mathit{Par}\,(k\,o)\,m_2\,k') \,/\, \sigma & \\
\mathit{Par}\,m_1\,(!\,\mathit{CPerf}\;v\,k)\,k' \,/\, \sigma & \longrightarrow & !\,\mathit{CPerf}\;v\,(\lambda o.\,\mathit{Par}\,m_1\,(k\,o)\,k') \,/\, \sigma & \\
\textbf{Delimited control} & & & \\
\mathit{Handle}\;(\mathit{Ret}\;v)\;h \,/\, \sigma & \longrightarrow & h\;(\mathit{Ret}_3\;v) \,/\, \sigma & \\
\mathit{Handle}\;(\mathit{Throw}\;v)\;h \,/\, \sigma & \longrightarrow & h\;(\mathit{Throw}_3\;v) \,/\, \sigma & \\
\mathit{Handle}\;(!\,\mathit{CPerf}\;v\,k)\;h \,/\, \sigma & \longrightarrow & h\;(\mathit{Perform}_3\;v\;\ell) \,/\, (\ell, k) :: \sigma & \text{if } \ell \notin \mathit{dom}(\sigma) \\
\mathit{Handle}\;\mathit{Crash}\;h \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma & \\
\mathit{Handle}\;m\;h \,/\, \sigma & \longrightarrow & \mathit{Handle}\;m'\;h \,/\, \sigma' & \text{if } m \,/\, \sigma \longrightarrow m' \,/\, \sigma' \\
!\,\mathit{CResume}\,(\ell, o)\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\;(k'\;o)\,k \,/\, (\ell, \lightning) :: \sigma & \text{if } \mathit{lookup}\;\sigma\;\ell = k' \\
!\,\mathit{CResume}\,(\ell, o)\,k \,/\, \sigma & \longrightarrow & \mathit{Crash} \,/\, \sigma & \text{otherwise} \\
!\,\mathit{CWrap}\,(\eta, \ell, bs)\,k \,/\, \sigma & \longrightarrow & \mathit{try}_2\,(\mathit{ret}\,\ell')\,k \,/\, (\ell', k') :: \sigma & \text{if } \ell' \notin \mathit{dom}(\sigma) \\
\multicolumn{4}{c}{\text{where } k' = \lambda o.\,\mathit{handle}\,(\mathit{resume}\;\ell\;o)\,(\mathit{wrap\_eval\_branches}\;\eta\;bs)}
\end{array}
$$

Fig. 6. The *micro* monad: small-step reduction

continuation *inject*$_2$ then this rule states that *please_eval* η e reduces to *eval* η e. In every reduction rule where a continuation k appears, it helps to read the rule in the special case where k is *inject*$_2$. The general case, where k is arbitrary, simply allows reduction under an evaluation context.

State. The system calls *CAlloc*, *CLoad*, and *CStore* implement the usual reduction semantics of mutable references. *CAlloc* v picks an unused memory location ℓ, initializes it with the value v, and returns ℓ. It cannot fail. *CLoad* ℓ reads the value stored at location ℓ, if this location has been allocated and stores a value; otherwise, it crashes. *CStore* (ℓ, v') overwrites the value at location ℓ with v', if this location has been allocated and stores a value; otherwise, it crashes.

Parallelism. A parallel composition $\mathit{Par}\,m_1\,m_2\,k$ allows the computations m_1 and m_2 to run in parallel. This is expressed by the first two rules in this group, which interleave the reduction steps of m_1 and m_2 in a non-deterministic manner.

The next rule, specialized to the trivial continuation, states that $\mathit{par}\,(\mathit{ret}\,v_1)\,(\mathit{ret}\,v_2)$ reduces to $\mathit{ret}\,(v_1, v_2)$. That is, if both m_1 and m_2 reach a result then $\mathit{par}\,m_1\,m_2$ returns a pair of these results. This is fork/join parallelism: a parallel composition terminates once both sides have finished.

The remaining six rules in this group define the behavior of a parallel composition in the situation where one side crashes, raises an exception, or performs a control effect.

A crash on either side is propagated: the parallel composition reduces to just *Crash*.

An exception on either side is also propagated: for example, $par\ (throw\ v)\ m_2$ reduces to $throw\ v$.

When a control effect $!\ CPerf\ v$ takes place under a parallel composition, the parallel composition itself is captured, as it forms one frame of the evaluation context. In the term $Par\ (!\ CPerf\ v\ k)\ m_2\ k'$, the continuation k represents an evaluation context that has been captured already, and the parallel composition $Par\ (\cdot)\ m_2\ k'$ forms one more frame, which has not yet been captured. This term reduces to a new term where the control effect $!\ CPerf\ v$ appears at the root and where the captured evaluation context $\lambda o.\ Par\ (k\ o)\ m_2\ k'$ is the composition of the original captured context k with this extra frame. This style of letting a control effect capture its evaluation context, one frame at a time, in a small-step operational semantics, is standard [Pretnar 2015, Fig. 4]. What is unusual and original here is that control effects and (non-deterministic) parallel composition interact.

Delimited control. The last group of rules in Figure 6 concerns *Handle*, which serves as a delimiter of control effects, and the system calls *CResume* and *CWrap*, which operate on stored continuations.

In $Handle\ m\ h$, the computation m is monitored by the handler h, a meta-level function whose argument has type $outcome_3\ val\ exn$. The first three reduction rules describe the three kinds of outcomes that the handler can observe. If the computation produces a result $Ret\ v$ then the handler is applied to the outcome $Ret_3\ v$. If it produces an exception $Throw\ v$ then the handler is applied to $Throw_3\ v$. If it performs an effect $!\ CPerf\ v\ k$ then the continuation k is captured: k is written in the heap at a fresh address ℓ, and the handler is applied to $Perform_3\ v\ \ell$. Thus, the handler receives access to the value v and to the stored continuation ℓ.

The next two rules state that a crash under a handler reduces to a crash and that reduction under $Handle\ (\cdot)\ h$ is permitted.

The first reduction rule for *CResume*, when specialized to the trivial continuation, states that if a continuation k' is stored at address ℓ then $resume\ \ell\ o\ /\ \sigma$ reduces to $k'\ o\ /\ (\ell, \not{\ell}) :: \sigma$. In words, $resume\ \ell\ o$ resumes the continuation that is stored at address ℓ by applying it to the outcome o and marks this continuation as *shot*. The next reduction rule states that attempting to resume a continuation that has already been shot causes a crash. OCaml and OLang support one-shot continuations only.

The last reduction rule, when specialized to the trivial continuation, states that $wrap\ \ell\ \eta\ bs\ /\ \sigma$ reduces to $ret\ \ell'\ /\ (\ell', k') :: \sigma$, where k' can be described as the stored continuation ℓ, wrapped in a copy of the closed handler (η, bs).

Basic properties. By design of this semantics, the terms $Par\ m_1\ m_2\ k$ and $Handle\ m\ h$ are never stuck; that is, they are always reducible. The same is true of a system call $!\ c\ v\ k$ except in the case where c is *CPerf*: indeed, a control effect cannot be reduced unless it occurs under *Par* or *Handle*. In summary, there are four kinds of irreducible terms, namely $ret\ v$, $throw\ v$, $crash$, and $!\ CPerf\ v\ k$. The last form represents an unhandled effect.

Our reduction semantics is compatible with evaluation contexts: that is, $m\ /\ \sigma \longrightarrow m'\ /\ \sigma'$ implies $try_2\ m\ k\ /\ \sigma \longrightarrow try_2\ m'\ k\ /\ \sigma'$.

6 Validation

The semantics of OCaml is folklore: it is not formally documented anywhere. The reference manual [Leroy et al. 2024] provides "precise syntax and informal semantics" and warns that "no attempt has been made at mathematical rigor". To validate (that is, to test) our formal semantics, we must compare it with existing implementations of OCaml. Ideally, this comparison should be automated and should involve a large number of test cases, including hand-written and randomly generated test cases. We are not there yet; in this section, we provide a preliminary status report.

Our monadic interpreter of OLang (§3) is executable. Rocq's `Extraction` command can produce OCaml code for it. By combining this interpreter with an interpreter of *micro* computations (which we write, by hand, in OCaml) and by composing it with our translator of OCaml to OLang, we obtain a stand-alone interpreter that is capable of executing self-contained OCaml programs.

