VERIFYING A MINIMALIST REVERSE-MODE AD LIBRARY

PAULO EMÍLIO DE VILHENA AND FRANÇOIS POTTIER

Inria, France
e-mail address: paulo-emilio.de-vilhena@inria.fr

Inria, France
e-mail address: francois.pottier@inria.fr

Abstract. By exploiting a number of relatively subtle programming language features, including dynamically-allocated mutable state, first-class functions, and effect handlers, reverse-mode automatic differentiation can be implemented as a library. One outstanding question, however, is: with which logical tools can one specify what this code is expected to compute and verify that it behaves as expected? We answer this question by using a modern variant of Separation Logic to specify and verify a minimalist (but concise and elegant) reverse-mode automatic differentiation library. We view this result as an advanced exercise in program verification, with potential future applications to more realistic automatic differentiation systems.

Introduction

Automatic differentiation (AD) is an important family of algorithms and techniques whose aim is to allow the efficient and exact evaluation of the derivative of a function that is defined programmatically. As very well put by the authors of the Wikipedia entry on the topic, *"automatic differentiation exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, multiplication, etc.). By applying the chain rule, derivatives can be computed automatically, accurately, and using at most a small constant factor more arithmetic operations than the original program."*

The family of automatic differentiation techniques can be coarsely divided into *forward-mode* and *reverse-mode* approaches.

Implementing forward-mode automatic differentiation as a library is fairly easy: it is a matter of replacing the ordinary arithmetic operations with the corresponding operations on *dual numbers*, that is, pairs of a value and a tangent. In many programming languages, overloading can be used to make this replacement transparent in the eyes of the end user.

Can reverse-mode automatic differentiation (RMAD) be implemented under a similar guise, that is, in the form of a library that provides replacements for the usual arithmetic operations? The answer is positive: this is also possible, albeit more challenging. Wang and Rompf [WR18] and Wang et al. [WZD+19] show that RMAD can be implemented as a

Key words and phrases: automatic differentiation, separation logic, effect handlers, program verification.

Preprint submitted to
Logical Methods in Computer Science

library, in a strikingly concise way, provided the programming language is equipped with dynamically-allocated mutable state and with some form of *delimited control*, such as Danvy and Filinski's `shift` and `reset` operators [DF90], which are available in Scala [RMO09]. Sivaramakrishnan [Siv18] and Sigal [Sig21] adapt Wang et al.'s approach to use *effect handlers* [Pre15], a more structured form of delimited control.

Wang and Rompf [WR18] make a particularly striking and provoking claim about their code: *"Our implementation is so concise that it can serve as a specification of reverse mode AD"*. While we agree that this code is indeed extremely concise, clearly structured, and offers an excellent basis for understanding and teaching RMAD, we nevertheless object that this code involves nontrivial programming language features, including dynamically-allocated mutable state and delimited control. Therefore, it is not entirely obvious how and why it works. We argue that this code is a concise and elegant *implementation* and that it deserves to be verified with respect to the simplest possible *specification*, which is: "this code computes a derivative".

This gives rise to a program verification challenge, which is *to verify that Wang and Rompf's implementation*, or a similar implementation expressed in some other programming language, *is a correct implementation of differentiation*. More specifically, because this code forms a library that is intended to be exploited in a larger application, it is reasonable to ask that the specification and verification effort be carried out in the setting of a *compositional program logic*, so that the library can be verified once and re-used as a building block in many verified applications.

To the best of our knowledge, no result of this kind has appeared in the literature. Wang et al. [WZD+19] publish detailed descriptions of several versions of their code, but no proof. The challenge seems nontrivial: indeed, one must somehow express and prove the fact that the use of mutable state and delimited control is correctly encapsulated and is invisible to a user of the library.

In this paper, we address this challenge. We transcribe Wang and Rompf's simple implementation of RMAD in *HH*, a core λ-calculus equipped with effect handlers. Then, using Iris [JKJ+18], a powerful evolution of Separation Logic, which in previous work [dVP21a] we have extended with support for effect handlers, we express a specification for an automatic differentiation algorithm and we verify that this code is correct with respect to this specification. Our proof is machine-checked and is available online [dVP21b].

For greater readability, the code that we present in the paper is expressed in a real-life programming language, namely Multicore OCaml 4.12.0 [DMS20], an experimental extension of OCaml with effect handlers. The code that we verify, however, is expressed in *HH*, a calculus whose syntax and semantics are formally defined inside the Coq proof assistant. Our *HH* code is not shown in the paper, but is available online [dVP21b], together with a short description of the correspondence between the Coq definitions and the paper [dVP21c]. We discuss the gap between Multicore OCaml and *HH* at the end of the paper (§7).

Although the code that we verify is arguably a toy implementation of RMAD, Wang et al. [WZD+19] show that its basic architecture is sound and (in combination with other techniques, such as staging) can offer competitive performance.

The paper is organized as follows. We present an API that a minimalistic AD library might offer to an end user (§1). We briefly introduce effect handlers (§2) and explain how they are exploited in the implementation of RMAD (§3). Then, we present the "first layer" of mathematical reasoning (§4). Finally, we present the "second layer" of Separation Logic reasoning (§5), where the specification and the proof of the code are given.

```
(* A record of the ring operations over a numeric type 'v. *)
type 'v num =
  { zero : 'v; one : 'v; add : 'v -> 'v -> 'v; mul : 'v -> 'v -> 'v }

(* An expression of one variable in tagless final style. *)
type exp =
  { eval : (* forall *) 'v. 'v num -> 'v -> 'v }

(* The automatic differentiation algorithm. *)
val diff : exp -> exp
```

Figure 1: API of a minimalistic automatic differentiation library

```
(* The expression (x+1)^3. *)
let e : exp =
  { eval =
      fun (type v) ({ zero; one; add; mul } : v num) x ->
        let ( + ), ( * ) = add, mul in
        let cube x = x * x * x in
        cube (x + one)
  }

(* Its derivative. *)
let e' : exp = diff f

(* The ring of the floating point numbers. *)
let float = { zero = 0.0; one = 1.0; add = ( +. ); mul = ( *. ) }

(* Evaluating e' in floating point at 4.0. *)
let () = assert (e'.eval float 4.0 = 75.0)
```

Figure 2: An example use of the library

For the sake of brevity, we omit introductions to AD, effect handlers, and Separation Logic. We attempt to recall and explain the key notions as they appear. For a comprehensive introduction to AD, the reader is referred to Griewank and Walther's book [GW08]. A tutorial introduction to effect handlers is given by Pretnar [Pre15]. The seminal paper by Reynolds [Rey02] offers a tutorial introduction to Separation Logic.

1. Automatic Differentiation as a Library

In this paper, we wish to present a very simple implementation of reverse-mode automatic differentiation and to verify its correctness. Before showing the code, let us give the interface (the API) that this library presents to an end user and present an example of its use.

This interface appears in Figure 1. It begins with the definitions of two types, `'v num` and `exp`. Then, the differentiation algorithm is presented to the user as a function, `diff`,

whose type is `exp -> exp`. Thus, differentiation transforms an expression into an expression. There remains to answer the question: what is an expression?

In this minimalistic library, we wish to handle expressions that are constructed out of the constants 0 and 1, addition, multiplication, and a single variable.[1] Now, how should expressions be represented? A natural idea might be to represent them as abstract syntax trees, which one would do by declaring `exp` as an algebraic data type. However, such an approach would not allow us to present `diff` as a function of type `exp -> exp`. Indeed, automatic differentiation is not symbolic differentiation: it does not expect a symbolic expression as an input, and does not construct a syntactic representation of the derivative. Here, a well-known alternative representation of expressions is better suited, namely the Church-Böhm-Berarducci encoding [Kis12], also known as the "tagless final" representation [CKS09, Kis10]. In this approach, *an expression is a computation*: provided it is given access to the four operations (zero, one, addition, multiplication) and to the value of the single variable, it produces a value. This is visible in Figure 1, where an expression is represented as a function of type `'v num -> 'v -> 'v` (line 7). The first argument, of type `'v num`, is a dictionary, that is, a record of four fields, containing the implementations of the four operations. The second argument, of type `'v`, is the value of the variable. The result, also of type `'v`, is the value of the expression. In fact, *an expression is a polymorphic computation*: indeed, we require it to be polymorphic in the type `'v`.[2] This allows an expression to be "evaluated" in many different ways: by applying it to suitable arguments, one can for instance evaluate it using integer arithmetic, evaluate it using floating-point arithmetic, or even convert it to a symbolic representation.

This representation of expressions is sufficiently flexible for reverse-mode automatic differentiation to be expressed as a function `diff` of type `exp -> exp`. Even though neither the algorithm nor its implementation using effect handlers is new, such a formulation of `diff` as a transformation of expressions in "tagless final" style seems new. Our implementation of `diff` (§3) evaluates the expression that one wishes to differentiate under a nonstandard implementation of the four operations, where addition and multiplication perform control effects. Thus, although this is not visible in the definition of the type `exp`,[3] our implementation exploits the fact that *an expression must be effect-polymorphic*: that is, it must be insensitive to the effects performed by the operations `add` and `mul`. This idea is explicitly spelled out in §5.2, where we propose a formal specification of `diff` in Separation Logic.

An example of the use of the library appears in Figure 2. We first build a representation `e` of the mathematical expression $(x + 1)^3$. This part is unfortunately rather verbose. Line 4 introduces the type parameter `v` and the five value parameters `zero`, `one`, `add`, `mul`, and `x`. Line 5 redefines the infix operators `+` and `*` as aliases for `add` and `mul`; unfortunately, OCaml does not allow similarly redefining `0` and `1` as aliases for `zero` and `one`. The "meat" of the definition of `e` is at lines 6–7. Then, we apply `diff` to `e` so as to obtain a representation `e'` of the derivative of `e` with respect to the variable x (line 11). This function call terminates

[1] For the sake of simplicity, we restrict our attention to expressions of one variable. Generalizing the algorithm and its proof so as to handle expressions of several variables should be straightforward, but would add a certain amount of clutter, which we prefer to avoid.

[2] Technically, the type `exp` is defined as a record with one field, named `eval`, which contains a polymorphic function: for every type `'v`, this function must have type `'v num -> 'v -> 'v`.

[3] In Multicore OCaml, at present, the type system does not keep track of the effects that a function might perform. In other words, function types are not annotated with effect information. Any function can in principle perform any effect. An unhandled effect causes a runtime error.

immediately: indeed, because an expression is represented as a computation, this call simply allocates and returns a closure. The reverse-mode automatic differentiation algorithm is actually executed only at line 17, where we request the evaluation of the expression `e'` using floating-point arithmetic. There, because we expect `e'` to be equivalent to the expression $3(x+1)^2$, and because we instantiate x with the value 4, we expect the result to be 75. Executing the code validates this expectation: the runtime assertion at line 17 succeeds.

2. Effects and Handlers

2.1. **Effect handling.** Effect handling can be understood as a generalization of exception handling, a familiar feature of many high-level programming languages, including Lisp, CLU [LS79], Ada, Modula-3, C++, Standard ML, OCaml, Java, and many more. Exception handling allows the execution of a computation to be monitored by a *handler*. The computation may at any time decide to *throw an exception*. In such an event, the computation is interrupted and the handler takes control. In the early days of exception handling, it was debated whether a computation that throws an exception should be terminated or possibly resumed after the handler has run. A consensus emerged in favor of the first option, the "termination model", because it was perceived to be easier to reason about and easier to implement efficiently than the second option, the "resumption model". Ryder and Soffa [RS03] offer a historical account.

Like exception handling, effect handling involves the interplay of a computation and a handler. At any time, the computation can interrupt itself by *performing an effect*. Control is then transferred to the handler. As a crucial new feature, the handler receives a first-class function, also known as a *delimited continuation*,[4] which represents the suspended computation: invoking this function resumes the computation. If and when the computation is resumed, another instance of the handler is installed, so the dialogue continues: the computation can perform another effect, causing control to be again transferred to the handler, and so on.

