Proof of Programs with Effect Handlers

Paulo Emílio de Vilhena

Université Paris Cité
École Doctorale 386 — Sciences Mathématiques de Paris Centre
Laboratoire — Inria Paris

Doctorat d'Informatique

Proof of Programs with Effect Handlers

Paulo Emílio de Vilhena

Thèse dirigée par François Pottier

et soutenue le 16 décembre 2022 devant le jury composé de :

Dariusz Biernacki	Associate Professor, University of Wrocław	Rapporteur
Robbert Krebbers	Associate Professor, Radboud University	Rapporteur
Roberto Amadio	Professeur, Université Paris Cité	Examinateur
Mihaela Sighireanu	Maître de Conférences, Université Paris Cité	Examinatrice
Yann Régis-Gianas	Software and Research Engineer	Examinateur
François Pottier	Directeur de Recherche, Inria	Directeur

Abstract This thesis addresses the problem of reasoning about programs that modify the heap and alter the control flow through effect handlers, a novel programming construct that provides a relatively simple interface to delimited control. This ability to manipulate the control flow is extremely powerful: many programming features – such as asynchronous programming and coroutines – that come as built-in packages of traditional programming languages can be expressed in terms of effect handlers. The status of effect handlers as a modular and expressive programming construct is attested by the development of the OCaml programming language, which will have support for handlers in its next major release. This event makes the problem of unveiling the logical principles that govern effect handlers even more pressing. In particular, how to reason abstractly about a continuation, rather than thinking concretely as a fragment of the stack? Moreover, can we reason separately about a program that performs effects and a program that handles these effects? This thesis answers these questions by introducing Hazel, a Separation Logic for effect handlers, built as an extension of Iris. Hazel introduces a novel specification language by means of which one can describe the behavior of programs, including continuations. The logic allows one to compose specifications in a modular fashion through familiar reasoning rules, such as the bind rule and the frame rule, and novel ones, such as the reasoning rules for handling and performing effects. To assess the applicability of Hazel as a tool to formally reason about programs, this thesis includes the verification of a number of case studies: (1) a program that transforms a higher-order iteration method into a lazy sequence; (2) a library for asynchronous computation; and (3) a library for reverse-mode automatic differentiation.

This thesis also explores variants of the Hazel logic for different languages in the design space of effect handlers. One such variant, for example, is Maze, a logic for handlers with multi-shot continuations, *i.e.*, continuations that can be resumed multiple times. The applicability of Maze is assessed through the verification of a simple SAT solver that uses multi-shot continuations to implement backtracking and through the design of reasoning rules for the undelimited-control operators `callcc` and `throw`. Another variant is TesLogic, a logic for a language with support for the dynamic generation of effect labels, identifiers used by both a program that performs effects and a program that handles effects as a way to specify the correspondence between these two programs.

The main application of TesLogic is in the study of type systems for effect handlers. The question of devising a static type discipline for effect handlers is the subject of an open debate, which seems to suggest a dichotomy: in order to achieve simple and strong subtyping rules, one side argues in favor of imposing a restriction to lexically scoped handlers, while the opposite side argues in favor of complex programming constructs, such as effect coercions. This thesis makes a contribution to this debate by introducing Tes, a type system that is not restricted to lexically scoped handlers, and that supports powerful subtyping rules without the introduction of effect coercions. The soundness of Tes follows from the interpretation of typing judgments as specifications written in TesLogic. The results of this thesis have been formalized in the Coq Proof Assistant.

Keywords Effect handlers, Formal verification, Program logics, Separation Logic, Type systems, Logical relations

Résumé Cette thèse s'intéresse à la conception des méthodes formelles pour raisonner sur les programmes impératifs qui peuvent modifier le flot de contrôle à travers les gestionnaires d'effets, une nouvelle primitive de programmation offrant une interface relativement simple aux opérateurs de contrôle délimité. Les gestionnaires d'effets sont extrêmement puissants en tant qu'outil de programmation: plusieurs primitives et modes de programmation – tel que la programmation asynchrone – souvent supportés par les langages traditionnels comme des parties intégrés de ces langages peuvent être implémentés à l'aide des gestionnaires d'effets. La réputation des gestionnaires d'effets en tant qu'un concept de programmation puissant et modulaire est attesté par le développement du langage OCaml, qui aura le support pour les gestionnaires d'effets dans sa prochaine version majeure. Cet événement fait de la recherche des principes logiques derrières les gestionnaires d'effets un problème encore plus pressant. En particulier, comment peut-on raisonner à propos d'une continuation de façon abstraite plutôt que de façon concrète en tant qu'un morceau de la pile d'exécution? Comment peut-on raisonner à propos d'un programme qui lance des effets séparément du programme qui attrape ces effets?

Cette thèse répond à ces questions en introduisant Hazel, une Logique de Séparation pour les gestionnaires d'effets. Hazel introduit un nouveau langage de spécification permettant la description du comportement des programmes, y compris des continuations. Cette logique permet aussi la composition des spécifications de façon modulaire, soit par l'application de règles de raisonnement habituelles, tel que la règle de liaison ou la règle de l'encadrement; soit par l'application de règles nouvelles, tel que les règles pour lancer ou capturer des effets. Pour évaluer l'applicabilité de Hazel en tant qu'outil pour le raisonnement formel sur les programmes, cette thèse inclut la vérification des nombreux cas d'études: (1) la conversion générique d'une méthode d'itération d'ordre supérieur vers une séquence paresseuse; (2) une bibliothèque pour la programmation asynchrone; (3) une bibliothèque pour la différentiation automatique en arrière. Cette thèse étudie aussi des variantes de Hazel pour les différentes conceptions de gestionnaires d'effets. Une telle variante est Maze, une logique pour les gestionnaires d'effets avec des continuations à plusieurs appels (multi-shot). L'applicabilité de Maze est évaluée par (1) la vérification d'un solveur SAT simple qui utilise des continuations pour implémenter le retour sur trace et par (2) la conception des règles de raisonnement pour `callcc` et `throw`.

Une autre variante est TesLogic, une logique pour raisonner sur la génération dynamique de noms d'effets. La principale application de TesLogic est dans l'étude des systèmes de types pour les gestionnaires d'effets. La conception d'un système de types pour les gestionnaires d'effets est le sujet d'un débat actif: pour offrir des règles simples et permissives de sous-typage, un coté soutient la restriction des gestionnaires d'effets aux gestionnaires d'effets de portée lexicale, tandis que l'autre coté soutient l'adoption des coercions d'effets. Cette thèse contribue à ce débat en introduisant Tes, un système de types qui (1) n'est pas restreint aux gestionnaires d'effets de portée lexicale, (2) n'introduit pas les coercions d'effets, et (3) admet des règles de sous-typage puissantes. La sûreté de Tes est prouvée à l'aide d'une interprétation des jugements de typage en tant que spécifications écrites en TesLogic. Les résultats de cette thèse sont formalisés dans l'assistant de preuve Coq.

Mots clés Gestionnaire d'effets, Vérification formel, Logique de programmes, Logique de Séparation, Système de types, Relations logiques

Contents

Acknowledgments

I offer my deepest thanks to François Pottier, my adviser, who has always been available for discussions and casual meetings, and who received my ideas with equal openness and consistent enthusiasm. Although the research for scientific knowledge brings great joy, exploring the unknown may also bring uncertainty. François knew how to be supportive in times of need, and how to share the excitement of discovery in times of success. As our paths will now go apart, I hope to keep clear in my heart the standard of virtue of which he provides proof every day.

During my short academic career, I have been extremely lucky to meet intelligent, kind people. I am thankful to Larry Paulson, who initiated me to research; to Jacques-Henri Jourdan, with whom I collaborated in my first published work; and to Amin Timany and Lars Birkedal, who invited me to visit their research group at Aarhus. I would like to thank my colleagues from Cambium, whose cheerful discussions I shall keep as an entertaining souvenir. And I thank Alexandre, Armaël, Carine, Florian, and Léo, with whom I shared many delightful breaks.

None of my academic accomplishments would be possible without the steady support of my family and my dear friends. There were no physical boundaries to my family's care and warmth, which I could feel despite the ocean that is placed between us. I thereby thank my parents, Vera Cristina and Paulo Emílio; and my siblings, João Paulo, Pedro Henrique, and Thyene. I am profoundly grateful to Véronique and Guy, who, throughout my stay in France, provided me shelter and the true love of a family. I thank Joana and Xandinho, and Shweta and Dudu, for the sweetness with which they always received me. I am thankful to Amanda and Renan, two friends who were constantly present during the writing of this thesis. I cannot repay the relief that their company provided me during this challenging undertaking. I thank Luis, my lockdown comrade, for our numerous tea-fuelled discussions; and Leandro, my apprentice in the practice of Minas's cuisine. Finally, I would like to thank Daniel, whose friendship does not cease to endure our ever-growing physical distance.

