Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library

Paulo Emílio de Vilhena, François Pottier

Logical Methods in Computer Science
Volume 19, Issue 4, 2023, pp. 5:1–5:51
https://lmcs.episciences.org/

Submitted Dec. 15, 2021
Published Oct. 23, 2023

VERIFYING AN EFFECT-HANDLER-BASED DEFINE-BY-RUN REVERSE-MODE AD LIBRARY

PAULO EMÍLIO DE VILHENA [a] AND FRANÇOIS POTTIER [b]

[a] Imperial College London, United Kingdom
e-mail address: p.de-vilhena@imperial.ac.uk

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

Abstract. We apply program verification technology to the problem of specifying and verifying automatic differentiation (AD) algorithms. We focus on define-by-run, a style of AD where the program that must be differentiated is executed and monitored by the automatic differentiation algorithm. We begin by asking, "what is an implementation of AD?" and "what does it mean for an implementation of AD to be correct?" We answer these questions both at an informal level, in precise English prose, and at a formal level, using types and logical assertions. After answering these broad questions, we focus on a specific implementation of AD, which involves a number of subtle programming-language features, including dynamically allocated mutable state, first-class functions, and effect handlers. We present a machine-checked proof, expressed in a modern variant of Separation Logic, of its correctness. We view this result as an advanced exercise in program verification, with potential future applications to the verification of more realistic automatic differentiation systems and of other software components that exploit delimited-control effects.

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."* Griewank and Walther's textbook [GW08] offers a comprehensive introduction to AD; Baydin et al. [BPRS18] survey its applications in machine learning.

There are two main families of AD *algorithms*, namely the *forward-mode* and *reverse-mode* algorithms. As noted above, a program that one wishes to differentiate performs a sequence of elementary arithmetic operations. A forward-mode algorithm processes this sequence in order: the earliest arithmetic operation is examined first. A reverse-mode algorithm processes it in reverse order: the earliest arithmetic operation is examined last. Reverse

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

DOI:10.46298/LMCS-19(4:5)2023

mode is attractive when differentiating a function whose number of outputs is several orders of magnitude smaller than the number of its inputs, such as a function of type $\mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$ [GW08, Chapter 1].

As an independent distinction, one can also identify several families of *programming-language techniques* that allow deploying AD in practice. One broad class of techniques relies on *program transformation*. The source code of the program P that one wishes to differentiate is supplied to an automatic differentiation tool, a special-purpose compiler, which produces source code for a differentiated program P'. Another broad class relies on *program monitoring*. In this approach, no source code for P is needed, and no source code for P' is produced. When the execution of P' is requested, the program P is executed instead. Its execution is monitored in such a way that the sequence of elementary arithmetic operations performed by P can be observed and the value of P' can eventually be computed. In some communities, implementations of AD based on program transformation are referred to as *define-then-run*, because the source code of P' is constructed in a separate phase, before P and P' are compiled and run, whereas implementations based on program monitoring are known as *define-by-run*, because they do not involve such a phase separation. These approaches have different pragmatic strengths and weaknesses. Because they have access to and rely on the syntax of the source program, the program-transformation approaches offer more scope for optimization, therefore potentially greater efficiency, but require greater implementation effort and are limited to a fixed programming-language subset. The program-monitoring approaches are usually less efficient but can be easier to implement and are not restricted to a fixed set of syntactic constructs.

In this paper, we wish to use program verification technology to *specify* and *verify* an implementation of automatic differentiation. That is, we wish to answer two main questions. Our first question is in fact two-pronged: (1A) *what is an implementation of AD?* and (1B) *what does it mean for an implementation of AD to be correct?* In other words, we ask what is the *type* and what is the *specification* of an AD implementation. Our second question is, (2) *how does one prove that a specific implementation of AD is correct?*

These are broad questions. Depending on which family of AD algorithms and which kind of implementation technique are considered, the difficulty of answering these questions may vary. In the following, we narrow down the scope of these questions and focus on specific situations where we are able to propose original answers to these questions.

Specifying Define-By-Run AD. We believe that, in the program-transformation approach, also known as define-then-run, the above questions are by now quite well understood. In this approach, an implementation of AD is a compiler. In other words, it is a function of type `exp -> exp`, where `exp` is an algebraic data type of abstract syntax trees. Such a compiler is correct if, when it is applied to an abstract syntax tree whose denotation is a mathematical expression E, it produces an abstract syntax tree whose denotation is E', the mathematical derivative of E. Admittedly, this high-level statement is somewhat sketchy and glosses over some details: for instance, we have implicitly assumed that the expression E has one free variable X, and we have written E' for the derivative of E with respect to X. Also, we have not defined the type `exp` of programmatic expressions or the type of mathematical expressions E. Nevertheless, this suggests how the correctness of an AD program transformation scheme can be stated. Many such schemes are proposed and proved correct in the literature; we discuss some of them at the end of the paper (§9).

For this reason, throughout this paper, we focus on the program-monitoring approach, also known as define-by-run. In this setting, our first contribution is an original answer to Question 1A. We remark that, *in the program-monitoring approach, an implementation of AD can still be viewed as a function of type* `exp -> exp`, as in the previous paragraph, *albeit under a different definition of the type* `exp`. Indeed, we remark that the *Church-Böhm-Berarducci encoding*, also known as the *tagless-final* representation, is a suitable definition of `exp` for this purpose. This representation does not allow inspecting the source code of a program, but does allow executing a program and monitoring its execution, to a certain extent. This is the topic of Section §1.

Under this definition of the type `exp`, a function `diff : exp -> exp` should not be viewed as a compiler: it does not consume or produce source code. It is an ordinary function, which can be compiled once and for all, placed in a library, and invoked by user programs that wish to exploit its functionality.

Building upon this approach, our next contribution is an original answer to Question 1B. We propose a formal specification for an implementation of define-by-run AD, that is, for a function `diff : exp -> exp`. We give this specification exactly the same general form as the correctness statement that was sketched earlier when discussing define-then-run AD: *when* `diff` *is applied to a value of type* `exp` *that denotes a mathematical expression* E*, it must produce a value of type* `exp` *that denotes the mathematical expression* E'. The crux is to define, under our new definition of the type `exp`, what it means for a runtime value of type `exp` to denote a mathematical expression E. This is where our second contribution lies, and this is the topic of Section §2.

Our specification is expressed in higher-order Separation Logic [Rey02, O'H08, Cha20, BB20]. Separation Logic is compositional: once the specification of a library has been fixed, an implementation of this library and a program that uses this library can be independently verified. Thus, an implementation of `diff` is verified with respect to the specification stated in the previous paragraph, without any knowledge of how `diff` might be used in a client program. Conversely, a client program is verified based only on this specification, without any knowledge of how `diff` is implemented. For example, under the assumption that `diff` satisfies this specification, it is easy to verify that if `e` denotes E then `diff (diff e)` denotes E'', the second derivative of E. In other words, it is immediately obvious that *every correct implementation of* `diff` *supports iterated differentiation*. This is a strong guarantee, which would be difficult to obtain in a naive way, by studying a specific implementation of `diff` and by attempting to understand what happens at runtime when two invocations of this function are nested.

At this point, a reader may ask whether our answers to Question 1A and Question 1B– that is, our proposed type and specification – are reasonable. To begin convincing this reader, we propose three minimalist implementations of AD, each of which fits in one page or less. They include a forward-mode implementation, based on dual numbers (Figure 4); a reverse-mode implementation that exploits a stack data structure (Figure 7); and a reverse-mode implementation that exploits *effect handlers* [Pre15] and does not explicitly involve a stack (Figure 10). This is the topic of Section §3. Each of these implementations has type `exp -> exp`, suggesting that this type is indeed a plausible answer to Question 1A.

Is our proposed specification also a plausible answer to Question 1B? To substantiate this claim, it would be desirable to verify that all three toy implementations are correct with respect to this specification. However, presenting three proofs would be space-consuming. Furthermore, the first two implementations use well-known techniques, so a reader who is

acquainted with AD should easily be convinced (at an informal level) that they are correct. For this reason, we narrow down the scope of our investigation and focus on the third implementation, which may seem more exotic. How does one verify that it is correct?

Verifying Effect-Based Define-By-Run Reverse-Mode AD. This question is a specific instance of Question 2. Answering this question is the third contribution of this paper. *We present a machine-checked proof that our effect-based implementation of AD is correct with respect to our specification of AD.* This proof is carried out in a variant of higher-order Separation Logic.

Our effect-based implementation of AD is inspired by Wang and Rompf [WR18] and by Wang et al. [WZD+19], who implement reverse-mode AD using dynamically allocated mutable state and the delimited-control operators `shift` and `reset` [DF90]. It is inspired also by Sivaramakrishnan [Siv18] and by Sigal [Sig21], who replace `shift/reset` with *effect handlers* [Pre15], a more structured form of delimited control.

Wang et al. [WZD+19] publish detailed descriptions of several versions of their code, but no proof of its correctness. In fact, 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"*. Although their code is indeed concise and clearly structured (and, we believe, so is our code in Figure 10), we object that it involves a combination of several nontrivial programming-language features, including first-class functions, dynamically allocated mutable state, and delimited control. Therefore, it is not at all obvious how and why this kind of code works. In particular, we argue that our code (Figure 10) is a concise and elegant implementation that deserves to be verified with respect to the simple specification that we have proposed.

To the best of our knowledge, this program verification challenge has not been addressed to this day, and seems highly nontrivial. To begin with, the literature offers very few proofs of programs that involve effect handlers, and virtually no proofs of programs that involve both effect handlers and primitive dynamically allocated mutable state. Furthermore, because, in our setting, `exp` is a second-order function type, `diff : exp -> exp` is a function of order three, which means that the dialogue between `diff` and the outside world is particularly complex. Yet, our proposed specification is particularly simple: when applied to a value that denotes E, `diff` must return a value that denotes E'. This specification allows no visible side effect: it states that `diff` must behave like a pure function. Thus, we face the challenge of proving that the use of mutable state and delimited control is encapsulated and cannot be observed by a user of `diff`.

Addressing this challenge is the third and main contribution of this paper. For greater readability, the code that is presented in the paper is expressed in a real-life programming language, namely Multicore OCaml 4.12.0 [SDW+21], an experimental extension of OCaml with effect handlers.[1] Unfortunately, OCaml is a large programming language, whose semantics is only loosely defined. For this reason, we cannot reason directly about the code in Figure 10. Instead, we introduce *HH*, a core λ-calculus equipped with effect handlers, whose syntax and semantics are defined inside the Coq proof assistant. We manually transcribe the code of Figure 10 into *HH*. (This manual transcription step is unverified. The resulting *HH* code is not shown in the paper.) Then, we use Hazel [dVP21], a Separation Logic for *HH*, to

[1] As of version 5, effects and effect handlers have been integrated in mainstream OCaml, albeit in the form of primitive functions: no syntactic constructs have been introduced in order to perform or handle effects.

express our specification of define-by-run AD and to verify that our *HH* code is correct with respect to this specification. Hazel, which we have introduced in previous work [dVP21], is built on top of the Iris framework [JKJ+18, KJJ+18], and includes support for reasoning about effect handlers. Hazel is defined inside Coq. The soundness of its reasoning rules is machine-checked. Our use of these reasoning rules is also machine-checked, so, in the end, we obtain a fully machine-checked proof of the correctness of our *HH* code.

An electronic supplement to this paper [dVP22a] offers the definition of the language *HH*, the definition and (machine-checked) soundness proof of Hazel, and the (machine-checked) correctness proof of our *HH* code. It also includes a short description of the correspondence between the Coq definitions and the paper [dVP22b]. The gap between Multicore OCaml and *HH* is discussed at the end of the paper (§10).

For the sake of simplicity, throughout the paper, we restrict our attention to expressions of one variable, and perform differentiation with respect to this variable. Generalizing our ideas to handle expressions of several variables is possible and should be straightforward, but would add a certain amount of clutter to the code, definitions, and proofs shown in this paper; we prefer to avoid it.

Summary. We view this paper primarily as an advanced exercise in program specification and verification in the presence of first-class functions, dynamically allocated mutable state, and delimited-control effects. We hope that it may also offer insights to readers who are interested in AD and in the connection between AD and delimited-control effects.

Outline. The paper is organized as follows. We propose a minimalist API (that is, a type) for a define-by-run AD library (§1). Then, we propose a specification for such a library (§2). We present three implementations of this API (§3). We briefly introduce effect handlers (§4) and explain how they are exploited in our third implementation of AD (§5). In preparation for the verification of this code, we recall a few basic properties of mathematical expressions and differentiation (§6), and briefly present a Separation Logic equipped with support for reasoning about effects and handlers (§7). Thus equipped, we offer a step-by-step presentation of the proof of our main result (§8). We review the related work (§9) and conclude (§10).

1. A Type for Define-By-Run AD

In this section, we propose a minimalist API for an AD library and describe a few examples of programs that use this API to construct and differentiate expressions.

1.1. The API.

Our proposed API appears in Figure 1. It is expressed as an OCaml module signature. It begins with the definitions of two types, `'v dict` and `exp`. Then, it requires the differentiation algorithm to be presented as a function `diff` of type `exp -> exp`. This API prescribes what functionality must be offered by an AD library, but does not mandate how the library must be implemented: as we will see (§3), a variety of algorithmic techniques and programming-language features can be used in an implementation of `diff`.

The type of `diff` seems easy to read: differentiation transforms an expression into an expression. For the reader to fully understand this type, there remains for us to explain the definition of the type `exp`, thereby answering the questions: what is a mathematical expression? What is the machine representation of a mathematical expression?

```
(* A record of the ring operations over a numeric type 'v. *)
type 'v dict =
  { 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 dict -> 'v -> 'v }

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

Figure 1. An OCaml API for define-by-run AD

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

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

(* 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 API