A continuation is an ordinary function, which an effect handler can use in a variety of ways. If the continuation is not invoked at all, then the computation is stopped. If the continuation is invoked at the end of the handler, then the computation is resumed after the handler has run. If the continuation is invoked somewhere in the middle of the handler, then part of the handler runs *before the computation is resumed* and part of it runs *after the computation has finished*. The example that we present shortly, as well as the reverse-mode automatic differentiation algorithm, exploit this pattern.

Yet other uses of the continuation can be imagined. In some applications, the continuation is not invoked by the handler, but is returned by the handler or stored by the handler in memory for use at a later time. In other applications, such as backtracking search, a continuation is invoked several times. This means that a computation that suspends itself *once* can be resumed *more than once*. This use of continuations is powerful, but requires care: it breaks the property that "a block of code, once entered, is exited at most once", and thereby compromises the frame rule [dVP21a], one of the most fundamental reasoning rules of Separation Logic. Both Multicore OCaml and our reasoning rules [dVP21a] require

[4]The literature offers a wide variety of delimited control operators, that is, operators that allow capturing delimited continuations [Fel88, DF90, Sit93]. Effect handling is equivalent in expressive power to many of these operators [FKLP19].

```
1  open Printf
2  effect Ask : int -> int
3  let ask x = perform (Ask x)
4  let handle (client : unit -> int) =
5    match client() with
6    | effect (Ask x) k ->
7        let y = x + 1 in
8        printf "I am queried at %d and I am replying %d.\n" x y;
9        continue k y;
10       printf "Earlier, I have been queried at %d and I have replied %d.\n" x y
11   | result ->
12       printf "The computation is finished and returns %d.\n" result
13 let () =
14   handle (fun () -> ask 2 + ask 7)
```

Figure 3: A simple demonstration of effect handlers

```
I am queried at 7 and I am replying 8.
I am queried at 2 and I am replying 3.
The computation is finished and returns 11.
Earlier, I have been queried at 2 and I have replied 3.
Earlier, I have been queried at 7 and I have replied 8.
```

Figure 4: Output of the program in Figure 3

that a continuation be invoked at most once. In the reverse-mode automatic differentiation algorithm, every continuation is invoked exactly once.

Effect handlers are found in several research programming languages, such as Eff [BP15, BP20], Effekt [BSO20a], Frank [LMM17], Koka [Lei14, Lei20], Links [HLA20], and Multicore OCaml [DEH+17, DMS20]. They have also been implemented as a library in mainstream programming languages such as Scala [BSO20b].

2.2. **Example.** We now illustrate effect handlers via a simple example. Although this example may seem somewhat artificial, it deserves to be understood, as it exploits effect handling exactly in the same way as the reverse-mode automatic differentiation algorithm that we wish to study.

Multicore OCaml offers three basic constructs for effect handling. **`perform`** `v` performs an effect: execution is interrupted and control is transferred to the nearest enclosing handler, which receives the value `v` and a continuation `k`. **`continue`** `k w` invokes the continuation `k`: the suspended computation is resumed, just as if **`perform`** `v` had returned the value `w`. Finally, the **`match`** construct wraps a computation in a handler.

The example in Figure 3 exploits all of these constructs in combination. It is a complete Multicore OCaml program, whose output appears in Figure 4.

In line 2, the effect `Ask` is declared, with signature `int -> int`. This means that the expression **`perform`** `(Ask x)` requires `x` to have type `int` and has type `int`. In line 3, `ask x` is defined as a shorthand for **`perform`** `(Ask x)`. Thus, the function `ask` has type `int -> int`.

The function `handle` at line 4 executes the computation `client()` under a handler for the effect `Ask`. The handler takes the form of a **`match`** construct whose first branch (line 6) takes control when the computation performs the effect `Ask` and whose second branch (line 11) takes control when the computation finishes.

The **`effect`** branch at line 6 specifies how the handler behaves when the client performs an effect. The handler receives the integer value `x` that was passed as an argument to `ask` as well as a continuation `k`. It defines `y` as `x+1` and applies the continuation `k` to `y` (line 9) between two `printf` statements. Thus, the first `printf` statement is executed before the suspended computation is resumed, whereas the second `printf` statement takes effect only after the suspended computation terminates. It is crucial to remark that the execution of **`continue`** `k y` may involve further effects and their handling.

The second branch, at line 11, specifies what to do after the client terminates normally and returns a value, `result`. The `printf` statement at line 12 displays this value.

We are now in a position to understand why the function application `handle (fun () -> ask 2 + ask 7)` at line 14 produces the output shown in Figure 4. Multicore OCaml happens to follow a right-to-left evaluation order, so the function call `ask 7` takes place first, causing control to be transferred to the handler, with `x` bound to `7`. The handler immediately prints "`I am queried at 7...`", then resumes the client by calling **`continue`** `k 8` at line 9. (The `printf` statement at line 10, whose effect is to print "`Earlier, I have been queried at 7...`", is delayed until this call returns.) The client then resumes its work and reaches the function call `ask 2`, again causing control to be transferred to the handler. This is in fact a new instance of the handler, where this time `x` is bound to `2`. This second effect is handled like the previous one. The handler immediately prints "`I am queried at 2...`", then resumes the client by calling **`continue`** `k 3`. (The `printf` statement at line 10, whose effect is to print "`Earlier, I have been queried at 2...`", is delayed until this call returns.) The client now computes `3 + 8` and terminates with the value `11`. The termination of the client is handled by a third and last instance of the handler: there, control reaches the `printf` statement at line 12, producing the third line of output. This third instance of the handler is then finished and disappears. The previous two instances of the handler, whose activation records still exist on the control stack, are allowed to complete their execution. Thus, the two `printf` statements that were delayed earlier are allowed to take effect. Naturally, the most recent handler instance is allowed to complete first: this explains the order in which the last two lines of output appear.

In summary, the execution of this code is divided in two phases. During the first phase, which lasts as long as the client runs, the client performs a sequence of effects, and the `printf` statements at line 8 are executed in order, while the `printf` statements at line 10 are accumulated on the control stack. During the second phase, which begins when the client terminates, the `printf` statements that have been delayed are popped off the control stack and are executed, in reverse order. Using Danvy and Goldberg's [DG05] terminology: the first phase occurs at *call time* whereas the second phase occurs at *return time*.

3. Implementing Reverse-Mode AD with Effect Handlers

We now propose an implementation of the API that was presented earlier (§1). This code is based on Sivaramakrishnan's implementation [Siv18], which itself was inspired by Wang et al.'s work [WR18, WZD+19]. The code appears in Figure 5. At this point, we give only an

```
type 'v num = { zero : 'v; one : 'v; add : 'v -> 'v -> 'v; mul : 'v -> 'v -> 'v }
type exp    = { eval : (* forall *) 'v. 'v num -> 'v -> 'v }

let diff (e : exp) : exp = { eval =
  fun (type v) ({ zero; one; add; mul } : v num) (n : v) ->
    let ( + ), ( * ) = add, mul in
    let open struct

      type t = O | I | Var of {v : v ; mutable d : v} (* zero | one | var *)
      effect Add : t * t -> t
      effect Mul : t * t -> t

      let mk n       = Var {v = n; d = zero}
      let get_v u    = match u with O -> zero | I -> one  | Var u     -> u.v
      let get_d u    = match u with O | I -> assert false | Var u     -> u.d
      let update u i = match u with O | I -> ()    | Var u -> u.d <- u.d + i

      let num =
        let zero = O
        and one = I
        and add a b = perform (Add (a, b))
        and mul a b = perform (Mul (a, b)) in
        { zero; one; add; mul }

      let x = mk n

      let () =
        match e.eval num x with
        | effect (Add (a, b)) k ->
           let u = mk (get_v a + get_v b) in
           continue k u;
           update a (get_d u);
           update b (get_d u)
        | effect (Mul (a, b)) k ->
           let u = mk (get_v a * get_v b) in
           continue k u;
           update a (get_d u * get_v b);
           update b (get_d u * get_v a)
        | y ->
           update y one

    end in
    get_d x
}
```

Figure 5: Reverse-mode automatic differentiation in Multicore OCaml

informal sketch of how the code works; a more detailed correctness argument is deferred to a later section (§5).

The definitions of the types `'v num` and `exp` are the same as in Figure 1 and have been explained earlier (§1).

The definition of the function `diff` begins with a number of abstractions over value and type parameters. The parameter `e` (line 4) is the expression that we must differentiate. The parameter `v` (line 5) is the type of numbers that the user chooses, and the parameters `zero`, `one`, `add`, `mul` are the arithmetic operations at this type, also chosen by the user. The parameter `n` (line 5) is the value at which the expression `diff e` must be evaluated. After these preliminaries, the bulk of the code of `diff` appears between lines 9 and 43.[5]

What happens next? Because the expression `e` is opaque, there is only one thing that we can do with it, namely evaluate it, by calling `e.eval`. We do so at line 28. Now, recall that `e.eval` is a polymorphic function (line 2): it is parameterized with a type `'v` of numbers and with a dictionary of type `'v num`. Thus, when we call `e.eval`, we are free to instantiate `'v` with a type of our choosing, provided that we supply implementations of `zero`, `one`, `add`, `mul` at this type. This is what we do next: between lines 9 and 23, we define a type `t` and construct a dictionary of type `t num`. These definitions are internal: to a user of `diff`, they are entirely invisible.

We use objects of type `t` to keep track of the intermediate results that appear during the evaluation of the expression `e`. We define `t` as an algebraic data type (line 9): an intermediate result is either `O`, which denotes the constant 0, or `I`, which denotes the constant 1, or a "variable", that is, a record that carries the tag `Var` and holds two fields named `v` and `d`, short for *value* and *derivative*.[6] As will become clear shortly, the `v` field is immutable and is initialized during the forward phase of the algorithm; the `d` field is mutable and is updated (possibly several times) during the backward phase.

Four auxiliary functions help construct and manipulate objects of type `t`. The function `mk` (line 13) has type `v -> t`. It allocates and initializes a fresh "variable". The functions `get_v` and `get_d` (lines 14 and 15) have type `t -> v`. They read the `v` and `d` fields of a "variable", and handle the constants `O` and `I` in a suitable way.[7] The function `update` (line 16) has type `t -> v -> unit`. It updates the `d` field of a "variable" by adding the value `i` to it. Applying `update` to the constant `O` or `I` does nothing.

A dictionary `num` of type `t num` is defined at lines 18–23. This dictionary is passed as an argument to `e.eval` on line 28.

A single "variable" `x` is allocated at line 25 and is given the value `n`. The `d` field of this variable serves as a placeholder where the final result of the differentiation algorithm is eventually read (line 43).

The bulk of the computation, which takes place between lines 27 and 40, exploits effects in the following way. The operations `num.add` and `num.mul`, which are defined at lines 21 and 22 and which can be invoked by `e.eval`, perform effects. Two effects, `Add` and `Mul`, are declared at lines 10 and 11. We handle these effects by wrapping the function call

[5] "**`let open struct`**" at line 7 is an OCaml idiosyncrasy that allows toplevel definitions (of types, effects, and values) to appear in the midst of an expression.

[6] One might wish to simplify things by getting rid of the data constructors `O` and `I` and by defining `zero` and `one` as "variables" whose `v` fields hold the values 0 and 1 and whose `d` fields are useless. This can work in practice, but involves redundant writes to the `d` fields. Furthermore, our proof in Separation Logic is easier if `zero` and `one` are immutable (therefore shareable) objects.

[7] The function `get_d` is never applied to a constant. The expression **`assert false`** denotes a dead branch.

`e.eval num x` in an effect handler (line 28). The structure of this handler is analogous to that found in our previous example (Figure 3). Because the continuation is invoked in the middle of the handler's code (lines 31 and 36), the execution of the algorithm is divided in two phases: a *forward phase* and a *backward phase*.