In principle, this stand-alone interpreter allows comparing our semantics with pre-existing implementations of OCaml. This said, the non-deterministic nature of the semantics of both OCaml and OLang creates a significant difficulty: picking one execution path on each side (whose choice is implementation-dependent and/or random) does not allow a meaningful comparison; exploring all execution paths is impractical. At this time, we have only one source of non-determinism, namely unspecified evaluation order (§2). Each existing implementation of OCaml uses a fixed deterministic evaluation order, which can be experimentally determined. Therefore, a reasonable approach is to manually define a deterministic variant of our semantics, which mimics this evaluation order, and to compare these two deterministic systems. In the future, different sources of non-determinism, namely concurrency and relaxed memory, will enter the picture. We plan to ensure that no test case involves both unspecified evaluation order and concurrency. Then, repeated random testing can be used to verify that the behaviors exhibited by the pre-existing implementation form a subset of the behaviors permitted by our semantics.

As far as the construction of a test suite is concerned, a natural starting point is the test suite of the OCaml compiler. This test suite is small; it contains fewer than 1700 test cases. Furthermore, many tests use primitive operations that our semantics does not yet support. At this time, we have manually implemented support for just a few input/output operations, and we have been able to run a small number of tests in the compiler's test suite. Furthermore, we have hand-written a number of additional tests. In total, our tests currently represent 29 files and about 900 lines of code. In the future, significant work is required to deal with non-determinism and to implement support for a large set of primitive operations, support for foreign function calls, and random generation and shrinking of test cases. We plan to take inspiration from previous projects that involve testing a compiler or a formal semantics in the presence of undefined behavior and/or non-determinism [Owens 2008; Livinskii et al. 2020; Wang and Jung 2024; Beck et al. 2025].

7 Horus

We say that a computation is "pure" if it does not involve divergence, state, or delimited control. Pure computations are commonplace in OCaml. It is possible to reason about their behavior using a stateless program logic, which is significantly simpler than a Separation Logic. Therefore, in this section, we present Horus, a total program logic for pure program fragments.

Making Horus a total logic, where divergence is forbidden, is a design choice. We could have made it a partial logic, where divergence is allowed. This would remove the obligation of proving that every recursive function definition is well-founded. Requiring the user to prove termination has a cost (more work for the user) and a benefit (a stronger guarantee about the code). We believe that the benefit often outweighs the cost. If a user cannot prove or does not wish to prove that a piece of "pure" code terminates, then they will have to use Osiris (§8) instead of Horus; but we believe that this should be fairly rare.

7.1 Pure Reduction

To clarify what we mean by "pure" computation, we introduce a *pure reduction* relation, $m \longrightarrow_{\mathrm{p}} m'$. In this paper, its definition is omitted. It is identical to the relation $\longrightarrow$ (§5), with two differences. First, it relates computations (m) rather than configurations ($m\ /\ \sigma$): thus, it does not involve the heap. Second, in this relation, a system call that needs access to the heap (*CAlloc*, *CLoad*, *CStore*, *CPerf*, *CResume*, *CWrap*) reduces to *Crash*. This reduction relation is not terminating, deterministic or confluent; these properties are not needed. The constructs *Par* and *Handle* are supported, and behave normally, if their children are pure.

PURE-RET
$$\frac{\varphi\ a}{\mathit{pure}\ (\mathit{ret}\, a)\ \varphi\ \psi}$$

PURE-THROW
$$\frac{\psi\ v}{\mathit{pure}\ (\mathit{throw}\ v)\ \varphi\ \psi}$$

PURE-BIND
$$\frac{\mathit{pure}\ m\ (\lambda a.\,\mathit{pure}\ (k\ a)\ \varphi\ \psi)\ \psi}{\mathit{pure}\ (\mathit{bind}\, m\, k)\ \varphi\ \psi}$$

PURE-PLEASE-EVAL
$$\frac{\mathit{pure}\ (\mathit{eval}\ \eta\ e)\ \varphi\ \psi}{\mathit{pure}\ (\mathit{please\_eval}\ \eta\ e)\ \varphi\ \psi}$$

PURE-CONSEQ
$$\frac{\begin{array}{c}\mathit{pure}\ m\ \varphi\ \psi\\ \forall a.\ \varphi\ a \Rightarrow \varphi'\ a\\ \forall v.\ \psi\ v \Rightarrow \psi'\ v\end{array}}{\mathit{pure}\ m\ \varphi'\ \psi'}$$

PURE-TRY2
$$\frac{\begin{array}{c}\mathit{pure}\ m\ (\lambda a.\,\mathit{pure}\ (k\ (\mathit{Ret}_2\ a))\ \varphi\ \psi)\\ (\lambda v.\,\mathit{pure}\ (k\ (\mathit{Throw}_2\ v))\ \varphi\ \psi)\end{array}}{\mathit{pure}\ (\mathit{try}_2\, m\, k)\ \varphi\ \psi}$$

PURE-PAR
$$\frac{\begin{array}{c}\mathit{pure}\ m_1\ \varphi_1\ \psi \qquad \mathit{pure}\ m_2\ \varphi_2\ \psi\\ \forall a_1 a_2.\ \varphi_1\ a_1 \land \varphi_2\ a_2 \Rightarrow \varphi\ (a_1, a_2)\end{array}}{\mathit{pure}\ (\mathit{par}\, m_1\, m_2)\ \varphi\ \psi}$$

PURE-HANDLE
$$\frac{\mathit{pure}\ m\ (\lambda a.\,\mathit{pure}\ (h\ (\mathit{Ret}_3\ a))\ \varphi\ \psi)\ (\lambda v.\,\mathit{pure}\ (h\ (\mathit{Throw}_3\ v))\ \varphi\ \psi)}{\mathit{pure}\ (\mathit{handle}\, m\, h)\ \varphi\ \psi}$$

PURE-INTERSECTION
$$\frac{\forall a : A.\ \mathit{pure}\ m\ (\varphi\ a)\ \psi}{\mathit{pure}\ m\ (\lambda v.\,\forall a : A.\ \varphi\, a\, v)\ \psi}$$

Fig. 7. Horus rules for *micro* computations (*pure*)

The pure reduction relation serves as a foundation for the lower layer of Horus, a stateless Hoare logic for pure *micro* computations (§7.2). This layer involves a single judgment, *pure*. On top of it, we construct the upper layer of Horus, a stateless Hoare logic for OCaml programs (§7.3). This layer involves several judgments: there is one judgment per syntactic category, including expressions, patterns, and so on. Furthermore, we introduce a specific judgment for function specifications (§7.4).

7.2 Micro Layer

The pure judgment, $\mathit{pure}\ m\ \varphi\ \psi$, states that $m : \mathit{micro}\ A\ E$ is a pure and terminating computation that must either return a value that satisfies the normal postcondition $\varphi : A \to \mathit{Prop}$ or raise an exception that satisfies the exceptional postcondition $\psi : E \to \mathit{Prop}$. It is inductively defined in terms of the pure reduction relation via the following three rules, which can be read as follows: either the computation is finished and the corresponding postcondition holds; or the computation is able to make a step and, after every possible step, *pure* holds again.

$$\frac{\varphi\ a}{\mathit{pure}\ (\mathit{ret}\, a)\ \varphi\ \psi} \qquad \frac{\psi\ v}{\mathit{pure}\ (\mathit{throw}\ v)\ \varphi\ \psi} \qquad \frac{\exists m'.\ m \longrightarrow_{\mathrm{p}} m' \qquad \forall m'.\ m \longrightarrow_{\mathrm{p}} m' \Rightarrow \mathit{pure}\ m'\ \varphi\ \psi}{\mathit{pure}\ m\ \varphi\ \psi}$$

The soundness of Horus with respect to the semantics (§5) is an immediate consequence of this definition: if $\mathit{pure}\ m\ \varphi\ \bot$ holds then executing m in an arbitrary heap σ cannot diverge, cannot crash, and cannot result in an unhandled exception or effect; it must reach a result $\mathit{ret}\, a$ such that $\varphi\ a$ holds and leave the heap σ unchanged.

With respect to this definition, we establish the validity of a number of reasoning rules (Figure 7). There is one rule for each combinator of the *micro* monad (Figure 2), excluding those that cannot be used in a pure computation. For example, PURE-BIND can be read as follows: to establish that the sequence $\mathit{bind}\, m\, k$ is pure and satisfies the postconditions φ and ψ, one must prove that (1) m is pure, (2) if m produces a normal result a then $k\, a$ is pure and satisfies φ and ψ, (3) if m produces an exceptional result then this result satisfies ψ.