```
(* The expression x^k. *)
let monomial (k : int) : exp = { eval =
  fun (type v) { one; mul; _ } (x : v) ->
    let ( * ) = mul in
    let result, x, k = ref one, ref x, ref k in
    while !k > 0 do
      result := !result * (if !k mod 2 = 0 then one else !x);
      x := !x * !x;
      k := !k / 2
    done;
    !result }
```

Figure 3. Another example use of the API: the function `monomial`

In this paper, we consider mathematical expressions that are constructed out of the constants 0 and 1, addition, multiplication, and a single variable x. How might these mathematical expressions be represented in memory? As noted earlier, a natural idea might be to represent them as abstract syntax trees. One would do this by declaring `exp` as an algebraic data type. However, that would lead to an API for *define-then-run* AD, that is, for an implementation of AD as a program transformation, accepting source code and producing source code. In this paper, instead, we wish to focus on *define-by-run* AD, a variant of AD that relies on a form of program monitoring. To do so, we adopt an alternative representation of expressions, namely the Church-Böhm-Berarducci encoding [Kis12], also known as the tagless-final representation [CKS09, Kis10]. Remarkably, provided we define the type `exp` in this way, we can keep the convention that `diff` has type `exp -> exp`. That is, the type `exp -> exp` can describe both define-then-run AD and define-by-run AD!

In the tagless-final representation, *an expression is represented as a computation.* Such a computation is given access to the four operations (zero, one, addition, multiplication) and to the value of the single variable and must produce a value. This is visible in Figure 1, where an expression is represented as a function of type `'v dict -> 'v -> 'v` (line 7). The first argument, of type `'v dict`, 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 represented as a polymorphic computation.* Indeed, the definition of the type `exp` includes a universal quantification over the type `'v`, thereby requiring every expression to be polymorphic in the type `'v`.[2] This means that an expression must not care what kind of numbers it is applied to. Because polymorphism in OCaml is parametric [Rey83], this requirement limits the ways in which an expression can interact with *numbers* – that is, with values of type `'v`. The only numbers to which an expression initially has access are its second argument and the dictionary fields `zero` and `one`. To create new numbers, an expression must call the dictionary operations `add` or `mul`. There is no way for an expression to inspect a number.

This representation of expressions allows an expression to be evaluated in many different ways, involving many different kinds of numbers. For instance, by applying it to a suitable type of numbers and to a suitable dictionary, one can evaluate it using integer arithmetic, evaluate it using floating-point arithmetic, or convert it to a symbolic representation. As will be demonstrated later on (§3, §5), this representation of expressions allows several different implementations of AD.

Our implementation of reverse-mode automatic differentiation using effect handlers (§5) evaluates the expression that one wishes to differentiate under a nonstandard implementation of the four operations, where addition and multiplication perform control effects. Although this is not visible in the definition of the type `exp`,[3] this implementation exploits the fact that *an expression must be an effect-polymorphic computation.* That is, an expression must not care what control effects (if any) are performed by the operations `add` and `mul`. This idea is explicitly spelled out in the next section (§2), where we propose a specification for `diff`.

[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 dict -> '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.

1.2. **Example Uses of the API.** A first example use of this API appears in Figure 2. It can be linked with an arbitrary implementation of this API, that is, with an arbitrary implementation of `diff`. This code first builds a representation `e` of the mathematical expression $(x + 1)^3$. This construction 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`. (One might wish to also redefine `0` and `1` as aliases for `zero` and `one`, but OCaml does not allow this.) The meat of the definition of `e` is at lines 6–7. Then, on line 11, `diff` is applied to `e`, yielding a representation `e'` of the derivative of `e` with respect to the variable x. Exactly what this function call does depends on how `diff` is implemented. With all three implementations shown in this paper, this function call terminates immediately: it simply allocates and returns a closure. Automatic differentiation really takes places only at line 17, which requests the evaluation of the expression `e'` using floating-point arithmetic. There, because we expect `e'` to be equivalent to the expression $3 \cdot (x + 1)^2$, and because we instantiate x with the value 4, we expect the result to be 75. Executing the code validates this expectation: with each of the three implementations of AD shown in this paper, the runtime assertion at line 17 succeeds.

A second example use of our API appears in Figure 3. The OCaml expression `monomial k` is intended to represent the mathematical expression x^k. This code is a textbook implementation of fast exponentiation. It uses a number of nontrivial programming-language features, including primitive operations on integers (division, modulus, comparisons), several control constructs (a conditional construct, a loop), and mutable references. One might naively fear that this might prevent the application of `diff` to `monomial k`: indeed, in general, a program that uses these features is not necessarily differentiable. However, such a fear is unwarranted. The use of these features in the definition of `monomial` is an implementation detail: it is not visible to an observer who monitors the behavior of `monomial k`. In particular, it cannot be observed by the function `diff` when `diff` is applied to `monomial k`.

We propose to reason about `monomial k`, and about an application of `diff` to `monomial k`, in the following way. To a user, how `monomial` is implemented does not matter; what matters is that `monomial k` represents the mathematical expression x^k. (The meaning of this claim is clarified in §2.) This is our proposed specification of the function `monomial`. This specification, together with our proposed specification of `diff`, guarantees that the function application `diff (monomial k)` is safe and that its result represents the mathematical expression $k \cdot x^{k-1}$.

This reasoning illustrates the appeal of a compositional approach to program verification. Thinking in terms of abstract specifications makes it easy to see that, if each of `diff` and `monomial` independently satisfies its specification, then `diff (monomial k)` must behave as intended. In contrast, unfolding the concrete definitions of `diff` and `monomial` and attempting to imagine what machine behavior results from the combination of these definitions would be much more difficult.

This example also illustrates a strength of define-by-run AD over define-then-run AD. Even though an implementation of `diff` has no knowledge of primitive integers, conditionals, loops, or mutable references, it can nevertheless be applied to a piece of OCaml code that uses these features, provided this code obeys its contract, which is to represent a certain mathematical expression E. The expression E must inhabit a fixed subset of the mathematical language, which (in this paper) includes the constants 0 and 1, addition, multiplication, and a single variable x. Thus, whereas define-then-run AD can be applied only to a fixed subset of programs, define-by-run AD can be applied to an arbitrary program, as long as its denotation inhabits a fixed subset of mathematical expressions.

2. A Specification for Define-By-Run AD

In the previous section (§1), we have presented an API for define-by-run AD. This API consists of a definition of the type `exp` and a type for the function `diff`. It is expressed in the polymorphic-λ-calculus fragment of OCaml: it involves product types, function types, and universal types. It describes the runtime representation of the values that are exchanged between the function `diff` and its environment, but does not describe the intended meaning of these runtime values.

We now go one step further and propose a specification for `diff`. This specification clarifies the connection between the runtime values that exist in the machine's memory and the mathematical objects that these values represent. Furthermore, it prescribes which values may or must be exchanged at each point of the dialogue between `diff` and its environment. It does not mandate a specific implementation of `diff`: as we shall see (§3, §5), several implementations are possible.

For the moment, we express this specification in English, in an informal yet precise style. Later on in the paper (§8.1), we show that this specification can be formally expressed in Hazel [dVP21], a variant of higher-order Separation Logic equipped with support for effects and effect handlers.

In the following, we write E for a mathematical expression in the fixed universe defined by $E ::= X \mid 0 \mid 1 \mid E + E \mid E \times E$. We write E' for the derivative of E with respect to the variable X. (There is just one variable, named X.) We use these mathematical expressions in specifications only; they do not exist at runtime.

We write e for a runtime value of type `exp`. We do not describe the syntax or memory layout of runtime values, as it is irrelevant: what matters is how runtime functions *behave* when invoked.

The specification of `diff` is very short:

Statement 2.1 (Informal Specification of Differentiation)**.** *If the value e represents E, then the function call* `diff` *e diverges or returns a value e' that represents E'.*

A reader who is curious to know how this statement is expressed in Separation Logic may wish to peek ahead at Statement 8.1.

Throughout the paper, we use a logic of partial correctness, which is why divergence (nontermination) is not forbidden by our specification. By convention, let us write *yields* for *diverges or returns*. A reformulation of the above statement, which uses this short-hand and avoids introducing the names e and e', is as follows: *when applied to a representation of E, the function* `diff` *yields a representation of E'.*

This specification is extremely simple, but relies on a key auxiliary definition, which we have not yet given. There remains to define the assertion *e represents E*, that is, to specify what it means for a runtime value e to represent a mathematical expression E. This is a bit more involved. Very roughly speaking, we wish to express the idea that applying the function e`.eval` to a dictionary of arithmetic operations $(0, 1, +, \times)$ and to a number r must yield the value of the expression E at the point $X = r$. However, such a sentence glosses over several important points.

First, as noted earlier (§1), e`.eval` must be polymorphic: it must be able to compute with numbers of an arbitrary nature, provided these numbers are equipped with sufficient structure. For our purposes, the structure of a semiring suffices. Thus, our definition must involve a universal quantification over a semiring $\mathcal{R}$. In the following, by convention, when we write *a mathematical number* or *a number*, we mean *an element of $\mathcal{R}$*. We write $+$ and

$\times$ for the addition and multiplication operations of the semiring $\mathcal{R}$ and 0 and 1 for their neutral elements.

Second, Hoare Logic and Separation Logic impose maintaining a distinction between a runtime value and the mathematical object that this value is meant to represent. Thus, we must distinguish between runtime values such as `zero` and `one` and numbers such as 0 and 1. We must also distinguish between the runtime values `add` and `mul` and the semiring operations $+$ and $\times$, and explain how they must be related.

From these remarks, it follows that we must quantify not only over the semiring $\mathcal{R}$, but also over the runtime values `zero`, `one`, `add`, `mul` and over the relation that describes what it means for a runtime value to represent a number. Quantifying over a relation may at first seem surprising, but is in fact perfectly natural: such a higher-order quantification is typically encountered in binary-logical-relation interpretations of universal types.

A last key aspect is the treatment of control effects. The definition of *e represents E* must specify which function invocations may perform effects and what effects they may perform. As noted earlier (§1), we would like e`.eval` to be effect-polymorphic. More precisely, we wish to allow the functions `add` and `mul` to perform arbitrary effects. Furthermore, we wish to forbid the function e`.eval` from handling these effects or performing any effects of its own. Thus, the effects performed by `add` and `mul`, and only these effects, can be observed outside e`.eval`. Later on (§7, §8.2.7), we introduce the notion of a *protocol* Ψ, which describes the effects that a function may perform. For now, let us posit that, to express that e is effect-polymorphic, the definition of *e represents E* must quantify over a protocol: the universal quantification over Ψ and the three occurrences of Ψ in the following definition reflect the idea that *whatever effects* `add` *and* `mul` *may perform,* e`.eval` *may perform these effects (and no more).*

Statement 2.2 (Informal Runtime Representation of Expressions)**.** *A runtime value e represents an expression E if*

- *for every semiring* $\mathcal{R}$*,*
- *for every protocol* Ψ*,*
- *for every possible meaning of the assertion "the runtime value n represents the number r",*
- *for all runtime values* `zero`*,* `one`*,* `add`*,* `mul` *such that*
  - `zero` *represents* 0*,*
  - `one` *represents* 1*,*
  - *when applied to representations of r and s,*
    `add` *yields a representation of* $r + s$
    and may perform effects permitted by Ψ*,*
  - *when applied to representations of r and s,*
    `mul` *yields a representation of* $r \times s$
    and may perform effects permitted by Ψ*,*
- *for every number r,*
- *when applied to the record* {`zero`; `one`; `add`; `mul`} *and to a representation of the number r, the function* e`.eval` *yields a representation of the value of the expression E at* $X = r$ *and may perform effects permitted by* Ψ*.*

Although this statement remains informal, it should give the reader a fairly exact preview of the formal definitions and statements that we present later on (§8.1). Only one technical aspect has been omitted above, namely the use of Iris's *persistence* modality in several places to indicate that certain runtime values can be used as many times as one wishes.

```
1  let diff (e : exp) : exp = { eval =
2    fun (type v) ({ zero; one; add; mul } : v dict) (n : v) ->
3      let ( + ), ( * ) = add, mul in
4      let open struct
5        type t = { v : v ; d : v }
6        let mk v d = { v; d }
7        let dict =
8          let zero = mk zero zero
9          and one = mk one zero
10         and add a b = mk (a.v + b.v) (a.d + b.d)
11         and mul a b = mk (a.v * b.v) (a.d * b.v + a.v * b.d)
12         in { zero; one; add; mul }
13       let x = mk n one
14       let y = e.eval dict x
15     end in
16     y.d
17 }
```

FIGURE 4. Forward-mode AD in OCaml

3. FORWARD-MODE AND STACK-BASED REVERSE-MODE IMPLEMENTATIONS OF AD

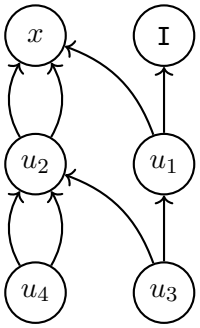
In this section, we briefly present two OCaml implementations of define-by-run AD. Each of them respects the API that we have proposed (§1), and we believe (but we have not formally verified) that each of them meets the specification that we have proposed (§2). The first implementation performs forward-mode AD. The second implementation performs reverse-mode AD and uses a dynamically allocated mutable stack.

There are several reasons why we present these two implementations. First, these examples help demonstrate that our API (§1) and specification (§2) do admit several quite different implementations. Second, they allow us to recall some of the basic principles of forward-mode and reverse-mode AD. Third, they prepare the reader to the presentation of a third implementation (§5), which exploits effect handlers and performs reverse-mode AD.

3.1. Forward-Mode AD.

A forward-mode implementation of `diff` appears in Figure 4. This code receives (1) a representation `e` of a mathematical expression E (line 1); (2) a representation of a semiring $\mathcal{R}$, in the form of a type `v` and a dictionary {`zero`; `one`; `add`; `mul`} (line 2); and (3) a representation `n` of a number $r \in \mathcal{R}$ (line 2). Its aim, according to Statements 2.1 and 2.2, is to evaluate the mathematical expression E' at r.

Because the algorithm has access to the function `e.eval`, it can evaluate the expression E. Furthermore, because this function is polymorphic, it can evaluate E in an arbitrary semiring of its choosing. The key idea of forward-mode AD is to evaluate E in the semiring $\mathcal{R}^2$ of *dual numbers* over $\mathcal{R}$. A dual number is a pair of numbers. Addition and multiplication of dual numbers are defined by $(a, \dot{a}) + (b, \dot{b}) = (a + b, \dot{a} + \dot{b})$ and $(a, \dot{a}) \times (b, \dot{b}) = (a \times b, \dot{a} \times b + a \times \dot{b})$. The neutral elements of addition and multiplication are $(0, 0)$ and $(1, 0)$. It is not difficult to see that, by virtue of these definitions, the value of E at $(r, 1)$ in the semiring $\mathcal{R}^2$ is a pair whose first component is the value of E at r in the semiring $\mathcal{R}$ and whose second component



(a) DAG view

$$\begin{array}{l}\mathsf{let}\ u_1 = x \times \mathtt{I}\ \mathsf{in}\\ \mathsf{let}\ u_2 = x \times x\ \mathsf{in}\\ \mathsf{let}\ u_3 = u_2 \times u_1\ \mathsf{in}\\ \mathsf{let}\ u_4 = u_2 \times u_2\ \mathsf{in}\\ u_3\end{array}$$

(b) Sequential view

Figure 5. DAG and stack built by the forward evaluation of `monomial 3`

is the value of E' at r in the semiring $\mathcal{R}$. The second component of this pair is the desired result.

The code in Figure 4 implements this idea in a straightforward way.[4] A type `t` of dual numbers is defined on line 5. A dual number is represented as a record with two fields named `v` and `d`, for "value" and "derivative", each of which is a number of type `v`. The operations on dual numbers and their neutral elements are defined and grouped in a dictionary `dict` on lines 7–12. This dictionary is used on line 14 to evaluate the expression E. This expression is evaluated at the dual number $(r, 1)$, represented by `x` (line 13). As previously explained, the result of this evaluation is a dual number whose derivative component is the desired result: the value of E' at r. This component is read and returned on line 16.

3.2. **Reverse-Mode AD.** We opened this paper with a quote from the Wikipedia entry on AD: *"every computer program executes a sequence of elementary arithmetic operations (addition, multiplication, etc.)"*. This sentence contains a key idea: instead of thinking of a mathematical expression as a tree-structured object, as suggested by the inductive syntax $E ::= X \mid 0 \mid 1 \mid E + E \mid E \times E$, one can equivalently represent it as a sequence of elementary operations whose arguments and results are identified by names $a, b, u, x, y, \ldots$ This view is described by the following alternate syntax:

$$S ::= \mathsf{let}\ u = a + b\ \mathsf{in}\ S \mid \mathsf{let}\ u = a \times b\ \mathsf{in}\ S \mid y$$

Although the leaves X, 0 and 1 seem to have disappeared in this syntax, they can be represented by three reserved names. In this form, an expression S is a sequence of operations. Each operation refers to its two operands by their names a and b, which must have been previously defined, and introduces the name u to stand for its result. At the end of the sequence, a result, identified by its name y, is produced.

A central quality of this alternate presentation of expressions is that it offers at the same time a view of an expression as a *sequence* of operations and a view of an expression as a *directed acyclic graph* (DAG). In the latter view, each vertex is identified by a name, and there are edges from u to a and from u to b if u is the result of an operation whose operands are a and b. The vertex y is the root vertex of the graph. Figures 5a and 5b offer an example of these views. The sequential view is valuable because it allows traversing the graph in the

[4] "`let open struct ... end`" at line 4 is an OCaml idiosyncrasy that allows toplevel definitions (of types, effects, and values) to appear in the midst of an expression.

		⟶ Time ⟶				
Space	x	0	0	0	$2n^2$	$3n^2$
	u_1	0	0	n^2	n^2	#
	u_2	0	0	n	#	#
	u_3	0	1	#	#	#
	u_4	1	#	#	#	#

FIGURE 6. Derivatives computed during the backward phase of `monomial 3`

forward direction (in dependency order) or in the reverse direction (in reverse dependency order), as desired. The graphical view is valuable because it keeps track of and allows taking advantage of sharing.

Reverse-mode AD exploits this presentation of expressions. It is organized in two phases. In the first phase, known as the *forward phase*, the expression that one wishes to differentiate is converted into this form: a sequential view and a DAG view are explicitly constructed in memory. The second phase, known as the *backward phase*, is where differentiation really takes place. During this phase, the graph is traversed in reverse dependency order: each vertex is processed in turn. During this traversal, a sequential expression S is gradually constructed: it evolves (grows) with time. The following *backward invariant* is maintained: at each point in time, for every vertex u that has not yet been processed, the partial derivative $\partial S/\partial u$ is computed and stored at vertex u. Initially, S is just y, the root vertex. Then, every time a vertex u is processed, the expression S is extended with a binding for u: for example, if u is a name for $a+b$, then S is updated to $\mathsf{let}\ u = a+b\ \mathsf{in}\ S$. The backward phase ends when every vertex has been processed except for x, the vertex that stands for the variable X. At this point, the expression S is equivalent to the expression E that one wishes to differentiate. The backward invariant then implies that the vertex x stores the derivative of E with respect to x. This is the desired result.

To illustrate the backward phase, Figure 6 shows the evolution of derivatives computed during this phase of the differentiation of `monomial 3`. We assume that, during the forward phase, this expression has been converted to the sequential view of Figure 5b. Furthermore, we assume that the derivative of `monomial 3` is queried at a number n. At each vertex and at each point in time, one number is stored. Thus, one column in Figure 6 shows a snapshot of the numbers stored at all vertices at a given point in time; whereas one line in Figure 6 shows the evolution through time of the number stored at a given vertex. Transitioning from a column to its right neighbor corresponds to processing one vertex. A processed vertex is marked with a hash symbol #. (Once a vertex has been processed, the value that is stored at this vertex loses its meaning and is no longer read or updated.) In the first column, we see that each vertex u stores the partial derivative of u_3 (the root vertex) with respect to u at n. In the second column, the vertex u_4 has been processed, so every unprocessed vertex u stores the partial derivative of $\mathsf{let}\ u_4 = u_2 \times u_2\ \mathsf{in}\ u_3$ with respect to u at n. By extending this reasoning to the remaining columns, one can see that, in the last column, the vertex x stores the partial derivative of the sequential expression of Figure 5b with respect to x at n.

3.3. **Stack-Based Reverse-Mode AD.** Having presented the key ideas of the reverse-mode approach to automatic differentiation, let us now discuss an implementation of this approach.