Chapter 1

Introduction

1.1 Proof of programs

The problem of verifying program correctness is perhaps one of the very oldest problems in Computer Science, as evidenced by the fact that, as early as 1949, Alan Turing was already interested in *checking large routines* [Tur49, MJ84]. A routine, in the sense studied by Turing, is a sequence of labeled instructions with jumps. Turing's main insight was that it is possible to split the task of verifying a routine into smaller subtasks: it is sufficient to reason separately about *"sequences of instructions without changes of control"*. His method regroups these sequences into boxes to which one must ascribe a pair of (1) a description of the contents of the machine before control enters the box, and (2) the relations among these contents after control exits the box in the case of a conditional jump.

Floyd [Flo67] independently envisions a similar technique, in which every command is annotated with two propositions: one relating the contents of program variables at the entry of the command and another one relating these variables at the exit. These propositions are respectively called *precondition* and *postcondition* in Hoare's seminal work [Hoa69], which reformulates Floyd's ideas into an arguably simpler setting, called *Hoare logic*. Hoare logic introduces the notation $\{P\}\, e\, \{Q\}$ ascribing a precondition P and a postcondition Q to a program: if P holds before the execution of e, then Q holds upon termination. Hoare logic invites one to think about a program in terms of an abstract *specification* of its behavior, rather than in terms of its implementation. Additionally, Hoare logic advocates for deductive reasoning: one can establish that a program satisfies a specification by assuming a set of axioms, that asserts the specification of the primitive instructions of the language, and by applying rules to derive the specification of compound programs. Hoare logic is thus an early and founding example of a *program logic*: a pair of (1) a specification language, to describe the properties of programs, and (2) a set of *reasoning rules*, to derive such specifications.

The method presented by Hoare [Hoa69], which applies to programs with local mutable variables and `while` loops, is already sufficient to provide a principled approach to understand and verify the following piece of code written in the Java programming language [GJSB00]:

```
(* Implementation of fast exponentiation in Java. *)
public long exp (long x, long y) {
  long b = x, e = y, res = 1;
  while (e > 0) {res *= (e % 2 == 0 ? 1 : b); b *= b; e /= 2;}
  return res;
}
```

This program computes x raised to the power of y, where x and y are positive integers with x^y (strictly) less than 2^{63}, using a technique called *fast exponentiation*. This technique precedes Hoare logic, but Hoare logic provides a clear and formal argument of why this program is correct. Indeed, Hoare logic formalizes the key intuition that, to reason about a `while` statement, it suffices to find a *loop invariant*: a relation among the program variables that holds at the beginning and at the end of every iteration of the loop. The loop invariant in this case is the following assertion: $res \cdot b^e = x^y \wedge e \geq 0$. It is easy to check that this assertion holds before entering the loop, and that this assertion is preserved by an arbitrary iteration of the loop. Therefore, if the loop terminates, the assertion holds, and, because e is equal to 0, we obtain the equality $res = x^y$, which justifies that the program is correct.

This example shows the importance of program logics as a way to formalize and to justify informal notions that arise from the practice of programming. An experienced programmer may already think about `while` loops informally in terms of loop invariants, by simulating the action of a small number of iterations and then generalizing this action to an arbitrary number of times needed to invalidate the guarding condition of the loop. However, Hoare logic captures this intuition as a formal and reusable reasoning rule, that is, a general principle that applies to an arbitrary `while` loop, regardless of the operational complexity of the iterated instructions.

Since the work of Floyd and Hoare, the field of Program Logics has seen many advances that extended the range of application of Hoare logic. A limitation of Hoare logic is the inability to reason modularly about mutable state: logical assertions in Hoare logic are not well-suited to describe unrelated portions of the memory. This deficiency leads to overly complicated reasoning principles about programs that manipulate mutable data structures. A satisfactory solution to this problem came in the early 2000's with the introduction of Separation Logic by O'Hearn, Reynolds, and Yang [ORY01, Rey02]. Among other features, Separation Logic extends the language of logical assertions of Hoare logic with the *separating conjunction*, a novel logical connective that can be used to succinctly describe disjoint parts of the heap. This connective allows the statement of the *frame rule*, which captures the idea that a program modifies only parts of the heap of which it is aware, thus achieving modular reasoning about state. This rule also internalizes the pervasive notion of *ownership*: the principle that, if a program *owns* a certain region of the heap, then only this program can modify this region.

The notion of ownership is especially important in the setting of concurrent programming: for example, if a region of the memory is protected by a lock, then a program receives ownership of this region when it acquires the lock, and transfers ownership to the lock upon release. Separation Logic has indeed been extended to support reasoning about concurrent programs [O'H07, GBC+07]. One of the most sophisticated examples of such extensions is Iris [JKJ+18], a Separation Logic that reaches an outstanding expressive power by adding support for two features: *user-defined higher-order ghost state* and *invariants*. Ghost state is fictional state that appears during the verification of a program. It is an old technique [AL88] in the field of Proof of Programs, but, in Iris, it proves itself unprecedentedly powerful because (1) Iris introduces an abstract notion of *resource*, which the user can define according to the task at hand, and (2) ghost state in Iris is higher-order, meaning that ghost state can contain Iris assertions. Invariants are used to reason about fine-grained concurrent programs: an invariant protects a shared region of

the memory, and allows ownership of this region to be transferred during atomic steps of execution.

A myriad of papers [VB21, TB19, FKB18, JJKD18] shows evidence of the success of Iris as a powerful tool for the verification of programs and for the study of programming languages. However, a class of programs has been largely ignored by separation logics, in general, and by Iris, in particular: programs with *delimited-control operators*. Delimited-control operators extend a language with the ability to capture a fragment of the stack and to reify it as a *continuation*. They form powerful programming constructs that are present in languages such as Scheme [sch] and Racket [Fla21a], and can be used, for example, to implement lightweight threads [DEH+17], coroutines [dMI09], and backtracking [DF90, BD04, KcSFS05]. However, these operators invalidate the intuition that a program is a code block with an entry and an exit. For this reason, delimited-control operators are often regarded as advanced programming techniques, that must be used with great care. The aim of this thesis is to remedy this situation: we wish to design reasoning rules for delimited-control operators, and thereby contribute to their understanding. The particular delimited-control operators studied in this thesis are *effect handlers*.

1.2 Effect handlers

Effect handlers were introduced by Plotkin and Pretnar [PP09] in the setting of the *algebraic approach* to the study of the semantics of programming languages. The algebraic approach [PP04] formalizes the semantics of a programming construct by means of equations stating which syntactically different usages of this construct have the same operational meaning. A construct that fits into this formalism is called an *algebraic effect* and its set of equations is called its *algebraic theory*. Formulating the algebraic theory of an effect allows one to study this construct in the abstract setting of Category Theory. The extension of a language $\mathcal{L}$ with an algebraic effect can then be seen as the *free model* of the associated algebraic theory: it is the set of $\mathcal{L}$ programs extended with this effect, modulo its algebraic equations. A *handler* of an algebraic effect emerges in this formalism as a *homomorphism* from the free model of the algebraic effect to a user-defined one.

In spite of this elegant mathematical formulation of algebraic effects and handlers, this thesis studies effect handlers from an operational point of view, which is better suited to our goal of devising a Separation Logic for effect handlers. From this perspective, effect handlers can be seen as a generalization of exception handlers. Akin to raising an exception, a program can *perform an effect* to interrupt the normal flow of execution and transfer control to an effect handler. Unlike an exception handler, an effect handler gains access to a *delimited continuation*, which represents the fragment of the evaluation context comprised between the point where the effect was performed and the point where the effect handler was installed. Therefore, this fragment of the evaluation context is not discarded as in the case of an exception handler, but reified as a first-class value. The handler may wish to ignore this continuation, in which case its behavior is similar to an exception handler, or it may wish to *invoke this continuation*, in which case the suspended computation is resumed.