The forward phase lasts as long as the function call `e.eval num x` runs. During this phase, the operations `num.add` and `num.mul` can be invoked, causing a sequence of effects to take place. When an effect occurs, it is serviced by the handler code that precedes the **`continue`** statement (lines 30 and 35). Control is then immediately handed back to the computation `e.eval num x`, while the execution of the handler code that follows the **`continue`** statement is postponed. The handler code before the **`continue`** statement uses the constructor function `mk` to allocate a fresh "variable" `u`. The `v` field of this variable, which is immutable, immediately receives its final value: during the forward phase, the expression `e` is evaluated in a standard way. The `d` field, which is mutable, is initialized with the number `zero`: it is meant to be updated during the backward phase.

The backward phase begins when the function call `e.eval num x` terminates and returns a "variable" `y`, which represents its final result. At this point, the effect handler receives control (line 39) and writes the number `one` into `y`'s `d` field. Then, the control stack is unwound and all of the handler code whose execution was postponed earlier is executed. Thus, for each `Add` effect that took place earlier, the code at lines 32–33 is executed, and for each `Mul` effect that took place earlier, the code at lines 37–38 is executed. The purpose of this code, in short, is to update the `d` fields of the operands of each intermediate addition or multiplication operation. These updates are performed *in reverse order* with respect to the earlier sequence of effects.

Once the backward phase is over, the code block at lines 27–40 is exited. Control moves on to the last line, where the desired derivative is now found in the `d` field of the "variable" `x` (line 43).

4. Mathematical Preliminaries

We now wish to explain more precisely what reverse-mode automatic differentiation computes and why it works. To this end, we must introduce a number of simple mathematical definitions. We distinguish a group of notions that are needed in the specification of the automatic differentiation algorithm and a group of notions that are needed in the proof of correctness of the algorithm. These groups are presented in §4.1 and §4.2.

4.1. Notions Used in the Specification of the Algorithm.

The algorithm is insensitive to the manner in which numbers are represented at runtime: the runtime values manipulated by the algorithm are not necessarily floating-point numbers. In line 5 of Figure 5, one can see that the code is parameterized over a type `v` of numbers and over a dictionary `{ zero; one; add; mul }` whose type is `v num`. Furthermore, when verifying the algorithm, we want our reasoning to be not only independent of the runtime representation of numbers, but also independent of the mathematical structure that these numbers inhabit. That is, the ideal mathematical numbers that appear in the proof of the algorithm need not be real numbers: they can be drawn from an arbitrary set $\mathcal{R}$, provided this set is equipped

with sufficient structure. For our purposes, the structure of a semiring suffices.[8] Thus, the proof of the algorithm is parameterized with a semiring $\mathcal{R}$, or, more precisely, with a tuple $(\mathcal{R}, 0, +, 1, \times, \equiv)$, where $\equiv$ is an equivalence relation on $\mathcal{R}$, and where the axioms of a semiring hold with respect to this equivalence relation. In the following, r ranges over $\mathcal{R}$. The proof is also parameterized with a binary predicate *isNum*, which provides a connection between runtime values and mathematical numbers: the relation v *isNum* r means that the runtime value v represents the number r. These aspects are explained in greater detail when we present the specification of the algorithm (§5.2).

Earlier (§1), we have informally defined what we mean by an "expression", and we have decided to represent expressions at runtime in "tagless final" style, that is, as computations of type `exp` (Figure 5). Nevertheless, when reasoning about the algorithm, it is natural to think of an expression as an abstract syntax tree. Thus, in this section, we define an (ideal, mathematical) expression as an abstract syntax tree. Later on, we define a predicate *isExp* which spells out the connection that exists between runtime values of type `exp` and ideal expressions (Definition 5.2).

An *expression* E is an abstract syntax tree whose nodes include (1) the constants "zero" and "one"; (2) applications of binary arithmetic operators (addition and multiplication), also known as "nodes"; and (3) other numeric constants or variables, also known as "leaves", drawn from some set $\mathcal{I}$.

Definition 4.1 (Expressions)**.** Let $\mathcal{I}$ be a finite or infinite set. Let $\imath$ range over $\mathcal{I}$. The set $Exp_{\mathcal{I}}$ of expressions whose variables are drawn from $\mathcal{I}$ is defined as follows:

$$\begin{aligned} Binop \ni op &::= Add \mid Mul \\ Exp_{\mathcal{I}} \ni E &::= Zero \mid One \mid Node\ op\ E\ E \mid Leaf\ \imath \end{aligned}$$

In the following, the parameter $\mathcal{I}$ is instantiated in several different ways:

(1) We sometimes wish to reason about closed expressions, that is, expressions without variables, whose leaves are arbitrary numeric constants. Then, we instantiate $\mathcal{I}$ with $\mathcal{R}$.
(2) The function `diff` operates on expressions of a single mathematical variable X. In this case, we take $\mathcal{I}$ to be a singleton set. In the paper, we write $\{X\}$ for this singleton set, where the name X is seen as a fixed abstract symbol. [9]
(3) While verifying the algorithm, we sometimes need to abstract away a subexpression and view it as a mathematical variable. At runtime, when the algorithm runs, this subexpression is represented by a value v of type `t` (line 9 of Figure 5). We use this value as a unique identifier. This leads us to instantiating $\mathcal{I}$ with the set *Val* of runtime values, which we introduce later on (§5).

To give meaning to expressions, we define $[\![E]\!]_\varrho$, the value of the expression E under an *environment* ϱ, a mapping of variables to numbers in $\mathcal{R}$. This is straightforward:

[8] We require just a semiring, as opposed to a ring, because the inverse of addition is not used in the algorithm or in its proof.

[9] In Coq, we use the `unit` type, whose single inhabitant is the unit value `tt`.

Definition 4.2 (Expression Evaluation)**.** The function $[\![\cdot]\!]_{(\cdot)} : \mathit{Exp}_{\mathcal{I}} \to (\mathcal{I} \to \mathcal{R}) \to \mathcal{R}$ is inductively defined as follows:

$$\begin{array}{rcl} [\![\mathit{Zero}]\!]_{\varrho} & = & 0 \\ [\![\mathit{One}]\!]_{\varrho} & = & 1 \\ [\![\mathit{Node\ Add\ E_1\ E_2}]\!]_{\varrho} & = & [\![E_1]\!]_{\varrho} + [\![E_2]\!]_{\varrho} \\ [\![\mathit{Node\ Mul\ E_1\ E_2}]\!]_{\varrho} & = & [\![E_1]\!]_{\varrho} \times [\![E_2]\!]_{\varrho} \\ [\![\mathit{Leaf\ \imath}]\!]_{\varrho} & = & \varrho(\imath) \end{array}$$

In the specification of the algorithm, we use a symbolic derivative, and in its proof, we use partial derivatives. We now provide definitions of these notions. We begin with $\partial E/\partial \jmath\ (\varrho)$, the partial derivative of an expression E with respect to a variable $\jmath$, evaluated at a point ϱ. In this definition, E is an expression of several variables: its variables range over $\mathcal{I}$. The variable $\jmath$ is an arbitrary member of $\mathcal{I}$. The environment ϱ has type $\mathcal{I} \to \mathcal{R}$; it can be thought of as an $\mathcal{I}$-dimensional point.

Definition 4.3 (Partial Derivative)**.** The function $\partial\cdot/\partial\cdot\ (\cdot) : (\mathcal{I} \to \mathcal{R}) \to \mathit{Exp}_{\mathcal{I}} \to \mathcal{I} \to \mathcal{R}$ is inductively defined as follows:

$$\begin{array}{rcll} \partial \mathit{Zero}/\partial \jmath\ (\varrho) & = & 0 & \\ \partial \mathit{One}/\partial \jmath\ (\varrho) & = & 0 & \\ \partial(\mathit{Node\ Add\ E_1\ E_2})/\partial \jmath\ (\varrho) & = & \partial E_1/\partial \jmath\ (\varrho) + \partial E_2/\partial \jmath\ (\varrho) & \\ \partial(\mathit{Node\ Mul\ E_1\ E_2})/\partial \jmath\ (\varrho) & = & \partial E_1/\partial \jmath\ (\varrho) \times [\![E_2]\!]_{\varrho} + [\![E_1]\!]_{\varrho} \times \partial E_2/\partial \jmath\ (\varrho) & \\ \partial(\mathit{Leaf\ \imath})/\partial \jmath\ (\varrho) & = & 1 & \text{if } \imath = \jmath \\ \partial(\mathit{Leaf\ \imath})/\partial \jmath\ (\varrho) & = & 0 & \text{otherwise} \end{array}$$

One recognizes in this definition the well-known laws that indicate how to compute a partial derivative of a sum, of a product, and of a variable. Most mathematicians would view the above equations as a set of laws that can be *proved*, based on a more primitive definition of derivation. We take these laws as the *definition* of derivation, because this is sufficient for our purposes and removes the need for us to engage in deeper mathematics.

The partial derivative $\partial E/\partial \jmath\ (\varrho)$ can be thought of as a number, an element of the semiring $\mathcal{R}$. However, provided an appropriate semiring is chosen, Definition 4.3 can also serve as the basis for a definition of the symbolic derivative. For this definition, we restrict our attention to *univariate expressions*, that is, expressions of a single variable X, and we instantiate $\mathcal{R}$ with the *free semiring* $\mathit{Exp}_{\{X\}}$ of univariate expressions.[10]

We write *Leaf* for the environment (that is, the function) of type $\{X\} \to \mathit{Exp}_{\{X\}}$ that maps the variable X to the expression *Leaf* X. We remark that evaluating an expression E in the free semiring with respect to this environment yields E itself: that is, the identity law $[\![E]\!]_{\mathit{Leaf}} = E$ holds. This remark helps understand the following definition:

Definition 4.4 (Symbolic Derivative)**.** Let $E \in \mathit{Exp}_{\{X\}}$ be a univariate expression. The symbolic derivative of E, written E', is defined by:

$$E' \ = \ \partial E/\partial X\ (\mathit{Leaf})$$

[10]The free semiring of univariate expressions is defined as follows. Its carrier is $\mathit{Exp}_{\{X\}}$, the set of univariate expressions. Its constants zero and one are the expressions *Zero* and *One*. Its addition and multiplication operations map E_1 and E_2 respectively to *Node Add* E_1 E_2 and *Node Mul* E_1 E_2. Its equivalence relation is inductively defined as the smallest equivalence relation that validates all of the semiring axioms. In a slight abuse of notation, we use the name $\mathit{Exp}_{\{X\}}$ both for the free semiring and for its carrier set.

This definition views the symbolic derivative E' as the partial derivative of E with respect to X, evaluated in the free semiring, under the environment *Leaf*. Although this definition might seem somewhat mysterious, one can easily check that it gives rise to the usual laws of symbolic derivation. In the case of multiplication, for instance, by combining the fourth equation of Definition 4.3 with the identity laws $[\![E_1]\!]_{Leaf} = E_1$ and $[\![E_2]\!]_{Leaf} = E_2$, one immediately obtains the familiar symbolic derivation law:

$$(Node\ Mul\ E_1\ E_2)' = Node\ Add\ (Node\ Mul\ E_1'\ E_2)\ (Node\ Mul\ E_1\ E_2').$$

4.2. **Notions Used in the Proof of the Algorithm.** As announced earlier, we now wish to introduce a series of definitions that play a role in the proof of the algorithm (§5.3), but not in its specification (§5.2). These definitions may be skipped upon first reading.

For the purposes of the proof, we must introduce an alternative view of syntactic expressions. During the execution of the automatic differentiation algorithm, the function call `e.eval num x` (line 28 of Figure 5) constructs an expression in an incremental manner, one node at a time. We need to reason about such a sequence of construction events and to convert it, when desired, to an expression.

In each such event, a new node u is built out of an arithmetic operator *op* and two preexisting nodes a and b. Technically, the node identifiers u, a and b are drawn from the set *Val* of runtime values, but this detail does not matter at this point. We write the 4-tuple (u, op, a, b) under the form $\mathsf{let}\ u = a\ op\ b$, so as to better reflect its intended meaning, which is that the identifier u is bound to the atomic expression $a\ op\ b$. We refer to such a 4-tuple as a *binding*. We write B for a single binding and K for a list of bindings, where, by convention, the left end of the list represents the earliest binding and its right end represents the newest binding.