```
let diff (e : exp) : exp = { eval =
  fun (type v) ({ zero; one; add; mul } : v dict) (n : v) ->
    let ( + ), ( * ) = add, mul in
    let open struct
      (* The graph. *)
      type t = O | I | Var of { v : v ; mutable d : v }
      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
      (* The stack. *)
      type op = Add | Mul
      type binding = Let of t * (t * op * t)
      open Stack
      let stack : binding Stack.t = create()
      (* The dictionary used in the forward phase. *)
      let dict =
        let zero = O
        and one = I
        and add a b =
          let u = mk (get_v a + get_v b) in
          push (Let (u, (a, Add, b))) stack; u
        and mul a b =
          let u = mk (get_v a * get_v b) in
          push (Let (u, (a, Mul, b))) stack; u
        in { zero; one; add; mul }
      (* The forward phase. *)
      let x = mk n
      let y = e.eval dict x
      (* The backward phase. *)
      let () =
        update y one;
        while not (is_empty stack) do
          match pop stack with
          | Let (u, (a, Add, b)) ->
              update a (get_d u);
              update b (get_d u)
          | Let (u, (a, Mul, b)) ->
              update a (get_d u * get_v b);
              update b (get_d u * get_v a)
        done
    end in
    get_d x
}
```

Figure 7. Stack-based reverse-mode AD in OCaml

This implementation of reverse-mode automatic differentiation appears in Figure 7. It is stack-based: it uses a stack to construct a DAG view and a sequential view of the expression that one wishes to differentiate. It begins like our forward-mode implementation (§3.1): it receives a representation `e` of a mathematical expression E (line 1); a representation of a semiring $\mathcal{R}$, in the form of a type `v` of numbers and a dictionary {`zero`; `one`; `add`; `mul`} (line 2); and a representation `n` of a number $r \in \mathcal{R}$ (line 2).

A type `t` of vertices is declared on line 6. A vertex is either `O`, which denotes the constant 0; or `I`, which denotes the constant 1; or a heap-allocated record, carrying the tag `Var`. The address of this record effectively serves as the name of the vertex. This record holds two fields, named `v` and `d`, for "value" and "derivative". The `v` field is immutable and is initialized during the forward phase. The `d` field is mutable and is updated (possibly several times) during the backward phase. These two fields allow us to associate two numbers, `u.v` and `u.d`, with each vertex `u`. Four auxiliary functions help construct and access vertices. The function `mk` (line 7) allocates and initializes a new vertex. The functions `get_v` and `get_d` (lines 8–9) read the `v` and `d` fields of a vertex, handling the constants `O` and `I` in a suitable way.[5] The function `update` (line 10) updates the `d` field of a vertex by adding the value `i` to this field. Applying `update` to the vertices `O` or `I` does nothing.

A stack is allocated at line 15. This stack is mutable and is initially empty. It stores a sequence of *bindings* of the form `let` $u = a \; op \; b$, where u, a, b are vertices and where op is `Add` or `Mul` (lines 12–13). Each such binding records an addition or multiplication operation, whose result is the vertex u and whose operands are the vertices a and b. It indicates the existence of two graph edges, from u to a and from u to b. This data structure is known in the literature as a *tape* or a *Wengert list*.

The functions `add` (line 20) and `mul` (line 23) create a new vertex `u`, extend the stack with a new binding for `u`, and return this vertex. The field `u.v` receives the sum or the product of the fields `a.v` and `b.v`. The constants `O` and `I` and the functions `add` and `mul` are grouped in a dictionary `dict`, for use during the forward phase.

After these preparations, the forward phase can take place. First, a vertex x is created to stand for the variable X (line 28). Then, the expression E is evaluated under the dictionary `dict`, yielding a vertex y (line 29). This evaluation serves two main purposes. First, the expression E is converted into the sequential representation S that was discussed earlier. Indeed, at the end of the forward phase, the bindings stored in the stack, together with the root vertex y, form such a sequential representation. Second, the expression E is evaluated in the semiring $\mathcal{R}$ at $X = r$. Indeed, the `v` field of every vertex stores the numeric value that corresponds to this vertex.

Then comes the backward phase (lines 31–41). Writing 1 into the `d` field of the vertex y (line 32) establishes the backward invariant that we have sketched earlier (§3.2). Indeed, for every vertex u, the partial derivative "$\partial y / \partial u$" is 1 if the vertices u and y are the same vertex, and it is 0 if they are distinct vertices. The backward invariant serves as a loop invariant for the loop that follows. In this loop, each vertex u is processed in turn, in reverse order: the most recently created vertices are processed first. As long as the stack is nonempty, an entry is popped off the stack (line 34). This entry records whether the vertex u was the result of an addition or multiplication operation, and what were the operands a and b of this operation. In each case, information is propagated along the edges from u to a and from u

[5] The function `get_d` is applied during the backward phase to the source vertex of an edge. The vertices `O` and `I` are never the source of an edge, so `get_d` is never applied to them. The OCaml expression **`assert false`** is used to denote these dead branches.

to b: the `d` fields associated with the vertices a and b are updated. These updates, which we do not explain in detail for now, serve to maintain the backward invariant. At the end of the loop, the backward invariant implies that the partial derivative "$\partial y/\partial x$" is stored in the `d` field of the vertex x. The code retrieves and returns this final result (line 43).

4. Effects and Handlers

4.1. **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*,[6] which represents the suspended computation: invoking this function resumes the computation. If the computation is resumed, then 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 shall present shortly (§4.2), as well as our effect-based reverse-mode automatic differentiation algorithm (§5), exploits 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* [dVP21], one of the most fundamental reasoning rules of Separation Logic. Both Multicore OCaml and our reasoning rules [dVP21] require that a continuation be invoked at most once. In our reverse-mode automatic differentiation algorithm, every continuation is invoked exactly once.

[6] 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 8. 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 9. Output of the program in Figure 8

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, SDW+21]. They have also been implemented as a library in mainstream programming languages such as Scala [BSO20b].

4.2. **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. The statement `perform v` interrupts execution and transfers control to the nearest enclosing handler, which receives the value `v` and a continuation `k`. The statement `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 8 exploits all of these constructs in combination. It is a complete Multicore OCaml program, whose output appears in Figure 9.

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 9. 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, then complete their execution. Thus, the two `printf` statements that were delayed earlier are allowed to take effect. The most recent handler instance completes 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 terminology [DG05], the first phase occurs at *call time* whereas the second phase occurs at *return time*.

5. Effect-Based Reverse-Mode AD

We now propose a third implementation AD, which, like the previous two, obeys the API (§1) and the specification (§2) proposed earlier. It is based on Sivaramakrishnan's implementation [Siv18], which itself was inspired by Wang et al.'s work [WR18, WZD+19]. Its code appears in Figure 10.

The overall structure of the function `diff` is the same as in our earlier two examples (Figures 4 and 7). The representation of vertices and the auxiliary functions on vertices (lines 6–10) are the same as in our earlier reverse-mode implementation (Figure 7). What is

