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

We present a formal definition of OLang, a nontrivial 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. This paper represents a first step towards a formal definition of OCaml and a foundational program verification environment for OCaml.

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.* 1, 1, Article 1 (January 2025), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A formal (mechanized) definition can be a valuable foundation for a programming language. A mechanized semantics rules out the inaccuracies that usually appear in informal specifications (such as reference manuals) 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; 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.

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.

2025. ACM 2475-1421/2025/1-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While certain simple and well-understood tools, such as small-step operational semantics [Plotkin 2004; Wright and Felleisen 1994], are expressive enough to describe the semantics of any programming language, these tools require significant skill and effort to put into practice. The search for new semantic styles that are more elegant, expressive, or 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]. It is still a challenge to find a semantic framework that is powerful enough to allow the desired semantics to be expressed and simple enough to allow this semantics to serve as a useful basis in projects such as the construction of an equational theory, a program logic, or a verified compiler.

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].

As of today, 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, the OCaml-to-JavaScript compilers `js_of_ocaml` [Vouillon and Balat 2014] and `melange` [Monteiro 2025], and the OCaml-to-WebAssembly compilers `wasocaml` [Andrès et al. 2023] and `wasm_of_ocaml` [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, and the Salto analyzer [Lermusiaux and Montagu 2024b,a] performs static analysis of OCaml programs. 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 fragment of OCaml. This fragment includes first-class functions, ordinary and extensible algebraic data types, pattern matching, references, exceptions, deep and shallow effect handlers,[1] 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 so-called *pure* programs, which cannot diverge or exploit mutable state or 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.

[1]We use OCaml 5.3, which has concrete syntax for deep effect handlers, and offers access to shallow handlers via a library.

Our work is carried out using Rocq; our results are machine-checked.[2] 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 move on to a presentation of Horus (§6) and Osiris (§7). The paper ends with discussions of related work (§8) and future work (§9). Extra material appears in the appendices.

2 Architecture and Design Choices

Untyped semantics. The semantics of OLang is untyped. There is a sum type of all values, *val*. Every value carries a tag. This tag is inspected by dynamic checks, whose failure causes a crash. This approach is standard: in the tradition of Milner [1978] and Wright and Felleisen [1994], some programs can go wrong (crash), but well-typed programs do not go wrong. It offers two benefits. First, it lets us assign a meaning to all programs, not just well-typed programs. This allows us to support some uses of unsafe type casts (§A). Second, it lets us avoid the need to define OCaml's type system, which would be a formidable task.

Type-based disambiguation. Our translator of OCaml into OLang is intended to be as simple as possible. This is important, as it is unverified and must be trusted. Yet, a potential difficulty is created by the fact that OCaml's syntax can be ambiguous. For example, two distinct algebraic data types can have data constructors named `A`. In OLang's syntax, this ambiguity does not exist. To avoid this difficulty and to keep our translator simple, we let the OCaml compiler perform type-based disambiguation. Our translator consumes a typed OCaml AST, which is produced by the OCaml type-checker. Thus, the OCaml parser and type-checker are part of our trusted code base.

Hybrid monadic/operational semantics. Our semantics is constructed as a modular composition of two layers, each of which adopts a distinct semantic style. The top layer is a monadic interpreter. This style seems easy to understand and review and lends itself well to execution, either inside Rocq or via extraction. It is *denotational* in the sense that the interpreter is defined by induction on the abstract syntax of the program. The bottom layer constructs the monad that the interpreter relies upon. This monad must support a large collection of effects (Figure 4). There, we do not attempt to perform a modular construction, where each effect is interpreted independently; we provide a monolithic construction. We represent a monadic computation as a tree whose nodes include final outcomes (*Ret*, *Throw*, *Crash*), observable events or *system calls* (*Stop*), parallel compositions (*Par*), and control effect delimiters (*Handle*). The behavior of a computation is given by a small-step reduction relation. This style is *operational.*

As in previous work on the free monad and its variants [Swierstra 2008; Kiselyov and Ishii 2015; Xia et al. 2020], the monad has a subtle dual appearance. Inside its implementation, the monad is just syntax: a computation is a tree. Outside the monadic abstraction, though, the monad's public API (Figure 4) offers a shallowly embedded DSL that is very convenient to use.

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

[2]Our code and proofs are provided as supplementary material.

or right-to-left; it can be a seemingly arbitrary permutation. The order that is actually chosen in practice varies across compilers, and can be difficult for the user to predict.

To account for this feature, our semantics must be non-deterministic: it must permit all of the permutations that the OCaml manual allows. In fact, we find that, by relaxing our semantics even further, we are able to simplify its definition. We allow *parallel evaluation* of the subexpressions. As an example of the simplicity that this buys us, we are able to 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 design decision implies that some programs that have only one possible result according to the OCaml manual can have non-deterministic behavior and several possible results in our semantics. 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: a programmer likely does not expect the above example program to return 1. Perhaps, in an effort to verify an OCaml compiler, one would prefer to adopt a stricter semantics of OCaml. One would separately prove that the two semantics are related.

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 can help offer a gentler learning curve. Furthermore, Horus can verify termination, whereas Osiris cannot: following most of the Iris literature, Osiris imposes partial correctness only, because this makes verifying concurrent programs much easier. Finally, Horus has helped us study certain problems (such as the treatment of pattern matching) in a simpler setting.

Exceptions versus effects. We do not view OCaml's exception handling mechanism as a special case of its effect handling mechanism where the handler discards the continuation. Here are three reasons why. First, in OCaml, discarding a continuation is considered a programming error, because it can prevent the suspended computation from performing cleanup tasks. The manual says: *every continuation must be eventually either continued or discontinued* [Leroy et al. 2024]. In other words, performing an effect creates a continuation, which must be considered linear, whereas raising an exception does not. Second, to *discontinue* a continuation means to resume the suspended computation with an exception. There is no means of resuming a continuation with an effect. This creates another asymmetry between exceptions and effects. Finally, we are able to view exceptions as "pure" and reason about them in Horus. Effects, on the other hand, must be viewed as an "impure" feature, because continuations are stored in the heap and are mutable (they are one-shot).

Undefined versus undesirable behavior. A crash represents a bad event. Our program logics rule out crashes: they have precondition *false*. Thus, a verified program cannot crash. That said, our current notion of crash conflates two kinds of bad events, namely those that OCaml's type

$$
\begin{array}{lcl}
\mathit{val} & := & \mathit{VInt}\ (i : \mathit{int}) \\
 & & \mathit{VTuple}\ (\mathit{vs} : \mathit{list}\ \mathit{val}) \\
 & & \mathit{VData}\ (c : \mathit{string})\ (\mathit{vs} : \mathit{list}\ \mathit{val}) \\
 & & \mathit{VXData}\ (\ell : \mathit{loc})\ (\mathit{vs} : \mathit{list}\ \mathit{val}) \\
 & & \mathit{VLoc}\ (\ell : \mathit{loc}) \mid \mathit{VCont}\ (\ell : \mathit{loc}) \\
 & & \mathit{VClo}\ (\eta : \mathit{env})\ (a : \mathit{anonfun})
\end{array}
\qquad
\begin{array}{lcl}
\mathit{loc} & := & \mathbb{Z} \\
\mathit{env} & := & \mathit{list}\ (\mathit{var} \times \mathit{val}) \\
\mathit{anonfun} & := & \mathit{AnonFun}\ (x : \mathit{var})\ (e : \mathit{expr}) \\
\mathit{exn} & := & \mathit{val} \\
\mathit{eff} & := & \mathit{val}
\end{array}
$$

Fig. 1. OLang's type of values

system can rule out, such as the failure of a dynamic tag check, and those that it cannot rule out, such as the failure of a runtime assertion, a division by zero, or falling off the end of a case analysis. Bad events of the first kind represent *undefined behavior*, whereas those of the second kind represent *undesirable behavior*. This distinction is useful because a compiler can assume the absence of undefined behavior, whereas undesirable behavior can occur and must cause the program to stop in a graceful way. We plan to introduce this distinction in the near future.

Names. We represent variables, record fields, and module fields in OLang as strings. This is straightforward and keeps our translator simple. A downside of this syntax is that it is not well-scoped by construction; a program can have unbound variables. It might seem tempting to adopt a well-scoped representation, based, for example, on de Bruijn indices. That said, in light of the complexity of OCaml's module language, it is not entirely clear what form such a representation would take or in what way we would benefit from it.

3 A Monadic Interpreter

Our semantics takes the form of a *monadic interpreter* [Liang et al. 1995] for OCaml. 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 (crashing), 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 4; 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 OCaml's surface syntax

The syntax of OCaml involves several categories, such as expressions, patterns, module expressions, and structure items. In this paper, we put emphasis on expressions for the sake of brevity. We use a deep embedding [Gibbons and Wu 2014]: that is, we represent OCaml's syntax in Rocq via several inductive types, including *expr* and *pat*. The definitions of these types (not shown) closely reflect the surface syntax of OCaml, so that it is easy to transform OCaml code into (Rocq definitions of) inhabitants of these types. We provide a straightforward 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 our interpreter accepts untyped code, 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 (not all cases are shown in Figure 1).

In *VData*, the name of the data constructor is a string (see §3.5). In *VXData*, the name of the data constructor is a memory location (§C.2). In the future, we want *VData* and *VXData* to also record the identity of the algebraic data type with which this constructor is associated. Indeed, this seems necessary in order to support unsafe type casts (§A).

In OCaml, 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.

3.3 Structure of the monadic interpreter

The interpreter is composed of multiple mutually recursive functions. There is typically one function for each syntactic category of OCaml along with a number of auxiliary functions. In this paper, we are mainly interested in the following function, which forms the heart of the monadic interpreter:

$$\mathit{eval\_expr} \quad : \quad \mathit{env} \longrightarrow \mathit{expr} \longrightarrow \mathit{micro}\ \mathit{val}\ \mathit{exn}$$

We write *eval* as a short-hand for *eval_expr*. The meta-level expression $\mathit{eval}\ \eta\ e$ evaluates the OCaml 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 OCaml value) or an abnormal result of type *exn* (an OCaml exception). It can also exhibit a range of other effectful behaviors, including crashing, diverging, and more; we discuss these later on. The function *eval* is defined by induction on its second argument, e, which is an abstract syntax tree.