In recent work, Dolan et al. [DEH+17] show that effect handlers have interesting applications to systems programming. In particular, they show how effect handlers can

```
1  (* The type of a higher-order iteration method. *)
2  type 'a iter = ('a -> unit) -> unit
3
4  (* The type of a lazy sequence. *)
5  type 'a  seq = unit -> 'a head
6   and 'a head = Nil | Cons of 'a * 'a seq
7
8  (* Implementation of [invert] in OCaml 4.12.0+effects+domains.*)
9  let invert (type a) (iter : a iter) : a seq = fun () ->
10   let open struct effect Yield : a -> unit end in
11   let yield x = perform (Yield x) in
12   match iter yield with
13   | effect (Yield x) k -> Cons (x, fun () -> continue k ())
14   | () -> Nil
```

Figure 1.1: Example of control inversion using effect handlers.

be used to write an *asynchronous computation* library, whose clients can fork lightweight threads and wait for a thread to produce a result. Another interesting application of effect handlers is *control inversion*, which allows converting a higher-order iteration method for a collection of elements into a lazy sequence of elements [dVP21], or in other words, converting a "push" producer into a "pull" producer. More generally, it has been argued that, "by separating effect signatures from their implementation, [effect handlers] provide a high degree of modularity" [KLO13]. Moreover, effect handlers can simulate most delimited-control operators [FKLP19]. For these reasons, effect handlers are finding their way into research programming languages such as Eff [BP15, BP20], Effekt [BSO20b, BSO20a], Frank [LMM17], Koka [Lei14, Lei20], Links [HLA20], and into mainstream programming languages such as OCaml 5 [SDW+21].

1.3 Goals and challenges

To illustrate the goals and challenges of this thesis, let us study the program `invert` that appears in Figure 1.1. This program is written in the OCaml programming language version `ocaml-variants.4.12.0+domains+effects`, [1] an experimental version of OCaml with support for effect handlers. It uses effect handlers to transform a *higher-order iteration* method for a collection of elements into a *lazy sequence* of these elements. We shall soon explain what each of these terms means. However, let us first introduce separately the four syntactic constructs used in this example.

- ***Introducing an effect.*** In OCaml, both the program fragment that performs effects and the one that installs an effect handler must carry an identifier called an *effect name*. In this way, the program that performs an effect can specify the handler that must be invoked. The construct to introduce a new effect name is called an *effect declaration* and it is written as follows:

[1] At the time of writing, installation instructions for this version of the OCaml compiler can be found at the following url: https://github.com/ocaml-multicore/multicore-opam.

```
effect <name> : <type> -> <type>
```

In addition to the new effect name, the user must also include two types: the *argument type* of the effect, in the left-hand side of the arrow; and the *return type*, in the right-hand side of the arrow. We shall explain the role of each of these types when studying the remaining syntactic constructs. An effect declaration can appear either at the toplevel or inside a module. In the implementation of `invert` (Figure 1.1), we must place an effect declaration inside the scope of the function `invert`, because the type `a`, on which this effect depends, is not known at the toplevel. Therefore, in line 10, we introduce the effect name `Yield` by means of the `open struct ... end` construct, which defines a module that is immediately opened.

- ***Performing an effect.*** To perform an effect, one must specify an effect name and an expression, called the *effect argument* or *payload*:

```
perform (<name> <expr>)
```

The type of the effect argument must coincide with the argument type of the specified effect name. OCaml assigns the return type of the effect name to this whole expression. Operationally, when a program performs an effect, this program is paused and the fragment of the stack up to the innermost handler able to handle this effect is reified into a continuation that is passed to this handler.

- ***Installing a handler.*** To install a handler over an expression, it suffices to wrap this expression over a `match ... with` construct with two branches: (1) an *effect branch*, and (2) a *return branch*.

```
match <expr> with
| effect (<name> <var>) <var> -> <expr>
| <var> -> <expr>
```

The expression monitored by the handler is called the *handlee*. The handler transfers control to the effect branch if the handlee performs an effect whose name coincides with the one specified by the effect branch. In this case, the first variable, from left to right, binds the effect argument, and the second variable binds the continuation. The handler transfers control to the return branch if the handlee terminates normally. In this case the variable in this branch binds the result of the handlee's evaluation.

- ***Invoking a continuation.*** The instruction `continue` can be used to invoke a continuation:

```
continue <expr> <expr>
```

The first expression, from left to right, is the continuation, and the second one is the expression with which the continuation is invoked. A continuation represents a paused handlee that performed an effect under a handler. Invoking a continuation with a certain value resumes the handlee as if the perform instruction had returned this value. Moreover, when a continuation is invoked, the handler is reinstalled over the handlee. This behavior corresponds to a *deep-handler* semantics, as opposed

to a *shallow-handler* semantics, under which invoking a continuation does not reinstall the handler. Despite of their differences, deep and shallow handlers are inter-expressible [HL18]. Finally, we note that continuations in OCaml are *one-shot*: they can be invoked at most once. In Chapter 2, we compare one-shot continuations to *multi-shot continuations*, which can be invoked multiple times.

The implementation of `invert` (Figure 1.1) is composed of three parts: (1) the declaration of the type of iteration methods in line 2; (2) the declaration of the type of lazy sequences in line 5; and (3) the implementation of `invert` in line 9.

An iteration method receives a function of type `'a -> unit`, called the *iteratee*, which consumes a single element of type `'a` and might perform some action (such as printing this element). An iteration method applies the iteratee to each element of a data structure. Given a list of elements `xs`, for example, one can define an iteration method for `xs` as follows:

```
let rec list_iter (xs : 'a list) (f : 'a -> unit) =
  match xs with [] -> () | u :: us -> (f u; list_iter us f)
```

Iteration methods for other data structures, such as trees and arrays, can be similarly defined. In fact, iteration methods are the standard approach in OCaml to repeat an action over each element of a collection, and a collection's API usually includes this feature.

Lazy sequences form yet another approach to repeating an action over elements of a structure. A lazy sequence is a thunk that, when forced, returns either a `Cons` value, which is a pair of one of the structure's elements and a lazy sequence representing the remaining elements, or an `Nil` value, indicating that all elements have been produced. With a lazy sequence, a program can thus access elements of a structure *on demand*: there is no need to traverse the whole structure at once, which is the case with iteration methods. Because elements are produced one at a time, it is possible to define a lazy sequence representing an infinite structure, such as the sequence of nonnegative integers:

```
let nonneg_integers_seq : int seq =
  let rec from n = fun () -> Cons (n, from (n + 1)) in from 0
```

An iteration method that traverses this sequence would not terminate:

```
let nonneg_integers_iter : int iter = fun f ->
  let rec from n = f n; from (n + 1) in from 0
```

This distinction creates a tension between iteration methods and lazy sequences: should a collection API support one of these features, or should it support both?

With effect handlers, however, this tension disappears: *one can convert an iteration method into a lazy sequence*. The idea is to use an effect to pause the iteration method and to resume its execution on demand. This is precisely what `invert` does. In line 10, the effect name `Yield` is introduced. In line 11, `invert` defines the function `yield`, which performs this effect `Yield`. Finally, in line 12, `invert` installs a handler over the expression `iter yield`. During the execution of this expression, every time the function `yield` is applied to an element x, the evaluation of `iter` is suspended and control is transferred to the effect branch of the handler. The effect branch has access to both the element x, to which `yield` was applied, and to a continuation representing the suspended iteration.

The element is used to form the left component of a `Cons` pair, while the continuation is used to build the sequence of the remaining elements. When, and if, the iteration terminates, the sequence produces the `Nil` marker.

We have presented an informal explanation of `invert`, but *how to formally specify and verify this piece of code?* Ideally, the specification of `invert` should translate into the formal setting of a program logic the following intuitive statement: *"*`invert` *converts an iteration method into a lazy sequence"*. The formalization of this statement can be divided into two tasks: (1) to write the specification of an iteration method; and (2) to write the specification of a lazy sequence. One can then write `invert`'s specification in a pre- and postcondition style, where the precondition states that `invert` takes an iteration method as an argument, and the postcondition states that `invert` produces a sequence.

In traditional higher-order Separation Logic, an iteration method `iter` for a collection of elements xs can be specified as follows:

$$\forall I\, f.(\forall u,\, us.\, \{I(us)\}\, f\, u\, \{I(us +\!\!+ u)\}) \mathrel{-\!\!*} \{I[]\}\, \texttt{iter}\ f\, \{I(xs)\} \tag{1.1}$$

The predicate I represents the internal state of the method. It is parameterized by the list of elements that have already been fed to f. The specification says that for any such predicate, if f updates the internal state of the iteration by passing from $I(us)$ to $I(us +\!\!+ u)$, then the complete traversal of the structure updates the initial state $I[]$ to the final state $I(xs)$.