Definition 4.5 (Bindings; contexts)**.** Let u, a, b range over *Val*. The syntax of bindings and contexts is defined as follows:

$$\begin{array}{rcl} LetBinding \ni B & ::= & \mathsf{let}\ u = a\ op\ b \\ Context \ni K & ::= & [] \ \mid\ B; K \end{array}$$

We use a semicolon to denote all three forms of concatenation, that is, $B; K$ for "cons", $K; B$ for "snoc", and $K; K'$ for general concatenation. Moreover, we use $defs(K)$ to denote the list of binders introduced in the context K, that is, $defs(\mathsf{let}\ u = a\ op\ b; K) = u; defs(K)$ and $defs([]) = []$.

The hole $[]$ at the right end of a list of bindings K can be viewed as a placeholder, waiting to be filled with a node identifier y. In other words, a sequence of bindings K can be viewed as a *context*. A pair of a context K and a node identifier y forms an alternative representation of an expression: it is a linear representation, where every subexpression is designated by an identifier, and where the order of construction is explicit. The identifier y designates a distinguished node: the root of the expression. The operation of filling the hole of a context K with an identifier y, defined next, offers a conversion from this alternative representation to ordinary expressions.

Definition 4.6 (Filling)**.** The function $\cdot[\cdot] : Context \to Val \to Exp_{\,Val}$ is inductively defined as follows:

$$\begin{array}{rcll} [][y] & = & Leaf\ y & \\ (K; \mathsf{let}\ u = a\ op\ b)[y] & = & Node\ op\ (K[a])\ (K[b]) & \text{if } u = y \\ (K; \mathsf{let}\ u = a\ op\ b)[y] & = & K[y] & \text{otherwise} \end{array}$$

Our last definition is the extension of an environment ϱ with a context K, resulting in an updated environment $\varrho\{K\}$.

Definition 4.7 (Extension of an Environment)**.** The function $\cdot\{\cdot\} : (\mathit{Val} \to \mathcal{R}) \to \mathit{Context} \to (\mathit{Val} \to \mathcal{R})$ is inductively defined as follows:

$$\varrho\{K\} = \lambda y.\ [\![K[y]]\!]_\varrho$$

This definition states that, in order to obtain the meaning of an identifier y under the environment $\varrho\{K\}$, one must first plug y into K, yielding an expression $K[y]$, then evaluate this expression under ϱ, yielding a number $[\![K[y]]\!]_\varrho$.

5. Formal Verification of Reverse-Mode AD with Effect Handlers

In this section, we use the mathematical definitions of the previous section (§4) in combination with Hazel [dVP21a], an extension of Separation Logic with support for effect handlers, described in previous work by the same authors. Hazel allows writing a formal specification of the library that was presented earlier (§3). Based on this specification, one can prove that the code in Figure 5 is correct: we do indeed provide a machine-checked proof of this claim. Based on this specification, without any knowledge of the code, one could also prove the correctness of a program that uses the automatic differentiation library; we do not do so in this paper.

We open this section with a short introduction to the core notions of the program logic Hazel (§5.1). Then, we propose a formal specification of our automatic differentiation library (§5.2). Finally, we present the proof that the library satisfies its specification (§5.3).

5.1. **Hazel.**

In traditional Separation Logic [Rey02, O'H19], one can write program specifications at a pleasant level of abstraction, which combines rigor and generality. A program specification expressed in Separation Logic is rigorous, as it has a well-defined mathematical meaning. It is general, because it does not mention the areas of memory that the program must not or need not access: it describes only the data structures that the program needs to access or modify.

Hazel extends this methodology to support programs that exploit effect handlers. Hazel consists of two main components, namely (1) a core programming language with support for effect handlers, equipped with a formal operational semantics; and (2) a program logic, where the standard notion *weakest precondition* [JKJ+18, §6] is extended with a *protocol* that describes a contract between the program that performs effects and the effect handler.

***HH* and its Operational Semantics.** To reason about programs in a rigorous way, a necessary step is to define what programs are and how they behave. In other words, one must define the syntax and the dynamic semantics of the programming language of interest. Here, we are interested in reasoning about Multicore OCaml programs. However, proposing a formal dynamic semantics for the complete OCaml language (let alone the Multicore extensions) would be a challenging endeavor. For this reason, we focus on a core calculus, a subset of Multicore OCaml, which suffices to express the automatic differentiation library of §3. A small set of features, including first-class functions, references, effect handlers and one-shot continuations, suffices. In previous work [dVP21a], we have defined such a calculus, dubbed *HH* (for "heaps and handlers"), and have endowed it with a small-step operational semantics. We do not recall the syntax or operational semantics of *HH*, whose details are

not relevant here. For the purposes of the present paper, it should suffice to say that there is an infinite set *Loc* of memory locations; there is a set *Val* of values, which includes the unit value, memory locations, binary products and sums, first-class functions, and first-class continuations; and the heap is modeled as a finite map of memory locations to values.

There are differences between *HH* and Multicore OCaml, most of which we believe are inessential. Perhaps the most critical difference is that an effect in Multicore OCaml carries a name and a value, and an **`effect`** declaration dynamically generates a fresh name, whereas an effect in *HH* is nameless: it carries just a value. For the time being, we gloss over this aspect, and discuss it in §7.

Program logic. An operational semantics defines the behavior of programs, but does not provide a convenient means of *describing* or *reasoning about* this behavior at a high level of abstraction. A program logic addresses this shortcoming. It offers a specification language in which one can describe how a program behaves in the eye of an outside observer, without exposing the details of its inner workings.

Hazel is a program logic for the programming language *HH*. Thus, it can describe programs that involve effects and effect handlers. Hazel is both an instance and an extension of the program logic Iris [JKJ+18]. Iris is programming-language-independent: we instantiate it for *HH*. Iris traditionally has no support for effect handlers: we extend it with such support. The main novel ingredient of Hazel's specification language is a richer *weakest precondition* predicate, which is parameterized with a *protocol*.

A traditional weakest precondition predicate, as found in propositional dynamic logic [TB19] or in Iris [JKJ+18, §6], takes the form $wp\ e\ \{\phi\}$, where e is a program (or an expression that is part of a larger program) and ϕ is a postcondition. A postcondition is a predicate that describes the result value and the final state: in short, the assertion $wp\ e\ \{\phi\}$ guarantees that the program e can be safely executed (that is, it will not crash) and that, if execution terminates, then, in the final state, the result value v satisfies the assertion $\phi(v)$. A condition P on the initial state, also known as a precondition, can be expressed via an implication: $P \mathrel{-\!\!*} wp\ e\ \{\phi\}$ means that if initially the assertion P holds, then it is safe to execute e and, once a value v is returned, $\phi(v)$ holds. In Iris, such an assertion is *affine*, which means that it represents a permission to execute e *at most once*. When this assertion is wrapped in the "persistence" modality $\Box$, it represents a permission to execute e as many times as one wishes. Thus, the persistent assertion $\Box(P \mathrel{-\!\!*} wp\ e\ \{\phi\})$ is equivalent to the traditional Hoare triple $\{P\}\, e\, \{\phi\}$ [JKJ+18, §6]. The distinction between affine and persistent assertions matters especially in Hazel because first-class continuations in *HH* are one-shot: an attempt to invoke a continuation twice causes a runtime failure. Hazel statically rules out this kind of failure: when proving the correctness of an algorithm, Hazel requires the user to prove that every continuation is invoked at most once.

In *HH*, a program can not only diverge, or terminate and return a value, but can also interrupt itself by performing an effect. For this reason, in contrast with a traditional weakest precondition, an *extended weakest precondition* in Hazel takes the form $ewp\ e\ \langle\Psi\rangle\{\phi\}$, where e is a program, ϕ is a postcondition, and Ψ is a *protocol*. A protocol describes the effects that a program may perform: it can be thought of as a contract between the program and the effect handler that encloses it. In short, the assertion $ewp\ e\ \langle\Psi\rangle\{\phi\}$ means that (1) it is safe to execute e, that (2) if a value v is returned, then $\phi(v)$ holds, and that (3) if e performs an effect, then this effect respects the protocol Ψ.

What is a protocol? We answer this question only partially at this point, because the specification of `diff` (§5.2) involves very few protocols. One important concrete protocol is the *empty protocol* $\bot$, which forbids all effects: the assertion $\mathit{ewp}\ e\ \langle\bot\rangle\{\phi\}$ guarantees that the program e does not perform any effect. Another important idiom is the use of an *abstract protocol*, that is, a universally quantified protocol variable Ψ. Abstract protocols are typically used to describe *effect-polymorphic* higher-order functions. As we shall see, an expression in tagless final style (§1) is an example of such a function.

5.2. **Specification.** Intuitively, `diff` computes the tagless final encoding of the derivative of a univariate expression $E \in \mathit{Exp}_X$, given the encoding of E. The key to formalizing this intuition is thus to define in logical terms what it means for a value to encode an expression.

Indeed, let us assume that we are able to introduce a binary predicate *isExp* capturing this notion: for any value e and expression $E \in \mathit{Exp}_X$, the assertion $e\ \mathit{isExp}\ E$ holds if e is an encoding of E. Then, it is easy to write the specification of `diff`. It suffices to state that if $e\ \mathit{isExp}\ E$ holds for some value e and expression E, then the program $\texttt{diff}\,e$ returns a value e' that encodes the derivative of E, that is, the assertion $e'\ \mathit{isExp}\ E'$ holds. In formal terms, we make the following statement:

Statement 5.1 (Formal specification)**.** *The specification of* `diff` *is expressed as follows:*

$$\forall e\,E.\ e\ \mathit{isExp}\ E \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{diff}\,e)\ \langle\bot\rangle\{e'.\ e'\ \mathit{isExp}\ E'\}$$

There remains to explain the definition of the predicate *isExp*, which is given below:

Definition 5.2 (*isExp*)**.** The predicate $\_\ \mathit{isExp}\ \_ : \mathit{Val} \to \mathit{Exp}_{\{X\}} \to \mathit{iProp}$, where *iProp* is the type of Hazel assertions, is defined as follows:

$$e\ \mathit{isExp}\ E\ = \forall \texttt{zero one add mul}.\ \left\{\begin{array}{ll} \mathit{ewp}\ (e.\texttt{eval}\ \{\texttt{zero}; \texttt{one}; \texttt{add}; \texttt{mul}\})\ \langle\bot\rangle\{f\ . & \text{(A)}\\ \quad \forall \Psi\ \mathcal{R}\ \mathit{isNum}. & \text{(B)}\\ \quad\quad \mathit{isNumDict}\ (\texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}, \mathit{isNum}, \Psi, \mathcal{R}) \mathrel{-\!\!*} & \\ \quad\quad\quad \forall n\ r.\ n\ \mathit{isNum}\ r \mathrel{-\!\!*} & \\ \quad\quad\quad\quad \mathit{ewp}\ (f\ n)\ \langle\Psi\rangle\{d.\ \exists s. & \text{(C)}\\ \quad\quad\quad\quad\quad d\ \mathit{isNum}\ s\ *\ s \equiv_{\mathcal{R}} [\![E]\!]_{(\lambda X.\ r)}\} & \\ \} & \end{array}\right.$$

The definition captures the idea that a value e is an encoding of the expression E if $e.\texttt{eval}$, applied to a dictionary and to a number, returns the value of E at this number. The assertion $e\ \mathit{isExp}\ E$ actually spells this idea out in two steps. The first step is the assertion on line (A): the claim that the application of $e.\texttt{eval}$ to any dictionary runs silently (it either loops or eventually terminates), returning a value f. The second step is the assertion on line (B): the postcondition of this application, that is, a claim about f.