In the following subsections (§3.4–§3.9), we present fragments of the definition of *eval* in Rocq to illustrate how the combinators of the *micro* monad are used.

3.4 Integer Arithmetic / Return, Bind, Crash

We use the evaluation of OCaml's integer arithmetic expressions as an illustration of the most basic combinators of the *micro* monad, whose full list appears in Figure 4. The computation *ret a* returns the result a. The combinator *bind* constructs the sequential composition of two computations. As usual, we write $x \longleftarrow m_1; m_2$ for $\mathit{bind}\, m_1\, (\lambda x.\, m_2)$. The combinator *crash* can be understood as a fatal failure or as undefined behavior—in any case, 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 $\mathbb{Z}$.[3] We convert i to a machine integer via *int.repr*, convert it to an integer value via *VInt*, then return it. An integer

[3]In OCaml, machine integers are signed and have a fixed bit width w. The value of w is unspecified. 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

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. This dynamic check is required because OLang is untyped.

3.5 Algebraic Data Types / Parallel Composition

OCaml supports user-defined algebraic data types, also known as variant types. In the expression *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 4). 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 the 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, as 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 OCaml's surface syntax, all functions are unary, and function application is binary. We represent a unary 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 4), 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.

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

To model OCaml's mutable references, we rely on three combinators offered by the *micro* monad, namely *alloc*, *load*, and *store* (Figure 4). 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 contains exception-handling branches of the form "**exception** p -> e", where p is a pattern.[4]

We interpret raising an exception by using the combinator *throw* (Figure 4).

```
| 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.[5] This guarantees that 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.

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 computation terminates; it successively handles all of the effects that this computation performs. In OCaml's surface syntax, 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. There is no surface syntax for shallow handlers; instead, the library Effect.Shallow offers access to shallow handlers via a number of

[4]We choose to view the more ancient exception-handling construct "**try** e1 **with** p -> e2" as syntactic sugar for "**match** e1 **with** x -> x | **exception** p -> e2".
[5]Declaring a new exception via "**exception** E **of** int" is sugar for "**type** exn += E **of** int".

```
| 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)
```

Fig. 2. Interpretation of effect-related constructs (a fragment of the definition of *eval η e*)