PURE-HANDLE is useful even in a pure setting (where delimited control effects cannot be used) because we use *handle* to interpret all **`match`** constructs. PURE-CONSEQ is the consequence rule. PURE-INTERSECTION is the intersection rule. There, X is a non-empty type. This rule pushes a universal quantification into the postcondition. All of the rules in Figure 7 are reversible. For example, out of a judgment about $\mathit{par}\, m_1\, m_2$, one can extract judgments about m_1 and m_2.

$$\textsc{EInt}\quad \frac{\varphi\ i}{\mathit{expr}\ \eta\ (\mathit{EInt}\ i)\ \varphi\ \psi}
\qquad
\textsc{EAdd}\quad \frac{\mathit{expr}\ \eta\ e_1\ \varphi_1\ \psi \qquad \mathit{expr}\ \eta\ e_2\ \varphi_2\ \psi \\ \forall i_1\ i_2.\ \varphi_1\ i_1 \Rightarrow \varphi_2\ i_2 \Rightarrow \varphi\ (i_1 + i_2)}{\mathit{expr}\ \eta\ \ (\mathit{EAdd}\ e_1\ e_2)\ \ \varphi\ \psi}
\qquad
\textsc{EIf}\quad \frac{\mathit{expr}\ \eta\ e\ (\lambda b.\ \mathit{expr}\ \eta\ (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2)\ \varphi\ \psi)\ \psi}{\mathit{expr}\ \eta\ (\mathit{EIfThenElse}\ e\ e_1\ e_2)\ \varphi\ \psi}$$

$$\textsc{ERaise}\quad \frac{\mathit{expr}\ \eta\ e\ \psi\ \psi}{\mathit{expr}\ \eta\ (\mathit{ERaise}\ e)\ \varphi\ \psi}
\qquad
\textsc{ESeq}\quad \frac{\mathit{expr}\ \eta\ e_1\ (\lambda\_.\ \mathit{expr}\ \eta\ e_2\ \varphi\ \psi)\ \psi}{\mathit{expr}\ \eta\ \ (\mathit{ESeq}\ e_1\ e_2)\ \ \varphi\ \psi}
\qquad
\textsc{ELet}\quad \frac{\mathit{bindings}\ \eta\ \mathit{bs}\ (\lambda\delta.\ \mathit{expr}\ (\delta +\!\!+\ \eta)\ e\ \varphi\ \psi)\ \psi}{\mathit{expr}\ \eta\ (\mathit{ELet}\ \mathit{bs}\ e)\ \varphi\ \psi}$$

$$\textsc{EMatch}\quad \frac{\mathit{expr}\ \eta\ e\ (\lambda a.\ \mathit{branches}\ \eta\ (\mathit{Ret}_3\ \#a)\ \mathit{bs}\ \varphi\ \psi) \\ (\lambda v.\ \mathit{branches}\ \eta\ (\mathit{Throw}_3\ v)\ \mathit{bs}\ \varphi\ \psi)}{\mathit{expr}\ \eta\ (\mathit{EMatch}\ e\ \mathit{bs})\ \varphi\ \psi}
\qquad
\textsc{BranchesCons}\quad \frac{\mathit{cpat}\ \eta\ \eta\ \mathit{cp}\ o\ (\lambda\eta'.\ \mathit{expr}\ \eta'\ e\ \varphi\ \psi)\ \zeta \\ \zeta \Rightarrow \mathit{branches}\ \eta\ o\ \mathit{bs}\ \varphi\ \psi}{\mathit{branches}\ \eta\ o\ (\mathit{Branch}\ \mathit{cp}\ e :: \mathit{bs})\ \varphi\ \psi}
\qquad
\textsc{BranchesNil}\quad \frac{o = \mathit{Throw}_3\ v \qquad \psi\ v}{\mathit{branches}\ \eta\ o\ []\ \varphi\ \psi}$$

Fig. 8. Selected Horus rules for OLang expressions (*expr*) and case analyses (*branches*)

The *pure* judgment satisfies a few additional deduction rules. For example, out of a postcondition, one can extract information: that is, $\mathit{pure}\ m\ \varphi\ \psi$ implies $(\exists a.\ \varphi\ a) \vee (\exists e.\ \psi\ e)$. Furthermore, the following two rules are valid, where $\perp$ stands for $\lambda\_.$ False:

$$\frac{\mathit{pure}\ m_1\ (\lambda a_1.\ \mathit{pure}\ m_2\ (\lambda a_2.\ \varphi_1\ a_1 \wedge \varphi_2\ a_2)\ \perp)\ \perp}{\mathit{pure}\ m_1\ \varphi_1\ \perp \wedge \mathit{pure}\ m_2\ \varphi_2\ \perp}
\qquad
\frac{\mathit{pure}\ m_1\ (\lambda a_1.\ \mathit{pure}\ m_2\ (\lambda a_2.\ \varphi\ a_1\ a_2)\ \psi)\ \perp}{\mathit{pure}\ m_2\ (\lambda a_2.\ \mathit{pure}\ m_1\ (\lambda a_1.\ \varphi\ a_1\ a_2)\ \perp)\ \psi}$$

7.3 OLang Layer

Although OLang is an untyped language, we give a typed view of its values in Horus. That is, instead of working with postconditions whose argument type is *val*, we want the user of the logic to write postconditions in Rocq with an argument type of their choosing, such as *unit*, *int*, *bool*, etc. For this purpose, we define a type class *Encode A* whose single method is $\mathit{encode} : A \longrightarrow \mathit{val}$. We write # as a short-hand for *encode*. It is a mapping of mathematical objects of type A into OLang values. It need not be injective. We define several commonly useful instances of this class. For example, the Rocq types $\mathbb{Z}$ and *unit* are instances of this class: #5 is *VInt* (*int.repr* 5), and #() is *VUnit*. This allows us to hide the tags *VInt* and *VUnit* from the user's view. In fact, we want the user to be entirely unaware of the manner in which typed OLang values are encoded as inhabitants of the type *val*, and to view *val* as an abstract type.

Reflecting this discussion, we define a judgment $\mathit{pure}_{\#}$ that takes an implicit parameter of type *Encode A* and where the postcondition φ has type $A \longrightarrow \mathit{Prop}$. Then, based on $\mathit{pure}_{\#}$ and *eval*, we define a judgment *expr* for pure OLang expressions, as well as similar judgments (not shown) for each of OLang's syntactic categories.

$$\begin{aligned}
\mathit{pure}_{\#}\ m\ \varphi\ \psi &:= \mathit{pure}\ m\ (\lambda v.\ \exists a.\ v = \#a \wedge \varphi\ a)\ \psi \\
\mathit{expr}\ \eta\ e\ \varphi\ \psi &:= \mathit{pure}_{\#}\ (\mathit{eval\_expr}\ \eta\ e)\ \varphi\ \psi
\end{aligned}$$

With respect to this definition of *expr*, we establish the validity of a number of reasoning rules, some of which are shown in Figure 8. In every *expr* judgment, the domain of the postcondition is implicit: for example, in EAdd, the postconditions φ_1 and φ_2 have argument type $\mathbb{Z}$; the variables i_1 and i_2 have type $\mathbb{Z}$ as well, as they are operands of +. Our reasoning rules for integer addition, subtraction, negation, and multiplication do *not* require the user to prove the absence of integer overflow.[6] Our reasoning rules for division and comparison do have such a requirement.

The sequential composition rule ESeq ignores the value produced by e_1. The more general sequential composition construct *ELet bs e* can bind any number of variables, so ELet is more complex; it relies on the auxiliary judgment *bindings* (not shown) to extend the environment with new bindings. It is instructive to examine two special cases of ELet, shown below, where the list *bs* contains only one binding. In ELet1Var a variable x is bound to the result of a subexpression e_1. In ELet1Pat a pattern p is used to deconstruct the result of e_1.

ELet1Var
$$\frac{expr\ \eta\ e_1\ (\lambda a.\ expr\ ((x, \#a) :: \eta)\ e_2\ \varphi\ \psi)\ \psi}{expr\ \eta\ (ELet\ [Binding\ (PVar\ x)\ e_1]\ e_2)\ \varphi\ \psi}$$

ELet1Pat
$$\frac{expr\ \eta\ e_1\ (\lambda a.\ pat\ \eta\ \eta\ p\ \#a\ (\lambda\eta'.\ expr\ \eta'\ e_2\ \varphi\ \psi)\ \bot)\ \psi}{expr\ \eta\ (ELet\ [Binding\ p\ e_1]\ e_2)\ \varphi\ \psi}$$

In ELet1Var, e_2 is examined under the environment $(x, \#a) :: \eta$, which extends η with a binding of the variable x to the value $\#a$ returned by e_1. In ELet1Pat, e_2 is examined under an environment η' that is obtained as the result of matching the value $\#a$ against the pattern p in environment η. This is expressed by the judgment *pat*, a Hoare-style judgment about pattern matching, which we define in terms of *pure* and *eval_pat*, as follows:

$$pat\ \eta\ \delta\ p\ v\ \varphi\ \zeta := pure\ (eval\_pat\ \eta\ \delta\ p\ v)\ \varphi\ (\lambda().\ \zeta)$$

The judgment $pat\ \eta\ \delta\ p\ v\ \varphi\ \zeta$ states that, starting with lookup-only environment η and extend-only environment δ, matching the value v against the pattern p cannot crash, must terminate, and either produces an environment that satisfies φ or fails by throwing $()$, in which case ζ holds. The function *eval_pat* (not shown) is part of our monadic interpreter. In short, $eval\_pat\ \eta\ \delta\ p\ v$ matches the value v against the pattern p. In case of success, the result is an extension of the environment (or fragment) δ with bindings for the bound variables of the pattern p. The environment η is used to look up data constructors of extensible algebraic data types.