```
let diff (e : exp) : exp = (* head: *) { eval =
  fun (type v) ({ zero; one; add; mul } : v dict) (n : v) -> (* body: *)
    let ( + ), ( * ) = add, mul in
    let open struct
      (* The graph. *)
      type t = O | I | Var of { v : v ; mutable d : v }
      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
      (* The dictionary. *)
      effect Add : t * t -> t
      effect Mul : t * t -> t
      let zero' = O
      and one' = I
      and add' a b = perform (Add (a, b))
      and mul' a b = perform (Mul (a, b))
      let dict = { zero = zero'; one = one'; add = add'; mul = mul'}
    end in
    (* The forward and backward phases. *)
    (* init: *)
    let x = mk n in
    (* heart: *)
    let _ =
      (* handle: *) match (* eval: *) e.eval dict x with
      | (* eff-add: *) effect (Add (a, b)) k ->
          (* add-body: *)
          let u = (* add-fwd: *) mk (get_v a + get_v b) in
          let _ = (* cont: *) continue k u in
          (* add-bwd: *)
          update a (get_d u);
          update b (get_d u)
      | (* eff-mul: *) effect (Mul (a, b)) k ->
          let u = mk (get_v a * get_v b) in
          let _ = continue k u in
          update a (get_d u * get_v b);
          update b (get_d u * get_v a)
      | (* ret: *) y ->
          (* seed: *) update y one
    in
    (* done: *) get_d x
}
```

FIGURE 10. Effect-based reverse-mode AD in Multicore OCaml

new here is that the stack disappears. In our earlier implementation, the call `e.eval dict x` (line 29 of Figure 7) represents just the forward phase. It fills up the stack, which is then emptied via an explicit loop. Here, in contrast, the dictionary `dict` is defined in such a way that the call `e.eval dict x` (line 25) represents both the forward phase and the backward phase. No loop is necessary.

The definition of the dictionary `dict` is almost trivial. Two effects, `Add` and `Mul`, are declared at lines 12–13. The operations `add` and `mul` perform these effects (lines 16–17). The meaning of these operations is defined by the way in which these effects are handled.

The bulk of the computation, which takes place between lines 24 and 39, exploits effects in the following way. Because the operations `dict.add` and `dict.mul` perform `Add` and `Mul` effects, and because the function call `e.eval dict x` can invoke these operations, it can itself perform `Add` and `Mul` effects. We handle these effects by wrapping this function call in a handler (line 25). The structure of this handler is analogous to that found in our previous example (Figure 8). Because the continuation is invoked in the middle of the handler's code (lines 29 and 35), 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 dict x` runs. During this phase, the operations `dict.add` and `dict.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 28 and 34). Control is then immediately handed back to the computation `e.eval dict x`, while the execution of the handler code that follows the **`continue`** statement is postponed.

The backward phase begins when the function call `e.eval dict x` terminates and returns a vertex `y`, which represents its final result. At this point, the handler receives control (line 38) and writes the number 1 into `y.d`. 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 31–32 is executed, and for each `Mul` effect that took place earlier, the code at lines 36–37 is executed. Once this is over, the code block at lines 24–39 is exited. Control moves on to the last line, where the desired result is found in the `d` field of the vertex `x`.

6. Mathematical Expressions

In preparation for the formal part of this paper, we define mathematical expressions and prove some of their basic properties. First, we introduce the syntax and evaluation of mathematical expressions (§6.1). Then, we present an alternative syntax, where a mathematical expression is viewed as a sequence of operations (§6.2). Finally, we define partial derivatives and present several forms of the Chain Rule (§6.3).

In our informal specification (§2), we have focused on expressions of one variable. We have written E for a mathematical expression defined by $E ::= X \mid 0 \mid 1 \mid E + E \mid E \times E$, and we have written E' for the derivative of E with respect to the variable X. However, our proof requires reasoning about expressions of several variables $\imath, \jmath, \ldots$ and about partial derivatives $\partial E/\partial \jmath$. Therefore, from here on, we work with expressions of several variables. We take a symbolic view of expressions and derivation: an expression is regarded as an abstract syntax tree; partial derivation $\partial\cdot/\partial\jmath$ is regarded as a transformation of an expression into an expression.

6.1. **Expressions; Evaluation; Free Semiring.** An *expression* E is an abstract syntax tree built out of (1) the constants $\mathbf{0}$ and $\mathbf{1}$; (2) applications of the binary arithmetic operators $\mathbf{+}$ and $*$, also known as *nodes*; and (3) other numeric constants or variables, also known as *leaves*, drawn from some set $\mathcal{I}$.

Definition 6.1 (Expressions)**.** Let $\mathcal{I}$ be a finite or infinite set. Let $\imath$ and $\jmath$ range over $\mathcal{I}$. The set $Exp_{\mathcal{I}}$ of expressions E whose variables are drawn from $\mathcal{I}$ is defined as follows:

$$\begin{aligned} op &::= \mathbf{+} \mid * \\ Exp_{\mathcal{I}} \ni E &::= \mathbf{Leaf}\ \imath \mid \mathbf{0} \mid \mathbf{1} \mid E\ op\ E \end{aligned}$$

When we wish to reason about expressions of one variable, we fix a name X and we instantiate $\mathcal{I}$ with the singleton set $\{X\}$. Thus, $Exp_{\{X\}}$ denotes the set of expressions of one variable. In Coq, the name X is encoded as the unit value `tt`, and the singleton set $\{X\}$ is encoded as the `unit` type, whose single inhabitant is `tt`.

An expression can be *evaluated* in an arbitrary semiring $(\mathcal{R}, 0, +, 1, \times, \equiv)$, where $\equiv$ is an equivalence relation on the carrier set $\mathcal{R}$, and where the axioms of a semiring hold with respect to this equivalence relation. *Evaluating* an expression $E \in Exp_{\mathcal{I}}$ under an *environment* ϱ, a mapping of variables in $\mathcal{I}$ to numbers in $\mathcal{R}$, yields a number $[\![E]\!]_\varrho \in \mathcal{R}$, the value of this expression.

Definition 6.2 (Expression Evaluation)**.** Evaluation $[\![\cdot]\!]_{(\cdot)}$ is inductively defined as follows:

$$\begin{aligned} [\![\mathbf{Leaf}\ \imath]\!]_\varrho &\triangleq \varrho(\imath) \\ [\![\mathbf{0}]\!]_\varrho &\triangleq 0 \\ [\![\mathbf{1}]\!]_\varrho &\triangleq 1 \\ [\![E_1\ \mathbf{+}\ E_2]\!]_\varrho &\triangleq [\![E_1]\!]_\varrho + [\![E_2]\!]_\varrho \\ [\![E_1\ *\ E_2]\!]_\varrho &\triangleq [\![E_1]\!]_\varrho \times [\![E_2]\!]_\varrho \end{aligned}$$

When equipped with the syntactic constructors $\mathbf{+}$ and $*$ as addition and multiplication, with the constants $\mathbf{0}$ and $\mathbf{1}$ as neutral elements, and with a suitable definition of equivalence, the set $Exp_{\mathcal{I}}$ forms a semiring, known as the *free semiring*.

Lemma 6.3 (Free Semiring)**.** *Let $\mathcal{I}$ be a set. Define $\equiv_{Exp_{\mathcal{I}}}$ as the smallest equivalence relation on $Exp_{\mathcal{I}}$ for which the semiring axioms hold. Then, $(Exp_{\mathcal{I}}, \mathbf{0}, \mathbf{+}, \mathbf{1}, *, \equiv_{Exp_{\mathcal{I}}})$ forms a semiring.*

The free semiring plays a role in the verification of the first phase of reverse-mode AD, whose purpose is to construct a sequential view and a DAG view of an expression E (§3.2). During this phase, the expression E, which can be evaluated in an arbitrary semiring (§1.1), is evaluated in the free semiring (§8.2.6).

6.2. **Sequential View of Expressions.** As pointed out earlier (§3.2), when reasoning about reverse-mode AD, it can be useful to have an alternate view of expressions as sequences of elementary operations. There, we proposed the abstract syntax $S ::= \mathsf{let}\ u = a\ op\ b\ \mathsf{in}\ S \mid y$. It is in fact more convenient to use the equivalent presentation $S ::= K[y]$, where a *context* K is defined as a list of *bindings* B:

Definition 6.4 (Bindings; contexts)**.** Bindings and contexts are defined as follows:

$$\begin{aligned} B &::= \mathsf{let}\ u = a\ op\ b \\ K &::= [] \ \mid\ B; K \end{aligned}$$

The identifiers u, a, b, y can be thought of as names for auxiliary variables. In our Coq proof, they are drawn from the set *Val* of the runtime values; in the paper, this detail does not matter much. 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 the right end represents the newest binding. 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 $\mathit{defs}(K)$ to denote the list of identifiers defined by the context K, that is, $\mathit{defs}(\mathsf{let}\ u = a\ \mathit{op}\ b; K) = u; \mathit{defs}(K)$ and $\mathit{defs}([]) = []$.

The hole [] at the right end of a list of bindings K can be viewed as a placeholder, waiting to be filled with an identifier y. This is why we refer to K as a *context*. A pair of a context K and an 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 the root of the expression. The operation of filling a context K with an identifier y, defined next, converts an expression represented as a pair (K, y) to an ordinary tree-structured expression.

Definition 6.5 (Filling)**.** The function $\cdot[\cdot]$ is inductively defined as follows:

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

If K is a context and $y \in \mathit{Val}$ is an identifier, then $K[y]$ is an expression in $\mathit{Exp}_{\mathit{Val}}$, that is, an expression whose leaves are identifiers.

Our last definition is the extension of an environment $\varrho \in \mathit{Val} \to \mathcal{R}$ with a context K, resulting in an updated environment $\varrho\{K\} \in \mathit{Val} \to \mathcal{R}$. An intuition is that $\varrho\{K\}$ can be obtained by starting from ϱ and by executing the bindings in K, in succession, from left to right. We could reflect this intuition by giving an inductive definition of $\varrho\{K\}$. Instead, we give the following direct definition, which is equivalent:

Definition 6.6 (Extension of an Environment)**.** $\varrho\{K\}$ is the environment that maps every identifier y to $[\![K[y]]\!]_\varrho$.

6.3. **Partial Derivatives; Chain Rule.** The partial derivative of an expression $E \in \mathit{Exp}_{\mathcal{I}}$ with respect to a variable $\jmath \in \mathcal{I}$ is an expression $\partial E/\partial \jmath \in \mathit{Exp}_{\mathcal{I}}$.

Definition 6.7 (Partial Derivative)**.** Partial derivation $\partial\cdot/\partial\cdot$ is inductively defined as follows:

$$\begin{array}{rcll} \partial(\mathbf{Leaf}\ \imath)/\partial\jmath & \triangleq & \mathbf{1} & \text{if } \imath = \jmath \\ \partial(\mathbf{Leaf}\ \imath)/\partial\jmath & \triangleq & \mathbf{0} & \text{otherwise} \\ \partial(\mathbf{0})/\partial\jmath & \triangleq & \mathbf{0} & \\ \partial(\mathbf{1})/\partial\jmath & \triangleq & \mathbf{0} & \\ \partial(E_1 \mathbin{+} E_2)/\partial\jmath & \triangleq & \partial E_1/\partial\jmath \mathbin{+} \partial E_2/\partial\jmath & \\ \partial(E_1 * E_2)/\partial\jmath & \triangleq & \partial E_1/\partial\jmath * E_2 \mathbin{+} E_1 * \partial E_2/\partial\jmath & \end{array}$$

One recognizes in the above definition the well-known laws that indicate how to compute a partial derivative of a variable, a constant, a sum, and a product. Most mathematicians would view the above equations as a set of laws that can be *proved*, based on a more semantic 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.

Definition 6.8 (Derivative)**.** Let $E \in \mathit{Exp}_{\{X\}}$ be a univariate expression. The derivative of E, written E', is also a univariate expression, defined by $E' = \partial E / \partial X$.

In traditional accounts of Calculus, the Chain Rule states how to compute the derivative of the composition of two differentiable functions. In our formalism, the Chain Rule states how to compute the partial derivative of the expression $[\![E]\!]_F$. This expression is the result of evaluating E in the free semiring $\mathit{Exp}_{\mathcal{J}}$ under an environment F that maps every variable $\imath \in \mathcal{I}$ to an expression $F(\imath)$. It can also be understood as the result of substituting $F(\imath)$ for $\imath$ in E, for every $\imath \in \mathcal{I}$, or as the sequential composition "$\mathsf{let}\ [\imath = F(\imath)]_{\imath \in \mathcal{I}}\ \mathsf{in}\ E$".

Lemma 6.9 (Chain Rule)**.** *Let $\mathcal{R}$ be a semiring. Let $\mathcal{I}$ and $\mathcal{J}$ be sets. Let $E \in \mathit{Exp}_{\mathcal{I}}$ be an expression whose variables inhabit $\mathcal{I}$. Let $F : \mathcal{I} \to \mathit{Exp}_{\mathcal{J}}$ be a map of $\mathcal{I}$ into the free semiring $\mathit{Exp}_{\mathcal{J}}$. Let the environment $\vartheta : \mathcal{J} \to \mathcal{R}$ be a map of $\mathcal{J}$ into $\mathcal{R}$. Then, the following equation holds:*

$$[\![\partial [\![E]\!]_F / \partial \jmath]\!]_\vartheta \quad \equiv_{\mathcal{R}} \quad \sum_{\imath \in \mathit{leaves}(E)} [\![\partial E / \partial \imath]\!]_{\lambda \imath . [\![F(\imath)]\!]_\vartheta} \times [\![\partial F(\imath) / \partial \jmath]\!]_\vartheta$$

The left-hand side of this equation is the partial derivative of $[\![E]\!]_F$ with respect to a variable $\jmath \in \mathcal{J}$, evaluated at ϑ. The sum in the right-hand side is indexed by the set $\mathit{leaves}(E)$. This set is a finite subset of $\mathcal{I}$: it is the set of the variables $\imath \in \mathcal{I}$ that occur in the expression E.

When reasoning about the backward phase of reverse-mode AD, we use the following specialized version of the Chain Rule:

Lemma 6.10 (Left-End Chain Rule)**.** *Let $\mathcal{R}$ be a semiring. Let a, b, u, x, $y \in \mathit{Val}$ be auxiliary variables. Let op be an operation. Let K_1 and K_2 be contexts. Let $\varrho : \mathit{Val} \to \mathcal{R}$ be an environment. Let B stand for the binding $\mathsf{let}\ u = a\ op\ b$. Suppose $x \neq u$. Then, the following equation holds:*

$$\begin{aligned} [\![\partial (B; K_2)[y] / \partial x]\!]_{\varrho\{K_1\}} \quad \equiv_{\mathcal{R}} \quad & [\![\partial K_2[y] / \partial x]\!]_{\varrho\{K_1;B\}} \\ & + [\![\partial K_2[y] / \partial u]\!]_{\varrho\{K_1;B\}} \times [\![\partial (\mathbf{Leaf}\ a)\ op\ (\mathbf{Leaf}\ b) / \partial x]\!]_{\varrho\{K_1\}} \end{aligned}$$

The left-hand side is the partial derivative of $(B; K_2)[y]$ with respect to x, and the right-hand side involves the partial derivatives of $K_2[y]$ with respect to x and u. Therefore, this lemma concerns the addition of a binding B at the *left* end of a sequence of bindings K_2. This is typical of reverse mode; if we wanted to reason about forward mode, we would instead wish to add a binding at the *right* end of a sequence.

Lemma 6.10 is obtained from Lemma 6.9 by instantiating both $\mathcal{I}$ and $\mathcal{J}$ with Val and by instantiating E, F, and ϑ as follows:

- $E : \mathit{Exp}_{\mathit{Val}}$ is the expression $K_2[y]$;
- $F : \mathit{Val} \to \mathit{Exp}_{\mathit{Val}}$ maps u to $(\mathbf{Leaf}\ a)\ op\ (\mathbf{Leaf}\ b)$ and is the identity elsewhere;
- $\vartheta : \mathit{Val} \to \mathcal{R}$ is $\varrho\{K_1\}$.

This lemma is key to the verification of the backward phase of reverse-mode AD, where it justifies how the d fields of auxiliary variables are incremented. Consider a vertex u, constructed during the forward phase as the result of an arithmetic operation op whose operands are two earlier vertices a and b. According to the backward invariant (Definition 8.15), during the backward phase, when the vertex u is about to be processed, its d field holds the partial derivative $\partial K_2[y] / \partial u$, for some context K_2 and value y. Similarly, the d fields of the vertices a and b hold the partial derivatives $\partial K_2[y] / \partial a$ and $\partial K_2[y] / \partial b$. After processing the vertex u, for the backward invariant to be preserved, the d fields of the vertices a and

b must hold the partial derivatives $\partial(B; K_2)[y]/\partial a$ and $\partial(B; K_2)[y]/\partial b$, where B stands for the binding $\mathsf{let}\ u = a\ op\ b$. Lemma 6.10, instantiated once with $x := a$ and once with $x := b$, tells us exactly what update instructions must be performed.

7. A Program Logic for Effect Handlers

Traditional Separation Logic [Rey02, O'H19] allows specifications to be written at a pleasant level of abstraction, combining 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.

The program logic Hazel, proposed by the authors in previous work [dVP21], extends this methodology to support programs that involve effects and effect handlers. Hazel consists of two main components, namely: (1) a core programming language with support for effect handlers, equipped with a small-step operational semantics; and (2) a program logic, where the standard notion of *weakest precondition* [JKJ+18, §6] is parameterized with a *protocol*. A protocol can be understood as a contract between a program that performs effects and a context that handles these effects.

7.1. *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. In this paper, we are interested in reasoning about Multicore OCaml programs. However, proposing a formal dynamic semantics for all of Multicore OCaml would be a challenging endeavor. For this reason, we focus on a core calculus, a subset of Multicore OCaml, which can express our effect-based implementation of AD (§5). A small set of features, including first-class functions, references, effect handlers and one-shot continuations, suffices. In previous work [dVP21], 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, the following basic facts should suffice. There is an infinite set *Loc* of memory locations. The set *Val* of values contains the unit value, memory locations, binary products, binary sums, first-class functions, and first-class continuations. 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 §10.

7.2. The program logic Hazel.

An operational semantics defines the behavior of programs, but does not provide a convenient means *to describe* or *to reason 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 *HH*. 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, this implication is an *affine* assertion, which means that it represents a permission to execute e *at most once*. When this assertion is wrapped in the *persistence* modality $\square$, it represents a permission to execute e as many times as one wishes. Thus, the persistent assertion $\square\,(P \mathrel{-\!\!*} wp\ e\ \{\phi\})$ 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` (§8.1) involves very few protocols. One important concrete protocol is the *empty protocol* $\bot$, which forbids all effects: the assertion $ewp\ e\ \langle\bot\rangle\{\phi\}$ 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. An expression in tagless-final style (§1) is an example of such a function.

8. Formal Verification of Effect-Based Reverse-Mode AD

We are equipped with a basic theory of mathematical expressions and derivation (§6) and with a program logic (§7). Therefore, we can now translate our earlier informal specification of AD (Statement 2.1) into a formal specification and prove that our effect-based implementation of reverse-mode AD (§5) meets this specification. We follow this path. We propose a formal specification of AD (§8.1) and we document our machine-checked proof that this implementation satisfies this specification (§8.2).

8.1. **Specification.** Our formal specification of automatic differentiation is a straightforward rendition in Hazel of our earlier informal specification (Statement 2.1).

Statement 8.1 (Specification of Differentiation)**.** *The specification of the function* `diff` *is expressed as follows:*

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

The use of the empty protocol $\bot$ in this statement indicates that the function call `diff` e does not perform any effect. This is in accord with the informal Statement 2.1, where `diff` e is allowed to diverge or return a value, but is not allowed to perform an effect.

This statement relies on the binary predicate *isExp*, which relates a runtime value e to a mathematical expression E. The assertion e *isExp* E reflects the idea that e *represents* E, for which we gave an informal definition in Statement 2.2. We now propose a formal definition:

Definition 8.2 (Runtime Representation of an Expression)**.** The predicate $\mathit{isExp} : \mathit{Val} \to \mathit{Exp}_{\{X\}} \to \mathit{iProp}$ is defined as follows:

$$\begin{array}{l} e\ \mathit{isExp}\ E \triangleq \\ \quad \Box \forall \mathcal{R},\ \Psi,\ \mathit{isNum},\ \texttt{zero},\ \texttt{one},\ \texttt{add},\ \texttt{mul}. \\ \quad \mathit{isDict}\ (\mathcal{R}, \Psi, \mathit{isNum}, \texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}) \mathrel{-\!\!*} \\ \quad \forall\, \texttt{n},\ r.\ \ \texttt{n}\ \mathit{isNum}\ r \mathrel{-\!\!*} \mathit{ewp}\ (e\texttt{.eval}\ \{\texttt{zero};\ \texttt{one};\ \texttt{add};\ \texttt{mul}\}\ \texttt{n})\ \langle\Psi\rangle\{y.\ y\ \mathit{isNum}\ [\![E]\!]_{\lambda X.r}\} \end{array}$$