```
Fixpoint eval_branches η o bs : micro val exn :=       Fixpoint wrap_eval_branches η bs o :=
  match bs with                                          o ← wrap_outcome η bs o ;
  | Branch cp e :: bs ⇒                                  eval_branches η o bs.
      try2 (eval_cpat η η cp o) (fun o ⇒
        match o with                                   Fixpoint eval_cpat η δ cp o
        | Ret2 η'  ⇒ eval η' e                         : micro env unit :=
        | Throw2 () ⇒ eval_branches η o bs               match cp, o with
        end)                                             | CVal p, Ret3 v ⇒
  | [] ⇒                                                     eval_pat η δ p v
      match o with                                       | CExc p, Throw3 v ⇒
      | Ret3 _    ⇒ crash                                    eval_pat η δ p v
      | Throw3 v ⇒ throw v                               | CEff pe pk, Perform3 e k ⇒
      | Perform3 v l ⇒                                       δ ← eval_pat η δ pe e ;
          try2 (perform v) (fun o ⇒ resume l o)              eval_pat η δ pk (VCont k)
      end                                                | _, _ ⇒ throw ()
  end.                                                   end.
```

Fig. 3. Case analysis on outcomes

primitive functions. We support both deep and shallow handlers. In this paper, we discuss deep handlers only, as they are more common and easier to use.

Performing an effect. Our interpretation of **perform** e appears in Figure 2. It uses the monadic combinator *perform* (Figure 4). This combinator is meant to interact with the combinator *handle*, which is discussed later on in this section.

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 are shown in Figure 2. Both expect the expression e1 to produce a stored continuation, that is, a value of the form *VCont* ℓ, where ℓ 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 4), whose arguments are a heap address where a continuation is stored and an outcome with which to resume this continuation. This outcome has type $outcome_2$ *val exn*, a sum type that can represent normal and exceptional results (Figure 4). In **continue**, the continuation is resumed with a normal outcome $Ret_2\ v$; in **discontinue**, it is resumed with an exceptional outcome $Throw_2\ v$.

Inductive $\mathit{outcome}_2\ (A\,E : \mathit{Type})\ : \mathit{Type} := \mathit{Ret}_2\ (a : A) \mid \mathit{Throw}_2\ (e : E)$

Inductive $\mathit{outcome}_3\ (A\,E : \mathit{Type})\ : \mathit{Type} := \mathit{Ret}_3\ (a : A) \mid \mathit{Throw}_3\ (e : E) \mid \mathit{Perform}_3\ (v : \mathit{eff})\ (\ell : \mathit{loc})$

$$
\begin{array}{lcrl}
\mathit{micro} & : & \mathit{Type} \longrightarrow \mathit{Type} \longrightarrow \mathit{Type} & \\
\mathit{ret} & : & A \longrightarrow \mathit{micro}\ A\ E & \\
\mathit{throw} & : & E \longrightarrow \mathit{micro}\ A\ E & \\
\mathit{try}_2 & : & \mathit{micro}\ B\ E' \longrightarrow (\mathit{outcome}_2\ B\ E' \longrightarrow \mathit{micro}\ A\ E) \longrightarrow \mathit{micro}\ A\ E & \\
\mathit{bind} & : & \mathit{micro}\ B\ E \longrightarrow (B \longrightarrow \mathit{micro}\ A\ E) \longrightarrow \mathit{micro}\ A\ E & \text{— derived from } \mathit{try}_2 \\
\mathit{orelse} & : & \mathit{micro}\ A\ E' \longrightarrow \mathit{micro}\ A\ E \longrightarrow \mathit{micro}\ A\ E & \\
\mathit{crash} & : & \mathit{micro}\ A\ E & \\
\mathit{please\_eval} & : & \mathit{env} \longrightarrow \mathit{expr} \longrightarrow \mathit{micro}\ \mathit{val}\ \mathit{exn} & \\
\mathit{alloc} & : & \mathit{val} \longrightarrow \mathit{micro}\ \mathit{loc}\ \mathit{exn} & \\
\mathit{load} & : & \mathit{loc} \longrightarrow \mathit{micro}\ \mathit{val}\ \mathit{exn} & \\
\mathit{store} & : & \mathit{loc} \longrightarrow \mathit{val} \longrightarrow \mathit{micro}\ \mathit{unit}\ \mathit{exn} & \\
\mathit{par} & : & \mathit{micro}\ A_1\ E \longrightarrow \mathit{micro}\ A_2\ E \longrightarrow \mathit{micro}\ (A_1 \times A_2)\ E & \\
\mathit{handle} & : & \mathit{micro}\ \mathit{val}\ \mathit{exn} \longrightarrow (\mathit{outcome}_3\ \mathit{val}\ \mathit{exn} \longrightarrow \mathit{micro}\ A\ E) \longrightarrow \mathit{micro}\ A\ E & \\
\mathit{perform} & : & \mathit{eff} \longrightarrow \mathit{micro}\ \mathit{val}\ \mathit{exn} & \\
\mathit{resume} & : & \mathit{loc} \longrightarrow \mathit{outcome}_2\ \mathit{val}\ \mathit{exn} \longrightarrow \mathit{micro}\ \mathit{val}\ \mathit{exn} & \\
\mathit{wrap} & : & \mathit{loc} \longrightarrow \mathit{env} \longrightarrow \mathit{handler} \longrightarrow \mathit{micro}\ \mathit{loc}\ \mathit{exn} &
\end{array}
$$

Fig. 4. The *micro* monad: public interface

Handling effects. Our interpretation of "**`match`** `e` **`with`** `bs`" also appears in Figure 2. To interpret this construct, we interpret the expression `e` in the scope of an effect handler. To install this handler and to delimit its scope, we use *handle* (Figure 4). In short, *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 4) 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 $\ell : \mathit{loc}$. Indeed, 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*, 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.

Wrapping a continuation in a handler. In the case where the outcome `o` is an effect ($\mathit{Perform}_3$), 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 4), which returns a new stored continuation. In the other two cases (Ret_3, Throw_3), *wrap_outcome* acts as an identity function. The definition of *wrap_outcome* is omitted (§C, Figure 13).

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* η δ 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.

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 4) 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.[6] *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).

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

Fig. 5. The *micro* monad: definition

4.1 Definition

An inductive type of computations. A mathematical object of type $\mathit{micro}\ A\ E$ 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 most fundamental combinator, try_2 (Figure 4). 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 $\mathit{micro}\ A\ E$ appears in Figure 5. 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 $\mathit{Stop}\,c\,v\,k$, the code c determines the types of the argument v and of the outcome expected by the continuation k. Indeed, if c has type $\mathit{code}\ X\ Y\ E'$ 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. In addition to $\mathit{Ret}\ v$, a trivial computation whose outcome is the result v, we have $\mathit{Throw}\ v$, a trivial computation whose outcome is the exception v, and *Crash*, a trivial computation that represents a fatal runtime failure. Whereas an exception can be caught and handled, a crash cannot be detected or handled. A crash can also be thought of as "undefined behavior" (§B). In any case, it is a "bad" event, which one wishes to avoid. By design, our program logics (§6, §7) ensure that a verified program cannot crash. The combinators *ret*, *throw*, and *crash* are just 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 $\mathit{try}_2\ m\,k$, the continuation k expects an outcome, that is, either a result or an exception. Crashes are propagated: $\mathit{try}_2\ (\mathit{crash})\ k$ is *crash*.

The sequential composition combinator *bind* is obtained as a special case of try_2. In $\mathit{bind}\,m\,k$, the continuation k expects a result. The monadic laws are satisfied: in particular, $\mathit{bind}\ (\mathit{ret}\,v)\ k$ is $k\,v$. Exceptions and crashes are propagated: $\mathit{bind}\ (\mathit{throw}\ v)\ k$ is $\mathit{throw}\ v$ and $\mathit{bind}\ (\mathit{crash})\ k$ 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, $\mathit{par}\,m_1\,m_2$ is defined as $\mathit{Par}\,m_1\,m_2\ \mathit{inject}_2$, where inject_2 is the trivial continuation.[6]

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 $\mathit{Handle}\ m\ h$ is the computation m running under the handler h. The combinator *handle* (Figure 4) is just a synonym for *Handle*.

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

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

Fig. 6. The *micro* monad: codes, also known as system calls

As indicated by the type of *Handle* in Figure 5, the handler h is a three-armed continuation: that is, it expects an outcome of type $\mathit{outcome}_3$ _ _.

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 $\mathit{outcome}_3$ *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 $\mathit{outcome}_3$ and *micro* are *not* mutually recursive. Indeed, $\mathit{outcome}_3$ is defined first (Figure 4); *micro* refers to it (Figure 5). The key reason why this is possible is that the second argument of the constructor $\mathit{Perform}_3$ 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 $\mathit{outcome}_3$ and *micro* would be mutually recursive. Furthermore, because $\mathit{outcome}_3$ appears in a negative position in the arguments of the constructor *Handle* (Figure 5), the definitions of the types $\mathit{outcome}_3$ 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 6. 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 produces a value (or an exception). The combinator *please_eval* (Figure 4) is defined by $\mathit{please\_eval}\ \eta\ e = \mathit{stop}\ \mathit{CEval}\ (\eta, e)$.

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

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 4) 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 4) 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 $(\eta, \ell, \mathit{bs})$, where bs is a *handler*, a list of branches. (This type is part of our abstract syntax of OCaml.) Its intended effect is to allocate a new

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\textbf{Divergence}} \\
!\, CEval\,(\eta, e)\, k \,/\, \sigma & \longrightarrow & try_2\,(eval\;\eta\;e)\,k \,/\, \sigma & \\
\multicolumn{4}{l}{\textbf{State}} \\
!\, CAlloc\, v\, k \,/\, \sigma & \longrightarrow & try_2\,(ret\,\ell)\,k \,/\, (\ell, v) :: \sigma & \text{if } \ell \notin dom(\sigma) \\
!\, CLoad\, \ell\, k \,/\, \sigma & \longrightarrow & try_2\,(ret\,v)\,k \,/\, \sigma & \text{if } lookup\;\sigma\;\ell = v \\
!\, CLoad\, \ell\, k \,/\, \sigma & \longrightarrow & Crash \,/\, \sigma & \text{otherwise} \\
!\, CStore\,(\ell, v')\, k \,/\, \sigma & \longrightarrow & try_2\,(ret\,())\,k \,/\, (\ell, v') :: \sigma & \text{if } lookup\;\sigma\;\ell = v \\
!\, CStore\,(\ell, v')\, k \,/\, \sigma & \longrightarrow & Crash \,/\, \sigma & \text{otherwise} \\
\multicolumn{4}{l}{\textbf{Parallelism}} \\
Par\, m_1\, m_2\, k \,/\, \sigma & \longrightarrow & Par\, m_1'\, m_2\, k \,/\, \sigma' & \text{if } m_1 \,/\, \sigma \longrightarrow m_1' \,/\, \sigma' \\
Par\, m_1\, m_2\, k \,/\, \sigma & \longrightarrow & Par\, m_1\, m_2'\, k \,/\, \sigma' & \text{if } m_2 \,/\, \sigma \longrightarrow m_2' \,/\, \sigma' \\
Par\,(Ret\; v_1)\,(Ret\; v_2)\, k \,/\, \sigma & \longrightarrow & try_2\,(ret\,(v_1, v_2))\,k \,/\, \sigma & \\
Par\; Crash\, m_2\, k \,/\, \sigma & \longrightarrow & Crash \,/\, \sigma & \\
Par\, m_1\; Crash\, k \,/\, \sigma & \longrightarrow & Crash \,/\, \sigma & \\
Par\,(Throw\; v)\, m_2\, k \,/\, \sigma & \longrightarrow & try_2\,(throw\; v)\,k \,/\, \sigma & \\
Par\, m_1\,(Throw\; v)\, k \,/\, \sigma & \longrightarrow & try_2\,(throw\; v)\,k \,/\, \sigma & \\
Par\,(!\, CPerf\, v\, k)\, m_2\, k' \,/\, \sigma & \longrightarrow & !\, CPerf\; v\;(\lambda o.\, Par\,(k\, o)\, m_2\, k') \,/\, \sigma & \\
Par\, m_1\,(!\, CPerf\, v\, k)\, k' \,/\, \sigma & \longrightarrow & !\, CPerf\; v\;(\lambda o.\, Par\, m_1\,(k\, o)\, k') \,/\, \sigma & \\
\multicolumn{4}{l}{\textbf{Delimited control}} \\
Handle\;(Ret\; v)\; h \,/\, \sigma & \longrightarrow & h\;(Ret_3\; v) \,/\, \sigma & \\
Handle\;(Throw\; v)\; h \,/\, \sigma & \longrightarrow & h\;(Throw_3\; v) \,/\, \sigma & \\
Handle\;(!\, CPerf\, v\, k)\; h \,/\, \sigma & \longrightarrow & h\;(Perform_3\; v\; \ell) \,/\, (\ell, k) :: \sigma & \text{if } \ell \notin dom(\sigma) \\
Handle\; Crash\; h \,/\, \sigma & \longrightarrow & Crash \,/\, \sigma & \\
Handle\; m\; h \,/\, \sigma & \longrightarrow & Handle\; m'\; h \,/\, \sigma' & \text{if } m \,/\, \sigma \longrightarrow m' \,/\, \sigma' \\
!\, CResume\,(\ell, o)\, k \,/\, \sigma & \longrightarrow & try_2\;(k'\; o)\,k \,/\, (\ell, \lightning) :: \sigma & \text{if } lookup\;\sigma\;\ell = k' \\
!\, CResume\,(\ell, o)\, k \,/\, \sigma & \longrightarrow & Crash \,/\, \sigma & \text{otherwise} \\
!\, CWrap\,(\eta, \ell, bs)\, k \,/\, \sigma & \longrightarrow & try_2\,(ret\,\ell')\,k \,/\, (\ell', k') :: \sigma & \text{if } \ell' \notin dom(\sigma) \\
\multicolumn{4}{c}{\text{where } k' = \lambda o.\; handle\,(resume\;\ell\;o)\,(wrap\_eval\_branches\;\eta\;bs)}
\end{array}
$$

Fig. 7. The *micro* monad: small-step reduction

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 $wrap\;\ell\;\eta\;bs = stop\; CWrap\,(\eta, \ell, bs)$.

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 $\lightning$, which denotes a continuation that has been "shot" already.

In the previous sentence, k has type $outcome_2\; val\; exn \longrightarrow micro\; val\; exn$. 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 7. 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 $eval\ \eta\ e$ 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 continuation $inject_2$ then this rule states that $please\_eval\ \eta\ e$ reduces to $eval\ \eta\ e$.

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. $CLoad\ \ell$ reads the value stored at location ℓ, if this location has been allocated and stores a value; otherwise, it crashes. $CStore\ (\ell, v')$ 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 $Par\, m_1\, m_2\, k$ 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, when specialized to the case where k is the trivial continuation ($inject_2$), states that $par\,(ret\, v_1)\,(ret\, v_2)$ reduces to $ret\,(v_1, v_2)$. That is, if both m_1 and m_2 reach a result then $par\, m_1\, m_2$ returns a pair of these results. This is fork/join parallelism: once both sides have finished, the continuation proceeds.

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. Writing $try_2\,(throw\ v)\,k$ is the natural way of expressing that the exception v is propagated into the continuation k. When k is $inject_2$, one finds that $par\,(throw\ v)\,m_2$ 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 7 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 that was passed as an argument to *CPerf* 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 case where k is the trivial continuation, states that if a continuation k' is stored at address ℓ then $\mathit{resume}\ \ell\ o\ /\ \sigma$ reduces to $k'\ o\ /\ (\ell, \not{\ell}) :: \sigma$. In words, $\mathit{resume}\ \ell\ o$ 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. Indeed, OCaml only supports one-shot continuations.

The last reduction rule, when specialized to the case where k is the trivial continuation, states that $\mathit{wrap}\ \ell\ \eta\ \mathit{bs}\ /\ \sigma$ reduces to $\mathit{ret}\,\ell'\ /\ (\ell', k') :: \sigma$, 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 $\mathit{Par}\,m_1\,m_2$ and $\mathit{Handle}\ m\ h$ 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 $\mathit{ret}\,v$, $\mathit{throw}\ v$, *crash*, and $!\ \mathit{CPerf}\ v\,k$. The last form represents an unhandled effect.

Our reduction semantics is compatible with evaluation contexts: that is, $m\ /\ \sigma \longrightarrow m'\ /\ \sigma'$ implies $\mathit{try}_2\,m\,k\ /\ \sigma \longrightarrow \mathit{try}_2\,m'\ k\ /\ \sigma'$. Such a property is often part of the definition of a small-step operational semantics: in our case, it is a lemma.

6 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 Hoare logic, which is significantly simpler than Separation Logic. We believe that this can be beneficial for the end user. 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. We believe that requiring termination makes the logic more practically useful.

6.1 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.

The pure reduction relation serves as a foundation for the lower layer of Horus, a stateless Hoare logic for pure *micro* computations (§6.2). 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 (§6.3). This layer involves several judgments: there is one judgment per syntactic category, including expressions, patterns, and so on. Furthermore, we introduce a specific assertion for function specifications (§6.4).

6.2 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 return either a value that satisfies the normal postcondition $\varphi : A \rightarrow \mathit{Prop}$ or an exception that satisfies the exceptional postcondition $\psi : E \rightarrow \mathit{Prop}$. It is inductively defined in terms of

$$\textsc{pure-ret}\quad\frac{\varphi\ a}{\mathit{pure}\ (\mathit{ret}\ a)\ \varphi\ \psi}\qquad \textsc{pure-throw}\quad\frac{\psi\ e}{\mathit{pure}\ (\mathit{throw}\ e)\ \varphi\ \psi}\qquad \textsc{pure-bind}\quad\frac{\mathit{pure}\ m\ (\lambda a.\,\mathit{pure}\ (k\ a)\ \varphi\ \psi)\ \psi}{\mathit{pure}\ (\mathit{bind}\,m\,k)\ \varphi\ \psi}\qquad \textsc{pure-please-eval}\quad\frac{\mathit{pure}\ (\mathit{eval}\ \eta\ e)\ \varphi\ \psi}{\mathit{pure}\ (\mathit{please\_eval}\ \eta\ e)\ \varphi\ \psi}$$

$$\textsc{pure-conseq}\quad\frac{\mathit{pure}\ m\ \varphi\ \psi\qquad \forall a.\ \varphi\ a \Rightarrow \varphi'\ a\qquad \forall e.\ \psi\ e \Rightarrow \psi'\ e}{\mathit{pure}\ m\ \varphi'\ \psi'}\qquad \textsc{pure-try2}\quad\frac{\mathit{pure}\ m\ (\lambda a.\,\mathit{pure}\ (k\ (\mathit{Ret}_2\ a))\ \varphi\ \psi)\ (\lambda e.\,\mathit{pure}\ (k\ (\mathit{Throw}_2\ e))\ \varphi\ \psi)}{\mathit{pure}\ (\mathit{try}_2\,m\,k)\ \varphi\ \psi}\qquad \textsc{pure-par}\quad\frac{\mathit{pure}\ m_1\ \varphi_1\ \psi\qquad \mathit{pure}\ m_2\ \varphi_2\ \psi\qquad \forall a_1 a_2.\ \varphi_1\ a_1 \wedge \varphi_2\ a_2 \Rightarrow \varphi\ (a_1, a_2)}{\mathit{pure}\ (\mathit{par}\,m_1\,m_2)\ \varphi\ \psi}$$

$$\textsc{pure-handle}\quad\frac{\mathit{pure}\ m\ (\lambda a.\,\mathit{pure}\ (h\ (\mathit{Ret}_3\ a))\ \varphi\ \psi)\ (\lambda e.\,\mathit{pure}\ (h\ (\mathit{Throw}_3\ a))\ \varphi\ \psi)}{\mathit{pure}\ (\mathit{handle}\,m\,h)\ \varphi\ \psi}\qquad \textsc{pure-intersection}\quad\frac{\forall x : X.\ \mathit{pure}\ m\ (\varphi\,x)\ \psi}{\mathit{pure}\ m\ (\lambda a.\forall x : X.\ \varphi\,x\,a)\ \psi}$$

Fig. 8. Horus rules for *micro* computations (*pure*)

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.

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

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 8). There is one rule for each combinator of the *micro* monad (Figure 4), 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, where X is a non-empty type. It pushes a universal quantification into the postcondition. All of the rules in Figure 8 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.

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 $\bot$ stands for $\lambda_.\,$False:

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

These rules help sequentialize subgoals. This can be convenient when reasoning about expressions with multiple subexpressions. For example, this can help avoid the creation of Rocq metavariables.

6.3 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

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

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

$$\textsc{EMatch}\ \frac{expr\ \eta\ e\ (\lambda a.\ branches\ \eta\ (Ret_3\ \#a)\ bs\ \varphi\ \psi)\ (\lambda v.\ branches\ \eta\ (Throw_3\ v)\ bs\ \varphi\ \psi)}{expr\ \eta\ (EMatch\ e\ bs)\ \varphi\ \psi} \qquad \textsc{BranchesCons}\ \frac{cpat\ \eta\ \eta\ cp\ o\ (\lambda\eta'.\ expr\ \eta'\ e\ \varphi\ \psi)\ \zeta \qquad \zeta \Rightarrow branches\ \eta\ o\ bs\ \varphi\ \psi}{branches\ \eta\ o\ (Branch\ cp\ e :: bs)\ \varphi\ \psi} \qquad \textsc{BranchesNil}\ \frac{o = Throw_3\ v \qquad \psi\ v}{branches\ \eta\ o\ []\ \varphi\ \psi}$$

Fig. 9. Selected Horus rules for OLang expressions (*expr*) and case analyses (*branches*)

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 *encode* : $A \longrightarrow val$. 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 $pure_\#$ that takes an implicit parameter of type *Encode A* and where the postcondition φ has type $A \longrightarrow Prop$. Then, based on $pure_\#$ 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.

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

With respect to this definition of *expr*, we establish the validity of a number of reasoning rules, some of which are shown in Figure 9. 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. 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) (§E.4) 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.

$$\textsc{ELet1Var}\ \frac{expr\ \eta\ e_1\ (\lambda a.\ expr\ ((x, \#a) :: \eta)\ e_2\ \varphi\ \psi)\ \psi}{expr\ \eta\ (ELet\ [Binding\ (PVar\ x)\ e_1]\ e_2)\ \varphi\ \psi} \qquad \textsc{ELet1Pat}\ \frac{expr\ \eta\ e_1\ (\lambda a.\ pat\ \eta\ \eta\ p\ \#a\ (\lambda\eta'.\ expr\ \eta'\ e_2\ \varphi\ \psi)\ \bot)\ \psi}{expr\ \eta\ (ELet\ [Binding\ p\ e_1]\ e_2)\ \varphi\ \psi}$$

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.

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) (§E.3). 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 9, 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 auxiliary syntactic categories (branches and patterns), we define specialized Hoare-style judgments using *pure*:

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

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, we reason about the application of the handler bs to this outcome via the judgment *branches*.