On line (B), we find three universally quantified terms: a protocol Ψ, a semiring $\mathcal{R}$ and a *representation predicate* $\_\ \mathit{isNum}\ \_ : \mathit{Val} \to \mathcal{R} \to \mathit{iProp}$. They all participate in the specification of the arithmetic operations in the dictionary. The protocol describes the effects that `add` and `mul` may perform. The semiring $\mathcal{R}$ is the set of numbers over which the numeric functions operate. The predicate *isNum* is the link between values and elements of $\mathcal{R}$: the assertion $n\ \mathit{isNum}\ r$ holds when the value n represents the number r.

The claim that Ψ is a description of `add` and `mul`'s effects and that they operate on representations of $\mathcal{R}$ elements is expressed by the premise *isNumDict*. This premise is defined

$$
\begin{array}{l}
\mathit{isNumDict}\,(\texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}, \mathit{isNum}, \Psi, \mathcal{R}) \;= \\
\quad \vdash\ \texttt{zero}\ \mathit{isNum}\ 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\quad \vdash\ \texttt{one}\ \mathit{isNum}\ 1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\quad \vdash\ \forall a\, b\, r\, s. a\ \mathit{isNum}\ r \mathrel{-\!\!*} b\ \mathit{isNum}\ s \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{add}\ a\ b)\ \langle\Psi\rangle\{c.\, c\ \mathit{isNum}\ (r+s)\} \quad \wedge \\
\quad \vdash\ \forall a\, b\, r\, s. a\ \mathit{isNum}\ r \mathrel{-\!\!*} b\ \mathit{isNum}\ s \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{mul}\ a\ b)\ \langle\Psi\rangle\{c.\, c\ \mathit{isNum}\ (r\times s)\} \quad \wedge \\
\quad \vdash\ \forall a\, r.\ a\ \mathit{isNum}\ r \mathrel{-\!\!*} \square(a\ \mathit{isNum}\ r)
\end{array}
$$

Figure 6: Specification of a dictionary of arithmetic operators

in Figure 6 as the conjunction of the following assertions: (1) the value `zero` represents the semiring element 0; (2) the value `one` represents the semiring element 1; (3) for all semiring elements r and s, `add` computes a representation of $r + s$ when applied to representations of r and s; (4) for all semiring elements r and s, `mul` computes a representation of $r \times s$ when applied to representations of r and s; (5) a representation of a number is persistent, which means that a value that represents a number can be used as many times as one wishes.

The rest of the postcondition on line (C) is a specification of f:

$$\forall n\, r.\ n\ \mathit{isNum}\ r \mathrel{-\!\!*} \mathit{ewp}\ (f\ n)\ \langle\Psi\rangle\{d.\ \exists s.\ d\ \mathit{isNum}\ s\ *\ s \equiv_{\mathcal{R}} [\![E]\!]_{(\lambda X.\ r)}\}$$

It reads as follows: for every value n and semiring element r, if n represents r, then $f\,n$ computes a representation of a number that is equivalent to the value of E under the environment $(\lambda X.\ r)$. [11] The specification includes the protocol Ψ to express that f is effect-polymorphic: the only observable effects that f may perform are those introduced by calls to `add` or `mul`.

5.3. **Proof.** Now, we present the formal arguments establishing the correctness of `diff`. Our purpose is not to give a step-by-step derivation of this fact but to explain the *structure of the proof* and to provide *convincing arguments* of why the proof works.

We follow a discipline in the naming of metavariables. All the metavariables introduced in the proof are in accordance with the names of variables both in the Multicore OCaml implementation (Figure 5) and in the definition of *isExp* (5.2). We use the following letters to designate values and memory locations during the proof:

(1) ***Intermediate variables.*** The letters a, b, u, x and y designate values of type `t` (line 9). We refer to them as *intermediate variables* because they only appear at runtime.
(2) ***Numeric values.*** The letters n, m and d designate values of type `v` (line 5). They represent numbers in an abstract semiring $\mathcal{R}$ that is introduced during the proof. We refer to these values as *numeric values*.
(3) ***Memory locations.*** The letters $\mathscr{p}$ and $\mathscr{q}$ designate memory locations.

[11] The fact that we only require a representation of an element equivalent to $[\![E]\!]_{(\lambda X.\ r)}$ allows optimizations in the encoding of E: we are not obliged to compose the result of the arithmetic operators in the exact same order as the corresponding nodes appear in E; we can, for instance, exploit the fact that E might contain several occurrences of a subexpression to share the intermediate results during the computation.

Preliminaries. When proving the correctness of `diff`, we reason about a specific application: we fix a value e, to which we refer as the client, and an expression E and we assume that e encodes E, that is, we assume that the assertion e *isExp* E holds. We then wish to prove that `diff` produces an encoding of E'.

Because the program `diff` e terminates immediately, producing a value e', our goal is to show that e' encodes E': we must show that the assertion e' *isExp* E' holds.

First, let us unfold the definition of *isExp* (5.2) in the current goal e' *isExp* E':

$$\forall \texttt{zero}\ \texttt{one}\ \texttt{add}\ \texttt{mul}.\ \left\{\begin{array}{l} \mathit{ewp}\ (e'.\texttt{eval}\ \{\texttt{zero}; \texttt{one}; \texttt{add}; \texttt{mul}\})\ \langle\bot\rangle\{f\ . \\ \quad \forall \Psi\ \mathcal{R}\ \mathit{isNum}. \\ \quad\quad \mathit{isNumDict}\ (\texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}, \mathit{isNum}, \Psi, \mathcal{R}) \mathrel{-\!\!*} \\ \quad\quad\quad \forall n\ r.\ n\ \mathit{isNum}\ r \mathrel{-\!\!*} \\ \quad\quad\quad\quad \mathit{ewp}\ (f\ n)\ \langle\Psi\rangle\{d.\ \exists s. \\ \quad\quad\quad\quad\quad d\ \mathit{isNum}\ s\ *\ s \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}\} \\ \} \end{array}\right.$$

Next, we introduce the universally quantified arithmetic operators `zero`, `one`, `add` and `mul`. The goal then becomes to prove a specification about the following program:

$$e'.\texttt{eval}\,\{\texttt{zero}; \texttt{one}; \texttt{add}; \texttt{mul}\}$$

Recall that e' stands for the result of applying `diff` to e. By studying the code in Figure 5, it is easy to see that e'`.eval` is a function of two arguments: a dictionary and a number. Therefore, when we supply its first argument – the dictionary {`zero`; `one`; `add`; `mul`} – it immediately reduces to a unary function, which we call f. The goal now becomes to prove the following claim:

$$\begin{array}{l} \forall \Psi\ \mathcal{R}\ \mathit{isNum}. \\ \quad \mathit{isNumDict}\ (\texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}, \mathit{isNum}, \Psi, \mathcal{R}) \mathrel{-\!\!*} \\ \quad\quad \forall n\ r.\ n\ \mathit{isNum}\ r \mathrel{-\!\!*} \\ \quad\quad\quad \mathit{ewp}\ (f\ n)\ \langle\Psi\rangle\{d.\ \exists s. \\ \quad\quad\quad\quad d\ \mathit{isNum}\ s\ *\ s \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}\} \end{array}$$

Again, let us introduce the leading universal quantifiers: a protocol Ψ, a semiring $\mathcal{R}$ and a representation predicate *isNum* that links runtime values with numbers in $\mathcal{R}$. Let us also introduce the premise *isNumDict* (`zero`, `one`, `add`, `mul`, *isNum*, Ψ, $\mathcal{R}$) that formally captures this intuitive meaning of Ψ, $\mathcal{R}$ and *isNum*. Finally, we introduce the variables n and r and the premise n *isNum* r: the claim that the runtime value n represents the ideal number r.

The goal thus reaches the following form:

Statement 5.3 (Current Goal)**.**

$$\mathit{ewp}\ (f\ n)\ \langle\Psi\rangle\{d.\ \exists s. d\ \mathit{isNum}\ s\ *\ s \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}\}$$

We must prove that $f\,n$ returns a value d that represents the term $[\![E']\!]_{(\lambda X.\ r)}$ up to equivalence. Moreover, f cannot introduce observable effects: the protocol Ψ allows f to call `add` or `mul`, which may throw effects, but obliges f to capture any other effects it may introduce.

Remember that f is the result of the application e'`.eval` {`zero`; `one`; `add`; `mul`}. So, the sequence of instructions that f performs is defined between lines 5 and 43 from Figure 5. We will consider these lines in the order as they appear. To clarify the discussion, we divide the execution of f into five phases:

(1) ***Initialization.*** This phase consists of the allocation of the variable x (line 25).
(2) ***Forward phase.*** This phase begins with the execution of the client e under the handler (line 28) and ends upon termination of the client.
(3) ***Transition phase.*** This phase corresponds to the execution of the return branch of the handler (line 40).
(4) ***Backward phase.*** This phase corresponds to the execution of the handler instructions that follow the continuation call.
(5) ***Finalization.*** The last phase corresponds to the read of the field `d` of x (line 43).

Initialization. The initial variable x introduced in line 25 is a `Var` pair containing the value n and a memory location $\not{p}$ that is used to *store a derivative*. (The exact meaning of this derivative – which expression is being differentiated, with respect to which variable, under what environment, over which semiring – will become clear later when we discuss the relation among the variables allocated during the backward phase.) The memory location $\not{p}$ initially holds the value `zero`. In summary, the following assertion holds after the allocation of x (after line 25):

$$x = \mathtt{Var}(n, \not{p}) \ * \ \not{p} \mapsto \mathtt{zero} \tag{5.1}$$

Another key notion is introduced in the initialization phase: the sequence of operations performed by the client. Although the algorithm does not store an explicit sequence of operations, such a sequence does appear in the proof. So it is necessary to introduce this sequence as a purely logical entity. Iris supports this kind of reasoning through a mechanism known as *ghost state*: at any time during the verification, it is possible to introduce a ghost variable γ that stores an element of a *camera* [JKJ+18] that determines how the variable can be updated and how its contents can be shared.

At this point in the proof, we introduce a ghost variable γ that stores a list of bindings K (Definition 4.5). Intuitively, γ keeps track of the sequence of operations performed by the client: each entry of K records one such operation as well as the intermediate variable allocated by the handler in response to this request.

The claim that γ currently stores a list of bindings K is written in the following fashion: $\boxed{\bullet K}^{\gamma}$. Since the execution of the client has not yet started, the initial state of γ is the empty list; that is, the assertion $\boxed{\bullet []}^{\gamma}$ holds at the end of line 25. This initial value is chosen at the moment we introduce the variable γ. In addition to its initial value, we also commit to the way in which we can update the contents of γ. The variable γ is *monotone*: it stores a sequence that *grows* as the execution of the program evolves; we can only add new entries to the context K stored in γ.

Although, on the one hand, this restriction limits our freedom to update γ, on the other hand, it allows us to define the assertion $\boxed{\circ B}^{\gamma}$, whose intuitive meaning is the binding B appears in the current context.

For completeness, Figure 7 includes the formal rules governing the state of γ. Rule Introduce states that γ can be introduced. Rule Spread states that the assertion $\boxed{\circ B}^{\gamma}$ can be shared. Rule Confront justifies the intuitive reading of $\boxed{\circ B}^{\gamma}$: if K is the current context and if the assertion $\boxed{\circ B}^{\gamma}$ holds, then $B \in K$ can be deduced. Rule Update dictates how to update the state of γ: we are allowed to only add entries to K.