The persistence modality $\Box$ that appears at the beginning of this definition strengthens the meaning of e *isExp* E and makes it a persistent assertion. Thus, once e *isExp* E has been established, this fact can be used as many times as one wishes, as opposed to at most once. As we will see shortly, e *isExp* E implies that e is a function: the persistence modality indicates that this function can be called as many times as one wishes.[7]

Following this modality, comes a series of universally quantified variables, of the same nature and in the same order as in Statement 2.2. An expression e in tagless-final style is polymorphic in a type of numbers (a semiring $\mathcal{R}$), in the runtime representation of these numbers (a predicate *isNum*), in the implementation of the semiring operations (the values `zero`, `one`, `add`, `mul`), and in a protocol Ψ.

The predicate *isNum* relates a runtime value `x` and a number $r \in \mathcal{R}$ and means that "*the value* `x` *represents the number* r". The universal quantification on *isNum* makes it an abstract predicate: the expression e must not know how numbers are represented. It may assume that the runtime values `zero`, `one`, `add`, `mul` are correct implementations of the semiring operations $0, 1, +, \times$: this is expressed by the hypothesis $\mathit{isDict}\ (\mathcal{R}, \Psi, \mathit{isNum}, \texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul})$, whose definition follows shortly.

The last line in the definition of *isExp* states that applying e`.eval` to two arguments, namely the dictionary {`zero`; `one`; `add`; `mul`} and a representation of the number r, must either produce a representation of the number $[\![E]\!]_{\lambda X.r}$ or perform an effect permitted by Ψ. The number $[\![E]\!]_{\lambda X.r}$ is the result of evaluating the expression E in the semiring $\mathcal{R}$ at the

[7] The persistence modality in the definition of *isExp* is in fact optional. With this modality, Statement 8.1 means that `diff` e may evaluate the expression e several times and returns an expression e' that can safely be evaluated several times. Without this modality, Statement 8.1 means that `diff` e evaluates e at most once and returns an expression e' that must be evaluated at most once. We have verified in Coq that both variants of the statement are true. (A Boolean parameter is used to avoid any duplication.) As far as we can see, neither variant of the statement implies the other. We thank one of the reviewers for pointing out the existence of the variant without the persistence modality.

point r. We have restricted our attention to expressions E of a single variable, which is why an expression is evaluated at a point $r \in \mathcal{R}$.

Definition 8.3 (Runtime Representation of a Semiring)**.** The predicate *isDict* is defined as follows:

$$
\begin{array}{l}
\mathit{isDict}\ (\mathcal{R}, \Psi, \mathit{isNum}, \texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}) \triangleq \\
\quad \Box\ \texttt{zero}\ \mathit{isNum}\ 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\quad \Box\ \texttt{one}\ \mathit{isNum}\ 1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\quad \Box \forall\, a,\, b,\, r,\, s.\ a\ \mathit{isNum}\ r \mathrel{-\!\!*} b\ \mathit{isNum}\ s \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{add}\ a\ b)\ \langle\Psi\rangle\{u.\ u\ \mathit{isNum}\ (r+s)\} \wedge \\
\quad \Box \forall\, a,\, b,\, r,\, s.\ a\ \mathit{isNum}\ r \mathrel{-\!\!*} b\ \mathit{isNum}\ s \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{mul}\ a\ b)\ \langle\Psi\rangle\{u.\ u\ \mathit{isNum}\ (r\times s)\} \wedge \\
\quad \Box \forall\, a,\, r,\, s.\ a\ \mathit{isNum}\ r \mathrel{-\!\!*} r \equiv_{\mathcal{R}} s \mathrel{-\!\!*} a\ \mathit{isNum}\ s \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\quad \Box \forall\, a,\, r.\ a\ \mathit{isNum}\ r \mathrel{-\!\!*} \Box\ a\ \mathit{isNum}\ r
\end{array}
$$

The right-hand side of this definition is a conjunction of the following claims: (1) the value `zero` represents 0; (2) the value `one` represents 1; (3) for all numbers r and s, when applied to representations of r and s, `add` computes a representation of $r+s$ and may perform effects permitted by Ψ; (4) for all numbers r and s, when applied to representations of r and s, `mul` computes a representation of $r \times s$ and may perform effects permitted by Ψ; (5) *isNum* is compatible with the equivalence relation $\equiv_{\mathcal{R}}$, that is, if the numbers r and s are equivalent, then a representation of r is also a representation of s; (6) a representation of a number is persistent. A look back at Statement 2.2 confirms that claims 1–4 were present there already. Claims 5–6 are more technical and were omitted there.

8.2. **Verification.** In this subsection, we present a formal proof of Statement 8.1, where we take `diff` to be a manual transcription in *HH* of the reverse-mode algorithm of Figure 10. To make it manageable, we organize this proof in many segments.

In the beginning (§8.2.1), we step over the first few lines of the code in Figure 10. Then, we define a Separation Logic predicate *isVar*, which describes auxiliary variables (§8.2.2), and step over the allocation of the first auxiliary variable at label `init` (§8.2.3). This brings us to the label `heart`.

At this point, we allocate a piece of *ghost state*, a key ingredient of the proof (§8.2.4). This ghost state keeps a record of the interaction between the differentiation algorithm and the expression that is being differentiated. Its evolution is monotonic. It appears in the main two invariants (§8.2.5). The *forward invariant* holds during the forward phase; the *backward invariant* holds during the backward phase. They describe the content of the `v` field and `d` field of every auxiliary variable.

Still at this point, we introduce another Separation Logic predicate, *isSubExp*, which equips vertices with semiring structure (§8.2.6). This predicate plays a role in the definition of the *protocol* (§8.2.7) that describes the interaction between the handlee and the effect handler.

After these definitions, we resume our step-by-step inspection of the code. We successively examine the combination of the handlee and the handler (§8.2.8), the handlee alone (§8.2.9), and the handler alone (§8.2.10), focusing in turn on the return branch (§8.2.11) and on the effect branch (§8.2.12). Along the way, we point out where Hazel's key reasoning rules are used. We do not show or explain these rules: for more information on Iris and Hazel, we refer the reader to the papers where these logics are presented [JKJ+18, dVP21].

8.2.1. *Initial Hypotheses and Goal.* The first step in the proof of Statement 8.1 is to introduce the metavariables e and E and the assumption e *isExp* E. This assumption is persistent, so it remains present until the end of the proof.

Hypothesis 8.4. *We assume that e is a value and that E is a mathematical expression. We assume that the persistent assertion e isExp E holds.*

The goal is then reduced to the following assertion:

$$\mathit{ewp}\ (\texttt{diff}\ e)\ \langle\bot\rangle\{e'.\ e'\ \mathit{isExp}\ E'\}$$

By definition of `diff`, the function call `diff` e reduces (in one step) to a value, namely, the value `{ eval = ... }` that appears at label head[8] in Figure 10, in which the variable `e` is replaced with the value e.

In the following, for the sake of readability, we elide the substitution $[e/\texttt{e}]$. We write that from here on, the variable `e` denotes the value e. We allow ourselves to write a goal where the variable `e` occurs, and we leave it to the reader to understand that, in such a goal, `e` means e. More generally, every time we reason about a reduction step that causes a substitution $[v/\texttt{x}]$ to arise, we elide this substitution. As this paper is accompanied with a machine-checked proof, we adopt a certain level of informality and choose to avoid the noise caused by these substitutions. Thus, whereas in our machine-checked proof the current goal always involves a closed term, in the paper our hypotheses, goals, and definitions have free variables. The reader must understand that each such variable actually denotes a certain closed value to which this variable has been bound at an earlier point in the proof.

With this convention in mind, we may write that `diff` e reduces to the value (head), eliding the substitution $[e/\texttt{e}]$. Thus, the goal can be transformed to:

$$(\texttt{head})\ \mathit{isExp}\ E'$$

where, by convention, we use the label head as a short-hand for the subexpression that is identified by this label in Figure 10.

The next step is to unfold *isExp* in this goal. The goal becomes:

$$\begin{array}{l}\square\,\forall\,\mathcal{R},\ \Psi_{\mathcal{R}},\ \mathit{isNum}_{\mathcal{R}},\ \texttt{zero}_{\mathcal{R}},\ \texttt{one}_{\mathcal{R}},\ \texttt{add}_{\mathcal{R}},\ \texttt{mul}_{\mathcal{R}}.\\ \mathit{isDict}\ (\mathcal{R}, \Psi_{\mathcal{R}}, \mathit{isNum}_{\mathcal{R}}, \texttt{zero}_{\mathcal{R}}, \texttt{one}_{\mathcal{R}}, \texttt{add}_{\mathcal{R}}, \texttt{mul}_{\mathcal{R}}) \mathrel{-\!\!*}\\ \forall\,\texttt{n}_{\mathcal{R}},\ r \in \mathcal{R}.\ \ \texttt{n}_{\mathcal{R}}\ \mathit{isNum}_{\mathcal{R}}\ r \mathrel{-\!\!*}\\ \mathit{ewp}\ ((\texttt{head}).\texttt{eval}\ \{\texttt{zero}_{\mathcal{R}};\ \texttt{one}_{\mathcal{R}};\ \texttt{add}_{\mathcal{R}};\ \texttt{mul}_{\mathcal{R}}\}\ \texttt{n}_{\mathcal{R}})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ y\ \mathit{isNum}_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r}\}\end{array}$$

We then introduce all of the metavariables and assumptions that appear in the first three lines of this goal. Again, these assumptions are persistent.

Hypothesis 8.5. *We assume a semiring $\mathcal{R}$, a protocol $\Psi_{\mathcal{R}}$, a predicate $\mathit{isNum}_{\mathcal{R}}$, four values $\texttt{zero}_{\mathcal{R}}$, $\texttt{one}_{\mathcal{R}}$, $\texttt{add}_{\mathcal{R}}$, $\texttt{mul}_{\mathcal{R}}$, a value $\texttt{n}_{\mathcal{R}}$, and a number $r \in \mathcal{R}$. We assume that the following two persistent assertions hold:*

$$\begin{array}{l}\mathit{isDict}\ (\mathcal{R}, \Psi_{\mathcal{R}}, \mathit{isNum}_{\mathcal{R}}, \texttt{zero}_{\mathcal{R}}, \texttt{one}_{\mathcal{R}}, \texttt{add}_{\mathcal{R}}, \texttt{mul}_{\mathcal{R}})\\ \texttt{n}_{\mathcal{R}}\ \mathit{isNum}_{\mathcal{R}}\ r\end{array}$$

The goal is now reduced to:

$$\mathit{ewp}\ ((\texttt{head}).\texttt{eval}\ \{\texttt{zero}_{\mathcal{R}};\ \texttt{one}_{\mathcal{R}};\ \texttt{add}_{\mathcal{R}};\ \texttt{mul}_{\mathcal{R}}\}\ \texttt{n}_{\mathcal{R}})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ y\ \mathit{isNum}_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r}\}$$

[8] In the electronic version of this paper, a label such as head is a hyperlink towards this label in Figure 10. In the reverse direction, a label in Figure 10 is a hyperlink to the point in the proof where this label is reached.

This expression reduces (in several steps) to the expression identified by the label `body`:

$$ewp\ (\texttt{body})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ y\ isNum_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r}\}$$

Again, in this goal, we have elided a substitution of actual arguments for formal parameters: from here on, the variables $\texttt{zero}, \texttt{one}, \texttt{add}, \texttt{mul}, \texttt{n}$ are bound to the values $\texttt{zero}_{\mathcal{R}}, \texttt{one}_{\mathcal{R}}, \texttt{add}_{\mathcal{R}}, \texttt{mul}_{\mathcal{R}}, \texttt{n}_{\mathcal{R}}$.

Between the label `body` and the label `init`, the code in Figure 10 contains a series of definitions. Each of these definitions binds a variable to a value: therefore, it has no side effect and reduces in one step, giving rise to a substitution of a value for a variable. In keeping with our convention, in the paper, we elide these substitutions. Thus, the previous goal reduces to the following goal:

Goal 8.6. *The goal is now:*

$$ewp\ (\texttt{init})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ y\ isNum_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r}\}$$

In words, the goal is to prove that the expression identified by the label `init` in Figure 10 obeys the protocol $\Psi_{\mathcal{R}}$ and eventually returns a representation of the number $[\![E']\!]_{\lambda X.r}$, which is the value of the expression E' at the point r.

8.2.2. *Auxiliary Variables.* During its forward and backward phases, the algorithm allocates objects of type `t` (Figure 10, line 6). We refer to a value of this type as a *vertex* (§3.2). In our proof (which is based not the OCaml code, but on the *HH* code), such a value is either the value `O` or the value `I` or an *auxiliary variable* (§6.2), that is, a value of the form `Var` (m, ℓ), where m is the content of the `d` field and ℓ is the address of the `v` field.

To describe the content of an auxiliary variable in a concise fashion, we introduce a predicate *isVar*. In short, the assertion $u\ isVar\,(s,\ \dot{s})$ means that u is an auxiliary variable whose `v` and `d` fields contain (representations of) the numbers s and $\dot{s}$. As is usual in Separation Logic, this assertion also represents the unique ownership of the auxiliary variable u. In other words, it represents a unique permission to read and update this auxiliary variable.

Definition 8.7. The predicate *isVar* is defined as follows:

$$\begin{array}{l} u\ isVar\,(s,\ \dot{s})\ \triangleq \\ \quad \exists\, m,\ \ell,\ d. \\ \qquad *\left\{\begin{array}{lcl} u = \texttt{Var}\ (m, \ell) & * & \ell \mapsto d \\ m\ isNum_{\mathcal{R}}\ s & * & d\ isNum_{\mathcal{R}}\ \dot{s} \end{array}\right. \end{array}$$

In this definition, u is a value and s and $\dot{s}$ are numbers, that is, inhabitants of the semiring $\mathcal{R}$, which was introduced in Hypothesis 8.5.

8.2.3. *The First Auxiliary Variable.* Let us now come back to the current goal, that is, Goal 8.6. The subexpression at label `init` begins with the definition **`let x = mk n`**. This definition has a side effect: it allocates a fresh auxiliary variable in the heap. The variable `x` becomes bound to a certain value, which we do not explicitly describe: instead, in keeping with our convention, we use the variable `x` to stand for it. Therefore, we can say that this fresh auxiliary variable is described by the assertion $\texttt{x}\ isVar\,(r,\ 0)$.[9]

[9] As the reader may recall, the variable `n` denotes the value $\texttt{n}_{\mathcal{R}}$, which, according to Hypothesis 8.5, is a runtime representation of the number r.

Because this assertion is not persistent, we do not present it as a displayed Hypothesis. Instead, we show it as part of the current goal.

Goal 8.8. *The previous goal, namely Goal 8.6, is replaced with:*

$$\mathsf{x}\ \mathit{isVar}\,(r,\ 0) \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{heart})\ \langle \Psi_{\mathcal{R}} \rangle \{ y.\ y\ \mathit{isNum}_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r} \}$$

In other words, we are now at label `heart` and we have allocated one auxiliary variable x.

8.2.4. *Ghost Theory.* During their forward phase, both the stack-based (§3.3) and the effect-based (§5) implementations of reverse-mode AD construct a sequential view of the expression E. To describe this view at the logical level, we have defined a *context* K to be a list of *bindings* B (§6.2).

While reasoning about the execution of the forward phase, we would like to speak of the *current context*, which records the bindings created so far. Furthermore, we would like to remark that the current context evolves in a monotonic manner: a new binding can be added at its right end, but an existing binding is never removed. Thus, once a binding B appears in the current context, this fact remains true forever.

These informal considerations can be made precise in Iris (therefore also in Hazel) using ghost state. In our setting, a single ghost cell, which holds the current context, suffices for this purpose. For the sake of brevity and simplicity, we do not describe the low-level implementation of our ghost state: we refer the reader to Timany and Birkedal's work [TB21] for general principles and to our Coq proof [dVP22a] for more detail. Instead, we describe the abstract API offered by our ghost state. This API is given by the following lemma, whose proof forms a small library that is separate from our main proof.

Lemma 8.9 (Custom Ghost State)**.** *The following closed assertion is valid:*

$$\dot{\Rightarrow}\ \exists\, \mathit{isContext}, \mathit{isBinding}.\ *\begin{cases} \mathit{isContext}\ [] \\ \textsc{NewBinding} \\ \quad \square\ \forall K, B.\ \mathit{isContext}\ K \mathrel{-\!\!*} \dot{\Rightarrow} * \begin{cases} \mathit{isContext}\ (K; B) \\ \mathit{isBinding}\ B \end{cases} \\ \textsc{ExploitBinding} \\ \quad \square\ \forall K, B.\ \mathit{isContext}\ K \mathrel{-\!\!*} \mathit{isBinding}\ B \mathrel{-\!\!*} B \in K \\ \textsc{PersistBinding} \\ \quad \square\ \forall B.\ \mathit{isBinding}\ B \mathrel{-\!\!*} \square\ \mathit{isBinding}\ B \end{cases}$$

The symbol $\dot{\Rightarrow}$ is the *update* modality: the assertion $\dot{\Rightarrow} P$ means that it is possible to update the ghost state in such a way that the assertion P holds. Thus, this lemma can be read as follows: provided a certain ghost update is performed (namely, the allocation of a ghost cell which holds the current context), it is possible to define two abstract predicates *isContext* and *isBinding* such that:

(1) initially $\mathit{isContext}\ []$ holds, that is, the current context is empty;
(2) as many times as one wishes, $\mathit{isContext}\ K$ can be transformed via a ghost update into $\mathit{isContext}\ (K; B) * \mathit{isBinding}\ B$, thus extending the current context with a new binding;
(3) the conjunction $\mathit{isContext}\ K * \mathit{isBinding}\ B$ implies $B \in K$, which means that if B has been once inserted into the current context, then it remains forever in the current context;
(4) the assertion $\mathit{isBinding}\ B$ is persistent.

The assertion *isContext* K states that the current context is K. It is not persistent: the current context changes over time. The assertion *isBinding* B states that the binding B appears in the current context. It is persistent: a binding, once created, exists forever in the current context.

Lemma 8.9 states that one *can* allocate a piece of ghost state that has the properties described above. The next step in our main proof is to *actually* allocate this ghost state. We do so by eliminating the ghost update and the existential quantifiers that appear in the statement of Lemma 8.9. This enriches our main proof with new persistent hypotheses:

Hypothesis 8.10. *We assume that the predicates isContext and isBinding satisfy the laws* NewBinding, ExploitBinding, *and* PersistBinding.

The assertion *isContext* [] is not persistent, so we do not list it as part of Hypothesis 8.10. Instead, we show it as part of the current goal.

Goal 8.11. *The previous goal, namely Goal 8.8, is replaced with:*

$$\texttt{x}\ \mathit{isVar}\ (r,\ 0) \mathrel{-\!\!*} \mathit{isContext}\ [] \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{heart})\ \langle \Psi_{\mathcal{R}} \rangle \{ y.\ y\ \mathit{isNum}_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r} \}$$

In other words, we are still at the program point `heart`, we have allocated the auxiliary variable `x`, and we have created a ghost current context, which is currently empty. It is now time to spell out the invariants of the forward phase and of the backward phase.

8.2.5. *Forward and Backward Invariants.* We are now ready to indicate exactly what values are stored, during the forward and backward phases, in the `v` field and `d` field of each auxiliary variable. For this purpose, we introduce two invariants. The *forward invariant* describes the content of these fields during the forward phase; the *backward invariant* describes their content during the backward phase.

The environment η, defined next, appears in the definitions of the forward and backward invariants. Its role is as follows. The variables `zero'`, `one'`, `x` (bound at lines 14, 15 and 22) stand for certain values.[10] We wish to express the idea that the values `zero'` and `one'` represent the numbers 0 and 1, respectively; and that the value `x` represents the number $r \in \mathcal{R}$, introduced in Hypothesis 8.5. The environment η serves this purpose:

Definition 8.12. Let $\eta : \mathit{Val} \to \mathcal{R}$ be an environment that maps the value `zero'` to 0, the value `one'` to 1, and the value `x` to r.