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 ($Ret_3\ v$), this premise must be proved by contradiction: 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.

6.4 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 name 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.

Spec-EAnonFun

$$\frac{\forall(\vec{a}:\vec{\tau}).\ P\ \vec{a}\ \ (eval\ ((\vec{x},\#\vec{a}) :: \eta)\ e)}{expr\ \eta\ \ (EAnonFun\ (AnonFun\ \vec{x}\ e))\ (\lambda c.\ Spec\ \vec{\tau}\ c\ P)\ \psi}$$

Spec-EApp

$$\frac{\begin{array}{c} expr\ \eta\ e\ (\lambda c.\ Spec\ \vec{\tau}\ c\ P)\ \psi \\ expr\ \eta\ e_1\ \varphi_1\ \psi \qquad \cdots \qquad expr\ \eta\ e_n\ \varphi_n\ \psi \\ \forall \vec{a}.\ \varphi_1\ a_1\ \Rightarrow \cdots \Rightarrow \varphi_n\ a_n \Rightarrow \forall m.\ P\ \vec{a}\ m \Rightarrow pure_{\#}\ m\ \varphi\ \psi \end{array}}{expr\ \eta\ \ (EApp\ e\ e_1 \cdots e_n)\ \ \varphi\ \psi}$$

Spec-ELetRec

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

Fig. 10. Selected Horus rules for OLang expressions (*expr*): *n*-ary function calls and function definitions

To reason about curried *n*-ary functions and give them specifications that take the natural form shown above, we introduce the proposition $Spec\ \vec{\tau}\ c\ P$, which means "c is a function with domain $\vec{\tau}$ and specification P." In this proposition, $\vec{\tau}$ 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 $P : \vec{\tau} \to micro\ val\ exn \to Prop$. 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 style in a partial correctness setting. Internally, we define $Spec\ \vec{\tau}\ c\ P$ by induction on the list $\vec{\tau}$, as follows:

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

In the base case (left), the function has one parameter of type A. In this case, for every 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 $Spec\ \vec{\tau}\ c'\ (P\ a)$.

The rules in Figure 10 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 $AnonFun\ \vec{x}\ e$ as a short-hand for the 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 $x \in \vec{x}$ is bound the corresponding actual parameter $\#a \in \#\vec{a}$. This takes place under a universal quantification over $\vec{a}$, as the actual parameters are unknown.

Spec-ELetRec governs the definition of one recursive function with an arbitrary number of formal parameters $\vec{x}$. Its first premise requires the user to exhibit a well-founded relation R, which applies to all parameters at once and has type $R : \vec{\tau} \to \vec{\tau} \to Prop$. Its second premise requires the user to prove that the function body e_f satisfies a specification P, under the assumption that 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 $\forall m.\ P\ \vec{a}\ m \Rightarrow pure_{\#}\ m\ \varphi\ \psi$. To better understand this proof obligation,

$$\textsc{impure-ret}\quad \frac{\varphi\ a}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{ret}\ a)\ \varphi\ \psi} \qquad \textsc{impure-throw}\quad \frac{\psi\ e}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{throw}\ e)\ \varphi\ \psi} \qquad \textsc{impure-bind}\quad \frac{\langle\Psi\rangle\ \mathit{impure}\ m\ (\lambda a.\, \langle\Psi\rangle\ \mathit{impure}\ k\ a\ \varphi\ \psi)\ \psi}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{bind}\, m\, k)\ \varphi\ \psi}$$

$$\textsc{impure-conseq}\quad \frac{\begin{array}{c}\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi\ \psi \\ \forall a.\ \varphi\ a \mathrel{-\!\!*} \varphi'\ a \\ \forall e.\ \psi\ e \mathrel{-\!\!*} \psi'\ e\end{array}}{\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi'\ \psi'} \qquad \textsc{impure-par}\quad \frac{\begin{array}{c}\langle\Psi\rangle\ \mathit{impure}\ m_1\ \varphi_1\ \psi \\ \langle\Psi\rangle\ \mathit{impure}\ m_2\ \varphi_2\ \psi \\ \forall a_1 a_2.\ \varphi_1\ a_1 \mathrel{-\!\!*} \varphi_2\ a_2 \mathrel{-\!\!*} \varphi\ (a_1, a_2)\end{array}}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{par}\, m_1\, m_2)\ \varphi\ \psi} \qquad \textsc{impure-handle}\quad \frac{\begin{array}{c}\langle\Psi\rangle\ \mathit{impure}\ m\ \varphi\ \psi \\ \mathit{shallow\text{-}handler}\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ h\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\}\end{array}}{\langle\Psi'\rangle\ \mathit{impure}\ (\mathit{handle}\, m\, h)\ \varphi'\ \psi'}$$

$$\textsc{impure-pure}\quad \frac{\mathit{pure}\ m\ \varphi\ \psi}{\langle\Psi\rangle\ \mathit{impure}\ m\ (\lambda a. \ulcorner \varphi\ a \urcorner)\ (\lambda e. \ulcorner \psi\ e \urcorner)} \qquad \textsc{impure-perform}\quad \frac{\Psi\ \mathit{allows\ perform}\ v\ \{\varphi \mid \psi\}}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{perform}\ v)\ \varphi\ \psi} \qquad \textsc{impure-resume}\quad \frac{\ell \mapsto k \qquad \triangleright \langle\Psi\rangle\ \mathit{impure}\ (k\ o)\ \varphi\ \psi}{\langle\Psi\rangle\ \mathit{impure}\ (\mathit{resume}\ \ell\ o)\ \varphi\ \psi}$$

Fig. 11. Selected Osiris rules for *micro* computations (*impure*)

consider how it is instantiated at a call site of *merge*. Then, P is P_{merge}, and the list $\vec{x}$ 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 $\mathit{sorted}\ l_1 \land \mathit{sorted}\ l_2$ 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:

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

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 $l_1 +\!\!+\ l_2$. She must then prove that the property that is eventually desired, $\varphi\ l$, follows from these facts.

7 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.

7.1 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 $\psi : E \longrightarrow \mathit{iProp}$. (*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 $\Psi : \mathit{eff} \longrightarrow (\mathit{outcome}_2\ \mathit{val}\ \mathit{exn} \longrightarrow \mathit{iProp}) \longrightarrow \mathit{iProp}$.

A protocol [de Vilhena and Pottier 2021] describes the effects that a computation is allowed to perform and the responses that the enclosing effect handlers are allowed to 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 $X \longrightarrow \mathit{iProp}$ 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 φ or ψ. The definition of this predicate, which we omit, fits in one line, but it helps to think of it abstractly.

The definition of *impure*, which we also omit (§F, Figure 15), is very similar to that of de Vilhena and Pottier’s *ewp* [2021]. A selection of the deduction rules for this judgment appear in Figure 11. 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 7) 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-PURE states that a Horus judgment implies a similar Osiris judgment. In other words, a pure computation can be viewed as an impure computation. The protocol Ψ in the conclusion is arbitrary, so the empty protocol $\bot$ could be used: a pure computation performs no control effects. This rule establishes a bridge between Horus and Osiris. Thanks to it, inside an Osiris proof, Horus can be used to reason about pure computations. In particular, since pattern matching is pure, we are able to re-use Horus’s support for pattern matching (§E.3) within Osiris proofs.

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 IMPURE-WRAP (omitted) paraphrase the reduction rules for *resume* and *wrap* in Figure 7. The “later” modality $\triangleright$ that appears in the second premise of these rules reflects the fact that one step of reduction has been made; this is standard in Iris [Jung et al. 2018b, §6.2].

The rules for heap access (*alloc*, *load*, *store*), are standard, and are also paraphrases of the small-step semantics; we omit them (§F, Figure 16).

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 a similar rule in Hazel [de Vilhena and Pottier 2021]. The complexity of this rule is delegated to the auxiliary judgment *shallow-handler*, whose definition and deduction rules are omitted (§F, Figure 18). The name and definition of this judgment reflect the fact that *handle* installs a *shallow* handler, which handles at most one effect, then vanishes.

7.2 OLang Layer

As we did in Horus (§6.3), in order to let the user entertain a typed view of values, we introduce an auxiliary judgment $impure_{\#}$ that takes an implicit parameter of type *Encode A* and where the postcondition φ has type $A \to iProp$. Then, we define a judgment *impure* about OLang expressions.

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

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 $\langle\Psi\rangle$.

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

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

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

Fig. 12. Selected Osiris rules for OLang expressions (*expr*)

Some deduction rules for the Osiris judgment *expr* appear in Figure 12. The three rules shown correspond to the OCaml expressions "`perform e`", "`continue e1 e2`", and "`match e with bs`". In impure-EContinue, a points-to assertion $\ell \mapsto k$ betrays the fact that a continuation is a heap-allocated object. This is an unnecessary detail; in the future, we plan to hide it by defining an abstract predicate *isCont* for first-class continuations. impure-EMatch relies on the auxiliary judgement *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'. Our definition of this judgement (omitted) (§F, Figure 18) is obtained by composing de Vilhena and Pottier's *deep-handler* judgement [2021] with the 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).

7.3 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 7.1. *Let m be a computation. If $\vdash \langle\bot\rangle\ \mathit{impure}\ m\ \varphi\ \bot$ holds then executing m in an empty heap cannot crash and cannot terminate with an unhandled effect or an unhandled exception.*

Corollary 7.2. *Let e be an OLang expression. If $\vdash \langle\bot\rangle\ \mathit{expr}\ \eta\ e\ \varphi\ \bot$ holds then evaluating e in environment η and in an empty heap cannot crash and cannot terminate with an unhandled effect or an unhandled exception.*

8 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].

Goose [Chajed et al. 2020] translates a subset of Go into a custom monad embedded in Rocq. This monad appears somewhat similar in spirit to ours. Unfortunately, the paper shows just the syntax of the monad; its semantics is not defined. Goose has been used to verify several realistic Go programs [Chajed et al. 2019, 2021].

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. The logic's main judgment, *wpi*, is defined by guarded recursion over trees. It is parameterized with an event signature and with a "logical event handler" that provides a specification for each event.[7] Vistrup et al. construct both event handlers and logical event handlers in a compositional way, as a combination of basic components that describe individual effects, including crashing, nondeterministic choice, state, and concurrency. Each component comes with an adequacy theorem that relates its event handler and its logical event handler; these adequacy theorems are then composed. We do not attempt to achieve this kind of modularity: we define the *micro* monad in a monolithic way. We do achieve a different kind of modularity, in our semantics and in our program logics, by distinguishing two layers, the *micro* layer and the OLang layer. Finally, 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.

9 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]. 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

[7]This seems technically similar to the manner in which de Vilhena and Pottier's judgment *ewp* [2021] is parameterized with a protocol, which provides a specification for each control effect that the program might perform.

that we do not yet support, concurrency (the ability of spawning new threads via *fork*) and weak memory seem most important and perhaps challenging. 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 hope to rely on these works. Appendix §9 lists more features of OCaml that we wish to support.

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.

References

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.

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.

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.

Jason Gross, Andres Erbsen, Jade Philipoom, Rajashree Agrawal, and Adam Chlipala. 2024. Towards a Scalable Proof Engine: A Performant Prototype Rewriting Primitive for Coq. *Journal of Automated Reasoning* 68, 3 (2024), 19.

Jason S. Gross. 2021. *Performance Engineering of Proof-Based Software Systems at Scale*. Ph. D. Dissertation. Massachusetts Institute of Technology.

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.

Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2023. Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 385–416.

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.

Pierre Lermusiaux and Benoît Montagu. 2024a. Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 14577)*, Stephanie Weirich (Ed.). Springer, 391–420.

Pierre Lermusiaux and Benoît Montagu. 2024b. The Salto Project: Static Analysis of OCaml Programs by Abstract Interpretation. *ERCIM News* 139, 138 (2024).

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.

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.

Gabriel Scherer, Thomas Réfis, and Nicholas Roberts. 2024. Pattern-matching on mutable values: danger!. In *ACM Workshop on ML*.

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.

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.

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.

A Supporting More Features of OCaml

Records. OCaml's record types can mix immutable and mutable fields. We currently support only immutable records, which are viewed as values—they do not have an address. Extending our semantics and our program logics with support for mutable records should be straightforward. In Osiris, it should be possible to offer either just a points-to assertion for the whole record or a points-to assertion for each field independently. In the latter case, a points-to assertion for an immutable field can be persistent.

Pattern matching on mutable data. We currently allow pattern matching on immutable data only. This simplifies our semantics and lets us view pattern matching as a "pure" operation (§6). Although OCaml does allow pattern matching on mutable data, this feature has a few inherent pitfalls [Scherer et al. 2024]. In the near future, we do *not* intend to support it.

Recursive values. OCaml's **`let rec`** construct can be used not only to define recursive functions, but also to construct cyclic data structures. For example, **`let rec`** `xs = 0 :: xs` constructs an infinite immutable list. We do not and cannot support this feature. Our type of values, *val*, is an inductive type, and we very much want it to remain so. The existence of infinite immutable lists is arguably a dubious feature of OCaml: indeed, it implies that the function `List.length` does not always terminate. We prefer to forbid infinite lists and to let user prove that `List.length` terminates.

OCaml's **`let rec`** construct also allows constructing cyclic mutable data structures. This is typically used, for example, to allocate a doubly-linked list cell that points to itself. We do not currently support this feature, but might support it in the future.

Modules. Our semantics has preliminary support for OCaml's module language, including (nested) structures, **`open`**, and **`include`**. A module value is represented as a dictionary, that is, a finite map of field names to values. This semantics is fairly straightforward because both variable names and field names are represented as strings. The operation of annotating a module M with a module type T, which is known as *signature ascription*, is not yet supported. This operation must produce a new module M', which is obtained from M by removing all fields whose existence is *not* advertised by the module type T. To do so, it is necessary to first "resolve" the module type T, that is, to convert it to a signature, a finite map of field names to types. This resolution step can and must take place during the translation of OCaml to OLang: indeed, during this phase, the OCaml type-checker can be queried. Because functor application involves an implicit signature ascription, it, too, requires a signature resolution step.

Unsafe type casts. As an experimental feature, which must be explicitly enabled by the user, our system supports certain uses of OCaml's unsafe type cast operation, `Obj.magic`. Our approach is simple: during the translation of OCaml to OLang, `Obj.magic` is erased. In other words, in our semantics, a type cast is invisible, and has no effect. Thus, no reasoning rules for `Obj.magic` are needed, and none are provided. In this approach, the way in which we define the type of values, *val*, plays a crucial role. Suppose that an integer value, say *VInt* 1, is cast from type *int* to type *int ref*. The result of such a cast is still the value *VInt* 1. If this value is passed as an argument to an operation that expects a memory location, such as the dereferencing operation !, then (in our semantics) the program crashes, because ! expects the tag *VLoc*, whereas this value carries the tag *VInt*. On the other hand, if this value goes through a second type cast, from type *int ref* back to type *int*, and is then subjected to an operation that expects an integer value, such as addition, then all goes well. In summary, our semantics allows distinguishing between "good" and "bad" uses of `Obj.magic`: a type cast is "good" if and only if (in our semantics) it does not cause a crash.

Is this approach “sound”? A semantics is just a definition, so, per se, it cannot be “sound” or “unsound”. The true question is, do the real-world implementations of OCaml obey this semantics? In other words, are the real-world OCaml interpreters and compilers correct with respect to this semantics? Answering such a question is not easy. It seems to us that a fundamental property, *deterministic memory layout*, should play a key role. This property relates our definition of the type *val* with the memory layout used by a real-world implementation of OCaml. It can be informally stated as follows: for every value $v : val$, there must exist *at most one way of laying out the value v in memory*. In other words, it must be possible, just by looking at the value v, to predict how this value is represented in memory; no extra information, such as the type of this value, must be required. This said, it seems difficult to give a formal statement of this property, to prove that it actually holds,[8] or to clarify what role it might play in a proof of correctness of an interpreter or compiler. We leave these questions to future work.

Primitive types and operations. OCaml's primitive types include `int`, `char`, `string`, `float`, `'a array`, and several more. Our support for these primitive types and operations is currently very limited. Extending it should not be difficult in principle, but will require substantial effort.

Polymorphic primitive operations. OCaml offers several few built-in polymorphic operations, including structural equality =, physical equality ==, structural comparison `compare`, and marshalling and demarshalling. These operations have a few inherent pitfalls and can be misused. For example, structural equality, applied to a closure, fails at runtime. Furthermore, structural equality violates type abstraction: it can report that two “set” data structures differ even though they represent the same mathematical set. To avoid these problems, our semantics currently supports only certain uses of these operations. We currently allow physical equality to be applied only to memory locations (*VLoc*). We currently allow structural equality to be applied only to integer values (*VInt*) and tuples (*VTuple*). One might wish to generalize it to a larger subclass of values, including data constructors, but we are worried that this would be unsound.

Labeled and optional parameters. OCaml allows function parameters to carry a label. At a function application site, labeled arguments can be provided out of order. Furthermore, OCaml allows a function parameter to be marked optional. At a function application site, optional arguments can be omitted. The details of these features are complex; we currently do not support them. As part of the the Salto analyzer [Lermusiaux and Montagu 2024b], Lermusiaux and Montagu have implemented a transformation that eliminates labeled and optional parameters. In the future, we may take advantage of their work.

B Design Choices

This section contains additional material about our design choices (§2).

Environment-based semantics. In a substitution-based semantics, certain reduction steps involve replacing variables with values. This style is popular in the literature on type systems [Wright and Felleisen 1994] and program logics [Jung et al. 2018b, §6.1]. In an environment-based semantics, instead, an explicit map of variables to values is maintained: it is extended when a variable is bound

[8]In fact, as of today, we know of a few reasons why this property does *not* hold. For example, we model the OCaml value `A`, which is an application of a data constructor named `A` to 0 arguments, as *VData* “A” [ ]. However, if the program defines two algebraic data types `type foo = A | B` and `type bar = B | A`, then the OCaml compiler can represent the constructor `A` either as the integer 0 or as the integer 1, depending on the type at which this constructor is viewed. This is a problem: a type cast from type `foo` to type `bar` behaves differently in our semantics and in the real world. To fix this problem, we plan to use a more precise model where the OCaml value `A` is modeled in our semantics either as *VData* “foo.A” [ ] or as *VData* “bar.A” [ ], depending on its construction site.