The forward phase starts with the execution of the client under the handler (line 28). Before we can reason about its evaluation, we must show that the client is indeed safe to

$$\text{(Introduce)} \qquad \dot{\Rrightarrow} \exists \gamma.\, \boxed{\bullet\, []}^{\gamma}$$

$$\text{(Spread)} \qquad \boxed{\circ\, B}^{\gamma} \mathrel{-\!\!*} \Box \boxed{\circ\, B}^{\gamma}$$

$$\text{(Confront)} \qquad \boxed{\bullet\, K}^{\gamma} \mathrel{-\!\!*} \boxed{\circ\, B}^{\gamma} \mathrel{-\!\!*} B \in K$$

$$\text{(Update)} \qquad \boxed{\bullet\, K}^{\gamma} \mathrel{-\!\!*} \dot{\Rrightarrow} \left\{ \begin{array}{l} \boxed{\bullet\, (K; B)}^{\gamma} \; * \\ \boxed{\circ\, B}^{\gamma} \end{array} \right.$$

Figure 7: Logical rules for ghost state

execute. In particular, we must show that the program

$$e.\texttt{eval}\,\texttt{num}\,x \tag{5.2}$$

satisfies *some* specification. It follows from the reading of *ewp* that any specification suffices for proving safety.

The only way to reason about program 5.2 is to eliminate the assertion e *isExp* E. We must find a semiring $\mathcal{R}'$, a protocol Ψ' and a representation predicate *isNum*$'$ with which we instantiate the assertion e *isExp* E.

The first question one should ask is: what is the type of numerical values on which the program 5.2 operates? The answer is: it computes on intermediate variables (values of type `t`). We give the initial intermediate variable x and the dictionary `num` to the client and every time the client calls one of the arithmetic operators contained in the dictionary `num`, the handler replies with a new intermediate variable. The key to proposing suitable definitions of $\mathcal{R}'$, Ψ' and *isNum*$'$ is thus to find what is the meaning of an intermediate variable – to which semiring it is linked and how this link (the representation predicate) is defined.

It is not difficult to see that $\mathcal{R}'$ should be the free semiring of univariate expressions $Exp_{\{X\}}$. An intermediate variable represents an expression. The initial variable x should represent the expression *Leaf* X, the value `0` should represent *Zero*, and the value `I` should represent *One*. Moreover, the arithmetic operators contained in the dictionary `num` should construct an expression out of two subexpressions.

The formal definition of *isNum*$'$ is given below.

Definition 5.4 (*isSubExp*)**.** The predicate $\mathit{isSubExp} : \mathit{Val} \to \mathit{Exp}_{\{X\}} \to \mathit{iProp}$ is inductively defined as follows:

$$\begin{array}{rcl} u\ \mathit{isSubExp}\ \mathit{Zero} & = & u = \texttt{0} \\ u\ \mathit{isSubExp}\ \mathit{One} & = & u = \texttt{I} \\ u\ \mathit{isSubExp}\ (\mathit{Leaf}\ X) & = & u = x \\ u\ \mathit{isSubExp}\ (\mathit{Node}\ \mathit{op}\ E_1\ E_2) & = & \exists a, b.\, \boxed{\circ\, \mathsf{let}\ u = a\ \mathit{op}\ b}^{\gamma} \; * \\ & & \quad a\ \mathit{isSubExp}\ E_1 \; * \\ & & \quad b\ \mathit{isSubExp}\ E_2 \end{array}$$

The name *isSubExp* reflects the idea that intermediate variables represent subexpressions of E. The expressions *Leaf* X, *Zero* and *One* are represented by x, `0` and `I` respectively. An expression *Node* op E_1 E_2 must be the result of a call by the client to one of the arithmetic operators. Therefore, there must exist a pair of intermediate variables a and b representing E_1 and E_2. There must exist a corresponding item in the current context K: this item

records the result u of the call, the type op of the operation, and the intermediate variables a and b.

The next step is to propose a suitable protocol Ψ'. This protocol should answer the question: what is the *contract* upon which client and handler agree? What is the service on which the client relies and which the handler supplies? The client relies on effects to combine new subexpressions that are represented by intermediate variables. Following this idea, we conceive the protocol below:

Definition 5.5 (Protocol)**.**

$$\begin{array}{rlll} \Psi' = & ! \, op \; a \; b \; E_a \; E_b & (op, a, b) & \{a \; isSubExp \; E_a \; * \; b \; isSubExp \; E_b\}. \\ & ? \, u & (u) & \{u \; isSubExp \; (Node \; op \; E_a \; E_b)\} \end{array}$$

We have not yet encountered this kind of protocol: a *send/receive protocol* [dVP21a]. In this particular instance, Ψ' means that when performing an effect, the client must provide a pair of intermediate variables – each of which represents a subexpression – and, in return, the client receives an intermediate variable that represents the combination of the subexpressions represented by the arguments.

Next, we must prove that we chose these structures coherently, that ring, protocol, predicate and `num`, together, satisfy the relation *isNumDict*:

$$isNumDict \; (\texttt{num.zero}, \texttt{num.one}, \texttt{num.add}, \texttt{num.mul}, isSubExp, \Psi', Exp_{\{X\}})$$

that is, to prove that the dictionary `num` defined at lines 18–23 provides a correct implementation of the semiring $Exp_{\{X\}}$. The proof is straightforward. The values `num.zero` and `num.one` unfold to `0` and `I`. Therefore, they represent the expressions *Zero* and *One* by definition of *isSubExp*. It is also easy to see that `num.add` and `num.mul` satisfy their specifications. They perform an effect under the protocol Ψ', which guarantees that the answer supplied by handler is a representation of the node built from the subexpressions represented by the arguments.

Forward phase. We are now in a position to consider the execution of program 5.2. The specification that we obtain by specializing the assertion $e \; isExp \; E$ to the semiring $Exp_{\{X\}}$, protocol Ψ' and representation predicate *isSubExp* ensures that the effects that this program might throw abide by the protocol Ψ':

$$ewp \; (e\texttt{.eval num}\, x) \; \langle \Psi' \rangle \{y. \; \exists S. \; y \; isSubExp \; S \; * \; S \equiv_{Exp_{\{X\}}} E\} \tag{5.3}$$

The handler at line 28 monitors the client's execution and replies to its effects in a way that is also consistent with Ψ'. At the same time, the handler also learns on which arithmetic operations the client relies. This information is not explicitly stored by the handler, but during the verification we identify it as the state of the variable γ. Our goal is thus to prove that after the client's execution (on line 39), γ stores a context K that corresponds to the complete list of computations performed by the client. In other words, K must be a linear representation of E, the expression computed by the client.

The key idea to prove this property about the final state of γ is to introduce a logical assertion describing an intermediate state of γ. We call this assertion the *forward invariant* because it holds throughout the forward phase. The forward invariant describes the state of γ and how it relates to the state of the heap: it says that the intermediate variables defined in the current context K have indeed been allocated and their contents have a precise meaning in terms of K. Here is the definition of the forward invariant:

Definition 5.6 (Forward Invariant)**.**

$$
\begin{array}{l}
\mathit{ForwardInv}\ K\ = \\
\quad \boxed{\bullet\, K}^{\gamma}\ * \\
\quad \left\{ \begin{array}{l} \forall u \in \{x\} \cup \mathit{defs}(K). \\ \quad \mathit{leaves}(K[u]) \subseteq \{\mathtt{0}, \mathtt{I}, x\}\ * \\ \quad \exists\, m\, \mathscr{q}. \left\{ \begin{array}{l} u = \mathtt{Var}\,(m, \mathscr{q})\ * \\ m\ \mathit{isNum}\ [\![K[u]]\!]_{\varrho}\ * \\ \mathscr{q} \mapsto \mathtt{zero} \end{array} \right. \end{array} \right. \\
\quad \text{where } \varrho : \mathit{Val} \to \mathcal{R}\ =\ (\lambda\_.\, r)[\mathtt{0} := 0][\mathtt{I} := 1]
\end{array}
$$

We decompose the forward invariant into three main assertions:

(1) *State of γ.* The first assertion states that K is the current context.
(2) *Well-formedness of K.* The second main assertion states that filling K with any intermediate variable that has been allocated yields an expression whose leaves [12] are either `0`, `I` or x. Intuitively, this assertion expresses the idea that the client builds expressions out of the intermediate variables that are initially available: `0`, `I` and x.
(3) *Contents of the intermediate variables.* The third assertion is the one that starts with an existential quantifier on m and $\mathscr{q}$. It claims that each intermediate variable u is a `Var`-tagged pair of a value m and a memory location $\mathscr{q}$. The value m represents the meaning of $K[u]$ in the ring $\mathcal{R}$ under the environment ϱ. [13] The memory location $\mathscr{q}$ stores the value `zero`. The intuitive reading is that during the forward phase, the field `v` stores the value of the subexpression, while the field `d` stores `zero`.

It is easy to see that the forward invariant initially holds of the empty context. Indeed, we have established that the assertion $\boxed{\bullet\, []}^{\gamma}$ holds at the end of the initialization phase. Moreover, x is the only intermediate variable in existence at this point, and it satisfies the claims of the invariant because of Assertion 5.1. The proof that the invariant is preserved during the forward phase involves more intricate reasoning. We skip the details, but the idea is that each time the handler receives a request, we can assume that it is in a state described by the invariant *ForwardInv* K, for some context K. The goal is thus to prove that the assertion *ForwardInv* $(K; \mathsf{let}\ u = a\ \mathit{op}\ b)$ holds after the handler has updated γ with a new entry $\mathsf{let}\ u = a\ \mathit{op}\ b$.

Because the invariant holds initially, and because it is preserved, we can deduce that, at the end of the forward phase, the assertion *ForwardInv* K holds for some context K. Moreover, upon termination, the client returns an intermediate variable y that represents the complete expression E.

Transition phase. The purpose of the forward phase is to extract information out of the client – which arithmetic operations it needs and in which order. The purpose of the transition phase and of the backward phase is to compute derivatives. Indeed, when the evaluation of the client terminates, the algorithm updates the "derivative" field of y to `one` (line 40).

[12] The function $\mathit{leaves}(\_) : \mathit{Exp}_{\mathcal{I}} \to \mathcal{P}(\mathcal{I})$ computes the set of leaves of an expression. Its definition is omitted.

[13] We write $\varrho[a := r]$ for the function that maps a to r and behaves like ϱ everywhere else.

Backward phase. In the backward phase, the algorithm continues to update the derivative component of intermediate variables, and it does so in the reverse order of their appearance in K. Therefore, at each step, one can view K as the concatenation of two contexts K_1 and K_2. The context K_2 is the suffix of K containing the entries already processed during the backward phase, while K_1 is the prefix of K containing the remaining entries, so K is $K_1; K_2$.

We can assign meaning to the derivative component of intermediate variables in K_2. At any point during this phase, if the algorithm has processed a variable u, then its derivative component is the derivative of $K_2[u]$ with respect to the variable u itself. The intuition is that, by traversing the context K in reverse order, the algorithm traverses the expression E from the upmost node to its leaves and computes the contribution of each intermediate variable to the expression it knows so far.

To formalize these ideas, we again proceed by introducing an invariant: a logical description of the state that holds throughout the backward phase. We call it the *backward invariant*. It is parameterized by K_1 and K_2 and is defined as follows:

Definition 5.7 (Backward Invariant)**.**

$$
\begin{array}{l}
\mathit{BackwardInv}\ K_1\ K_2\ y\ = \\
\quad \mathsf{let}\ K = K_1; K_2\ \mathsf{in} \\
\quad [\![E']\!]_{(\lambda X.\, r)} \equiv_{\mathcal{R}} \partial(K[y])/\partial x\ (\varrho)\ * \\
\quad \left\{ \begin{array}{l} \forall u \in \{x\} \cup \mathit{defs}(K_1). \\ \qquad \exists\, d\, \mathscr{q}. \left\{ \begin{array}{l} u = \mathtt{Var}\,(\_, \mathscr{q})\ * \\ \mathscr{q} \mapsto d\ * \\ d\ \mathit{isNum}\ (\partial(K_2[y])/\partial u\ (\varrho\{K_1\})) \end{array} \right. \end{array} \right. \\
\quad \text{where } \varrho : \mathit{Val} \to \mathcal{R}\ =\ (\lambda\_.\, r)[\mathtt{0} := 0][\mathtt{I} := 1]
\end{array}
$$

We decompose the backward invariant into two main assertions:

(1) *Derivative with respect to x.* The first conjunct claims that filling K with y yields an expression whose derivative with respect to x (the initial variable) is $[\![E']\!]_{(\lambda X.r)}$.