In the premise of ELet1Pat, the use of $\bot$ as an exceptional postcondition of the *pat* judgment indicates that pattern matching is not allowed to fail; it must be exhaustive.

With respect to this definition of *pat*, we establish the validity of a number of reasoning rules (not shown). These rules support deeply nested patterns. An end user need not be aware of these rules: since the pattern is always statically known, our tactics are able to automatically apply these rules in such a way that the remaining subgoal is an *expr* judgment, requesting the user to verify a branch, under the assumption that this branch has been entered, and that the previous branches could not be entered.

Coming back to Figure 8, the rule EMatch deals with the OCaml expression "**match** e **with** bs" where each branch in the list *bs* is composed of a computation pattern *cp* (§3.9) and a body *e*. In order to reason about these syntactic categories (namely, branches and computation patterns), we define two Hoare-style judgments:

$$branches\ \eta\ o\ bs\ \varphi\ \psi := pure_{\#}\ (eval\_branches\ \eta\ o\ bs)\ \varphi\ \psi$$
$$cpat\ \eta\ \delta\ p\ v\ \varphi\ \zeta := pure\ (eval\_cpat\ \eta\ \delta\ cp\ v)\ \varphi\ (\lambda().\ \zeta)$$

EMatch states that one must first reason about the scrutinee (that is, the expression e), which produces either a normal result $\#a$ or an exceptional result v; then, one reasons about the application of the handler *bs* to this outcome via the judgment *branches*.

[6]How is this possible? Recall (§3.4) that the type *int* of machine integers is the semi-open interval $[-n, n)$, where n is 2^{w-1}. The function *int.repr* is the projection of $\mathbb{Z}$ into *int*. This function is idempotent and commutes with addition, so $int.repr\ (int.repr\ (x) + int.repr\ (y))$ is $int.repr\ (x + y)$, where x and y have type $\mathbb{Z}$. In other words, the result of adding the values $\#x$ and $\#y$ is always the value $\#(x + y)$, even if *overflow*, perhaps better described as *wraparound*, takes place. The same is true for negation, subtraction, and multiplication.

The rules BranchesCons and BranchesNil allow reasoning about each branch in turn. In the second premise of BranchesCons, the implication $\zeta \Rightarrow \cdots$ allows each branch to be verified under the assumption that the previous branches did not match. When *bs* is the empty list and *o* is a normal outcome ($Ret_3\ v$), the user must check that ζ contains a contradiction. When *bs* is the empty list and *o* is an exceptional outcome ($Throw_3\ v$), it can be proved by applying BranchesNil.

An end user normally does not encounter the judgments *branches* or *cpat*: indeed, we provide tactics that automatically apply BranchesCons, compute exceptional postconditions, and attempt to extract contradictions out of them, so the only remaining subgoals are *expr* judgments.

7.4 Function Specifications

To a *merge* function on sorted lists of integers, we wish to give a specification of this form:

$$P_{merge} := \lambda\ l_1\ l_2\ m.\ sorted\ l_1 \wedge sorted\ l_2 \Rightarrow pure_{\#}\ m\ (\lambda l.\ sorted\ l \wedge permutation\ l\ (l_1 +\!\!+\ l_2))\ \bot$$

Here, l_1 and l_2 are two Rocq lists, whose type is *list* $\mathbb{Z}$. The variable *m*, whose type is *micro val exn*, serves as an abstract placeholder for the function application. This specification requires the lists l_1 and l_2 to be sorted (a precondition) and guarantees that the function call produces a sorted list *l* that is a permutation of $l_1 +\!\!+\ l_2$ (a postcondition).

OLang's functions are unary (§3.6), so, by "*n*-ary function", we mean *n* nested λ-abstractions. Indeed, in OCaml, this "curried" style is the most popular style, as opposed to the "uncurried" style where an *n*-ary function expects an *n*-tuple as an argument.

To reason about curried *n*-ary functions and give them specifications that take the natural form shown above, we introduce the proposition $Spec\ \overline{\tau}\ c\ P$, which means "*c* is a function with domain $\overline{\tau}$ and specification *P*." In this proposition, $\overline{\tau}$ is a non-empty list of the Rocq types of the function's parameters (these types must be instances of *Encode*), *c* is a value (which represents the function—*c* is for "closure"), and *P* describes the behavior of the function. The specification itself has type $P : \overline{\tau} \rightarrow micro\ val\ exn \rightarrow Prop$. Its parameters are the function's parameters and a monadic computation, which represents an application of the function to its actual arguments. This style of specification is inspired by Moine et al. [2023], who use a similar specification predicate in a partial correctness setting. Internally, we define $Spec\ \overline{\tau}\ c\ P$ by induction on the list $\overline{\tau}$, as follows:

$$\frac{\forall (a : A).\ P\ a\ (call\ c\ \#a)}{Spec\ [A]\ c\ P} \qquad \frac{\forall (a : A).\ pure_{\#}\ (call\ c\ \#a)\ (\lambda c'.\ Spec\ \overline{\tau}\ c'\ (P\ a))\ \bot}{Spec\ (A :: \overline{\tau})\ c\ P}$$

In the base case (left), the function has one parameter of type *A*. In this case, for any argument $a : A$, the function call $call\ c\ \#a$ must satisfy the specification $P\ a$. (*call* was introduced in §3.6.) In the inductive case (right), the first parameter has type *A*, and there are more parameters. In that case, the function call $call\ c\ \#a$ must return a closure c' which itself satisfies $Spec\ \overline{\tau}\ c'\ (P\ a)$.

The rules in Figure 9 form the public API of the abstract predicate *Spec*. (*Spec* also enjoys a consequence rule, which we omit.) Spec-EAnonFun lets one prove that the expression $EAnonFun\ (\cdots)$, an *n*-ary function, produces a value *c* (a closure) that satisfies the specification *P*. We write $AnonFun\ \overline{x}\ e$ as a short-hand for a series of nested λ-abstractions. The rule's single premise requires the user to prove that the function's body *e* abides by the specification *P*. This proof is carried out under an environment where each formal parameter $x \in \overline{x}$ is bound to the corresponding actual parameter $\#a \in \#\overline{a}$. This takes place under a universal quantification over $\overline{a}$, as the actual parameters are unknown.

Spec-ELetRec governs the definition of one recursive function with an arbitrary number of formal parameters $\overline{x}$. Its first premise requires the user to exhibit a well-founded relation *R*, which applies to all parameters at once and has type $R : \overline{\tau} \rightarrow \overline{\tau} \rightarrow Prop$. Its second premise requires the user to prove that the function body e_f satisfies a specification *P*, under the assumption that

$$\textsc{Spec-EAnonFun}\quad \frac{\forall(\overline{a} : \overline{\tau}).\ P\ \overline{a}\ \ (\mathit{eval}\ ((\overline{x}, \#\overline{a}) :: \eta)\ e)}{\begin{array}{c}\mathit{expr}\ \eta\ (\mathit{EAnonFun}\ (\mathit{AnonFun}\ \overline{x}\ e))\\ (\lambda c.\, \mathit{Spec}\ \overline{\tau}\ c\ P)\ \psi\end{array}}$$

$$\textsc{Spec-EApp}\quad \frac{\begin{array}{c}\mathit{expr}\ \eta\ e\ (\lambda c.\, \mathit{Spec}\ \overline{\tau}\ c\ P)\ \psi\\ \mathit{expr}\ \eta\ e_1\ \varphi_1\ \psi \qquad \cdots \qquad \mathit{expr}\ \eta\ e_n\ \varphi_n\ \psi\\ \forall \overline{a}.\ \varphi_1\ a_1\ \Rightarrow \cdots \Rightarrow \varphi_n\ a_n \Rightarrow \forall m.\ P\ \overline{a}\ m \Rightarrow \mathit{pure}_{\#}\ m\ \varphi\ \psi\end{array}}{\mathit{expr}\ \eta\ \ (\mathit{EApp}\ e\ e_1 \cdots e_n)\ \ \varphi\ \psi}$$

$$\textsc{Spec-ELetRec}\quad \frac{\begin{array}{c}\mathit{wf}\ R\\ \forall c\ \overline{a}.\ \mathit{Spec}\ \overline{\tau}\ c\ (\lambda \overline{a}'\ m.\, R\ \overline{a}'\ \overline{a} \Rightarrow P\ \overline{a}'\ m) \Rightarrow P\ \overline{a}\ (\mathit{eval}\ ((\overline{x}, \#\overline{a}) :: (f, c) :: \eta)\ e_f)\\ \forall c.\ \mathit{Spec}\ \overline{\tau}\ c\ P \Rightarrow \mathit{expr}\ ((f, c) :: \eta)\ e\ \varphi\ \psi\end{array}}{\mathit{expr}\ \eta\ (\mathit{ELetRec}\ [\mathit{RecBinding}\ f\ \ (\mathit{AnonFun}\ \overline{x}\ e_f)]\ e)\ \varphi\ \psi}$$

Fig. 9. Selected Horus rules for OLang expressions (*expr*): n-ary function calls and function definitions

recursive calls (with strictly smaller arguments) obey the specification P. The third premise allows the user to assume that the function (represented by the closure c) satisfies P while verifying the right-hand side of the **`let rec`** construct.