```
Definition wrap_outcome η bs o :=
  match o with
  | Perform3 e k ⇒
      k ← install true k η bs ;
      ret (Perform3 e k)
  | _ ⇒
      ret o
  end.
```

Fig. 13. The auxiliary function *wrap_outcome*

and looked up when a variable is referenced. Our interpreter is parameterized with an environment, because this is the most natural most efficient thing to do. Thus, our semantics is environment-based.

In our program logics, we naturally use the same style. In every judgment, the environment and the code are distinct parameters. In Rocq, with some engineering effort, each of these two components can be displayed in a natural and lightweight manner. Via the `set` tactic, the structure of the environment (an association list whose domain is statically known) can be made part of the assumptions, instead of appearing in the goal. Because the code is never subject to a substitution, it is always a fragment of the source code that the user knows. The user can think of it as a *program point*, and it can be displayed as a textual fragment of concrete syntax.

Micro-level versus OCaml-level reasoning. We name our monad *micro* because it offers an intermediate language, or “microcode”, into which OCaml code is expanded. Indeed, our interpreter can be used as a compiler: by applying the interpreter to an OCaml 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 OCaml code into *micro* code and let the user perform program verification at the *micro* level (Figures 8 and 11). Thus, one would save the work of building a program logic for OCaml. We did experiment with this idea, but were unable to make it work to our satisfaction. To avoid an explosion in the size of the goal, one must carefully control the manner in which the application of the interpreter to the OCaml AST is reduced. Unfortunately, Rocq does not seem to offer sufficient control on its reduction strategy. Perhaps we could learn from successful approaches to partial evaluation in Rocq [Gross 2021; Gross et al. 2024]; perhaps other proof assistants, such as F*, would be better suited to this approach [Ho et al. 2023].

Chasing mistakes in the semantics. Is our semantics “correct”, and in what sense? Two questions arise. (A) Is it consistent with the ideal notion of OCaml that we have in mind? (B) Is it consistent with real-world implementations of OCaml? Our program logics, whose soundness with respect to the semantics has been verified, offer reasoning rules that correspond to our expectations. This suggests that the answer to question A is positive. Concerning question B, we do not yet have an answer. Large-scale testing seems necessary, but has not yet been carried out. Our interpreter can in principle be executed. Indeed, it can be extracted from Rocq to OCaml; there, it can be linked with a native (trusted) OCaml implementation of the *micro* monad. However, the fact that our semantics is non-deterministic creates an efficiency problem. We envision augmenting our interpreter with a heuristic oracle that selects one execution strategy among the many strategies that our semantics allows.

C A Monadic Interpreter

This section contains additional material about the monadic interpreter (§3).

C.1 Loops/Divergence

OCaml's **while** loops do not necessarily terminate. Yet, our host language, Rocq, allows writing terminating functions only. In particular, in the definition of *eval* η e, Rocq checks that every recursive call takes the form *eval* η' e' where e' is a strict subterm of e. This creates a difficulty in the interpretation of **while** loops: indeed, a naive definition of *eval* η (*EWhile e body*) seems to require a recursive call of the form *eval* η (*EWhile e body*). Our approach is to use the *please_eval* combinator. We use it in the interpretation of **while** loops:

```
| EWhile e body ⇒
    b ← as_bool (eval η e) ;
    if (b : bool) then
      _ ← eval η body ;
      please_eval η (EWhile e body)
    else
      ret VUnit
```

The dynamic check *as_bool* is analogous to *as_int* (§3.4). It checks that the value produced by the expression e is a Boolean value and extracts a Rocq Boolean b out of it. If b is true then we evaluate the loop body, discard its result, and evaluate the whole loop again, using *please_eval*. Otherwise we return the unit value *VUnit*.

C.2 Extensible Algebraic Data Types

OCaml has primitive support for extensible algebraic data types. An extensible algebraic data type grows at runtime: it initially has no constructors, but can be dynamically extended with new constructors. In OCaml's surface syntax, an extensible algebraic data type is created by **type** `t = ..` and is extended with a new constructor by **type** `t += A` **of** `int`. Such a constructor declaration is in fact a memory allocation instruction in disguise: indeed, its effect is to allocate a fresh memory location ℓ and to bind the name `A` to the value *VLoc* ℓ. Thus, the address ℓ, a dynamic name, is heap-allocated at runtime, whereas `A`, a static name, is treated like a variable: we bind `A` to *VLoc* ℓ by extending the environment, in the same way as we bind a variable to a value. This is done by the auxiliary function *type_extension* (right):