The environment η is useful because our invariants involve expressions in $\mathit{Exp}_{\mathit{Val}}$, that is, expressions whose leaves are values. We typically evaluate these expressions in the environment η or in an extension of this environment. For this purpose, we define two shorthands:

Definition 8.13 (Expression Evaluation Shorthands)**.** We write $(\!|E|\!)$ as a shorthand for $[\![E]\!]_{\eta}$. We write $(\!|E|\!)_K$ as a shorthand for $[\![E]\!]_{\eta\{K\}}$.

We can now give the definitions of the forward and backward invariants.

The forward invariant states that, during the forward phase:

- every auxiliary variable u created so far represents the mathematical expression $K[u]$, where the current context K records all of the operations performed so far; and
- the `v` field of the auxiliary variable u stores the value of this expression.

[10] `zero'` and `one'` denote the values `O` and `I`, while `x` denotes the memory location allocated at line 22.

This is expressed as follows:

Definition 8.14 (Forward Invariant)**.** The forward invariant *ForwardInv* K, an assertion parameterized by a context K, is defined as follows:

$$\begin{array}{rcl} \mathit{ForwardInv}\ K & \triangleq & \\ \mathit{isContext}\ K & * & \left(\begin{array}{l} \forall u \in \{\mathtt{x}\} \cup \mathit{defs}(K). \\ \quad \mathit{leaves}(K[u]) \subseteq \{\mathtt{zero'}, \mathtt{one'}, \mathtt{x}\} * \\ \quad u\ \mathit{isVar}\ (\!|K[u]|\!),\ 0) \end{array} \right) \end{array}$$

The first conjunct, *isContext* K, asserts that K is the current context (§8.2.4). The second conjunct asserts that K is *well-formed*: that is, filling K with an auxiliary variable u, where $u \in \{\mathtt{x}\} \cup \mathit{defs}(K)$, yields an expression $K[u]$ whose leaves are (a subset of) the values `zero'`, `one'`, and `x`. Therefore, it makes sense to evaluate the expression $K[u]$ under the environment η. The forward invariant asserts that the result of this evaluation, namely the number $(\!|K[u]|\!)$, is stored in the `v` field of the auxiliary variable u.

Let us now move on to the backward invariant.

During the forward phase, a sequence of arithmetic operations takes place, which we represent by the current context K, a sequence of bindings. At the end of the forward phase, this current context becomes fixed. During the backward phase, the bindings in K are examined and processed in reverse order: that is, the rightmost bindings in the list K are processed first. Therefore, during the backward phase, it is natural to split K into two contexts, that is, to write K as a concatenation $K_1; K_2$, where K_1 contains the *pending bindings*, which have not yet been processed, and K_2 contains the *processed bindings*.

The backward invariant is defined as follows:

Definition 8.15 (Backward Invariant)**.** The backward invariant *BackwardInv* K_1, an assertion parameterized by a context K_1, is defined as follows:

$$\mathit{BackwardInv}\ K_1 \triangleq \exists K_2, y.\ \mathit{BackwardInvBody}\ K_1\ K_2\ y$$

$$\mathit{BackwardInvBody}\ K_1\ K_2\ y \triangleq * \begin{cases} \text{(A)} & [\![E']\!]_{\lambda X.r} \equiv_{\mathcal{R}} (\!|\partial(K_1; K_2)[y]/\partial \mathtt{x}|\!) \\ \text{(B)} & \forall\, u \in \{\mathtt{x}\} \cup \mathit{defs}(K_1). \\ & \quad u\ \mathit{isVar}\ (\!|K_1[u]|\!),\ (\!|\partial K_2[y]/\partial u|\!)_{K_1}) \end{cases}$$

The backward invariant is parameterized by the pending bindings K_1 and begins with an existential quantification over the processed bindings K_2 and over the root auxiliary variable y. It states that:

(A) the number that the algorithm aims to compute, namely the value of the symbolic derivative E', is the value of the partial derivative $\partial K[y]/\partial x$.

(B) the `d` field of every *pending* auxiliary variable u stores the partial derivative of the expression $K_2[y]$ with respect to u.

Point A intuitively follows from the fact that the expression $K[y]$ is the expression E. Technically, however, we cannot write $K[y] = E$ because $K[y]$ inhabits $\mathit{Exp}_{\mathit{Val}}$ (it is an expression whose leaves are values) whereas E inhabits $\mathit{Exp}_{\{X\}}$ (it is an expression whose sole leaf is the variable X).

Point B reveals a fundamental distinction between processed and pending auxiliary variables during the backward phase. A processed auxiliary variable $u \in \mathit{defs}(K_2)$ is never read or written again: in fact, its ownership has been abandoned, as it is not mentioned in the invariant. At the logical level, because the binding that defines u is part of K_2, this

binding influences the meaning of the expression $K_2[y]$. Thus, the auxiliary variable u is merely a name for a subexpression of the expression $K_2[y]$. In contrast, a pending auxiliary variable $u \in \{\mathtt{x}\} \cup \mathit{defs}(K_1)$ can be thought of as a variable in a mathematical sense: indeed, it can be a leaf of the expression $K_2[y]$, and the value of the partial derivative $\partial K_2[y]/\partial u$ is stored in memory in the `d` field of the auxiliary variable u.

8.2.6. *Abstract Semiring Structure for Vertices.* We have just given much detail about the representation of vertices in memory and about their role during the forward and backward phases. However, in the eyes of the expression `e : exp` with which the differentiation algorithm interacts, vertices must be presented as abstract objects equipped with semiring structure. Indeed, an expression `e : exp` is polymorphic in a semiring.

To equip vertices with semiring structure, we exploit the idea that a vertex represents an expression in $\mathit{Exp}_{\{X\}}$. Expressions have semiring structure: indeed, $\mathit{Exp}_{\{X\}}$ is a semiring, the free semiring (§6.1).

To express the idea that a vertex u represents an expression $E \in \mathit{Exp}_{\{X\}}$, we must define an assertion $u\ \mathit{isSubExp}\ E$, where $u \in \mathit{Val}$ is a vertex and $E \in \mathit{Exp}_{\{X\}}$ is an expression. Anticipating on the fact that we will need to instantiate the parameter *isNum* in the definition of *isDict* (Definition 8.3) with *isSubExp*, we want *isSubExp* to be compatible with the relation $\equiv_{\mathit{Exp}_{\{X\}}}$ and to be persistent.

We would like the definition of *isSubExp* to express the intuitive idea that a vertex u represents the expression $K[u]$, where K is the current context. However, in the definition of *isSubExp*, we cannot rely on the assertion $\mathit{isContext}\, K$, because it is not persistent: it represents the unique ownership of the ghost current context. Fortunately, we can instead rely on the predicate *isBinding*, which is persistent. In so doing, we exploit the fact that the current context evolves in a monotonic manner.

Definition 8.16 (Abstract View of Vertices)**.** The assertion $u\ \mathit{isSubExp}\ E$ and the assertion $u\ \mathit{isSubExpRaw}\ E$ are inductively defined as follows:

$$
\begin{aligned}
u\ \mathit{isSubExp}\ E &\triangleq \exists T.\quad u\ \mathit{isSubExpRaw}\ T \quad * \quad T \equiv_{\mathit{Exp}_{\{X\}}} E \\
u\ \mathit{isSubExpRaw}\ \mathbf{0} &\triangleq u = \mathtt{zero'} \\
u\ \mathit{isSubExpRaw}\ \mathbf{1} &\triangleq u = \mathtt{one'} \\
u\ \mathit{isSubExpRaw}\ (\mathbf{Leaf}\ X) &\triangleq u = \mathtt{x} \\
u\ \mathit{isSubExpRaw}\ (E_a\ \mathit{op}\ E_b) &\triangleq \exists a, b.\quad \mathit{isBinding}\,(\mathsf{let}\ u = a\ \mathit{op}\ b)\ * \\
&\qquad\qquad a\ \mathit{isSubExpRaw}\ E_a\ *\ b\ \mathit{isSubExpRaw}\ E_b
\end{aligned}
$$

The vertices `zero'`, `one'` and `x` respectively represent the expressions $\mathbf{0}$, $\mathbf{1}$, and $\mathbf{Leaf}\ X$. If a vertex u is the result of an arithmetic operation *op* whose operands were the vertices a and b, where a represents E_a and b represents E_b, then u represents the expression $E_a\ \mathit{op}\ E_b$. In summary, the assertion $u\ \mathit{isSubExpRaw}\ E$ means that decoding the DAG that exists in memory, beginning at the vertex u, yields the tree E; and *isSubExp* is just the compatible closure of *isSubExpRaw*.

At this point, the reader might expect us to prove that we have successfully equipped vertices with semiring structure, by establishing an *isDict* assertion. Indeed, we are almost ready to do this; we will do so in Lemma 8.18. Before we can state this lemma, however, we must provide a description of the effects that the functions `add'` and `mul'` are allowed to perform. In Hazel, this is done by defining a *protocol*.

8.2.7. *An Effect Protocol.* The arithmetic operations `add'` (line 16) and `mul'` (line 17) perform effects. In Hazel, an effect is described by a protocol [dVP21], which describes the precondition and the postcondition of the **`perform`** instruction. A protocol serves as a contract that mediates the interaction between a handlee and a handler. We must now define the protocol that is in use in the reverse-mode AD algorithm of Figure 10.

The functions `add'` and `mul'` are trivial: their bodies consist of a single **`perform`** instruction. This remark helps us guess how to define the protocol: the protocol should literally paraphrase the specifications that we wish to give for the functions `add'` and `mul'`.

What should these specifications be? In the eyes of the expression `e : exp` with which the differentiation algorithm interacts, a vertex is an abstract object, which represents an expression (§8.2.6). We wish to state that `add'`, applied to two arguments a and b that represent two expressions E_a and E_b, produces a result u that represents the expression E_a**+**E_b. We wish to make a similar statement about `mul'`.

This leads us to define the protocol *OP* (for "operation") in the following way.

Definition 8.17 (Protocol)**.** The protocol *OP* is defined as follows:

$$\begin{array}{lll} OP \triangleq ! \; op \; a \; b \; E_a \; E_b & (op, a, b) & \{a \; isSubExpRaw \; E_a \; * \; b \; isSubExpRaw \; E_b\}. \\ \quad ? \; u & (u) & \{u \; isSubExpRaw \; (E_a \; op \; E_b)\} \end{array}$$

This is a *send-receive* protocol [dVP21]. The first line describes the message that is sent by the handlee; the second line describes the reply that is sent by the handler. Each line begins with a series of binders, followed with the value that must be sent (on the first line, the triple (op, a, b); on the second line, the value u), followed with an assertion that must be satisfied.

Intuitively, this protocol states that, to perform an effect *op* with operands a and b, there must exist expressions E_a and E_b, such that a represents E_a and b represents E_b; and that, if these conditions are met, then a vertex u representing E_a *op* E_b can be expected as a result of performing this effect.

Thanks to this definition, we can now state and prove that we have successfully equipped our vertices with semiring structure:

Lemma 8.18 (Semiring Structure for Vertices)**.** *This persistent assertion holds:*

$$isDict \; (Exp_{\{X\}}, OP, isSubExp, \texttt{zero'}, \texttt{one'}, \texttt{add'}, \texttt{mul'})$$

The proof is easy. A look at Definition 8.3 helps recall the six proof obligations: the values `zero'`, `one'`, `add'`, `mul'` must implement the four semiring operations; the predicate *isSubExp* must be compatible and persistent.

8.2.8. *At the Heart of the Algorithm.* After a long digression, it may seem, we come back to the main narrative of the proof. Our current goal is still Goal 8.11:

$$\texttt{x} \; isVar \; (r, \; 0) \mathrel{-\!\!*} isContext \; [] \mathrel{-\!\!*} ewp \; (\texttt{heart}) \; \langle \Psi_{\mathcal{R}} \rangle \{y. \; y \; isNum_{\mathcal{R}} \; [\![E']\!]_{\lambda X.r}\}$$

Our next step is to establish that, at this point, the forward invariant holds. It is not difficult to check that the entailment $isContext \; [] \; * \; \texttt{x} \; isVar \; (r, \; 0) \vdash ForwardInv \; []$ holds, so the current goal can be changed to:

$$ForwardInv \; [] \mathrel{-\!\!*} ewp \; (\texttt{heart}) \; \langle \Psi_{\mathcal{R}} \rangle \{y. \; y \; isNum_{\mathcal{R}} \; [\![E']\!]_{\lambda X.r}\}$$

Because the expression at label `heart` is the sequential composition of two subexpressions, which are respectively identified by the labels `handle` and `done`, this goal can be decomposed (via the sequencing rule of Separation Logic) into the following two goals:

Goal 8.19. *The correctness of the forward and backward phases is expressed by this goal:*

$$\mathit{ForwardInv}\ [] \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{handle})\ \langle \Psi_{\mathcal{R}} \rangle \{ \_.\ \mathit{BackwardInv}\ [] \}$$

Goal 8.20. *The correctness of the final read is expressed by this goal:*

$$\mathit{BackwardInv}\ [] \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{done})\ \langle \Psi_{\mathcal{R}} \rangle \{ y.\ y\ \mathit{isNum}_{\mathcal{R}}\ [\![E']\!]_{\lambda X.r} \}$$

We have ingenuously guessed that the assertion *BackwardInv* [] holds at the program point `done`.

The proof of Goal 8.20 is easy. Indeed, when K_1 is empty, the backward invariant implies that the desired result, namely the number $[\![E']\!]_{\lambda X.r}$, is equal to the number $(\!|\partial K_2[y]/\partial \mathsf{x}|\!)$, which is stored in the `d` field of the auxiliary variable `x`.

Therefore, the sole outstanding goal is Goal 8.19. It is arguably the most interesting goal in this entire proof, because both the forward and the backward phases occur during the execution of the subexpression identified by the label `handle`, and because this subexpression is a **`match ... with effect ...`** construct, that is, a handlee wrapped in a deep handler. The ability to reason about this construct is a unique feature of Hazel, in contrast with plain Iris. For this purpose, Hazel offers the following reasoning rule [dVP21]:

$$\frac{\textsc{Try-With-Deep} \qquad \mathit{ewp}\ \mathtt{e}\ \langle \Psi \rangle \{ \phi \} \qquad \mathit{deep\text{-}handler}\ \langle \Psi \rangle \{ \phi \}\ (\mathtt{h} \mid \mathtt{r})\ \langle \Psi' \rangle \{ \phi' \}}{\mathit{ewp}\ (\textbf{match}\ \mathtt{e}\ \textbf{with}\ \mathtt{h} \mid \mathtt{r})\ \langle \Psi' \rangle \{ \phi' \}}$$

In the syntax of *HH*, a handler consists of two branches: an effect branch `h` and a return branch `r`. This reasoning rule allows the handlee `e` and the handler (`h` | `r`) to be separately verified, provided that they agree on a protocol Ψ, which describes the effects that the handlee performs and that the handler's effect branch intercepts; and on a postcondition ϕ, which describes the value that the handlee returns and that the handler's return branch receives. The rule's first premise corresponds to the verification of the handlee. The rule's second premise, a *deep-handler judgment*, requires the verification of the handler.

We now apply the reasoning rule Try-With-Deep to Goal 8.19, choosing to instantiate the metavariable Ψ with the protocol *OP* and the metavariable ϕ with the postcondition $\lambda y.\ y\ \mathit{isSubExp}\ E$. The metavariables Ψ' and ϕ' must be instantiated with the protocol $\Psi_{\mathcal{R}}$ and with the postcondition $\lambda \_.\ \mathit{BackwardInv}\ []$. This results in two goals, Goal 8.21 and Goal 8.23, which we separately examine.

8.2.9. *Reasoning About the Handlee.* The first goal that results from the application of Try-With-Deep is the following:

Goal 8.21. *The correctness of the handlee is expressed as follows:*

$$\mathit{ewp}\ (\texttt{eval})\ \langle \mathit{OP} \rangle \{ y.\ y\ \mathit{isSubExp}\ E \}$$

This goal requires verifying that the code at label `eval` computes a value y that represents the expression E. During its execution, this code may perform effects according to the protocol *OP*.

A look at Figure 10 shows that the code at label `eval` is the function call `e.eval dict x`.

Goal 8.21 is proved as follows. In the assertion e *isExp* E (Hypothesis 8.4), we unfold *isExp* (Definition 8.2) and we instantiate the universally quantified metavariables $\mathcal{R}$, Ψ, *isNum*, `zero`, `one`, `add`, and `mul` respectively with the free semiring $Exp_{\{X\}}$, the protocol OP, the predicate *isSubExp*, and the values `zero'`, `one'`, `add'`, and `mul'`. We obtain the following fact:

$$\begin{array}{l} \mathit{isDict}\,(\mathit{Exp}_{\{X\}}, \mathit{OP}, \mathit{isSubExp}, \texttt{zero'}, \texttt{one'}, \texttt{add'}, \texttt{mul'}) \mathrel{-\!\!*} \\ \forall\, \texttt{n}.\ \forall r \in \mathit{Exp}_{\{X\}}. \\ \texttt{n}\ \mathit{isSubExp}\ r \mathrel{-\!\!*} \\ \mathit{ewp}\ (\texttt{eval})\ \langle \mathit{OP} \rangle \{y.\ y\ \mathit{isSubExp}\ [\![E]\!]_{\lambda X.r}\} \end{array}$$

Thanks to Lemma 8.18, the requirement on the first line holds. We instantiate the metavariables `n` and r on the second line with `x` and **Leaf** X. By Definition 8.16, the instantiated assertion on the third line, `x` *isSubExp* (**Leaf** X), simplifies to $\texttt{x} = \texttt{x}$. Thanks to the identity $[\![E]\!]_{\lambda X.\mathbf{Leaf}\ X} = E$ (which is proved by induction on E), the fourth line becomes Goal 8.21.

8.2.10. *Reasoning About the Handler.* The second goal that results from the application of Try-With-Deep is a *deep-handler judgment*. In general, a deep-handler judgment has the following shape:

$$\mathit{deep\text{-}handler}\ \langle \Psi \rangle \{\Phi\}\ (\ \texttt{h}\ \mid\ \texttt{r}\ )\ \langle \Psi' \rangle \{\Phi'\} \tag{8.1}$$

We omit the definition of the deep-handler judgment, which the reader can find in a previous paper by the authors [dVP21]. For the purposes of this proof, it is sufficient to know that this judgment includes correctness statements for both the effect branch `h` and the return branch `r`.

To present the second goal that results from the application of Try-With-Deep, we introduce an abbreviation, *isHandler*. In anticipation for an inductive proof, we parameterize this abbreviation with a context K_1.

Definition 8.22 (Correct Handler)**.**

$$\begin{array}{l} \mathit{isHandler}\ K_1\ \triangleq \\ \quad \mathit{deep\text{-}handler}\ \langle \mathit{OP} \rangle\ \{y.\ y\ \mathit{isSubExp}\ E\} \\ \qquad\qquad (\ \texttt{eff-add}\ \&\ \texttt{eff-mul}\ \mid\ \texttt{ret}\ ) \\ \qquad\qquad \langle \Psi_{\mathcal{R}} \rangle\ \{\_.\ \mathit{BackwardInv}\ K_1\} \end{array}$$

The assertion *isHandler* K_1 states that our effect handler handles effects according to the protocol OP and is itself allowed to perform effects according to the protocol $\Psi_{\mathcal{R}}$. It also states that the handlee must return a value y that represents the expression E and that the handler (that is, the handler's effect branch and return branch) must terminate in a state where the bindings that have not yet been processed (during the backward phase) are K_1.

This abbreviation allows us to state the goal as follows:

Goal 8.23. *The correctness of the first instance of the handler is expressed as follows:*

$$\mathit{ForwardInv}\ [] \mathrel{-\!\!*} \mathit{isHandler}\ []$$

This is the sole outstanding goal.

We remark that the application of Try-With-Deep requires the handlee and the handler to be separately verified. Therefore, the assertion *ForwardInv* [], which was present before the rule was applied (Goal 8.19), must be transferred either to the handlee or to the handler.

This assertion is not needed while reasoning about the handlee (§8.2.9); therefore, we have transferred it to the handler. This explains why it appears in Goal 8.23.

In order to establish Goal 8.23, we transform it into a more general statement, which is amenable to a proof by induction:

Goal 8.24. *The correctness of an arbitrary instance of the handler is expressed as follows:*

$$\forall\, K_1.\ \mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \mathit{isHandler}\ K_1$$

Although we do not show the definition of the deep-handler judgment, we can reveal that it is a *non-separating conjunction* of a statement about the effect branch and a statement about the return branch. Thus, the assertion $\mathit{ForwardInv}\ K_1$ serves as a precondition both for the effect branch and for the return branch.

The universal quantification on K_1 allows us to reason about a point in time where the effect branch or the return branch is entered and the current context is K_1. The occurrence of K_1 in $\mathit{ForwardInv}\ K_1$ means that, when the handler is invoked, a sequence of arithmetic operations described by K_1 has been performed already. The occurrence of K_1 in $\mathit{isHandler}\ K_1$ means that, when the effect branch or the return branch terminates, the bindings that have *not yet been processed* in the backward phase are exactly the bindings in K_1. This reflects the well-bracketed manner in which each instance of the handler is executed: the bindings that have already been constructed (in the forward phase) when the execution of the handler begins are also the bindings that have not yet been processed (in the backward phase) when the execution of the handler ends.

To prove Goal 8.24, we use *Löb induction* [JKJ+18]. The reason why induction is required is the recursive nature of deep handlers: a deep handler is syntactic sugar for a recursive function in which a shallow handler is installed [dVP21]. The application of Löb's induction principle gives rise to the following induction hypothesis:

Hypothesis 8.25 (Induction Hypothesis)**.**

$$\triangleright\ (\forall\, K_1.\ \mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \mathit{isHandler}\ K_1)$$

The symbol $\triangleright$ is the *later* modality [JKJ+18]. It prevents circular proofs: in its absence, Hypothesis 8.25 would coincide with Goal 8.24, so the proof would be finished—but that would of course be an unsound form of reasoning. The later modality can be eliminated only after the program has performed at least one execution step, thereby forbidding this kind of circular reasoning.

The current goal is still Goal 8.24. This goal begins with a universal quantification on K_1, which we now introduce. This yields the following hypothesis and goal:

Hypothesis 8.26. *We assume that a context K_1 is given.*

Goal 8.27. *The goal is now:*

$$\mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \mathit{isHandler}\ K_1$$

Unfolding *isHandler*, unfolding the definition of the deep-handler judgment, and applying the introduction rule for a non-separating conjunction reduces this goal to two subgoals: the *verification of the return branch* `ret` and the *verification of the effect branch* `eff-add` & `eff-mul`. The former is discussed below (Goal 8.28). The latter is subdivided into one subgoal related to the addition branch `eff-add` (Goal 8.29) and one subgoal related to the multiplication branch `eff-mul`, whose description we omit, as it is analogous.

8.2.11. *The Return Branch.* The body of the return branch is the expression labeled `seed`.

Goal 8.28. *The correctness of the return branch is expressed as follows:*

$$\begin{array}{l} \mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \\ y\ \mathit{isSubExp}\ E \mathrel{-\!\!*} \\ \mathit{ewp}\ (\texttt{seed})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInv}\ K_1\} \end{array}$$

The return branch is invoked when the function call `e.eval dict x` terminates. At this moment, the forward phase ends and the backward phase begins. The return branch consists of just one instruction, namely `update y one`. Under the assumption that the forward invariant holds and that K_1 is the current context, we must prove that this instruction establishes the backward invariant, where every binding in K_1 is pending. We may also assume that the value y satisfies the postcondition of the handlee, which is $\lambda y.\ y\ \mathit{isSubExp}\ E$.

Confronting $\mathit{ForwardInv}\ K_1$ and $y\ \mathit{isSubExp}\ E$ allows us to deduce $y \in \mathit{defs}(K_1)$. This step exploits the reasoning rule ExploitBinding. What is more, confronting these assertions allows us to establish a link between the expression E and the expression $K_1[y]$. In the interest of space, we omit the details.

To justify that `update y one` can be safely executed, an *isVar* assertion, which represents a permission to update the `d` field of y, must be presented. Because $y \in \mathit{defs}(K_1)$ holds, this permission can be obtained from the forward invariant $\mathit{ForwardInv}\ K_1$.

After executing `update y one`, this *isVar* assertion is updated so as to reflect the fact that the `d` field of y now contains the number 1, and the goal is to establish the postcondition:

$$\exists\, K_2,\ y.\ \mathit{BackwardInvBody}\ K_1\ K_2\ y$$

To do so, the existentially quantified variables K_2 and y are instantiated with [] and y. We omit the details of the remainder of the proof of Goal 8.28. In short, verifying that the backward invariant holds involves checking that the `d` field of the auxiliary variable y contains the number 1 and that the `d` field of every other auxiliary variable u contains the number 0. Indeed, very roughly speaking, $\partial y/\partial y$ is 1 and $\partial y/\partial u$ is 0.

8.2.12. *The Addition Effect Branch.* The body of the effect branch, in the case of addition, is the expression labeled `add-body`.

Goal 8.29. *The correctness of the addition effect branch is expressed as follows:*

$$\begin{array}{l} \forall\, a,\ b,\ E_a,\ E_b,\ k. \\ \mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \\ a\ \mathit{isSubExpRaw}\ E_a \mathrel{-\!\!*} b\ \mathit{isSubExpRaw}\ E_b \mathrel{-\!\!*} \\ \mathit{isCont}\ k \mathrel{-\!\!*} \\ \mathit{ewp}\ (\texttt{add-body})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInv}\ K_1\} \end{array}$$

The manner in which K_1 is shared between the precondition $\mathit{ForwardInv}\ K_1$ and the postcondition $\mathit{BackwardInv}\ K_1$ expresses the fact that the execution of this branch begins at an arbitrary point of the forward phase, after a sequence of arithmetic operations described by K_1 has been performed, and ends at the corresponding point in the backward phase, when the sequence of pending bindings is K_1.

The precondition also includes the assertions $a\ \mathit{isSubExpRaw}\ E_a$ and $b\ \mathit{isSubExpRaw}\ E_b$, which mean that the values a and b represent two expressions E_a and E_b. The effect handler can make these assumptions about a and b thanks to the protocol OP (Definition 8.17),

where these assertions appear as a requirement (a precondition) from the point of view of the code that performs an effect.

The last part of the precondition is the assertion $\mathit{isCont}\, k$, whose definition is as follows. For the sake of simplicity, we present a definition of *isCont* where two universal quantifiers have been instantiated in a wise manner. This makes the hypothesis $\mathit{isCont}\, k$ weaker than it could be, but still sufficient for our purposes. This allows us to use the abbreviation *isHandler* in the definition of *isCont* and thereby to present a simplified definition of *isCont*.

Definition 8.30. The assertion $\mathit{isCont}\, k$ is defined as follows:

$$
\begin{array}{l}
\mathit{isCont}\, k \;\triangleq \\
\quad \forall\, u. \\
\quad \mathit{let}\ K_1' := (K_1; \mathsf{let}\ u = a + b)\ \mathit{in} \\
\quad \triangleright\, \mathit{isHandler}\ K_1' \mathrel{-\!\!*} \\
\quad u\ \mathit{isSubExpRaw}\ (E_a + E_b) \mathrel{-\!\!*} \\
\quad \mathit{ewp}\ (k\ u)\ \langle \Psi_{\mathcal{R}} \rangle \{ \_.\ \mathit{BackwardInv}\ K_1' \}
\end{array}
$$

The assertion $\mathit{isCont}\, k$ expresses a hypothesis about the continuation k. It is the *specification of the continuation*: it states under what condition one can invoke the continuation, and what property one can expect to hold as a result of this call.

This specification takes the form of an *ewp* judgment about a function call of the form $k\ u$, where u is a value. Because an *ewp* judgment is *not* persistent, the assertion $\mathit{isCont}\, k$ is not persistent either. It represents a permission to invoke the continuation k *at most once*.

In essence, this specification states that k can be applied to a vertex u, provided that u represents the expression $E_a + E_b$. The handler must satisfy this constraint when invoking k because of the protocol *OP* (Definition 8.17), where this constraint appears as a guarantee (a postcondition) from the point of view of the code that performs an effect.

The continuation invocation $k\ u$ is subject to another precondition, $\mathit{isHandler}\ K_1'$, and its postcondition is $\mathit{BackwardInv}\ K_1'$, where we have chosen to let K_1' stand for $K_1; \mathsf{let}\ u = a + b$, that is, for the extension of the sequence K_1 with the new binding $\mathsf{let}\ u = a + b$. We have made this decision because we know that this invocation begins during the forward phase after the sequence of operations K_1' has been performed and ends during the backward phase when the sequence of pending bindings is K_1'. The precondition $\mathit{isHandler}\ K_1'$ reflects the fact that this is a deep handler: a new instance of the handler is reinstalled as the bottommost frame inside the continuation k. Thus, in order to be allowed to invoke this continuation, we must prove that this new instance of the handler is also a correct handler.

Let us now attack the proof of Goal 8.29. The code at `add-body` is a sequential composition of three segments, namely: the allocation of a new auxiliary variable, at `add-fwd`, part of the forward phase; the invocation of the continuation, at `cont`; and two update instructions, at `add-bwd`, part of the backward phase. The sequencing rule and the frame rule of Separation Logic let us verify each segment independently, as follows.

Segment 1: Forward Phase.

Goal 8.31. *The correctness of the instruction at label* `add-fwd` *is expressed as follows:*

$$
\begin{array}{l}
\mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \\
a\ \mathit{isSubExpRaw}\ E_a \mathrel{-\!\!*} b\ \mathit{isSubExpRaw}\ E_b \mathrel{-\!\!*} \\
\mathit{ewp}\ (\texttt{add-fwd})\ \langle \Psi_{\mathcal{R}} \rangle \{ u. \\
\quad \mathit{let}\ K_1' := (K_1; \mathsf{let}\ u = a + b)\ \mathit{in} \\
\quad \mathit{ForwardInv}\ K_1' \ast u\ \mathit{isSubExpRaw}\ (E_a + E_b) \}
\end{array}
$$

The appearance of the forward invariant *ForwardInv* in the pre- and postcondition indicates that the execution of `add-fwd` takes place during the forward phase.

The fact that *ForwardInv* K_1 is transformed into *ForwardInv* K_1' reflects the fact that a new arithmetic operation has taken place. To perform this transformation, one must begin with the following two steps:

- unfold *ForwardInv* K_1 (Definition 8.14), thereby revealing *isContext* K_1;
- perform a ghost update, by applying NewBinding, thereby replacing *isContext* K_1 with the conjunction of *isContext* $(K_1; \mathsf{let}\ u = a + b)$, which is necessary to reestablish the forward invariant, and *isBinding* $(\mathsf{let}\ u = a + b)$, which is necessary to prove that u represents $E_a + E_b$.

To establish *ForwardInv* K_1', one must also check that, for every auxiliary variable $x \in \{\mathsf{x}\} \cup \mathit{defs}(K_1')$, the assertion $x\ \mathit{isVar}\ (\!|K_1'[x]|\!),\ 0)$ holds. This is easy; we say no more.

Segment 2: Continuation Invocation. From this point on, the variable `u` (bound at line 28) is in scope, and (in our proof) so is the value u that this variable denotes. Therefore, we may introduce the following abbreviations: we let K_1' stand for $(K_1; \mathsf{let}\ u = a + b)$, and we let K_2' stand for $(\mathsf{let}\ u = a + b; K_2)$.

Goal 8.32. *The correctness of the continuation invocation is expressed as follows:*

$$\begin{array}{l} \mathit{isCont}\ k \mathrel{-\!\!*} \\ \mathit{ForwardInv}\ K_1' \mathrel{-\!\!*} \\ u\ \mathit{isSubExpRaw}\ (E_a + E_b) \mathrel{-\!\!*} \\ \mathit{ewp}\ (\texttt{cont})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInv}\ K_1'\} \end{array}$$

This goal amounts to checking that invoking the continuation is permitted. By unfolding the definition of *isCont* (Definition 8.30), it is easy to see that this assertion allows such an invocation.

Goal 8.33. *Goal 8.32 reduces to the following goal:*

$$\mathit{ForwardInv}\ K_1' \mathrel{-\!\!*} \triangleright\, \mathit{isHandler}\ K_1'$$

In other words, we must check that, after the forward phase advances from K_1 to K_1', the handler continues to behave correctly.

Goal 8.33 is a direct consequence from our induction hypothesis (Hypothesis 8.25), which is applicable because the conclusion of this goal is guarded by a later modality.

Segment 3: Backward Phase.

Goal 8.34. *The correctness of the instructions at label* `add-bwd` *is expressed as follows:*

$$\mathit{BackwardInv}\ K_1' \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{add-bwd})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInv}\ K_1\}$$

This goal states that the two update instructions identified by the label `add-bwd` advance the backward invariant by one step. More precisely, if the backward invariant holds of the list of pending bindings K_1', then, after the execution of these two update instructions, the backward invariant holds of the list K_1. Hence, these update instructions correctly process the binding $\mathsf{let}\ u = a + b$: they remove it from the list of pending bindings.

The proof of Goal 8.34 begins by unfolding the backward invariant *BackwardInv* K_1' (Definition 8.15) and eliminating the existential quantifiers that appear as a result. This yields the following hypothesis and new goal:

Hypothesis 8.35. *We assume that a context K_2 and an auxiliary variable y are given.*

Goal 8.36. *Goal 8.34 is replaced with:*

$$\mathit{BackwardInvBody}\ K_1'\ K_2\ y \multimap \mathit{ewp}\ (\texttt{add-bwd})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInv}\ K_1\}$$

The next step is to unfold $\mathit{BackwardInv}\ K_1$ in the postcondition and to introduce the existential quantifiers that appear as a result. We instantiate them with K_2' and y.

Goal 8.37. *Goal 8.36 is replaced with:*

$$\mathit{BackwardInvBody}\ K_1'\ K_2\ y \multimap \mathit{ewp}\ (\texttt{add-bwd})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInvBody}\ K_1\ K_2'\ y\}$$

There remains to unfold the two occurrences of *BackwardInvBody* (Definition 8.15) and prove that the two update instructions achieve the desired effect. The first occurrence, which describes the state before these updates have taken place, is expanded as follows:

$$\mathit{BackwardInvBody}\ K_1'\ K_2\ y \quad \equiv \quad * \begin{cases} \text{(EA1)} & [\![E']\!]_{\lambda X.r} \equiv_{\mathcal{R}} (\!|\partial(K_1'; K_2)[y]/\partial \mathsf{x}|\!) \\ \text{(EB1)} & \forall\, x \in \{\mathsf{x}\} \cup \mathit{defs}(K_1').\ x\ \mathit{isVar}\ ((\!|K_1'[x]|\!),\ (\!|\partial K_2[y]/\partial x|\!)_{K_1'}) \end{cases}$$

The second occurrence, which describes the state after these updates have taken place, is expanded as follows:

$$\mathit{BackwardInvBody}\ K_1\ K_2'\ y \quad \equiv \quad * \begin{cases} \text{(EA2)} & [\![E']\!]_{\lambda X.r} \equiv_{\mathcal{R}} (\!|\partial(K_1; K_2')[y]/\partial \mathsf{x}|\!) \\ \text{(EB2)} & \forall\, x \in \{\mathsf{x}\} \cup \mathit{defs}(K_1).\ x\ \mathit{isVar}\ ((\!|K_1[x]|\!),\ (\!|\partial K_2'[y]/\partial x|\!)_{K_1}) \end{cases}$$

Claims EA1 and EA2 coincide, because $K_1'; K_2$ and $K_1; K_2'$ are the same sequence.

Establishing that Claim EB2 holds after the two update instructions is the crux of this proof. Intuitively, this claim asserts that the d field of every pending auxiliary variable is correctly updated: after the update, the d field of every auxiliary variable $x \in \{\mathsf{x}\} \cup \mathit{defs}(K_1)$ must hold the partial derivative of $K_2'[y]$ with respect to x.

The key mathematical tool that is used in the proof of Claim EB2 is the Left-End Chain Rule (Lemma 6.10). The proof begins with a three-way case disjunction on x: it must be the case that either (1) $x \notin \{a, b\}$, or (2) $a \neq b$ and $x \in \{a, b\}$, or (3) $x = a = b$.

In case (1), Lemma 6.10 boils down to the following simple equality:

$$(\!|\partial K_2'[y]/\partial x|\!)_{K_1} \equiv_{\mathcal{R}} (\!|\partial K_2[y]/\partial x|\!)_{K_1'}$$

This equality means that the d field of the auxiliary variable x does not need to be updated. In this case, the update instructions at `add-bwd` are indeed correct, since they do not update x: they update only the auxiliary variables a and b.

In case (2), Lemma 6.10 yields the following equality:

$$(\!|\partial K_2'[y]/\partial x|\!)_{K_1} \equiv_{\mathcal{R}} (\!|\partial K_2[y]/\partial x|\!)_{K_1'} + (\!|\partial K_2[y]/\partial u|\!)_{K_1'}$$

This equality implies that, in order to reach the final state described by Claim EB2, it suffices to increment the d field of the auxiliary variable x by the quantity $(\!|\partial K_2[y]/\partial u|\!)_{K_1'}$. By Claim EB1, this number is stored in the d field of the auxiliary variable u. In this case, the update instructions at `add-bwd` are indeed correct, since they update x (which is either a or b, but not both) precisely in the desired way.

Finally, in case (3), Lemma 6.10 yields the following equality, where 2 denotes $1 + 1$:

$$(\!|\partial K_2'[y]/\partial x|\!)_{K_1} \equiv_{\mathcal{R}} (\!|\partial K_2[y]/\partial x|\!)_{K_1'} + 2 \times (\!|\partial K_2[y]/\partial u|\!)_{K_1'}$$

This equality implies that the `d` field of the auxiliary variable a must be incremented by twice the number $(\!|\partial K_2[y]/\partial u|\!)_{K_1'}$. Again, this is precisely the behavior of the update instructions at `add-bwd`, since in this case they update x (which is the same as a and b) twice in succession.

This concludes the proof. We have verified that Statement 8.1 holds. Therefore, we have verified that the reverse-mode AD algorithm in Figure 10 is correct.

9. Related Work

We organize our review of the related work into three categories: sources of inspiration for the specification and the code that are presented in this paper (§9.1); approaches to reasoning about effect handlers (§9.2); and formal presentations and correctness arguments for automatic differentiation algorithms (§9.3).

9.1. **Sources of Inspiration.** Our minimalist API for define-by-run AD (Figure 1), which relies on a tagless-final representation of expressions [CKS09, Kis10], appears to be new. This API seems remarkable insofar as it is very simple: in short, `diff` has type `exp -> exp`. It is used as the basis for an arguably similarly simple specification in Hazel (Statement 8.1).

This API can be implemented in several ways. We provide three implementations: a forward-mode algorithm based on dual numbers (Figure 4), a reverse-mode algorithm that exploits an explicit stack (Figure 7), and a reverse-mode algorithm that exploits effect handlers (Figure 10). One can imagine other implementations, such as a reverse-mode algorithm that constructs backpropagator functions [PS08, BMP20, SV23]. These implementations may involve complex programming-language features, such as dynamically allocated mutable state, higher-order functions, and effect handlers, but this complexity is (to a large extent) abstracted away in our API (Figure 1 and Statement 8.1), as this API describes only the *interaction* between the differentiation algorithm and the outside world.

It is worth remarking that this outside world is in fact split in two components, namely the *subject* (that is, the program fragment that is differentiated) and the *customer* (that is, the program fragment that requests the evaluation of the derivative at a certain point). This is reflected in our API by the fact that the type `exp` occurs twice in the type of `diff` (Figure 1) and, similarly, the predicate *isExp* occurs twice in the specification of `diff` (Statement 8.1). One occurrence describes the interaction between the subject and the differentiation algorithm: in this dialogue, the subject plays the role of an expression, which the differentiation algorithm is allowed to query. The other occurrence describes the interaction between the differentiation algorithm and the customer: there, the differentiation algorithm plays the role of an expression, which the customer queries.

The code that we present in Figure 10 and that we verify is inspired by Wang et al. [WR18, WZD+19] and by Sivaramakrishnan [Siv18]. Wang et al.'s key observation is that the use of delimited-control operators, such as `shift` and `reset`, allows an elegant compositional presentation of reverse-mode AD. Furthermore, by making clever use of staging, they are able to propose both define-by-run and define-then-run implementations that share this common compositional architecture. They present an implementation in Scala and evaluate its performance. Inspired by Wang et al.'s ideas, Sivaramakrishnan [Siv18] proposes a minimalist implementation in Multicore OCaml that uses effect handlers instead of `shift` and `reset`. We retain the essence of this implementation and we adapt it so as to satisfy our more abstract API.

Another implementation of reverse-mode AD, written in Frank, is documented by 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 other AD algorithms, including a reverse-mode algorithm that performs checkpointing.

9.2. **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 feature can be seen as a restricted form of effect handlers where the continuation is not a first-class object and must be invoked while the effect handler is running. Goodenough describes the exception-handling methods of the time and compares them according to several criteria, including implementation difficulty, efficiency, and readability.

Effects and effect handlers offer an interface to delimited control; that is, they offer 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 large family of delimited-control operators had been studied already. 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 simulate various forms of effects in a pure language. Intuitively, the `reflect` construct allows the programmer to introduce an effect by giving its implementation 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 shows how these constructs can be implemented using `call/cc` and a single mutable cell. He proves the correctness of this encoding using a logical-relations argument.