Spec-EApp allows reasoning about n nested function applications as a single n-ary application. The first premise asks that the function e satisfy an n-ary specification P. The following n premises require the subexpressions e_i to be verified. In the last premise, their results are named a_i. There, the user must prove $\forall m.\ P\ \overline{a}\ m \Rightarrow \mathit{pure}_{\#}\ m\ \varphi\ \psi$. To better understand this proof obligation, consider how it is instantiated at a call site of *merge*. Then, P is P_{merge}, and the list $\overline{x}$ consists of the variables l_1 and l_2. The proof obligation takes the form:

$$\forall m.\ P_{merge}\ l_1\ l_2\ m \Rightarrow \mathit{pure}_{\#}\ m\ \varphi\ \psi$$

where l_1 and l_2 represent the actual arguments at this call site. Unfolding the definition of P_{merge} reveals a judgment "$\mathit{pure}_{\#}\ m\ \ldots$" on the left-hand side of the implication. Thus, after proving that the precondition $\mathit{sorted}\ l_1 \wedge \mathit{sorted}\ l_2$ holds, one can apply pure-conseq to eliminate the judgments "$\mathit{pure}_{\#}\ m\ \ldots$" on both sides of the implication. This yields a goal of the form:

$$\forall l.\ \mathit{sorted}\ l \wedge \mathit{permutation}\ l\ (l_1 \mathbin{+\!\!+} l_2)\ \ \Rightarrow\ \ \varphi\ l$$

In this goal, the variable l stands for the result of the function call. The user is allowed to assume that the postcondition of *merge* holds: that is, the list l is sorted and is a permutation of $l_1 \mathbin{+\!\!+} l_2$. She must then prove that the property that is eventually desired, $\varphi\ l$, follows from these facts.

8 Osiris

We now present Osiris, a Separation Logic for OLang. Osiris allows reasoning about OLang programs that exhibit all kinds of effects (§3), including divergence, state, and control effects, which Horus forbids. Osiris is based on Iris [Jung et al. 2018b] and borrows ideas from Hazel [de Vilhena and Pottier 2021], a variant of Iris that supports effect handlers.

8.1 Micro Layer

The lower layer of Osiris is a Separation Logic for monadic computations in the *micro* monad. Its main judgment, $\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi\ \psi$, means that the computation $m : \mathit{micro}\ A\ E$ cannot crash and that if it terminates then it must produce either a normal result that satisfies $\varphi : A \longrightarrow \mathit{iProp}$ or an exceptional result that satisfies $\psi : E \longrightarrow \mathit{iProp}$. ($\mathit{iProp}$ is the type of Iris assertions.) Furthermore, along the way, this computation may perform a sequence of zero, one or more control effects in accordance with the protocol $\Psi : \mathit{eff} \longrightarrow (\mathit{outcome}_2\ \mathit{val}\ \mathit{exn} \longrightarrow \mathit{iProp}) \longrightarrow \mathit{iProp}$.

IMPURE-RET
$$\frac{\varphi\ a}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{ret}\ a)\ \varphi\ \psi}$$

IMPURE-THROW
$$\frac{\psi\ e}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{throw}\ e)\ \varphi\ \psi}$$

IMPURE-BIND
$$\frac{\langle\Psi\rangle\ \mathit{impure}\ m\ (\lambda a.\ \langle\Psi\rangle\ \mathit{impure}\ k\ a\ \varphi\ \psi)\ \psi}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{bind}\ m\ k)\ \varphi\ \psi}$$

IMPURE-CONSEQ
$$\frac{\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi\ \psi \quad \forall a.\ \varphi\ a \mathrel{-\!\!*} \varphi'\ a \quad \forall e.\ \psi\ e \mathrel{-\!\!*} \psi'\ e}{\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi'\ \psi'}$$

IMPURE-PAR
$$\frac{\langle\Psi\rangle\ \mathit{impure}\ m_1\ \varphi_1\ \psi \quad \langle\Psi\rangle\ \mathit{impure}\ m_2\ \varphi_2\ \psi \quad \forall a_1 a_2.\ \varphi_1\ a_1 \mathrel{-\!\!*} \varphi_2\ a_2 \mathrel{-\!\!*} \varphi\ (a_1, a_2)}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{par}\ m_1\ m_2)\ \varphi\ \psi}$$

IMPURE-HANDLE
$$\frac{\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi\ \psi \quad \mathit{shallow\text{-}handler}\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ h\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\}}{\langle\Psi'\rangle\ \mathit{impure}\ (\mathit{handle}\ m\ h)\ \varphi'\ \psi'}$$

IMPURE-PERFORM
$$\frac{\Psi\ \mathit{allows\ perform}\ v\ \{\varphi \mid \psi\}}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{perform}\ v)\ \varphi\ \psi}$$

IMPURE-RESUME
$$\frac{\mathit{isCont}\ \ell\ k \qquad \triangleright \langle\Psi\rangle\ \mathit{impure}\ (k\ o)\ \varphi\ \psi}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{resume}\ \ell\ o)\ \varphi\ \psi}$$

Fig. 10. Selected Osiris rules for *micro* computations (*impure*)

A protocol [de Vilhena and Pottier 2021] describes the effects that a computation may perform and the responses that the enclosing effect handlers may provide. Our definition of protocols is the same as de Vilhena and Pottier's, except that we change the type of a response from *val* to $\mathit{outcome}_2\ \mathit{val}\ \mathit{exn}$, because, by continuing or discontinuing a continuation, a handler can respond with a normal result or with an exceptional result. If $X \rightarrow \mathit{iProp}$ is read informally as "a set of X's" then the type of protocols can be understood as "a set of pairs of an effect and a set of responses". Thus, a protocol describes which effects are permitted, and for each permitted effect, which responses are permitted.

Following de Vilhena and Pottier [2021], we write Ψ *allows perform* $v\ \{\varphi \mid \psi\}$ to mean that the protocol Ψ allows the request v and guarantees that the response will satisfy the postconditions φ and ψ.

A selection of the deduction rules for the judgment *impure* appears in Figure 10. In each rule, the premises are separated, and the horizontal bar is a magic wand. These rules are not meant to be surprising in any way: they are essentially a paraphrase of our small-step reduction rules (Figure 6) in the style of an Iris-based Separation Logic.

The rules IMPURE-RET, IMPURE-THROW, IMPURE-BIND, and IMPURE-PAR are analogous to PURE-RET, PURE-THROW, PURE-BIND, and PURE-PAR. The rule IMPURE-PAR is in fact the parallel composition rule of Separation Logic [O'Hearn 2007]. Therefore, to verify a parallel composition, one must split the current resource and separately verify each side.

The consequence rule, IMPURE-CONSEQ, is also known as the frame rule. A reader who is not familiar with this formulation is referred to the rule WP-MONO in Iris [Jung et al. 2018b, §6.2]. The *absence* of a persistence modality in the second and third premises of IMPURE-CONSEQ makes it a true frame rule, and reflects the fact that a computation terminates at most once—which is true in our setting because there are no multi-shot continuations.

IMPURE-PERFORM states that performing an effect v, and expecting its outcome to satisfy the postconditions φ and ψ, is permitted if and only if the protocol says so. IMPURE-RESUME and an analogous rule for *wrap* (omitted) paraphrase the reduction rules for *resume* and *wrap* in Figure 6, but use the abstract predicate *isCont* to hide the fact that a continuation is a heap-allocated object.