```
| EXData x es ⇒
    l ← as_loc (lookup η x) ;
    vs ← evals η es ;
    ret (VXData l vs)
```

```
Definition type_extension η (x : var) :=
  l ← alloc VUnit ;
  ret ((x, VLoc l) :: η).
```

In *eval* (left), to interpret an application of the data constructor x, we look up the name x in the environment, check that it is bound to a memory location ℓ, and use ℓ to identify this data constructor in the value *VXData* ℓ *vs*.

In OCaml's surface syntax, by looking at a data constructor application `A 3` in isolation, one cannot tell whether this expression constructs an ordinary algebraic data type (*EData*) or an extensible one (*EXData*). We rely on the OCaml type-checker to disambiguate this for us: our translator expects a typed abstract syntax tree as its input.

Runtime Assertions / Non-Deterministic Choice. When assertion checking is enabled, OCaml's runtime assertion construct **assert** `e` evaluates the expression `e` and checks that the result is a Boolean value b. If b is true, it returns a unit value; otherwise, it causes a failure, which we consider fatal. When assertion checking is disabled, the instruction **assert** `e` has no effect: it is erased. Assertion checking is enabled or disabled, at the level of a compilation unit, via compiler flags. Because we want our semantics (and program logics) to be independent of which compiler flags are used, we adopt a nondeterministic model in which each assertion independently may be either executed or erased. We achieve this thanks to the monadic combinator *choose* : *micro A E* ⟶ *micro A E* ⟶ *micro A E* (currently not shown in Figure 4). The computation *choose* m_1 m_2 picks one of m_1 and m_2 in a non-deterministic way and executes it.