(2) *Contents of the intermediate variables.* The second assertion is the claim that each intermediate variable u is a `Var`-tagged pair of a value and a memory location $\mathscr{q}$. The memory location $\mathscr{q}$ stores a value d that represents the derivative of $K_2[y]$ with respect to u under the environment ϱ extended with K_1.

During the backward phase, the invariant evolves as the algorithm processes the entries from K. For example, during an arbitrary run of the lines 32 and 33, we assume:

$$\mathit{BackwardInv}\ (K_1; \mathsf{let}\ u = a\ \mathit{Add}\ b)\ K_2\ y$$

and we must prove that at the end of line 33, the following assertion holds:

$$\mathit{BackwardInv}\ K_1\ (\mathsf{let}\ u = a\ \mathit{Add}\ b; K_2)\ y$$

The formal justification of this step relies on the properties of the derivative function $\partial\cdot/\partial\cdot\ (\cdot)$. For instance, we exploit a restricted form of the chain rule.

Finalization. Because the backward invariant holds at the beginning of the backward phase (after line 40) and because it is preserved by the updating instructions (lines 32–33 and 37–38), we can deduce that at the end of the backward phase (after the execution of line 27), the assertion $\mathit{BackwardInv}\ []\ K\ y$ holds. Therefore, in line 43, we can exploit the invariant to derive that the value d that is read from the derivative component of x is a representation of

$\partial(K[y])/\partial x\ (\varrho)$. Moreover, by the first assertion of the invariant, this term is equivalent to $[\![E']\!]_{(\lambda X.\ r)}$. So, if one chooses s to be $\partial(K[y])/\partial x\ (\varrho)$ (an element of the ring $\mathcal{R}$), then we can prove that the value d, the result of the algorithm, represents $[\![E']\!]_{(\lambda X.\ r)}$ up to equivalence. And the proof is complete.

5.3.1. *Technical remarks.* In order to keep the discussion lightweight, we have omitted the application of Hazel's formal reasoning rules. However, here we present the reasoning rule for handlers, in the interest of providing a complete documentation of the proof.

The rule for handlers is stated as follows:

$$\textsc{Try-With-Deep}\quad \frac{\begin{array}{c} \mathit{ewp}\ \texttt{client}\ \langle\Psi_1\rangle\{\phi_1\} \\ \mathit{deep\text{-}handler}\ \langle\Psi_1\rangle\{\phi_1\}\ \texttt{h} \mid \texttt{r}\ \langle\Psi_2\rangle\{\phi_2\} \end{array}}{\mathit{ewp}\ (\textbf{match}\ \texttt{client}\ \textbf{with}\ \texttt{h}\ \mid\ \texttt{r})\ \langle\Psi_2\rangle\{\phi_2\}}$$

The variables `h` and `r` stand for the effect and return branches of the handler. The rule states that the correctness of the combination of client and handler can be established separately as long as they agree on a protocol Ψ_1 and a postcondition ϕ_1. Indeed, the rule enables the verification of client plus handler given (1) a specification of the client and (2) that the assertion *deep-handler*, the *handler judgement*, holds. The handler judgement states the correctness of the handler. It is a concise way to write the specification of a handler. It does not depend on the client; it depends only on the logical interface declared by the client: the protocol Ψ_1 and the postcondition ϕ_1.

During the proof of `diff`, we apply rule Try-With-Deep to establish a specification of the expression comprised between lines 28 and 40:

$$\mathit{ForwardInv}\ [] \mathrel{-\!\!*} \mathit{ewp}\ (\text{lines } 28\text{–}40)\ \langle\Psi\rangle\{\_.\ \exists K\ y.\ \mathit{BackwardInv}\ []\ K\ y\}$$

It is the assertion that the forward invariant holds at the beginning of the forward phase and that the backward invariant holds at the end of the backward phase.

We fulfill the first requirement of the rule, a specification for the client, with Assertion 5.3.

We fulfill the second requirement of the rule, the proof that the assertion *deep-handler* holds, by proving a more general statement where we replace the parameter of the forward invariant with an abstract context K_1:

$$\forall K_1.\ \mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \left\{\begin{array}{l} \mathit{deep\text{-}handler}\ \langle\Psi'\rangle\ \{y.\ \exists S.\ y\ \mathit{isSubExp}\ S\ *\ S \equiv_{\mathit{Exp}_{\{X\}}} E\} \\ \qquad (\text{lines } 29\text{–}38) \mid (\text{lines } 39\text{–}40) \\ \qquad \langle\Psi\rangle\ \{\_.\ \exists K_2\ y.\ \mathit{BackwardInv}\ K_1\ K_2\ y\} \end{array}\right.$$

The proof follows by Löb's induction which reflects the recursive behavior of deep handlers.

6. Related Work

We divide the related work into two categories: (1) the conception of methods for reasoning about effect handlers and (2) the study of algorithms for automatic differentiation.

Reasoning about effect handlers. Goodenough [Goo75] provides a meticulous study of resumable exceptions, a mechanism that allows the exception handler to resume the suspended computation. This programming feature can be seen as a restricted form of an effect handler where the call to the continuation lies on the scope of the effect branch. Goodenough describes the existing exception handling methods of the time and compares them according to different criteria – implementation difficulty, efficiency, readability, etc. His discussion on resumable exceptions shows a precise understanding of their operational behavior.

In its full generality, an effect handler can be described as a delimited control operator, that is, an effect handler extends a language with the ability to capture a delimited continuation and to reify it as a first-class value. Before Plotkin and Pretnar's seminal work [PP09], which introduced effect handlers, a lot of study was already dedicated to other delimited control operators. For instance, Filinski [Fil96] shows that many of these operators can be expressed in terms of `reify` and `reflect`, two novel programming constructs that he introduces to leverage the simulation of effects in a pure language. Intuitively, the `reflect` construct allows the programmer to introduce an effect in direct-style by giving its specification as a term in a monad. The `reify` construct then translates a program to a monadic expression by transforming let-bindings into monadic binds. Filinski also shows how these constructs can be implemented using `call/cc` and mutable state and justifies his claims through a logical relations argument.

The idea that effects are means to construct computations and that to evaluate a program one must deconstruct such computations is one that will be often revisited. For instance, this idea is at the core of Plotkin and Pretnar's work [PP09], where the authors introduce a denotational semantics for a language with effect handlers. This semantics allows one to think of a computation as a tree whose nodes are effectful operations, and to think of an effect handler as a *deconstructor* of such computations: an effect handler traverses the tree and substitutes each effectful operation with an implementation. Their denotational semantics is justified only when handlers are correct: the handler must satisfy the equations of the algebraic theory associated to an effect. The authors also show how to adapt their previous equational logic [PP08] to account for effect handlers. This logic allows one to state and prove that two programs are equivalent. Once such a logical judgement is proven, the soundness statement of the logic implies equality between the denotational interpretations of each program.

Xia et al. [XZH+20] build a Coq library, ITrees, which defines a coinductive data structure, *interaction trees*. An interaction tree is a possibly infinite tree-like structure whose nodes are either effectful operations or silent reduction steps. Handlers act on interaction trees by providing an interpretation of the operations into an user-defined monad. On the Related Work Section, Paragraph 8.2, the authors observe that the current library does not support handlers in their most general form. In particular, the handler does not have access to the continuation.

Brady [Bra13b] presents Effects, a programming language embedded in Idris [Bra13a], where the type of a program includes a list of the effects that this program might perform. An interesting feature is that this list of effects can be locally extended: the programmer can introduce new effects locally and they won't interfere with the type of the complete program. Effect handlers are declared at the top level through Idris's type class mechanism.

On much of the work previously discussed the focus was on reasoning about the mathematical model of programs with handlers: Plotkin and Pretnar's equational logic allows

one to establish that two programs have the same denotation; Interaction trees is a purely mathematical structure that can be the target of a denotational semantics. Another approach is to reason directly about programs with handlers through contextual equivalence: one wishes to establish when two implementations of a program can be exchanged without affecting the complete program.

Biernacki et al. [BLP20] show that contextual equivalence in the setting of a restricted and untyped programming language with handlers is equivalent to bisimilarity: two programs are equivalent if and only if their execution traces are related by a bisimulation. To simplify the proof of bisimilarity results, the authors propose *up-to techniques*, which they illustrate through a number of simple examples. However, it remains to see if this verification methodology scales to the setting of a major programming language.

In the context of a typed language, one can reason about contextual equivalence by means of logical relations. Biernacki et al. [BPPS18] present the first logical relations model of a type system with support for effect handlers. The type system tracks effects through rows of effects and also has support for effect polymorphism. The logical relations model allows proving equivalences between two programs that might exploit handlers. The paper provides an interesting example: to supply a constant function to a higher-order function f is equivalent to supply f with an effect which we interpret as a lookup operation to a constant variable. Later, Biernacki et al. [BPPS20] propose a logical relations model of a type system with support for *lexically-scoped handlers*, a restricted kind of effect handler where generating a fresh effect name and installing an effect handler are combined into a single operation. This restriction eases reasoning about the dynamic behavior of handlers because with lexically-scoped handlers one has the static guarantee of which handler is invoked when an effect is performed.

Our work differs from the papers cited above in two aspects: (1) we consider a programming language with both effect handlers and dynamically-allocated mutable state and (2) we propose a Hoare-style logic, that is, we propose a method to write the specification of a program as a logical description of its behavior. We believe that Hoare-style specifications lead to program documentations that are easier to understand. Moreover, we believe that Separation Logic leads to powerful reasoning principles about heap-modifying programs. This article confirms our beliefs: we provide a simple specification and a machine-checked proof of correctness for a piece of code that involves higher-order functions, dynamically-allocated mutable state and effect handlers. And our proof is compositional: we prove that the library works correctly in the presence of an arbitrary client, provided the client respects the requirements imposed by our specification.

Study of automatic differentiation. Krawiec et al. [KKP+22] study RMAD from a denotational semantics point of view. They introduce a pure higher-order calculus that includes standard constructs such as sums and pairs as well as real numbers and arithmetic operations on real numbers, such as addition and multiplication. The calculus is endowed with a type system such that a well-typed program denotes a multivariate mathematical function from reals to reals. One of the contributions of the paper is to present several source-to-source transformations that take a well-typed program as input and construct a program that denotes the *derivative* of the function denoted by the original program. (The meaning of derivative is defined as an operation over the Jacobian of the function denoted by the original program.) These transformations follow different strategies (forward-mode or reverse-mode) and have different time and space requirements. The main contribution

of the paper is the proof that one of these transformations, arguably the most intricate one, is correct. The proof is informal and justifies the correctness claim only in the case where programs are interpreted as real-valued functions. Later on, the paper proposes generalizing both the semantics and the program transformations to an arbitrary type instead of the fixed domain of real numbers. However, the correctness claim for the general case remains unjustified. The paper also discusses informally how one could turn the theoretical presentation into a library written in a functional programming language such as Haskell or OCaml.

Pearlmutter and Siskind [PS08] introduce VLAD, a calculus that is capable of expressing functional, pure programs. VLAD includes a first-class construct $\overleftarrow{\mathcal{J}}$ to compute derivatives, which allows programs to take higher-order derivatives. The method to effectively compute derivatives is defined on paper as a source-to-source transformation: given a VLAD program as input, the method produces a VLAD program that does not use $\overleftarrow{\mathcal{J}}$. The paper announces a prototype implementation of an interpreter for VLAD, named Stalin∇. To handle $\overleftarrow{\mathcal{J}}$, Stalin$\nabla$ relies on the interpreter's ability to reflectively inspect programs at runtime.