IMPURE-HANDLE reflects the fact that installing a handler via *handle* changes the description of a computation from Ψ, φ, ψ to Ψ', φ', ψ'. It is modeled after Hazel's shallow-handler rule [de Vilhena and Pottier 2021, Figure 6] so we do not explain it here. The name and definition of this judgment reflect the fact that *handle* installs a *shallow* handler, which handles at most one effect, then vanishes.

$$\textsc{impure-EPerform}\quad \frac{\langle\Psi\rangle\ \mathit{expr}\ \eta\ e\ \varphi'\ \psi \qquad \forall v.\ \varphi'(v) \twoheadrightarrow \Psi\ \mathit{allows\ perform}\ v\ \{\varphi \mid \psi\}}{\langle\Psi\rangle\ \mathit{expr}\ \eta\ (\mathit{EPerform}\ e)\ \varphi\ \psi}$$

$$\textsc{impure-EContinue}\quad \frac{\langle\Psi\rangle\ \mathit{expr}\ \eta\ e_1\ \varphi_1\ \psi \qquad \langle\Psi\rangle\ \mathit{expr}\ \eta\ e_2\ \varphi_2\ \psi \qquad \forall \ell, v.\ \varphi_1(\ell) \twoheadrightarrow \varphi_2(v) \twoheadrightarrow \exists k.\ \mathit{isCont}\ \ell\ k * \triangleright \langle\Psi\rangle\ \mathit{impure}\ (k\ (\mathit{Ret}_2\ v))\ \varphi\ \psi}{\langle\Psi\rangle\ \mathit{expr}\ \eta\ (\mathit{EContinue}\ e_1\ e_2)\ \varphi\ \psi}$$

$$\textsc{impure-EMatch}\quad \frac{\langle\Psi\rangle\ \mathit{expr}\ \eta\ e\ \varphi\ \psi \qquad \mathit{olang\text{-}deep\text{-}handler}\ \eta\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ \mathit{bs}\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\}}{\langle\Psi'\rangle\ \mathit{expr}\ \eta\ (\mathit{EMatch}\ e\ \mathit{bs})\ \varphi'\ \psi'}$$

Fig. 11. Selected Osiris rules for OLang expressions (*expr*)

8.2 OLang Layer

As we did in Horus (§7.3), in order to let the user entertain a typed view of values, we introduce an auxiliary judgment $\mathit{impure}_\#$ that takes an implicit parameter of type *Encode A* and where the postcondition φ has type $A \longrightarrow \mathit{iProp}$. Then, we define a judgment *expr* about OLang expressions.

$$\langle\Psi\rangle\ \mathit{impure}_\#\ m\ \varphi\ \psi := \langle\Psi\rangle\ \mathit{impure}\ m\ (\lambda v.\ \exists a.\ \ulcorner v = \#a \urcorner * \varphi\ a)\ \psi$$
$$\langle\Psi\rangle\ \mathit{expr}\ \eta\ e\ \varphi\ \psi := \langle\Psi\rangle\ \mathit{impure}_\#\ (\mathit{eval\_expr}\ \eta\ e)\ \varphi\ \psi$$

In this paper, the Horus judgment *expr* and the Osiris judgment *expr* are visually distinguished by the fact that the latter begins with an extra parameter $\langle\Psi\rangle$.

Some deduction rules for the Osiris judgment *expr* appear in Figure 11. The three rules shown correspond to the OCaml expressions "`perform e`", "`continue e1 e2`", and "`match e with bs`". impure-EContinue manages resuming a continuation with a value. If e_1 produces a stored continuation ℓ and e_2 produces a value v then this rules requires that it be safe to resume ℓ with $\mathit{Ret}_2\ v$. impure-EMatch relies on the auxiliary judgment *olang-deep-handler* to express the fact that the closed handler (η, bs) changes the description of the program's behavior from Ψ, φ, ψ to Ψ', φ', ψ'. The definition of this judgment (not shown) is obtained by composing de Vilhena and Pottier's *deep-handler* judgment [2021] with our function *eval_branches* (§3.9), which transforms the syntactic handler (η, bs) into a semantic handler (a function of a three-armed outcome to a computation).

8.3 Interaction Between the Two Logics

Sometimes, somewhere in the middle of an Osiris proof, the user faces a pure fragment of the program that they wish to verify. Then, they can "drop down" from Osiris to Horus. Consider this definition of *find_first*, which uses a local exception to cause an early return out of `List.iter`:

```
let find_first (type a) l pred =
  let open struct type exn += Found of a end in
  let scan x = if pred x then raise (Found x) in
  match List.iter scan l with
  | () -> None
  | exception Found x -> Some x
```

The instruction "`type exn += Found of a`" (which, technically, is a *structure item*) dynamically adds a new constructor to the extensible algebraic data type `exn`. In our semantics, executing this instruction allocates a fresh memory location ℓ and binds the name `Found` to this memory location. Because this instruction extends the store, it is not pure. Therefore, *find_first* cannot be verified using Horus alone. The specification of *find_first* must be expressed at the level of Osiris, and its verification must begin at this level. Nevertheless, once the focus of the proof moves past this instruction and reaches the beginning of line 3, the remaining code is pure. Thus, at this point, the user can exploit the following rule to drop down from Osiris to Horus:

$$\textsc{impure-pure}\quad \frac{\mathit{pure}\ m\ \varphi\ \psi}{\langle\Psi\rangle\ \mathit{impure}\ m\ (\lambda a.\ulcorner \varphi\ a \urcorner)\ (\lambda e.\ulcorner \psi\ e \urcorner)}$$

This rule states that a Horus judgment implies a similar Osiris judgment. In other words, a pure computation can be viewed as an impure computation. The rule can also be read from bottom to top: to prove an Osiris judgment about a computation *m*, it suffices to prove a Horus judgment about this computation. The protocol Ψ in the conclusion is arbitrary, so the empty protocol $\perp$ can be used: a pure computation has no control effects.

The specification of *find_first*, expressed at the level of Osiris, is as follows:

$$\begin{array}{rcl} P_{\mathit{find}} & := & \lambda\, l\ \mathit{pred}\ m.\ \forall\,(\varphi : A \to \mathit{Prop}).\\ & & \ulcorner \mathit{Spec}\ [A]\ \mathit{pred}\ (\lambda\, a\ m.\ \mathit{pure}_{\#}\ m\ (\lambda P.\ P \equiv \varphi\ a)\ \perp)\urcorner \twoheadrightarrow \\ & & \langle\perp\rangle\ \mathit{impure}_{\#}\ m\ \left(\begin{array}{c}\lambda\, o.\ \ulcorner o = \mathit{Some}\ a \land a \in l \land \varphi\ a\ \lor \\ o = \mathit{None} \land \forall\, a \in l.\ \neg\varphi\ a \quad \urcorner\end{array}\right)\ \perp \end{array}$$

This specification states that *find_first* expects a list *l* and a function *pred*. It requires *pred* to be a pure function whose Boolean result encodes the truth value of some predicate φ. It states that the application of *find_first* to *pred* and *l* (represented by the placeholder *m*) performs no control effects, raises no exceptions, and returns an option *o* such that if *o* is *Some a* then $\varphi\ a$ holds and if *o* is *None* then no element of *l* satisfies φ.

The verification of *find_first* goes as follows. We start with the goal of proving that *find_first* satisfies the specification P_{find}. We enter the function's body, introducing *find_first*'s parameters. We introduce φ as well as the hypothesis $\mathit{Spec}\ [A]\ \mathit{pred}\ \ldots$. We use one of Osiris's reasoning rules (not shown in this paper) to reason about the dynamic allocation instruction at line 2, which binds the name `Found` to a fresh memory location. At this point, the goal has the form:

$$\langle\perp\rangle\ \mathit{expr}\ \eta\ (\texttt{\textbf{let} scan x = ... \textbf{in} ...})\ \left(\begin{array}{c}\lambda\, o.\ \ulcorner o = \mathit{Some}\ a \land a \in l \land \varphi\ a\ \lor \\ o = \mathit{None} \land \forall\, a \in l.\ \neg\varphi\ a \quad \urcorner\end{array}\right)\ \perp.$$

In this goal, η is a concrete environment, where the bindings of the names `l`, `pred`, and `Found` are recorded. By applying IMPURE-PURE to this goal, we descend into Horus, with a similar goal: the current environment, code fragment, and postconditions are unchanged. From there on, the proof continues inside Horus. By exploiting a Horus specification of the function `List.iter`, which must have been previously established, as well as the Horus specification of the function *pred*, it is easy to finish the proof.

8.4 Soundness

We state the soundness of Osiris first at the level of the *micro* monad, then at the level of OLang. This property is known as "adequacy" in the Iris literature. In short, if a computation or program has been verified in Osiris under an empty protocol and an empty exceptional postcondition then it can diverge or return a value but cannot crash or terminate abruptly.