```
| EAssert e ⇒
    choose
      (ret VUnit)                                                   (*  ignore runtime assertion *)
      (b ← as_bool (eval η e) ; if b then ret VUnit else crash) (* execute runtime assertion *)
| EAssertFalse ⇒
    crash
```

The construct **`assert false`** is treated in a special way by the OCaml compiler and by us. It is never erased and always fails.

D The Micro Monad

This section contains additional material about the *micro* monad (§4).

Even though *micro* is an inductive type, it is able to represent potentially divergent computations. There is no paradox or contradiction. As far as Rocq is concerned, every computation can be converted to a normal form, whose head constructor can be *Ret*, *Throw*, *Crash* (which mean that the computation is finished) or *Stop* (which means that the computation is paused at a system call), among other other possibilities. A "divergent" computation is a computation out of which there exists an infinite sequence of reduction steps, according to the small-step reduction relation that we define in the next section (§5).

The reader may feel that the architecture of our semantics is recursive in a strange way. On the one hand, the function *eval* produces a result of type *micro val exn*. So, the *micro* monad must exist before the function *eval* can be defined. On the other hand, the combinator *please_eval* is part of the public API of the *micro* monad, and the corresponding code, *CEval*, is part of the definition of the *micro* monad. So, as we define the *micro* monad, we must be already aware of the *future* existence of the function *eval*. Also, the types *env*, *expr*, *val* and *exn*, which appear in the type of *CEval*, must exist before the *micro* monad is defined. Fortunately, no mutual recursion is required. The definitions of the types *val* and *exn* do not refer to *micro*. The definition of *micro* does not refer to *eval*. The connection between the code *CEval* and the function *eval* is established *a posteriori* by the small-step operational semantics (§5). Our semantics includes other examples of this phenomenon: for instance, the right-hand side of the reduction rule for the system call *CWrap* refers to the function *eval_branches* (Figure 7).

System calls that raise exceptions. In every line in Figure 6, the parameter E is instantiated with *exn*, the type of OCaml exceptions. Indeed, the system calls *CEval*, *CPerf*, and *CResume* can produce an exception. *CEval* evaluates an OCaml expression: this can produce an exception. *CPerf* performs a control effect and captures a continuation: if this continuation is later discontinued then this system call appears to produce an exception. *CResume* resumes a captured continuation: in other words, it resumes the evaluation of an OCaml expression. This can produce an exception. The remaining system calls, namely *CAlloc*, *CLoad*, *CStore*, and *CWrap*, cannot produce exceptions. Assigning them the type *exn* is a convenient over-approximation.

E Horus

E.1 Micro-level rules

Rule PURE-CHOOSE is used for **`assert`**.

$$\text{PURE-CHOOSE}\quad \frac{\mathit{pure}\ m_1\ \varphi\ \psi \qquad \mathit{pure}\ m_2\ \varphi\ \psi}{\mathit{pure}\ (\mathit{choose}\, m_1\, m_2)\ \varphi\ \psi}$$

E.2 Judgments

We have a judgment for each syntactic category:

$$\begin{aligned}
\mathit{pure}_{\#}\ m\ \varphi\ \psi &:= \mathit{pure}\ m\ (\lambda v.\, \exists a.\, v = \#a \land \varphi\ a)\ \psi \\
\mathit{expr}\ \eta\ e\ \varphi\ \psi &:= \mathit{pure}_{\#}\ (\mathit{eval\_expr}\ \eta\ e)\ \varphi\ \psi \\
\mathit{branches}\ \eta\ o\ \mathit{bs}\ \varphi\ \psi &:= \mathit{pure}_{\#}\ (\mathit{eval\_branches}\ \eta\ o\ \mathit{bs})\ \varphi\ \psi \\
\mathit{bindings}\ \eta\ \mathit{bs}\ \varphi\ \psi &:= \mathit{pure}\ (\mathit{eval\_bindings}\ \eta\ \mathit{bs})\ \varphi\ \psi \\
\mathit{pat}\ \eta\ \delta\ p\ v\ \varphi\ \zeta &:= \mathit{pure}\ (\mathit{eval\_pat}\ \eta\ \delta\ p\ v)\ \varphi\ (\lambda ().\, \zeta) \\
\mathit{cpat}\ \eta\ \delta\ \mathit{cp}\ o\ \varphi\ \zeta &:= \mathit{pure}\ (\mathit{eval\_cpat}\ \eta\ \delta\ \mathit{cp}\ o)\ \varphi\ (\lambda ().\, \zeta) \\
\mathit{pats}\ \eta\ \delta\ \mathit{ps}\ \mathit{vs}\ \varphi\ \zeta &:= \mathit{pure}\ (\mathit{eval\_pats}\ \eta\ \delta\ \mathit{ps}\ \mathit{vs})\ \varphi\ (\lambda ().\, \zeta)
\end{aligned}$$

We have not given the definition of *eval_bindings*, in fact even the definition of *bindings* is not needed for the purpose of the program logic. What is important is that *bindings* is introduced when applying ELet, and that it can be established using BindingsNil and BindingsCons.

E.3 Pattern matching

We introduce the *pat* judgment for reasoning over pattern matching. The judgment $\mathit{pat}\ \eta\ \delta\ p\ v\ \varphi\ \zeta$ states that, in the environment η, matching the pattern p against the value v is safe, and either results in extending the environment δ into one that satisfies φ, or fails by throwing a non-fatal meta-level exception (*throw* ()) and guaranteeing ζ. It is defined over the auxiliary function *eval_pat*, which syntactically matches patterns against values.

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

We similarly define the judgments *cpat* and *pats* (over *eval_cpat* and *eval_pats*). We omit the definition of *eval_pat*, and instead give syntactic reasoning rules over *pat* in Figure 14.

As an appetizer, consider the rules for matching a wildcard pattern pat-PAny, or a value pattern pat-PVar. Because these matches always succeed, the failure postcondition ζ can be anything (in particular, it can be *false*). The difference between the two rules is whether or not any bindings get added to the environment.

An or-pattern fails only if both sub-patterns fail, so its failure postcondition is a *conjunction* in pat-POr. In the second premise of pat-POr, we can assume the failure condition of the first premise since in OCaml the second branch of an or-pattern is considered only if the first fails.

Matching on n-ary tuples and on variant values requires matching on their components, which is the role of the judgment *pats* in the hypotheses of pat-PTuple and pat-P(X)Data. For *pats* judgments the success postcondition is the composition of all corresponding *pat* judgments; the failure postcondition is the corresponding disjunction, since a mismatch for any of the argument is a mismatch for the compound (pats-nil, pats-cons).

$$\textsc{pat-PAny}\quad \frac{\varphi\ \delta}{pat\ \eta\ \delta\ PAny\ v\ \varphi\ \zeta} \qquad \textsc{pat-PVar}\quad \frac{\varphi\ ((x,v) :: \delta)}{pat\ \eta\ \delta\ (PVar\ x)\ v\ \varphi\ \zeta} \qquad \textsc{pat-PAlias}\quad \frac{pat\ \eta\ \delta\ p\ v\ (\lambda\delta.\ \varphi\ ((x,v) :: \delta))\ \zeta}{pat\ \eta\ \delta\ (PAlias\ p\ x)\ v\ \varphi\ \zeta}$$

$$\textsc{pat-POr}\quad \frac{pat\ \eta\ \delta\ p_1\ v\ \varphi\ \zeta_1 \qquad (\zeta_1 \Rightarrow pat\ \eta\ \delta\ p_2\ v\ \varphi\ \zeta_2)}{pat\ \eta\ \delta\ (POr\ p_1\ p_2)\ v\ \varphi\ (\zeta_1 \wedge \zeta_2)} \qquad \textsc{pat-PTuple}\quad \frac{pats\ \eta\ \delta\ ps\ vs\ \varphi\ \zeta}{pat\ \eta\ \delta\ (PTuple\ ps)\ (VTuple\ vs)\ \varphi\ \zeta}$$

$$\textsc{pat-PData}\quad \frac{c = c' \Rightarrow pats\ \eta\ \delta\ ps\ vs\ \varphi\ \zeta}{pat\ \eta\ \delta\ (PData\ c\ ps)\ (VData\ c'\ vs)\ \varphi\ (\zeta \vee c \neq c')}$$

$$\textsc{pat-PXData}\quad \frac{lookup\ \eta\ x = VLoc\ \ell' \qquad (\ell = \ell' \Rightarrow pats\ \eta\ \delta\ ps\ vs\ \varphi\ \zeta)}{pat\ \eta\ \delta\ (PXData\ x\ ps)\ (VXData\ \ell\ vs)\ \varphi\ (\zeta \vee \ell \neq \ell')}$$

Fig. 14. Syntactic rules for *pat*

$$\textsc{pats-nil}\quad \frac{\varphi\ \eta}{pats\ \eta\ \delta\ []\ []\ \varphi\ \zeta} \qquad \textsc{pats-cons}\quad \frac{pat\ \eta\ \delta\ p\ v\ (\lambda\delta.\ pats\ \eta\ \delta\ ps\ vs\ \varphi\ \zeta_2)\ \zeta_1}{pats\ \eta\ \delta\ (p :: ps)\ (v :: vs)\ \varphi\ \ (\zeta_1 \vee \zeta_2)}$$

An operational reading of pat's reasoning rules. In proofs, we expect the occurrence of the pattern p to be concrete, while often the value v will be abstract. When matching a single pattern p and a value v, the repeated application of the *pat* rules has the effect of translating the pattern matching operation into a positive formula which has to be proven, and a negative formula. The positive formula accumulates a sequence of quantifiers and equations that describe what is learned when pattern matching succeeds. The negative formula describes what is learned when pattern matching fails. For example, consider the following derived rule for matching on a list's `(::)` constructor.

$$\frac{\forall h\ t.\ v = h :: t \Rightarrow pat\ \eta\ \delta\ p\ h\ (\lambda\delta'.\ pat\ \eta\ \delta'\ ps\ t\ \varphi\ (\psi_2\ t))\ (\psi_1\ h)}{pat\ \eta\ \delta\ (PData\ \ ::\ \ [p, ps])\ v\ \varphi\ (v = [] \vee \exists h\ t.\ l = h :: t \wedge (\psi_1\ h \vee \psi_2\ t))}$$

The premise of the rule becomes the positive formula, where we universally quantify over the head and tail of the list. The negative formula is found in the failure postcondition of the conclusion of the rule. In this case the negative formula reads as "the pattern fails if the list is empty, or if the list is a cons and either the head fails to match or the tail fails to match". We can derive such rules for any variant type, and automate the application of the syntactic rules over *pat*, making it so the user never has to reason over the *pat* and associated judgments directly. The user only has to prove the instantiate positive formula, and gets to use the negative formula in the following branches of a pattern match.

E.4 Support for expressions

Missing from the main text is the rule for assertions EAssert that does not show any trace of their nondeterministic semantics, and the rules for *bindings*, the auxiliary judgment used for rule ELet.

EAssert

$$\frac{expr\ \eta\ e\ (\lambda b.\, b = true)\ \psi}{expr\ \eta\ (EAssert\ e)\ (\lambda\_.\, \mathrm{True})\ \psi}$$

BindingsNil

$$\frac{\varphi\ []}{bindings\ \eta\ []\ \varphi\ \psi}$$

BindingsCons

$$\frac{\begin{array}{c} expr\ \eta\ e\ (pat\ \eta\ []\ p\ \#a\ \varphi_1\ \bot)\ \psi \\ bindings\ \eta\ bs\ \varphi_2\ \psi \\ \forall \delta\ \eta'.\ \varphi_1\ \delta \Rightarrow \varphi_2\ \eta' \Rightarrow \varphi\ (\delta +\!\!+\ \eta') \end{array}}{bindings\ \eta\ (Binding\ p\ e :: bs)\ \varphi\ \psi}$$

F Osiris

$$\begin{array}{ll}
\text{(IMP1)} & \langle\Psi\rangle\ impure\ (ret\, v)\ \varphi\ \psi \triangleq {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ \varphi(v) \\
\text{(IMP2)} & \langle\Psi\rangle\ impure\ (throw\ e)\ \varphi\ \psi \triangleq {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ \psi(e) \\
\text{(IMP3)} & \langle\Psi\rangle\ impure\ (crash)\ \varphi\ \psi \triangleq \bot \\
\text{(IMP4)} & \langle\Psi\rangle\ impure\ (Stop\ CPerf\ v\, k)\ \varphi\ \psi \triangleq {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ \Psi\ \text{allows do}\ \ v\ \{w.\ \triangleright\ \langle\Psi\rangle\ impure\ (k\ w)\ \varphi\ \psi\} \\
\text{(IMP5)} & \langle\Psi\rangle\ impure\ e\ \varphi\ \psi \triangleq \forall\sigma.\, \mathcal{S}(\sigma)\, {}^{\mathcal{E}}{\Rrightarrow\!\ast}^{\emptyset} \\
 & \quad \forall\sigma', e'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ {}^{\emptyset}{\Rrightarrow\!\ast}^{\emptyset}\ \triangleright\ {}^{\emptyset}{\Rrightarrow}^{\mathcal{E}}\ \mathcal{S}(\sigma') \ast \langle\Psi\rangle\ impure\ e'\ \varphi\ \psi\} \\
 & \quad \textit{(e not an outcome nor an effect)}
\end{array}$$

Fig. 15. Definition of $\langle\Psi\rangle\ impure\ e\ \varphi\ \psi$

impure-alloc

$$\frac{\forall \ell, \ell \mapsto v \mathrel{-\!\!\ast} \varphi(\ell)}{\langle\Psi\rangle\ impure\ (alloc\ v)\ \varphi\ \psi}$$

impure-store

$$\frac{\begin{array}{c} \ell \mapsto v \\ \triangleright(\ell \mapsto v' \mathrel{-\!\!\ast} \varphi(())) \end{array}}{\langle\Psi\rangle\ impure\ (store\ (\ell, v'))\ \varphi\ \psi}$$

impure-load

$$\frac{\begin{array}{c} \ell \mapsto v \\ \triangleright(\ell \mapsto v \mathrel{-\!\!\ast} \varphi(v)) \end{array}}{\langle\Psi\rangle\ impure\ (load\ \ell)\ \varphi\ \psi}$$

impure-ERef

$$\frac{\begin{array}{c} \langle\Psi\rangle\ expr\ \eta\ e\ \varphi'\ \psi \\ \forall v.\ \varphi'(v) \mathrel{-\!\!\ast} \forall \ell.\ \ell \mapsto v \mathrel{-\!\!\ast} \varphi(\ell) \end{array}}{\langle\Psi\rangle\ expr\ \eta\ (ERef\ e)\ \varphi\ \psi}$$

impure-ELoad

$$\frac{\begin{array}{c} \langle\Psi\rangle\ expr\ \eta\ e\ \varphi'\ \psi \\ \forall \ell.\ \varphi'(\ell) \mathrel{-\!\!\ast} \exists q, v.\ \ell \mapsto_q v \ast \triangleright(\ell \mapsto_q v \mathrel{-\!\!\ast} \varphi(v)) \end{array}}{\langle\Psi\rangle\ expr\ \eta\ (ELoad\ e)\ \varphi\ \psi}$$

impure-EStore

$$\frac{\begin{array}{c} \langle\Psi\rangle\ expr\ \eta\ e_1\ \varphi_1\ \psi \\ \langle\Psi\rangle\ expr\ \eta\ e_2\ \varphi_2\ \psi \\ \forall \ell_1, v_2.\ \varphi_1(\ell_1) \ast \varphi_2(v_2) \mathrel{-\!\!\ast} \exists v_1.\ \ell_1 \mapsto v_1 \ast \triangleright(\ell_1 \mapsto v_2 \mathrel{-\!\!\ast} \varphi(())) \end{array}}{\langle\Psi\rangle\ expr\ \eta\ (EStore\ e_1\ e_2)\ \varphi\ \psi}$$

Fig. 16. Osiris memory-related rules

impure-wrap

$$\frac{\begin{array}{c} k' = \lambda o.\ handle\ (resume\ \ell\ o)\ (wrap\_eval\_branches\ \eta\ bs) \\ \forall \ell'.\ \ell' \mapsto k' \mathrel{-\!\!\ast}\ \triangleright\ \varphi\ \ell' \end{array}}{\langle\Psi\rangle\ impure\ (wrap\ \ell\ \eta\ bs)\ \varphi\ \psi}$$

Fig. 17. Osiris *micro*-layer reasoning rule for *wrap*

$$
\begin{array}{l}
\textit{shallow-handler}\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ h\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\} \triangleq \\
\quad (\forall o.\ \varphi\ o \mathrel{-\!\!*} \triangleright\langle\Psi'\rangle\ \textit{impure}\ (h\ o)\ \varphi'\ \psi') \land \\
\quad (\forall v, k.\ \Psi \text{ allows do } v\ \{ \\
\qquad \lambda o.\ \forall \Psi'', \varphi'', \psi''. \\
\qquad\quad \triangleright\ \langle\Psi''\rangle\ \textit{impure}\ (\textit{resume}\ k\ o)\ \varphi''\ \psi'' \\
\qquad\quad \} \mathrel{-\!\!*} \\
\qquad \triangleright\ \langle\Psi'\rangle\ \textit{impure}\ (h\ (\textit{Perform}_3\ v\ k))\ \varphi'\ \psi')
\end{array}
$$

$$
\begin{array}{l}
\textit{olang-deep-handler}\ \eta\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ bs\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\} \triangleq \\
\qquad \textit{deep-handler}\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ (\lambda o,\ \mathsf{eval\_branches}\ \eta\ o\ bs)\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\} \\
\textit{deep-handler}\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ h\ \langle\Psi'\rangle\ \{\varphi' \mid \psi'\} \triangleq \\
\quad (\forall o.\ \varphi\ o \mathrel{-\!\!*} \triangleright\langle\Psi'\rangle\ \textit{impure}\ (h\ o)\ \varphi'\ \psi') \land \\
\quad (\forall v, k.\ \Psi \text{ allows do } v\ \{ \\
\qquad \lambda o.\ \forall \Psi'', \varphi'', \psi''. \\
\qquad\quad \triangleright\ \textit{deep-handler}\ \langle\Psi\rangle\ \{\varphi \mid \psi\}\ h\ \langle\Psi''\rangle\ \{\varphi'' \mid \psi''\} \mathrel{-\!\!*} \\
\qquad\quad \langle\Psi''\rangle\ \textit{impure}\ (\textit{resume}\ k\ o)\ \varphi''\ \psi'' \\
\qquad\quad \} \mathrel{-\!\!*} \\
\qquad \triangleright\ \langle\Psi'\rangle\ \textit{impure}\ (h\ (\textit{Perform}_3\ v\ k))\ \varphi'\ \psi')
\end{array}
$$

Fig. 18. Definition of the *olang-deep-handler* and *shallow-handler* predicate