However, this specification says nothing about the effects that either f or `iter` might perform. It is important to include this information, because `invert` assumes that `iter` does not intercept the effects that the iteratee might perform. For instance, `invert` does not correctly convert the following iteration method into a sequence:

```
let ocaml_iter = fun f ->
  match f 'o'; f 'c'; f 'a'; f 'm'; f 'l' with
  | effect _ _ -> ()
  | _ -> ()
```

This program iterates over the list of characters `['o', 'c', 'a', 'm', 'l']`, but it installs a "catch-all" handler over the applications of the iteratee so that, if f performs an effect, then this effect is intercepted by `ocaml_iter` and `ocaml_iter` terminates immediately. In particular, a `Yield` effect never reaches the handler installed by `invert`. The function `ocaml_iter` is not a valid input of `invert`, even though it satisfies Specification 1.1 if one reads the triple $\{P\}\, e\, \{Q\}$ as saying "*if P holds, then e either diverges or terminates in a state where Q holds*". This informal reading of specifications says nothing about the effects that a program fragment e might perform during its execution. This problem highlights the first challenge which one must overcome to specify and to verify `invert`:

> **Challenge 1** ***(Specification)*** *To extend Separation Logic with means to describe the effects that a program might perform.*

Assuming that one has found means to describe the effects that a program fragment might perform, then comes the next challenge: to verify that programs exploiting effects

and effect handlers, such as the function `invert`, meet their specifications; that is, one must find rules to reason about delimited-control operators:

> **Challenge 2** ***(Verification)*** *To extend Separation Logic with rules to reason about delimited-control operators.*

The main goal of this thesis is to design a logic that addresses these limitations of Separation Logic, and enables the verification of programs with effect handlers, such as `invert`. Moreover, this logic should provide means *to reason modularly about a program fragment that performs effects and the program fragment that handles these effects.* This feature is already essential in the example of `invert`, because we would like the specification of `iter` to be independent of the `Yield` handler, which is part of the implementation of `invert` and is unrelated to `iter`.

1.4 Overview

Chapter 2 shows how this thesis accomplishes its main goal: it presents Hazel, a novel Separation Logic with support for effect handlers and one-shot continuations. Chapter 3 then shows how one may apply this logic to the verification of `invert`. Chapters 4 and 5 illustrate further applications of Hazel: Chapter 4 shows the verification of a library for lightweight threads and Chapter 5 shows the verification of a library for reverse-mode automatic differentiation. We then explore variants of Hazel. Chapter 6 presents Maze, a logic for effect handlers with multi-shot continuations, and illustrates the application of this logic to the verification of a simple SAT solver. Chapter 7 presents TesLogic, a logic for effect handlers with dynamic generation of effect labels, and applies this logic to the study of type systems. Moreover, Chapter 7 introduces Tes, a type system for effect handlers with dynamic labels, and proves the soundness of Tes through the semantic interpretation of typing judgments as specifications in TesLogic.

Coq formalization. At the time of writing, the Coq formalization of the results from Chapters 2 to 6 can be found at:

`https://gitlab.inria.fr/cambium/hazel`

Moreover, the Coq formalization of the results from Chapter 7 can be found at:

`https://gitlab.inria.fr/cambium/tes`

Alternatively, a snapshot of both projects is available at the following persistent location:

`https://doi.org/10.5281/zenodo.7371093`

Chapter 2

A Separation Logic for Effect Handlers

In this chapter, we introduce Hazel, a Separation Logic with support for effect handlers. This logic enjoys most forms of modular reasoning permitted by Separation Logic: (1) it admits the bind rule, thus allowing *context-local reasoning*; and (2) it admits the frame rule, thus allowing *local reasoning about the state*. Through the novel notion of *protocols*, the Hazel logic allows one to reason separately about a program that performs effects and a program that handles these effects. The contents of this chapter have been presented in a published paper [dVP21].

2.1 Syntax and semantics of *HH*

To reason formally about effect handlers, we must first formalize their operational semantics. To this end, we introduce *HH*, a calculus with support for both shallow handlers (as a primitive construct) and deep handlers (as a derived construct), dynamically allocated mutable state, one-shot continuations, and *unnamed effects*. [1]

Design choices

Mutable state. We choose to support mutable state because there are important applications of effect handlers that exploit this feature (Chapters 4 and 5 provide examples of such applications), and because this feature is also supported by OCaml, a major and realistic programming language to which we would like to apply the reasoning principles that we might discover for *HH*.

One-shot versus multi-shot. Another design choice is the discipline guiding the usage of continuations: should *HH* support multi-shot continuations or should it support one-shot continuations, that is, should continuations be allowed to be called multiple times or at most once? In OCaml, continuations are one-shot, so, if we follow our intent to apply the discovered principles to OCaml, we are inclined to opt for this flavor of continuations. However, there is a more fundamental reason why we wish to impose a one-shot discipline: *multi-shot continuations break the frame rule of Separation Logic*. This means that, in the presence of multi-shot continuations, assuming the frame rule leads to unsound reasoning. The following OCaml program illustrates how the frame rule, in a language with multi-shot continuations, leads to the derivation of an unsound specification.

[1] Chapter 7 applies the key ideas from this chapter to a setting with named effects.

```
1  let call f b =
2    b := 0;          (* Set [b] to zero.                          *)
3    f();             (* Apply frame rule; keep ownership of [b]. *)
4    b := !b + 1;     (* Increment contents of [b] by one.         *)
5    assert (!b = 1)  (* Now, [b] must hold the value one.         *)
```

The function `call` sets `b` to 0, calls a function `f` given as an argument, and then increments `b` by one. Under the assumption that `call` is used in the traditional subset of OCaml without effect handlers, the following specification holds:

$$\forall f.\ \{\mathsf{True}\}\,\mathtt{f}\,()\,\{\mathsf{True}\} \mathrel{-\!\!*} \{\mathtt{b} \mapsto -\}\,\mathtt{call\ f\ b}\,\{\mathtt{b} \mapsto 1\} \tag{2.1}$$

This specification guarantees, in particular, that the assertion on line 5 succeeds. The key step to derive this specification is the following application of the frame rule:

$$\frac{\{\mathsf{True}\}\,\mathtt{f}\,()\,\{\mathsf{True}\}}{\{\mathtt{b} \mapsto x\}\,\mathtt{f}\,()\,\{\mathtt{b} \mapsto x\}}$$

This instance of the frame rule lets one extend the footprint of `f` and argue that the ownership of `b` is recoverable upon `f`'s return.

Specification 2.1, however, is incorrect in a setting with effect handlers and multi-shot continuations. The essential reason why the specification would no longer hold is that, in the presence of multi-shot continuations, a function might terminate multiple times; therefore, the update instruction on line 4 might be executed multiple times, thus invalidating the assertion on line 5. Here is a program that leads to such a behavior:

```
effect Escape : unit -> unit
let f() = perform (Escape ())
let b = ref 0
let _ =
  match call f b with
  | effect (Escape ()) k ->
     continue (Obj.clone_continuation k) ();
     continue k () (* Assertion fails! *)
  | () -> ()
```

This program introduces the effect name `Escape`, defines a function `f` that performs this effect, allocates a reference `b`, and then installs an `Escape` handler over the application `call f b`. When `f` is called, during the execution of `call`, it performs an `Escape` effect and control is transferred to the handler that receives a continuation `k` representing the suspended execution of `call`. This handler then invokes the continuation twice; both invocations trigger the execution of line 4, so `b` is incremented twice, and, during the second time, the assertion in line 5 fails. (OCaml dynamically checks that continuations are resumed at most once, but, here, we bypass this dynamic check by using the unsafe feature `Obj.clone_continuation`, which produces a copy of the continuation. [2])

Unnamed effects. Unlike effects in OCaml, unnamed effects do not carry a name, they carry only a payload value. Once a program performs an unnamed effect, this effect is

[2] This feature is available in version `ocaml-variants.4.12.0+domains+effects` of the OCaml compiler, but not in version 5.