THEOREM 8.1. *Let m be a computation. If* $\vdash \langle\perp\rangle\ \mathit{impure}\ m\ (\lambda v.\ulcorner\varphi\ v\urcorner)\ \perp$ *holds then executing m in an empty heap cannot crash and cannot terminate with an unhandled effect or an unhandled exception. Furthermore, if this computation returns a value v then* $\varphi\ v$ *holds.*

Corollary 8.2. *Let e be an OLang expression. If $\vdash \langle\bot\rangle$ expr η e $(\lambda v.\ulcorner\varphi\ v\urcorner)$ $\bot$ holds then evaluating the expression e in environment η and in an empty heap cannot crash and cannot terminate with an unhandled effect or an unhandled exception. Furthermore, if this computation returns a value v then $\varphi\ v$ holds.*

9 Related Work

Formalizations of realistic ML-family languages. The semantics and type system of Standard ML have been the subject of early mechanization attempts [Syme 1993; VanInwegen and Gunter 1993], and later fully formalized [Lee et al. 2007; Harper and Crary 2014]. The semantics and type system of a subset of OCaml, which is also a subset of OLang, are formalized by Owens [2008]. He defines a small-step operational semantics and a deterministic executable interpreter, and proves that they agree. He chooses a fully specified evaluation order (right-to-left), because this makes testing easier. The CakeML compiler, whose source language is a large subset of Standard ML, is fully mechanized and verified [Kumar et al. 2014; Tan et al. 2019; Myreen 2021]. The semantics of CakeML is expressed as in "functional big-step" style [Owens et al. 2016]. Like ours, this interpreter takes the form of a recursive *eval* function. However, it is not monadic: it uses an explicit fuel parameter, explicit store passing, and explicit case analyses on outcomes. It is deterministic; external non-determinism is simulated by taking a stream of events as an extra parameter.

Program logics for ML-family languages. CFML [Charguéraud 2010, 2011, 2020] is a mechanized Separation Logic for an untyped subset of OCaml, which does not have exceptions or control effects. It uses characteristic formulae, which can be viewed as a syntax-directed presentation of Separation Logic. A similar mechanized Separation Logic has been defined for CakeML [Guéneau et al. 2017], and has been extended to enable reasoning about the input-output behavior of non-terminating programs [Pohjola et al. 2019]. A large part of Iris [Jung et al. 2018b], a powerful Separation Logic, is language-independent. Nevertheless, Iris is often used in conjunction with HeapLang, an untyped λ-calculus extended with mutable state and unstructured concurrency. Many verified algorithms and data structures in the Iris literature have been first translated from a realistic language into HeapLang, often through a manual transcription. We automate the translation of OCaml to OLang, so using Iris (Osiris) to verify OCaml code becomes easier. Compared to HeapLang, OLang adds exceptions, control effects, and unspecified evaluation order, but does not yet support concurrency.

Our treatment of delimited control effects is based de Vilhena and Pottier's work [2021]. They emphasize that forbidding multi-shot continuations allows the frame rule to remain everywhere valid. van Rooij and Krebbers [2025] extend their work to a calculus that offers both one-shot and multi-shot continuations and propose a variant of Separation Logic where the frame rule is applicable only in areas where no multi-shot effects take place.

Semantics and logics for other realistic languages. There have been several efforts to mechanize C [Norrish 1998; Ellison and Rosu 2012; Krebbers et al. 2014; Krebbers 2015]. The CompCert verified compiler uses CompCert C, a variant of C, as its source language [Leroy 2006, 2009, 2024]. The separation-logic-based verification frameworks for C include unverified systems such as VeriFast [Jacobs and Piessens 2008] and CN [Pulte et al. 2023] and verified systems such as VST [Appel 2011; Cao et al. 2018], Refined C [Sammler et al. 2021], and Iris/CompCert C [Mansky and Du 2024].

WebAssembly has been fully mechanized using small-step operational semantics [Watt 2021] and in several other styles, including a big-step semantics [Watt et al. 2019] and a monadic interpreter [Watt et al. 2023]. Its small-step semantics has been extended with delimited control effects [Phipps-Costin et al. 2023]. Several Separation Logics for WebAssembly have been proposed [Watt et al. 2019; Rao et al. 2023].

Early versions of Goose [Chajed et al. 2020], an Iris-based Separation Logic for Go, translate Go into a custom monad embedded in Rocq. This monad appears somewhat similar in spirit to ours. The paper shows just its syntax; its semantics is not defined. Later versions of Goose translate Go to GooseLang, an extension of HeapLang. Like HeapLang, GooseLang is equipped with a small-step substitution-based operational semantics. Goose has been used to verify several realistic Go programs [Chajed et al. 2019, 2021; Sharma et al. 2023].

Computation trees and modular semantics. The freer monad [Kiselyov and Ishii 2015] offers a representation of computations as finite trees. Its constructors correspond to our *Ret* and *Stop* (§4): thus, our monad is a custom extension of the freer monad. Interaction trees (ITrees) [Xia et al. 2020], a co-inductive variant of the freer monad, represent computations as possibly infinite trees, thereby offering native support for divergence. We prefer to work with finite trees and encode general recursion via the system call *CEval*.

The freer monad and the ITree monad are parameterized with an event signature, that is, a set of "events", or "system calls". They do not assign any semantics to events: this is done by defining an "event handler", that is, a monad morphism into some other monad—possibly an instance of the freer monad or ITree monad with a different event signature. A complex event handler can be constructed in several layers, that is, as the composition of several event handlers [Yoon et al. 2022]. This technique has been demonstrated in the Vellvm project [Zakowski et al. 2021] with a modular construction of the semantics of LLVM IR.

In contrast with most of the Iris literature so far, which is based on small-step operational semantics, Vistrup et al. [2025] define a generic Iris-based Separation Logic for ITrees. An important common point between their work and ours is their organization in two layers: there, the HeapLang layer and the ITree layer; here, the OLang layer and the *micro* layer. In both cases, the top layer is a monadic interpreter, that is, a denotational semantics of the surface language; the bottom layer is a computation tree monad. In Vistrup et al.'s work, the main judgment of the ITree-level logic, *wpi*, is defined by guarded recursion over trees. It is parameterized with an effect signature and with a "logical effect handler" that provides a specification for each effect. A logical effect handler is very much the same thing as a "protocol" [de Vilhena and Pottier 2021]. In de Vilhena and Pottier's paper, as in the present paper, protocols are used to describe user-level effects and handlers, whereas in Vistrup et al.'s work, logical effect handlers are used to describe meta-level effects and handlers that serve as basic components in a modular description of the semantics. Vistrup et al. construct an effect handler, a logical effect handler, and an adequacy theorem for each effect independently, including crashing, nondeterministic choice, state, and concurrency. These handlers, and their adequacy theorems, are then composed. We do not attempt to achieve this kind of modularity: we define the *micro* monad in a monolithic way. Although our computation trees (§4) are in some ways similar to ITrees, the main judgment of our logic, *impure*, is not defined by recursion over trees, like Vistrup et al.'s *wpi*; instead, following a more traditional approach, it is defined in terms of the small-step reduction relation that we have defined for our trees.

A limitation of ITrees is that they cannot describe computations or events whose arguments or results are computations. A naive attempt to extend ITrees with such a capability leads to an ill-formed type, whose definition involves a negative occurrence of itself. This makes it difficult to model languages that involve first-class functions or first-class continuations. We avoid this problem via an indirection through syntax: in our inductive type of values (*val*), a first-class function is represented by its environment and its code (*VClo*), and a first-class continuation is represented by its address (*VCont*). Instead of following this path, Frumin et al. [2024] introduce Guarded Interaction Trees (GITrees), whose definition relies on guarded recursion instead of co-induction, therefore tolerates negative self-references. Using GITrees, they give a denotational semantics to a calculus equipped with first-class functions, and Stepanenko et al. [2025] give a semantics to calculi equipped with several forms of control effects. Both papers define an Iris-based judgment *wp* for GITrees. Like our judgment *impure*, and unlike Vistrup et al.'s *wpi*, this judgment appears to be defined in terms of a reduction relation on trees.

The ITree literature places emphasis on using the equational theory of ITrees to justify program transformations. With this motivation in mind, Chappe et al. [2023] develop Choice Trees, an extension of ITrees with non-determinism, which also enjoys a rich equational theory. In contrast, we currently have no tools to compare two monadic computations. To address this need, in the future, we would like to develop relational Separation Logics for the *micro* monad and for OLang.

10 Future Work

We have formalized the abstract syntax and dynamic semantics of a substantial fragment of OCaml. Our semantic style is a modular combination of a monadic interpreter and a custom monad, whose definition is original and relies on a small-step operational semantics. We have constructed two program logics, Horus and Osiris, whose soundness we have machine-checked.