Sherman et al. [SMC21] focus on expressiveness: they wish to extend the applicability of automatic differentiation. They introduce λ_S, a programming language that includes higher-order functions, higher-order derivatives, and constructs for integration, root-finding and optimization. To assign meaning to λ_S programs, the authors follow a denotational approach. In doing so, they must solve the problem of interpreting programs, such as `max` and `min`, whose denotations are not differentiable everywhere. Their key idea is to generalize the notion of derivative to *Clarke derivatives*, which are defined even at points where a function is not differentiable in the traditional sense. Therefore, automatic differentiation of λ_S programs is the computation of their Clarke derivatives. The paper discusses the implementation of a Haskell library to compute the Clarke derivatives, with arbitrary precision, of programs written in a Haskell embedding of λ_S.

An efficient implementation of effect handlers, with support for one-shot continuations only, has appeared in Multicore OCaml [SDW+21]. The Multicore OCaml code that we present (Figure 5) is inspired by Wang et al. [WR18, WZD+19] and by Sivaramakrishnan [Siv18]. However, the manner in which we package reverse-mode automatic differentiation as a library (Figure 1), using a tagless final representation of expressions [CKS09, Kis10], seems new. This minimalist library is remarkable insofar as it offers a very simple, pure interface, yet its implementation is arguably rather subtle and involves dynamically-allocated mutable state, higher-order functions, and effect handlers. Another implementation of reverse-mode automatic differentiation, written in Frank, is documented by Jesse Sigal [Sig21]. It differs from ours in several aspects. First, Frank does not have primitive mutable state, so it is simulated via effect handlers. Second, Sigal presents several automatic differentiation algorithms, including a forward-mode algorithm and a reverse-mode algorithm that performs checkpointing.

7. Conclusion

In this paper, we have verified a concise library for reverse-mode automatic differentiation. The code exploits dynamically-allocated mutable state, higher-order functions, and effect handlers. We have proposed an original API for this library as a transformation of expressions in tagless final style (Figure 1). We have written a specification of the library in Hazel [dVP21a], a variant of higher-order Separation Logic (§5.2), and provided a machine-checked proof

of the correctness of the code with respect to this specification (§5.3). We view this as a nontrivial exercise in modular program verification and an illustration of the power of Separation Logic, in the presence of mutable state, higher-order functions, and effect handlers. To the best of our knowledge, this is a first: outside of the authors' previous paper [dVP21a], no correctness proofs for programs that involve primitive mutable state and primitive effect handlers have appeared in the literature.

Our specification of `diff` illustrates how effects are described in Hazel. According to Statement 5.1, `diff` itself performs no effects: it is a function from expressions to expressions. According to Definition 5.2, an "expression" in tagless final style is a computation that is parameterized with a dictionary of arithmetic operations. These arithmetic operations may perform unknown effects, represented by an abstract protocol Ψ: the expression is not allowed to handle these effects or to perform any effects of its own. It can perform effects internally, but if it does so, then it must handle them, so they are not observable.

By virtue of working in Separation Logic, we naturally reap the benefits of modular reasoning. The verified function `diff` can be safely combined with other software components, provided they respect the expectations expressed by the specification of `diff`. The use of mutable state and effect handlers inside `diff` cannot interact in unexpected ways with these foreign components. As an example of particular interest, the specification of `diff` allows composing `diff` with itself: it is clear (and one can easily verify) that `fun e -> diff (diff e)` computes a second-order derivative. This is a nontrivial result: it would be very difficult to reason operationally about the manner in which mutable state and effect handlers are used when the expression `diff (diff e)` is evaluated.

By now, many programmers are familiar with the fundamental principles of Hoare logic and Separation Logic: they know (at least informally) that one reasons about a loop with the help of a loop invariant, and that one reasons about a function through a precondition and a postcondition. We would like this paper to popularize the idea that one reasons about an effect through a protocol, a pair of a precondition and a postcondition, which represents a contract between the client (which performs an effect) and the handler (which handles this effect). Our proof illustrates both client-side and handler-side reasoning. On the client side, performing an effect is akin to calling a function: the protocol describes the pre- and postcondition of the effect. On the handler side, naturally, the postcondition of the effect becomes the precondition of the captured continuation. Less obviously, because a deep handler is sugar for a shallow handler wrapped in a recursive function [HL18], verifying a deep handler requires spelling out the pre- and postcondition of this recursive function: in our proof, the forward invariant (Definition 5.6) and the backward invariant (Definition 5.7) play these roles. Finally, when writing down the protocol that the client and handler obey, it is often necessary to introduce custom Separation Logic assertions, whose definition may involve ghost state. In our proof, the predicate *isSubExp* (Definition 5.4), which appears in the protocol that describes the effects `Add` and `Mul`, is defined in terms of a ghost history K of the past effects. Ghost history variables are common in proofs of concurrent and distributed algorithms [AL88, LM17]; here, although the code is sequential, the client and the handler form two distinct logical "threads", so it should not be surprising that the need for a history variable appears.

Our work is limited in several ways. Because our emphasis is on program verification techniques, as opposed to automatic differentiation techniques, the code that we verify has been distilled to the simplest possible form. It is limited to expressions that involve one variable, two constants (zero and one) and two primitive arithmetic operations (addition and

multiplication), and it is not at all optimized for efficiency. We see no obstacle in principle to supporting multiple variables and a richer set of primitive arithmetic operations. In future work, it would be interesting to investigate which real-world AD libraries rely on effect handlers and what it would take to verify these libraries. Another caveat about our work is that there exists a gap between the code that we present in the paper and the code that we verify. While the code that we present (Figure 5) is written in Multicore OCaml 4.12.0, the code that we verify is expressed in *HH*, a λ-calculus with mutable state and effect handlers, whose syntax and operational semantics are defined in Coq. The main difference between them is that the declarations **`effect Add`** and **`effect Mul`** in Multicore OCaml generate fresh effect names at runtime, whereas *HH* does not have fresh name generation nor effect names, so we encode `Add` and `Mul` in *HH* using left and right injections. We believe that this difference should not fundamentally influence the manner in which one reasons about effect handlers. To provide evidence for this belief, in future work, we would like to propose a Separation Logic that allows reasoning about effect handlers in the presence of multiple effect names and dynamic generation of fresh effect names. This is currently an active area of research [BPPS18, BPPS19, BPPS20, ZM19, BSO20b], where the final word has not yet been said.

References

[AL88] Martin Abadi and Leslie Lamport. The existence of refinement mappings. In *Logic in Computer Science (LICS)*, pages 165–175, July 1988.

[BLP20] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. A complete normal-form bisimilarity for algebraic effects and handlers. In *Formal Structures for Computation and Deduction (FSCD)*, volume 167 of *LIPIcs*, pages 7:1–7:22, 2020.

[BP15] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[BP20] Andrej Bauer and Matija Pretnar. Eff. `http://www.eff-lang.org/`, 2020.

[BPPS18] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages*, 2(POPL):8:1–8:30, 2018.

[BPPS19] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):6:1–6:28, 2019.

[BPPS20] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages*, 4(POPL):48:1–48:29, 2020.

[Bra13a] Edwin Brady. Idris, a general purpose dependently typed programming language: design and implementation. *Journal of Functional Programming*, 23(5):552—-593, 2013.

[Bra13b] Edwin C. Brady. Programming and reasoning with algebraic effects and dependent types. In *International Conference on Functional Programming (ICFP)*, pages 133–144, September 2013.

[BSO20a] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):126:1–126:30, 2020.

[BSO20b] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming*, 30:e8, 2020.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[DEH+17] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In

Trends in Functional Programming (TFP), volume 10788 of *Lecture Notes in Computer Science*, pages 98–117. Springer, June 2017.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 151–160, 1990.

[DG05] Olivier Danvy and Mayer Goldberg. There and back again. *Fundamenta Informaticæ*, 66(4):397–413, 2005.

[DMS20] Stephan Dolan, Anil Madhavapeddy, and KC Sivaramakrishnan. Multicore OCaml. https://github.com/ocaml-multicore/ocaml-multicore/wiki, 2020.

[dVP21a] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021.

[dVP21b] Paulo Emílio de Vilhena and François Pottier. Verifying a minimalist reverse-mode AD library: Coq formalization. https://gitlab.inria.fr/pdevilhe/hazel, 2021.

[dVP21c] Paulo Emílio de Vilhena and François Pottier. Verifying a minimalist reverse-mode AD library: Guide to the Coq formalization. https://gitlab.inria.fr/pdevilhe/hazel/-/blob/master/papers/LMCS-RMAD.md, 2021.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages (POPL)*, pages 180–190, January 1988.

[Fil96] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1996.

[FKLP19] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Journal of Functional Programming*, 29:e15, 2019.

[Goo75] John B. Goodenough. Structured exception handling. In *Principles of Programming Languages (POPL)*, pages 204–224, January 1975.

[GW08] Andreas Griewank and Andrea Walther. *Evaluating derivatives – principles and techniques of algorithmic differentiation, Second Edition*. SIAM, 2008.

[HL18] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, December 2018.

[HLA20] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30:e5, 2020.

[HSV21] Mathieu Huot, Sam Staton, and Matthijs Vákár. Higher order automatic differentiation of higher order functions. https://arxiv.org/abs/2101.06757, 2021.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[Kis10] Oleg Kiselyov. Typed tagless final interpreters. In *International Spring School on Generic and Indexed Programming (SSGIP)*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, March 2010.

[Kis12] Oleg Kiselyov. Beyond Church encoding: Boehm-Berarducci isomorphism of algebraic data types and polymorphic lambda-terms. http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html, April 2012.

[KKP+22] Faustyna Krawiec, Neel Krishnaswami, Simon Peyton-Jones, Tom Ellis, Andrew Fitzgibbon, and Richard A. Eisenberg. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022.

[Lei14] Daan Leijen. Koka: Programming with row polymorphic effect types. In *Workshop on Mathematically Structured Functional Programming (MSFP)*, volume 153, pages 100–126, April 2014.

[Lei20] Daan Leijen. Koka. https://www.microsoft.com/en-us/research/project/koka/, 2020.

[LM17] Leslie Lamport and Stephan Merz. Auxiliary variables in TLA+. *CoRR*, abs/1703.05121, 2017.

[LMM17] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *Principles of Programming Languages (POPL)*, January 2017.

[LS79] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6):546–558, 1979.

[O'H19] Peter W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[PP08] Gordon D. Plotkin and Matija Pretnar. A logic for algebraic effects. In *Logic in Computer Science (LICS)*, pages 118–129, June 2008.

[PP09] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, March 2009.

[Pre15] Matija Pretnar. An introduction to algebraic effects and handlers. In *Mathematical Foundations of Programming Semantics*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, June 2015.

[PS08] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems*, 30(2):7:1–7:36, 2008.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[RMO09] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS transform. In *International Conference on Functional Programming (ICFP)*, pages 317–328, September 2009.

[RS03] Barbara G. Ryder and Mary Lou Soffa. Influences on the design of exception handling. *ACM SIGSOFT Software Engineering Notes*, 28(4):29–35, 2003.

[SDW+21] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Programming Language Design and Implementation (PLDI)*, pages 206–221, June 2021.

[Sig21] Jesse Sigal. Automatic differentiation via effects and handlers: An implementation in Frank. https://arxiv.org/abs/2101.08095, 2021.

[Sit93] Dorai Sitaram. Handling control. In *Programming Language Design and Implementation (PLDI)*, pages 147–155, June 1993.

[Siv18] KC Sivaramakrishnan. Reverse-mode algorithmic differentiation using effect handlers. https://github.com/ocaml-multicore/effects-examples/blob/master/algorithmic_differentiation.ml, February 2018.

[SMC21] Benjamin Sherman, Jesse Michel, and Michael Carbin. λ_S: Computable semantics for differentiable programming with higher-order functions and datatypes. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, 2021.

[TB19] Nicolas Troquard and Philippe Balbiani. Propositional dynamic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2019.

[WR18] Fei Wang and Tiark Rompf. A language and compiler view on differentiable programming. In *International Conference on Learning Representations (ICLR), Workshop Track*, 2018.

[WZD+19] Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP):96:1–96:31, 2019.

[XZH+20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):51:1–51:32, 2020.

[ZM19] Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proceedings of the ACM on Programming Languages*, 3(POPL):5:1–5:29, 2019.