Values, expressions, and operations

$$
\begin{array}{rcl}
\mathit{Op} \ni \odot & ::= & \texttt{+} \mid \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{==} \\
\mathit{Val} \ni h, r, v & ::= & () \mid b\ (\in \mathit{Bool}) \mid i\ (\in \mathit{Int}) \mid \ell\ (\in \mathit{Loc}) \mid \odot\ (\in \mathit{Op}) \\
& \mid & \texttt{rec}\ f\ x.\ e \mid (v, v) \mid \texttt{inj}_i\ v \mid \texttt{cont}\ (\ell, N) \\
\mathit{Expr} \ni e & ::= & v \mid x \mid e\ e \mid (e, e) \mid \texttt{proj}_i\ e \mid \texttt{inj}_i\ e \\
& \mid & \texttt{match}\ e\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{ref}\ e \mid \texttt{!}e \mid e \coloneqq e \\
& \mid & \texttt{do}\ e \mid \texttt{eff}\ v\ N \mid \texttt{try}\ e\ \texttt{with}\ (v \mid v)
\end{array}
$$

Evaluation contexts

$$
\begin{array}{rcl}
\mathit{Ectx} \ni K & ::= & \bullet \mid e\ K \mid K\ v \mid (e, K) \mid (K, v) \mid \texttt{proj}_i\ K \mid \texttt{inj}_i\ K \\
& \mid & \texttt{match}\ K\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ K\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \texttt{ref}\ K \mid \texttt{!}K \mid e \coloneqq K \mid K \coloneqq v \mid \texttt{do}\ K \\
& \mid & \texttt{try}\ K\ \texttt{with}\ (v \mid v) \\
\mathit{Nctx} \ni N & ::= & \bullet \mid e\ N \mid N\ v \mid (e, N) \mid (N, v) \mid \texttt{proj}_i\ N \mid \texttt{inj}_i\ N \\
& \mid & \texttt{match}\ N\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ N\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \texttt{ref}\ N \mid \texttt{!}N \mid e \coloneqq N \mid N \coloneqq v \mid \texttt{do}\ N
\end{array}
$$

Figure 2.1: Syntax of *HH*.

captured by the innermost handler. The reason for adopting this simplification is that it lets us focus on the heart of the problem of devising reasoning principles for effect handlers. Once we solve Challenges 1 and 2, we should be able to adapt the gained insights to variants of *HH*.

Shallow effect handlers. Shallow and deep handlers are inter-expressible [HL18] (in a calculus with recursive functions and binary sums), so this design choice is not important. The reason we opt for shallow handlers is that the encoding of deep handlers on top of shallow handlers is simpler than the encoding in the reverse direction.

Syntax

Figure 2.1 shows the syntax of values, expressions, and evaluation contexts. Most of the constructs are standard. They include recursive functions, binary sums, binary and unary operations, and references. We use infix notation when writing the application of a binary operation to a pair of arguments. We define a non-recursive function as an anonymous recursive function, $\lambda x.\ e \triangleq \texttt{rec}\ \_\ x.\ e$, and we define a let binding as the application of a function to the expression whose result is being bound, $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq (\lambda x.\ e_2)\ e_1$. Non-standard constructs include the instruction for performing an effect, $\texttt{do}\ e$, *active effects*, $\texttt{eff}\ v\ N$, first-class continuations, $\texttt{cont}\ (\ell, N)$, and shallow handlers, $\texttt{try}\ e\ \texttt{with}\ (h \mid r)$. The argument e of the instruction $\texttt{do}\ e$ is called the *effect payload*. The values h and r of a handler expression are called the *effect branch* and the *return branch*, respectively. Active effects play a role in the definition of the operational semantics as we shall explain in the next segment. A continuation carries a location ℓ, which stores a Boolean indicating whether this continuation has been called, thus allowing

Reduction relation $\boxed{e \,/\, \sigma \;\to\; e \,/\, \sigma}$

$$\textsc{BetaStep}\quad (\texttt{rec}\, f\, x.\, e)\; v \,/\, \sigma \;\to\; e\{(\texttt{rec}\, f\, x.\, e)/f\}\{v/x\} \,/\, \sigma$$

$$\textsc{ProjStep}\quad \texttt{proj}_i\; (e_1, e_2) \,/\, \sigma \;\to\; e_i \,/\, \sigma \qquad \textsc{CaseStep}\quad \texttt{match}\; \texttt{inj}_i\; v\; \texttt{with}\; (v_1 \mid v_2) \,/\, \sigma \;\to\; v_i\; v \,/\, \sigma$$

$$\textsc{IfStep}\quad \texttt{if}\; b\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \,/\, \sigma \;\to\; \mathit{if}\; (b = \texttt{true})\; \mathit{then}\; e_1\; \mathit{else}\; e_2 \,/\, \sigma$$

$$\textsc{AllocStep}\quad \frac{\ell \notin \mathit{dom}(\sigma)}{\texttt{ref}\; v \,/\, \sigma \;\to\; \ell \,/\, \sigma[\ell \mapsto v]} \qquad \textsc{ReadStep}\quad \frac{\sigma(\ell) = v}{!\,\ell \,/\, \sigma \;\to\; v \,/\, \sigma} \qquad \textsc{WriteStep}\quad \frac{\ell \in \mathit{dom}(\sigma)}{\ell \coloneqq v \,/\, \sigma \;\to\; () \,/\, \sigma[\ell \mapsto v]}$$

$$\textsc{DoStep}\quad \texttt{do}\; v \,/\, \sigma \;\to\; \texttt{eff}\; v\; \bullet \,/\, \sigma \qquad \textsc{EffStep}\quad \frac{N_1 \neq \bullet}{N_1[\texttt{eff}\; v\; N_2] \,/\, \sigma \;\to\; \texttt{eff}\; v\; (N_1[N_2[\bullet]]) \,/\, \sigma}$$

$$\textsc{TryWithEffectStep}\quad \frac{\ell \notin \mathit{dom}(\sigma)}{\texttt{try}\; (\texttt{eff}\; v\; N)\; \texttt{with}\; (h \mid r) \,/\, \sigma \;\to\; h\; v\; (\texttt{cont}\; (\ell, N)) \,/\, \sigma[\ell \mapsto \texttt{false}]}$$

$$\textsc{TryWithReturnStep}\quad \texttt{try}\; v\; \texttt{with}\; (h \mid r) \,/\, \sigma \;\to\; r\; v \,/\, \sigma \qquad \textsc{ResumeStep}\quad \frac{\sigma(\ell) = \texttt{false}}{(\texttt{cont}\; (\ell, N))\; v \,/\, \sigma \;\to\; N[v] \,/\, \sigma[\ell \mapsto \texttt{true}]}$$

$$\textsc{ContextStep}\quad \frac{e \,/\, \sigma \;\to\; e' \,/\, \sigma'}{K[e] \,/\, \sigma \;\to\; K[e'] \,/\, \sigma'}$$

Figure 2.2: Reduction rules of *HH*.

the implementation of a dynamically enforced one-shot policy. Moreover, a continuation carries an evaluation context N corresponding to the computation that performed an effect under a handler. This evaluation context belongs to the syntactic category of *neutral evaluation contexts*, a subset of usual evaluation contexts K that does not include handlers. Because effects are unnamed, when a handler captures an evaluation context, this context must be neutral.

Semantics

The operational semantics of *HH* is defined by means of a *reduction relation* on pairs of an expression e and a store σ, which is a finite map from memory locations to values. The definition of the reduction relation appears in Figure 2.2. Under rule DoStep, an instruction `do` v reduces to an active effect carrying the value v and the empty context.

This active effect will then gradually *swallow* its surrounding context. Indeed, under rule EffStep, an active effect `eff` v N_2 that evaluates under a neutral context N_1 reduces to the active effect carrying the extended context $N_1[N_2[\bullet]]$. Because this rule applies only to neutral contexts, an active effect does not swallow a handler. Under rule TryWithEffectStep, when an active effect `eff` v N reaches a handler, the handler's effect branch h is called with two arguments: (1) the value v and (2) the reification of N as a first-class continuation. This continuation contains a fresh memory location ℓ that indicates whether the continuation has been called. Initially, the location ℓ stores the value `false`, but, under rule ResumeStep, when a program invokes a continuation, this location is updated to `true`. Therefore, a one-shot policy is indeed enforced. Finally, under rule TryWithReturnStep, if the handlee reduces normally to a value v, then the return branch r is called with v as its argument.