We have tested our semantics by executing a small number of examples. Much more work is needed to ensure that our semantics is consistent with existing implementations of OCaml. We have tested the expressiveness and usability of our program logics by verifying a few small programs. Using Horus, we have verified a merge sort and some operations on splay trees. Using Osiris, we have verified de Vilhena and Pottier's short but challenging "control inversion" example [2021, §5]. The program logics are expressive enough to verify these examples in a straightforward manner. However, the ratio of lines of verification over lines of code still seems quite high. Much more work is needed to assess and improve the usability of our program logics.

In the future, we wish to enlarge OLang, so as to make progress towards a complete formal definition of the dynamic semantics of OCaml, and so as to be able to offer Horus and Osiris as practical tools for the interactive verification of OCaml programs. Among the features of OCaml that we do not yet support, concurrency (the ability of spawning new threads via *fork*) and weak memory seem most important. We believe that these features cannot be modeled using *Par*; instead, we plan to introduce a separate notion of thread. Concurrency is a well-understood feature of Iris [Jung et al. 2018b], and there exists a variant of Iris that accounts for OCaml's weak memory model [Mével et al. 2020]; we plan to rely on these works. Several other major features that we do not yet support are functors, objects and classes, and labeled and optional arguments.

In the long term, we would like to widen the scope of our program logics so as to verify liveness properties of possibly non-terminating, effectful, concurrent programs; time and space complexity properties; or security properties. Furthermore, we are interested in defining relational program logics and in connecting our formal semantics of OLang with a verified compilation toolchain such as CakeML [Kumar et al. 2014] or a future verified OCaml compiler.

Data Availability Statement

The latest version of our mechanized definitions and proofs is available online [The Osiris Project 2025] and an artifact has been created to accompany this publication [Seassau et al. 2025].

Acknowledgments

This work has been partially funded by ANR under Grant ANR-22-CE48-0013-01 "GOSPEL".

References

Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 2337–2363.

Léo Andrès, Pierre Chambart, and Jean-Christophe Filliâtre. 2023. Wasocaml: compiling OCaml to WebAssembly. In *Implementation of Functional Languages (IFL)*.

Andrew W. Appel. 2011. Verified Software Toolchain. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 6602)*. Springer, 1–17.

Calvin Beck, Hanxi Chen, and Steve Zdancewic. 2025. Vellvm: Formalizing the Informal LLVM. In *NASA Formal Methods (NFM)*. Springer, 91–99.

John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 142:1–142:28.

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *Principles of Programming Languages (POPL)*. 87–100.

Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 44:1–44:31.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1-4 (2018), 367–422.

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Symposium on Operating Systems Principles (SOSP)*. 243–258.

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2020. Verifying concurrent Go code in Coq with Goose. In *Workshop on Coq for Programming Languages*.

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *Symposium on Operating Systems Design and Implementation*. 423–439.

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1770–1800.

Arthur Charguéraud. 2010. Program Verification Through Characteristic Formulae. In *International Conference on Functional Programming (ICFP)*. 321–332.

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming (ICFP)*. 418–430.

Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 116:1–116:34.

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Transactions on Programming Languages and Systems* 45, 1 (2023), 5:1–5:43.

Nathanaëlle Courant, Julien Lepiller, and Gabriel Scherer. 2022. Debootstrapping without Archeology - Stacked Implementations in Camlboot. *Art, Science, and Engineering of Programming* 6, 3 (2022), 13.

Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021).

Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Principles of Programming Languages (POPL)*. 533–544.

Dan Frumin, Léon Gondelman, and Robbert Krebbers. 2019. Semi-automated Reasoning About Non-determinism in C Expressions. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 60–87.

Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 332–361.

Yoshihiko Futamura. 1999a. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

Yoshihiko Futamura. 1999b. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation* 12, 4 (1999), 377–380.

Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. 2015. A Trusted Mechanised Specification of JavaScript: One Year On. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 9206)*. Springer, 3–10.

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *International Conference on Functional Programming (ICFP)*. 339–347.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 584–610.

Robert Harper and Karl Crary. 2014. The Mechanization of Standard ML. (Feb. 2014). Available online.

Bart Jacobs and Frank Piessens. 2008. *The VeriFast Program Verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 41:1–41:32.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Haskell symposium*. 94–105.

Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems* 28, 4 (2006), 619–695.

Robbert Krebbers. 2014. An operational and axiomatic semantics for non-determinism and sequence points in C. In *Principles of Programming Languages (POPL)*. 101–112.

Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph. D. Dissertation. Radboud University Nijmegen.

Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. 2014. Formal C Semantics: CompCert and the C Standard. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science, Vol. 8558)*. Springer, 543–548.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL)*. 179–192.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Programming Language Design and Implementation (PLDI)*. 618–632.

Isabella Leandersson. 2022. Six Surprising Reasons the OCaml Programming Language is Good for Business. https://tarides.com/blog/2022-11-22-six-surprising-reasons-the-ocaml-programming-language-is-good-for-business/.

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of Standard ML. In *Principles of Programming Languages (POPL)*. 173–184.

Xavier Leroy. 2000. A modular module system. *Journal of Functional Programming* 10, 3 (2000), 269–303.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*. 42–54.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Xavier Leroy. 2024. The CompCert C compiler. http://compcert.org/.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2024. The OCaml system.

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Principles of Programming Languages (POPL)*. 333–343.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 196:1–196:25.

Andreas Lochbihler. 2012. Java and the Java Memory Model – A Unified, Machine-Checked Formalisation. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 497–517.

David MacQueen, Robert Harper, and John H. Reppy. 2020. The history of Standard ML. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 86:1–86:100.

William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 148–174.

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Principles of Programming Languages (POPL)*. 378–391.

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, Vol. 9129)*. Springer, 257–275.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 718–747.

Antonio Vinhas Nunes Monteiro. 2025. Melange. https://github.com/melange-re/melange.

Magnus O. Myreen. 2021. The CakeML Project's Quest for Ever Stronger Correctness Theorems. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 193)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:10.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (June 2020).

Michael Norrish. 1998. *C formalised in HOL*. Technical Report UCAM-CL-TR-453. University of Cambridge.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (May 2007), 271–307.

Scott Owens. 2008. A Sound Semantics for OCamllight. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 4960)*. Springer, 1–15.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9632)*. Springer, 589–615.

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 460–485.

Gordon D. Plotkin. 2004. The origins of structural operational semantics. *Journal of Logical and Algebraic Methods in Programming* 60-61 (2004), 3–15.

Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. 2019. Characteristic Formulae for Liveness Properties of Non-Terminating CakeML Programs. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 141)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19.

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.

Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 1 (Jan. 2023), 32 pages.

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1096–1120.

Didier Rémy and Jérôme Vouillon. 1998. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems* 4, 1 (1998), 27–50.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Programming Language Design and Implementation (PLDI)*. 158–174.

Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml - Artifact. doi:10.5281/zenodo.15863614

Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Symposium on Operating Systems Principles (SOSP)*. 113–129.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 113:1–113:30.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Programming Language Design and Implementation (PLDI)*. 206–221.

Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, and Lars Birkedal. 2025. Context-Dependent Effects in Guarded Interaction Trees. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*. Springer.

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436.

Don Syme. 1993. Reasoning with the Formal Definition of Standard ML in HOL. In *Higher Order Logic Theorem Proving and its Applications (HUG) (Lecture Notes in Computer Science, Vol. 780)*. Springer, 43–60.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), e2.

The Osiris Project. 2025. Osiris. https://gitlab.inria.fr/fpottier/osiris.

Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. *Proceedings of the ACM on Programming Languages* 9, POPL, Article 5 (Jan. 2025).

Myra VanInwegen and Elsa Gunter. 1993. HOL-ML. In *Higher Order Logic Theorem Proving and its Applications (HUG) (Lecture Notes in Computer Science, Vol. 780)*. Springer, 61–74.

Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 11:1–11:32.

Jérôme Vouillon. 2023. `wasm_of_ocaml`. https://cambium.inria.fr/seminaires/transparents/20231213.Jerome.Vouillon.pdf. Slides.

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the `Js_of_ocaml` compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972.

Qian Wang and Ralf Jung. 2024. Rustlantis: Randomized Differential Testing of the Rust Compiler. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1955–1981.

Conrad Watt. 2021. *Mechanising and evolving the formal semantics of WebAssembly: the Web's new low-level language*. Ph. D. Dissertation. University of Cambridge.

Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. 2019. A Program Logic for First-Order Encapsulated WebAssembly. In *European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics, Vol. 134)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:30.

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods (FM) (Lecture Notes in Computer Science, Vol. 13047)*. Springer, 61–79.

Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 100–123.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 51:1–51:32.

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal reasoning about layered monadic interpreters. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 254–282.

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–30.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Principles of Programming Languages (POPL)*. 427–440.

Received 2025-02-27; accepted 2025-06-27