Plotkin and Pretnar [PP09] introduce a denotational semantics for a programming language equipped 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 an implementation for each effectful operation. Plotkin and Pretnar require handlers to be correct in the sense that the operational behavior of the handler must satisfy an effect theory, that is, a set of equations, which is presented to an end user as a tool to understand and reason about effects. However, they do not investigate through what technical means one might prove that a handler is correct. Plotkin and Pretnar also adapt their previous equational logic [PP08] to account for effect handlers. This logic allows one stating and proving that two programs are equivalent. Once such a logical judgement is proven, the soundness of the logic implies an equality between the denotational interpretations of the programs.

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 a user-defined monad. Xia et al. [XZH+20, §8.2] observe that the library currently does not support handlers in their most general form: in particular, the handler does not have access to the continuation.

In Plotkin and Pretnar's work and in interaction trees, the idea is to reason about programs that involve effect handlers through a denotational model. Another approach is to reason directly in terms of contextual equivalence between programs.

In the setting of a restricted and untyped programming language equipped with effect handlers, Biernacki et al. [BLP20] show that contextual equivalence coincides with bisimilarity.

To simplify proofs of bisimilarity, 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 realistic programming language.

In the setting of a typed language, one can establish 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. Later, Zhang and Myers [ZM19] and Biernacki et al. [BPPS20] propose a logical-relations model of a type system with support for *lexically scoped handlers*, a restricted use case of effect handlers where generating a fresh effect name and installing a handler are combined into a single operation.

Our work differs from the papers cited above in two respects. First, we consider a programming language with both effect handlers and dynamically allocated mutable state. Second, we propose a compositional program logic, in the style of Hoare logic and Separation Logic. That is, we propose a method that allows writing a logical specification of the expected behavior of a program component in isolation. The proof that is presented in this paper emphasizes the benefits of compositionality: we prove that our define-by-run AD library works correctly in an arbitrary context, provided, of course, that this context respects the requirements imposed by our specification.

9.3. **Formal Presentations and Proofs of AD Algorithms.** There is a huge literature on real-world implementations of AD and on their engineering aspects. We cannot give a survey of this literature, and refer the reader to Griewank and Walther's textbook [GW08] for an introduction to the field.

Focusing more specifically on the programming-language literature, many researchers have contributed to setting AD on a firm theoretical footing by presenting formal definitions of AD algorithms, by proposing formal proofs of the correctness of these algorithms, and by making these algorithms applicable in richer settings.

A large part of this work seems dedicated to define-then-run presentations of AD, where AD takes the form of a program transformation, which transforms an abstract syntax tree into an abstract syntax tree. In this style, an AD algorithm is essentially a compiler, and its proof is a *compiler correctness* argument. Our paper may be the first formal study of define-by-run AD, where the AD algorithm is packaged as a library, and where its proof requires a program verification task. Our paper certainly offers the first proof of correctness of an AD algorithm that involves delimited-control operators: the similar algorithms previously described by Wang et al. [WR18, WZD+19], Sivaramakrishnan [Siv18], or Sigal [Sig21] are not accompanied with a proof. Furthermore, our proof is machine-checked.

Whereas the define-then-run approach involves an inspection of the syntax of the subject (that is, the program that one wishes to differentiate) and therefore requires limiting the set of programming-language constructs that the subject may use, the define-by-run approach involves executing the subject and monitoring its execution, therefore requires limiting the set of behaviors that the subject may exhibit. So, these approaches may seem quite different. This said, the dividing line can in fact be blurry. For instance, a common approach to implementing AD involves overloading the arithmetic operations. In Haskell, this can be done by exploiting type classes. Because overloading itself can be implemented either via a compile-time program transformation or via runtime dictionary passing, it can be difficult to decide in which category this approach falls. As another example, the implementation of a *differentiable programming language*, where differentiation is a primitive construct, may well involve a marriage of program transformation and program monitoring techniques.

In the following, we briefly list some of the recent work on AD that has appeared in the programming-language literature, with a focus on its semantics and on the correctness of its implementation. This is not a true survey: we provide these citations merely as starting points for the interested reader.

Karczmarczuk [Kar98, Kar01] implements forward-mode and reverse-mode AD in Haskell. He uses type classes to overload the arithmetic operators. His forward-mode implementation computes not just the first derivative, but the infinite lazy sequence of all higher-order derivatives. His reverse-mode implementation uses backpropagator functions. Also in Haskell, Kmett, Pearlmutter and Siskind [KPS10] implement forward-mode, reverse-mode, and mixed-mode AD combinators, with a common API, as a library. Their implementation relies on mutable state, stable names [PME99], and on a form of reflection.

Pearlmutter and Siskind [PS08] present VLAD, a functional programming language where differentiation is a primitive construct. They propose a program transformation that eliminates this construct. Because this transformation is *non-compositional*, it is not necessarily an attractive implementation technique. For this reason, in their prototype implementation of VLAD, Pearlmutter and Siskind use a different technique: they implement an interpreter, and rely on the fact that an interpreter has access to the source code of a function at runtime.

Elliott [Ell09] presents an implementation of higher-dimensional, higher-order, forward-mode AD in the general setting of calculus on manifolds. Starting from the specification of AD, Elliott [Ell18] derives a general AD algorithm, which he then specializes in various ways by varying the representation of derivatives. His implementation in Haskell relies on a compiler plugin which performs a form of reflection and allows him to alter the conventional meaning of function abstraction and function application.

A number of authors present AD as a program transformation for calculi that do *not* include differentiation as a primitive construct. Shaikhha et al. [SFVJ19] implement forward mode first, then show how reverse mode can be reconstructed by combining forward mode with a number of standard compiler optimizations. Their calculus is simply-typed and does not allow a function to return a function. Barthe et al. [BCLG20] prove the correctness of a fragment of Shaikhha et al.'s transformation. Alvarez-Picallo et al. [AGSZ21] prove the correctness of a simplified version of Pearlmutter and Siskind's transformation [PS08]. They phrase the proof in terms of hierarchical string diagrams, or *hypernets*, and rewriting rules. Brunel, Mazza and Pagani [BMP20] define a reverse-mode transformation for simply-typed λ-calculus The transformation is compositional and does not use mutable state; backpropagator functions are used instead. A *linear-factoring* reduction rule, which is built into the semantics of the calculus, is required for the transformation to be cost-preserving. Smeding and Vákár [SV23] improve on Brunel, Mazza and Pagani's work by showing how their approach can be efficiently implemented in a standard programming language, whose semantics does not include a linear-factoring rule. Mazza and Pagani [MP21] prove the soundness of AD transformations in the setting of PCF, a typed λ-calculus equipped with real numbers, recursion, and conditionals. In the presence of conditionals, AD cannot be expected to yield a correct result everywhere: Mazza and Pagani show that it yields a correct result almost everywhere. With similar motivation, Lee et al. [LYRY20] isolate a class of functions for which an *intensional derivative* always exists and coincides almost everywhere with the standard derivative. Huot et al. [HSV20, HSV22] present a denotational semantics based on diffeological spaces for a simply-typed λ-calculus equipped with algebraic data types. They use this semantics to state and prove the correctness of a forward-mode transformation.

Extending Elliott's work [Ell18], Vákár et al. [Vák21, NV21, VS22] present forward- and reverse-mode transformations for pure λ-calculi equipped with rich type disciplines. Their proofs of correctness exploit logical relations. Krawiec et al. [KKP+22] present forward-mode and reverse-mode transformations for a simply-typed λ-calculus. The transformations are cost-preserving. The correctness proof involves logical relations. Radul et al. [RPF+23] propose a modular presentation of reverse-mode AD in a typed calculus equipped with reals, tuples, and first-order functions. The transformation is decomposed in three steps, namely forward-mode AD, *unzipping*, and transposition. They argue that this decomposition allows a better understanding and a more economical implementation of reverse-mode AD.

Treading in the footsteps of Pearlmutter and Siskind [PS08], several authors propose semantics and implementation techniques for differentiable programming languages, that is, for calculi that *do* include differentiation as a primitive construct. Vytiniotis et al. [VBW+19] sketch a compilation scheme for a simply-typed, higher-order language. Abadi and Plotkin [AP20] give operational and denotational semantics for a first-order language. They prove that these semantics coincide, thereby establishing the correctness of the AD algorithm that is built into the operational semantics. Cockett et al. [CCG+20] propose "a starting point to build categorical semantics of differentia[ble] programming languages". Mak and Ong [MO20] design a higher-order differentiable programming language, where the reduction strategy simulates reverse-mode AD. The correctness of this reduction strategy is proved via a categorical interpretation. Sherman et al. [SMC21] describe λ_S, a higher-order programming language that includes higher-order derivatives as well as constructs for integration, root-finding and optimization. Its denotational semantics is based on Clarke derivatives. They describe an implementation of λ_S as an embedded language inside Haskell.

10. Conclusion

In this paper, we have verified a very small implementation of reverse-mode AD, packaged as a library. The code exploits dynamically allocated mutable state, higher-order functions, and effect handlers. We have proposed an original API for this library (Figure 1) as a transformation of expressions in tagless-final style. We have used Hazel [dVP21], a variant of higher-order Separation Logic, to write a specification of the library (§8.1) and to construct a proof of its correctness. (§8.2). This proof is machine-checked [dVP22a]. 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 our own previous work [dVP21], no correctness proofs for programs that involve primitive mutable state and effect handlers have appeared in the literature.

Our specification of `diff` illustrates how effects are described in Hazel. According to Statement 8.1, `diff` itself performs no effects: it is a function from expressions to expressions. According to Definition 8.2, 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 how 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 handlee (which performs an effect) and the handler (which handles this effect). Our proof illustrates both handlee-side and handler-side reasoning. On the handlee 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, as well as any universal quantifiers that are shared between the pre- and postcondition. In our proof, the forward invariant (Definition 8.14) and the backward invariant (Definition 8.15) serve as pre- and postcondition for the handler, and they are combined in a universally quantified statement that expresses the correctness of an arbitrary instance of the handler (Goal 8.24). Finally, when writing down the protocol that the handlee and handler obey, it is often necessary to introduce custom Separation Logic assertions, whose definition may involve ghost state. In our proof, the predicate *isSubExpRaw* (Definition 8.16), which appears in the protocol that describes the effects `Add` and `Mul` (Definition 8.17), is defined in terms of a ghost history K of the past effects. This ghost history appears in the forward invariant as well. Ghost history variables are common in proofs of concurrent and distributed algorithms [AL88, LM17]; here, although the code is sequential, the handlee 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). 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 obstacles remain before these libraries can be verified.

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 10) 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 into a binary sum. In other words, in our encoding, `Add` and `Mul` are essentially

global names. We believe that this difference should not fundamentally influence the manner in which one reasons about effect handlers. Nevertheless, in the future, it would be desirable 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, dVP23], where the final word has not yet been said.

Acknowledgments

The authors wish to thank the anonymous reviewers, whose comments have greatly helped improve the presentation of the paper.

References

[AGSZ21] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *CoRR*, abs/2107.13433, 2021.

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

[AP20] Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):38:1–38:28, 2020.

[BB20] Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic. Lectures notes, September 2020.

[BCLG20] Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. On the versatility of open logical relations – continuity, automatic differentiation, and a containment theorem. In *European Symposium on Programming (ESOP)*, volume 12075 of *Lecture Notes in Computer Science*, pages 56–83. Springer, April 2020.

[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 *Leibniz International Proceedings in Informatics*, pages 7:1–7:22, 2020.

[BMP20] Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *Proceedings of the ACM on Programming Languages*, 4(POPL):64:1–64:27, 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.

[BPRS18] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[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.

[CCG+20] J. Robin B. Cockett, Geoff S. H. Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon D. Plotkin, and Dorette Pronk. Reverse derivative categories. In *Computer Science Logic*, volume 152 of *Leibniz International Proceedings in Informatics*, pages 18:1–18:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, January 2020.

[Cha20] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP):116:1–116:34, 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.

[dVP21] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021.

[dVP22a] Paulo Emílio de Vilhena and François Pottier. Verifying an effect-handler-based define-by-run reverse-mode AD library: Coq formalization. `https://gitlab.inria.fr/pdevilhe/hazel`, 2022.

[dVP22b] Paulo Emílio de Vilhena and François Pottier. Verifying an effect-handler-based define-by-run reverse-mode AD library: Guide to the Coq formalization. `https://gitlab.inria.fr/pdevilhe/hazel/-/blob/master/papers/LMCS-RMAD.md`, 2022.

[dVP23] Paulo Emílio de Vilhena and François Pottier. A type system for effect handlers and dynamic labels. In *European Symposium on Programming (ESOP)*, volume 13990 of *Lecture Notes in Computer Science*, pages 225–252. Springer, April 2023.

[Ell09] Conal M. Elliott. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*, pages 191–202, September 2009.

[Ell18] Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):70:1–70:29, 2018.

[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.

[HSV20] Mathieu Huot, Sam Staton, and Matthijs Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 12077 of *Lecture Notes in Computer Science*, pages 319–338. Springer, April 2020.

[HSV22] Mathieu Huot, Sam Staton, and Matthijs Vákár. Higher order automatic differentiation of higher order functions. *Logical Methods in Computer Science*, 18(1), 2022.

[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.

[Kar98] Jerzy Karczmarczuk. Functional differentiation of computer programs. In *International Conference on Functional Programming (ICFP)*, pages 195–203, September 1998.

[Kar01] Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, 2001.

[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.

[KJJ+18] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages*, 2(ICFP):77:1–77:30, 2018.

[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.

[KPS10] Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. `ad`: Automatic differentiation. `https://hackage.haskell.org/package/ad`, 2010.

[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.

[LRCH18] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In *Formal Methods (FM)*, volume 10951 of *Lecture Notes in Computer Science*, pages 338–354. Springer, July 2018.

[LRCH21] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effects handlers. *Formal Aspects of Computing*, 33(1):127–150, 2021.

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

[LYRY20] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. On correctness of automatic differentiation for non-differentiable functions. In *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, December 2020.

[MO20] Carol Mak and Luke Ong. A differential-form pullback programming language for higher-order reverse-mode automatic differentiation. *CoRR*, abs/2002.08241, 2020.

[MP21] Damiano Mazza and Michele Pagani. Automatic differentiation in PCF. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–27, 2021.

[NV21] Fernando Lucatelli Nunes and Matthijs Vákár. CHAD for expressive total languages. *CoRR*, abs/2110.00446, 2021.

[O'H08] Peter W. O'Hearn. Separation logic tutorial. In *International Conference on Logic Programming (ICLP)*, volume 5366 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2008.

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

[PME99] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in Haskell. In *Implementation of Functional Languages (IFL)*, volume 1868 of *Lecture Notes in Computer Science*, pages 37–58. Springer, September 1999.

[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.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.

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

[RPF+23] Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin. You only linearize once: Tangents transpose to gradients. *Proceedings of the ACM on Programming Languages*, 7(POPL), January 2023.

[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.

[SFVJ19] Amir Shaikhha, Andrew W. Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages*, 3(ICFP):97:1–97:30, 2019.

[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. 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.

[SV23] Tom Smeding and Matthijs Vákár. Efficient dual-numbers reverse AD via well-known program transformations. *Proceedings of the ACM on Programming Languages*, 7(POPL), January 2023.

[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.

[TB21] Amin Timany and Lars Birkedal. Reasoning about monotonicity in separation logic. In *Certified Programs and Proofs (CPP)*, pages 91–104, January 2021.

[Vák21] Matthijs Vákár. Reverse AD at higher types: Pure, principled and denotationally correct. In *European Symposium on Programming (ESOP)*, volume 12648 of *Lecture Notes in Computer Science*, pages 607–634. Springer, April 2021.

[VBW+19] Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. The differentiable curry. Presented at NeurIPS, 2019.

[VS22] Matthijs Vákár and Tom Smeding. CHAD: Combinatory homomorphic automatic differentiation. *ACM Transactions on Programming Languages and Systems*, 44(3):1–49, September 2022.

[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.