Deep handlers Here is the encoding of deep handlers on top of shallow handlers:

$$\texttt{deep-try}\ e\ \texttt{with}\ (h \mid r) \triangleq \\ (\texttt{rec}\ \mathit{deep}\ \mathit{tk}.\ \texttt{try}\ \mathit{tk}\,\texttt{()}\ \texttt{with}\ (\lambda x\, k.\ h\ x\ (\lambda y.\ \mathit{deep}\ (\lambda\_.\ k\ y)) \mid r))\ (\lambda\_.\ e)$$

The function *deep* takes a thunk *tk* as input and evaluates this thunk under a shallow handler. If the handlee terminates normally, then the return branch r is called. If the handlee performs an effect, then the effect branch h is called with a modified version of the continuation that reinstalls *deep*. Therefore, *deep* handles further effects performed by the continuation. The deep handler construct `deep-try` e `with` $(h \mid r)$ is then simply defined as the application of *deep* to the thunk $\lambda\_.\ e$.

2.2 Logical assertions and protocols

The key idea to address Challenge 1 – which addresses the inability of traditional Separation Logic to specify the effects that a program may perform – is to introduce the notion of a *protocol*. A protocol is a contract established between handler and handlee: it describes a functionality – for example, mutating the state of a memory cell, or forking a thread – on which the handlee can rely, and which the handler must implement.

Instead of building the notion of protocols from scratch, we introduce protocols as a derived notion in Iris [JKJ+18, BB18], a modern and expressive Separation Logic. We choose to work with Iris for the following reasons:

1. *Core logic:* Iris includes of a core logic with Separation-Logic inspired connectives and modalities. This core logic is *language independent*: it is not tied to a particular choice of programming language. For instance, Iris includes a default notion of *weakest precondition*, but this notion is built on top of the core logic. A user is thus free to work with a language of her own choice and to build a notion of weakest precondition according to her own needs.

2. *Ghost state:* Iris has support for *ghost state*, a verification technique that allows one to introduce fictional state that keeps track of logical entities that change during the execution of a program but which do not appear in the program itself. All the case studies in this thesis make use of this technique.

3. *Mechanization:* Iris is formalized in the Coq Proof Assistant [Coq20]. This formal development not only brings confidence of the absence of flaws in the logic to the user, but it also provides an interface with which one can interactively study Iris and its possible extensions [KTB17]. Also, one can easily integrate Iris with other Coq developments formalizing, for example, useful mathematical concepts for the verification of a given program.

2.2.1 Background on Iris

In this subsection, we give a brief explanation of Iris. Our purpose is to give a bird's-eye view of how this logical system works by informally explaining some of its subtle notions, namely ghost state. The reader not acquainted with Iris should not feel discouraged if some notions remain obscure. Throughout this thesis, we recall the key notions as they reappear. Moreover, on all case studies, we hide the use of ghost state behind abstractions that suffice for understanding their core technical content (for example, the use of protocols). If the reader wishes to acquire a solid grasp of Iris, then we recommend Birkedal and Bizjak's lecture notes [BB18] as an introductory reading and Jung et al.'s journal paper [JKJ+18] as an advanced reading, where the model of Iris is documented.

Here is the syntax of a subset of Iris assertions:

$$iProp \ni P \quad ::= \quad P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \triangleright P \mid \dot{\Rightarrow} P \mid \ell \mapsto v \mid \boxed{a}^{\gamma} \mid \dots$$

In a first approximation, an Iris assertion holds relatively to a heap. The separating conjunction $P * Q$ holds of heaps composed of two disjoint parts, one satisfying P and one satisfying Q. The points-to assertion $\ell \mapsto v$ holds of heaps where the location ℓ stores v. The *persistently modality* $\Box$ is used to describe immutable regions of the heap. In particular, if the assertion $\Box P$ holds, then P can be *duplicated*: $P * P$ holds. Moreover, this modality can be used to introduce the notion of a *persistent* assertion P: whenever P holds, the assertion $\Box P$ holds. The *later modality* $\triangleright$ is used to construct recursive definitions. The *update modality* $\dot{\Rightarrow}$ is used to describe heap updates, such as the allocation of references, or writes to a memory location.

This approximation becomes inadequate when dealing with ghost state. In this situation, one must adjust the interpretation of Iris assertions as holding relatively to the set of *ghost variables* $\{\gamma_i\}$ introduced during a verification task. At any point during the verification of a program, the user can introduce a ghost variable γ storing an element of a *camera* M. A camera is an algebraic structure that dictates how the contents of γ can be updated (the complete algebraic characterization of a camera is defined in [JKJ+18]). A ghost variable γ can thus be seen as a global cell holding an element c of a camera M. This element can be *split* into several pieces. Indeed, a camera includes a binary operator $\_ \cdot \_$ such that, if $c = a \cdot b$, then c splits into a and b. The assertion $\boxed{a}^{\gamma}$ means that one of the pieces of the element stored in the global cell γ is a. Therefore, if the assertion $\boxed{a \cdot b}^{\gamma}$ holds, then one can further decompose the global state into the individual pieces a and b; that is, the assertion $\boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ holds. To update the contents of γ, one does not need to collect all these pieces. One can update each individual piece separately as long as their composition remains *well-defined*. (The meaning of *well-defined* is given by the *validity predicate* $\mathcal{V}$, which is also included in the definition of a camera.) In Iris terminology, such updates are qualified as *frame-preserving*. The update modality

is used to describe frame-preserving updates: if updating the piece of ghost state a to b is frame-preserving, then the assertion $\boxed{a}^{\gamma}$ entails $\dot{\Rightarrow}\boxed{b}^{\gamma}$. Finally, when dealing with ghost state, the meaning of the persistently modality is adjusted by means of the (partial) function *core* $|\_|$. The intuition is that, when defined, this function assigns an element c to its *duplicable core*; that is, an element $|c|$ that can be split into two copies of itself: $|c| = |c| \cdot |c|$. The assertion $\Box P$ holds relatively to the state obtained by applying $|\_|$ to the contents of every introduced ghost cell. If the core is not defined at the contents of a particular ghost cell, then this ghost cell is simply ignored. As a simple example, consider a ghost cell γ storing an element c. If $|c|$ is well-defined, then the assertion $\Box\boxed{a}^{\gamma}$ means that a is a piece of $|c|$, rather than c; that is, there exists b, such that $|c| = a \cdot b$.

2.2.2 Protocols

A protocol Ψ is an inhabitant of the type

$$\mathit{Protocol} \triangleq \mathit{Val} \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp}.$$

Therefore, a protocol is a relation between a value u and a predicate Φ. The value u represents the payload used by a program when performing an effect, whereas the predicate Φ represents a specification of the continuation of this program: the assertion $\Phi(w)$ must hold when resuming the program with a value w.

This relation between an effect payload and a specification of a continuation can be used to describe the functionality on which a program fragment relies when performing an effect. For example, one can define the *predicate-pair protocol* $(\Phi_{\text{pre}}; \Phi_{\text{post}})$ that attaches a precondition Φ_{pre} and a postcondition Φ_{post} to performing an effect.

Definition 2.1 (Predicate-pair protocol) *Let Φ_{pre} and Φ_{post} be predicates. The predicate-pair protocol is defined as follows:*

$$(\Phi_{pre}; \Phi_{post}) \triangleq \lambda u\, \Phi.\ \Phi_{pre}(u) * (\forall\, w.\ \Phi_{post}(w) \mathrel{-\!\!*} \Phi(w))$$

This protocol expresses the following contract: to perform an effect with value u, the assertion $\Phi_{\text{pre}}(u)$ must hold; moreover, for some value w, when resuming the continuation with w, the assertion $\Phi_{\text{post}}(w)$ can be assumed to hold. Therefore, from the eyes of the handlee, performing this effect seems as calling a function specified by precondition Φ_{pre} and postcondition Φ_{post}: if the input u satisfies Φ_{pre}, then the output satisfies Φ_{post}.

Send-receive protocol

As the upcoming examples and case studies demonstrate, the pattern of constructing a protocol that attaches a precondition and a postcondition to an effect is very common. Predicate-pair protocols, however, are not well-suited to describe contracts with some form of *dependency* between the precondition and the postcondition. This deficiency is illustrated by the simple example where the effect one wishes to describe is "*to push elements into a stack*". To perform this effect, one must supply a pair of a reference q to the stack and an element u to be inserted into the stack. The precondition should state that the stack currently contains the elements of a list us, and that one has the permission $\mathit{isStack}\, q\, us$ to modify this stack:

$$\Phi^{\text{stack}}_{\text{pre}}(q, u) = \exists\, us.\ \mathit{isStack}\, q\, us$$

The postcondition should state that the value returned is `()`, and that the stack has been updated with the new element u:

$$\Phi^{\text{stack}}_{\text{post}}(()) = \exists\, u\, us.\ \mathit{isStack}\, q\, (u :: us)$$

However, because u is hidden behind an existential quantifier, this postcondition says only that the stack is nonempty. For instance, it does not say that the element on top of the stack corresponds to the last inserted element. It is possible to express this constraint by means of a variable x that stands for u and occurs free in both the precondition and the postcondition. The precondition would state that x corresponds to the element to be inserted in the stack, and the postcondition would state that the stack is extended with x. This variable would have to be bound in such a way that every time a program performs a push operation, it can choose a different instance of x. The *send-receive protocol* includes this flexibility.

Definition 2.2 (Send-receive protocol) *Let $\vec{x}$ and $\vec{y}$ be lists of binders, let v and w be values, and let P and Q be assertions. The send-receive protocol is defined as follows:*

$$!\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\} \triangleq \lambda u\,\Phi.\,\exists\,\vec{x}.\ u = v * P * (\forall\,\vec{y}.\ Q \mathrel{-\!\!*} \Phi(w))$$

This protocol expresses the following contract: to perform an effect with value u, one must find instances of the variables $\vec{x}$ such that $u = v$ and the assertion P holds; moreover, for every instance of the variables $\vec{y}$, the assertion Q can be assumed to hold when resuming the continuation with w, the *return value*.

The scope of a binder in $\vec{x}$ or in $\vec{y}$ includes every term that appears at the right of this binder in the writing of the send-receive protocol. In particular, a binder in $\vec{x}$ can occur free in v, P, w or in Q. Therefore, to describe a push operation of a stack, it suffices to introduce a binder u occurring free in both v and Q:

$$!\,u\,us\,q\,(q,u)\,\{\mathit{isStack}\,q\,us\}.\,?\ (())\,\{\mathit{isStack}\,q\,(u :: us)\}$$

Send-receive protocols subsume predicate-pair protocols. Indeed, it suffices to introduce a binder x that stands for the effect argument and to introduce a binder y that stands for the return value:

$$(\Phi_{\text{pre}}; \Phi_{\text{post}}) \quad \textit{can be written as} \quad !\,x\,(x)\,\{\Phi_{\text{pre}}(x)\}.\,?\,y\,(y)\,\{\Phi_{\text{post}}(y)\}$$

Bottom protocol

The *bottom protocol* declares the absence of effects.

Definition 2.3 *The bottom protocol is defined as follows:*

$$\bot \triangleq\ !\,x\,(x)\,\{\mathsf{False}\}.\,?\,y\,(y)\,\{\mathsf{True}\}$$

This protocol states that, to perform an effect, the assertion False must hold. Therefore, a program that abides by this protocol does not perform effects: it does not rely on a service provided by a handler, and can thus be evaluated in a context deprived of handlers (and, in particular, in the empty context).

Protocol sum

A program usually relies on more than one effect. These effects could be the operations to read and write a memory cell, or operations to fork a thread and wait for the execution of this thread. In such cases, ideally, one would like first to specify each operation by an individual protocol and then to combine these protocols to specify the entire set of operations. The *protocol sum* $\_ + \_$ is a combinator on protocols allowing one to proceed in this way.

Definition 2.4 (Protocol sum) *Let Ψ_1 and Ψ_2 be protocols. The protocol $\Psi_1 + \Psi_2$ is defined as follows:*

$$\Psi_1 + \Psi_2 \triangleq \lambda u\, \Phi.\ \Psi_1\, u\, \Phi \lor \Psi_2\, u\, \Phi$$

Given two protocols Ψ_1 and Ψ_2, the protocol $\Psi_1 + \Psi_2$ expresses the following contract: when performing an effect, a program can abide either by the protocol Ψ_1 or by the protocol Ψ_2.

Protocol equivalence

What are the algebraic properties of the protocol sum? Is it an *associative* combinator? Does it admit a *neutral element*? To answer these questions, we endow protocols with the following notion of equivalence.

Definition 2.5 (Protocol equivalence) *Let Ψ_1 and Ψ_2 be protocols. The relation $\Psi_1 \equiv \Psi_2$ is defined as follows:*

$$\Psi_1 \equiv \Psi_2 \triangleq \vdash \forall\, u\, \Phi.\ (\Psi_1\, u\, \Phi \mathrel{-\!\!*} \Psi_2\, u\, \Phi) * (\Psi_2\, u\, \Phi \mathrel{-\!\!*} \Psi_1\, u\, \Phi)$$

The turnstile symbol $\vdash$ denotes a sequent of the Iris logic: let P be an Iris assertion, the sequent $\vdash P$ is a *meta-level assertion* stating that P is derivable from Iris proof rules [BB18]. In accordance to the mechanization of Iris in the Coq Proof Assistant, our meta-level logic is the Calculus of Inductive Constructions (CIC), an intuitionistic logic for which Coq offers an interface. Therefore, meta-level assertions inhabit the type *Prop* of CIC propositions.

Protocol equivalence is an *equivalence relation* on protocols: it is symmetric, transitive, and reflexive. Moreover, under this notion of equivalence, the triple $(\mathit{Protocol}, +, \bot)$ forms a *commutative monoid*:

$$\begin{array}{rcll} \Psi_1 + \Psi_2 & \equiv & \Psi_2 + \Psi_1 & (+ \text{ is commutative}) \\ \Psi_1 + (\Psi_2 + \Psi_3) & \equiv & (\Psi_1 + \Psi_2) + \Psi_3 & (+ \text{ is associative}) \\ \bot + \Psi & \equiv & \Psi & (\bot \text{ is a left neutral element of } +) \\ \Psi + \bot & \equiv & \Psi & (\bot \text{ is a right neutral element of } +) \end{array}$$

Upward closure

A key notion when working with protocols is that of a *monotonic protocol*.

Definition 2.6 (Monotonic) *A protocol Ψ is monotonic if the following sequent is derivable:*

$$\vdash \forall u, \Phi, \Phi'.\ (\forall\, w.\ \Phi(w) \mathrel{-\!\!*} \Phi'(w)) \mathrel{-\!\!*} \Psi\, u\, \Phi \mathrel{-\!\!*} \Psi\, u\, \Phi'$$

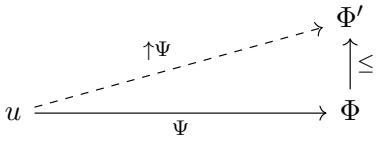
Intuitively, a monotonic protocol Ψ expresses the idea that, if, according to the protocol Ψ, the handler-provided answer to performing an effect with payload z satisfies the predicate Φ, then this answer satisfies a weaker predicate Φ'.

Most of the examples of protocol that we have seen so far are monotonic: send-receive protocols are monotonic, and, consequently, so is the bottom protocol. However, an arbitrary protocol is not necessarily monotonic. In such cases, one can take its *upward closure*.

Definition 2.7 (Upward closure) *The upward closure of a protocol Ψ is defined as follows:*

$$\uparrow \Psi \triangleq \lambda\, u\, \Phi'.\, \exists\, \Phi.\, \Psi\, u\, \Phi \;*\; (\forall\, w.\, \Phi(w) \mathrel{-\!\!*} \Phi'(w))$$

The upward closure is monotonic by construction. Graphically, it is the dashed arrow in the following diagram:



The protocol-annotated arrows represent a protocol relation between the value u and a predicate. The $\leq$-annotated arrow means that Φ is *stronger* than Φ': for every w, $\Phi(w)$ implies $\Phi'(w)$.

Here is a summary of the properties of the upward closure:

Lemma 2.1 (Properties of the upward closure) *The upward closure is monotonic:*

For every Ψ, the protocol $\uparrow \Psi$ is monotonic.

Moreover, the upward closure satisfies the following properties:

1. *The upward closure has no action over monotonic protocols:*

$$\uparrow \Psi \equiv \Psi \quad \textit{(for every monotonic protocol } \Psi\textit{)}$$

   In particular, the following equations hold:

$$\begin{array}{rcl} \uparrow (!\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\}) & \equiv & !\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\} \\ \uparrow \bot & \equiv & \bot \end{array}$$

2. *The upward closure distributes over the protocol sum:*

$$\uparrow (\Psi_1 + \Psi_2) \equiv \uparrow \Psi_1 + \uparrow \Psi_2 \quad \textit{(for every } \Psi_1 \textit{ and } \Psi_2\textit{)}$$

Protocol ordering

Finally, to conclude the presentation of protocols, we introduce the notion of *protocol ordering*. This notion is important in the statement of the monotonicity property of the program logic.

Weakest precondition $\boxed{\mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\}}$

(EWP1) $\quad \mathit{ewp}\ v\ \langle\Psi\rangle\{\Phi\} \triangleq \dot{\Rrightarrow}\Phi(v)$

(EWP2) $\quad \mathit{ewp}\ (\mathtt{eff}\ v\ N)\ \langle\Psi\rangle\{\Phi\} \triangleq (\uparrow\Psi)\, v\, (\lambda w.\ \triangleright\, \mathit{ewp}\ N[w]\ \langle\Psi\rangle\{\Phi\})$

(EWP3) $$\mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\} \triangleq \forall\sigma.\, S(\sigma) \Rrightarrow\!\!\!\!\ast \begin{cases} \exists e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ * \\ \forall e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma' \Rrightarrow\!\!\!\!\ast \triangleright \dot{\Rrightarrow} \\ \qquad S(\sigma')\ *\ \mathit{ewp}\ e'\ \langle\Psi\rangle\{\Phi\} \end{cases}$$

Figure 2.3: Definition of *ewp*.

Definition 2.8 (Protocol ordering) *The protocol Ψ_1 is stronger than Ψ_2 if the following assertion holds:*

$$\Psi_1 \sqsubseteq \Psi_2 \triangleq \Box\forall\, u\, \Phi.\ \Psi_1\, u\, \Phi \mathrel{-\!\!*} \Psi_2\, u\, \Phi$$

The assertion $\Psi_1 \sqsubseteq \Psi_2$ means that, for every value u and predicate Φ, if the protocol Ψ_1 allows performing an effect with u in a context specified by Φ, then so does the protocol Ψ_2. The use of the persistently modality ensures that this weakening argument can be applied as many times as a program performs effects.

2.3 Program logic

In this section, we introduce a rich language with which one can describe the behavior of programs with effect handlers. This language is the result of the combination of the Iris base logic, the notion of protocols, and the reduction relation of *HH* programs. The knot that ties all of these concepts together is the notion of *weakest precondition*. In Subsection 2.3.1, we present the definition of this notion, and in Subsection 2.3.2, we study its properties.

2.3.1 Weakest precondition

In traditional Separation Logic, the notion of weakest precondition is an assertion that relates a program e to a postcondition Φ. The informal meaning of this assertion is the claim that e can be safely executed (it does not get *stuck*) and that, if this program produces an output v, then the assertion $\Phi(v)$ holds. With *HH* programs, we have to construct the notion of weakest precondition in a slightly different way: indeed, since *HH* programs can perform effects, a postcondition is no longer sufficient to describe all the states in which a program fragment can exit; we must also specify the state when a program performs an effect. To this end, we introduce a novel Separation Logic called Hazel, where the notion of weakest precondition is parameterized with a protocol Ψ. The weakest precondition is thus an assertion with the following shape: $\mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\}$, where *ewp* stands for *extended weakest precondition*. Its informal reading is a threefold claim about the execution of e. First, e can be safely executed. Second, e abides by the protocol Ψ when performing effects. Third, if e produces an output v, then $\Phi(v)$ holds.

The formal definition of *ewp* appears in Figure 2.3. [3] It is presented as a set of three defining laws. The law (EWP1) covers the case where the expression e is a value. The law (EWP2) is applicable when e is an effect, that is, an expression of the form $\texttt{eff}\ v\ N$. The law (EWP3) covers the remaining cases.

The later modalities $\triangleright$ that appear in (EWP2) and (EWP3) guard the occurrences of *ewp* on the right-hand side and thereby ensure that this recursive definition is accepted (that is, these equations do have a solution).

The law (EWP1) formalizes the claim that, if a program reduces to a value v, then v must satisfy the postcondition Φ.

The law (EWP2) defines the meaning of the assertion $\mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\}$ when e is an active effect $\texttt{eff}\ v\ N$. This law expresses two requirements. First, it must be the case that, according to the protocol Ψ, the request v is permitted. Second, it must be the case that, for every reply w that the protocol Ψ permits, plugging the value w into the evaluation context N yields a term $N[w]$ that behaves in accordance with the protocol Ψ and the postcondition Φ. According to this law, after performing one effect that conforms to the protocol step Ψ, the program must still conform to the protocol Ψ.

The law (EWP3) describes the case where e is neither a value nor an effect. Then, we expect e to be able to make one step of computation. (Indeed, e would otherwise be stuck. That would represent a runtime error, which we want to forbid.) Regardless of which step of computation is performed, we expect it to result in an expression e' that satisfies the specification $\mathit{ewp}\ e'\ \langle\Psi\rangle\{\Phi\}$.

State interpretation. The predicate S that appears in law (EWP3) is the *state-interpretation* predicate. It encodes an invariant about the store; for this reason, the right-hand side of the law involves an assumption $S(\sigma)$ and a goal $S(\sigma')$. More precisely, this predicate states ownership over a piece of a fixed ghost cell γ_{heap}. The contents of this cell inhabit an *authoritative camera* [JKJ+18]. An authoritative camera is a construction $\textsc{Auth}(M)$ that builds a camera structure from a given camera M. The elements of this construction are of two kinds: (1) there are *authoritative elements*, written $\bullet\, c$, where $c \in M$; and (2) there are *fragments*, written $\circ\, a$, where $a \in M$. An authoritative element $\bullet\, c$ is a distinguished element, which, by the composition rules of the authoritative camera, cannot be split into smaller pieces. Therefore, the ghost cell γ_{heap} contains one (and exactly one) such piece. Fragments, on the other hand, follow the composition rules of the camera M:

$$\circ\,(a_1 \cdot_M a_2) = \circ\, a_1 \cdot_{\textsc{Auth}(M)} \circ\, a_2$$

The validity-predicate $\mathcal{V}$ of $\textsc{Auth}(M)$ is defined in such a way that every fragment $\circ\, a$ is *smaller* than the unique authoritative element $\bullet\, c$; that is, there exists b such that $c = a \cdot_M b$. The idea is thus to set the physical heap σ as the authoritative-element piece of γ_{heap} and to define the points-to predicate $\ell \mapsto v$ as asserting that one of the fragment pieces of γ_{heap} is $\circ\,\{\ell \mapsto v\}$. Indeed, we let the contents of γ_{heap} range over the following camera:

$$\textsc{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} \textsc{Ex}(\mathit{Val})) \tag{2.2}$$

[3] This definition is similar to the one presented in the paper [dVP21]. The only difference is that, in the paper, the weakest precondition is parameterized by a *mask*, which is part of Iris's mechanism to support invariants. None of the case studies exploit this verification technique, so we choose to simplify the presentation.

And we define the predicate S and the points-to predicate as follows:

$$S(\sigma) \triangleq \boxed{\bullet\,\sigma}^{\gamma_{heap}} \qquad \ell \mapsto v \triangleq \boxed{\circ\,\{\ell \mapsto v\}}^{\gamma_{heap}}$$

The construction $\mathrm{Ex}(A)$, called the *exclusive camera* [JKJ+18], builds a camera from a given set A. The set of elements of this camera includes a special element $\lightning$ and includes the injection of elements of A through the map $\mathsf{ex}(-) : A \to \mathrm{Ex}(A)$. In the definitions of S and of the points-to predicate, to pass from a physical heap to an element of the camera $\mathit{Loc} \xrightarrow{\text{fin}} \mathrm{Ex}(\mathit{Val})$, we implicitly apply the following coercion:

$$\{\ell \mapsto v\} \quad \longmapsto \quad \{\ell \mapsto \mathsf{ex}(v)\}$$

Elements of an exclusive camera do not split, therefore we obtain the property that a points-to predicate $\ell \mapsto v$ is non-duplicable, thus expressing full ownership over the location ℓ. Moreover, the choice of Camera 2.2, from which the contents of γ_{heap} are taken, induces the following logical rules (where the operation $\_ \sqcup \_$ denotes disjoint map union):

$$\text{(ExtendHeap)} \qquad \boxed{\bullet\,\sigma}^{\gamma_{heap}} \mathrel{-\!\!*} \dot{\Rrightarrow} \left\{ \begin{array}{l} \boxed{\bullet\,(\sigma \sqcup \{\ell \mapsto v\})}^{\gamma_{heap}} \\ \boxed{\circ\,\{\ell \mapsto v\}}^{\gamma_{heap}} \end{array} \right. *$$

$$\text{(UpdateHeap)} \qquad \begin{array}{l} \boxed{\bullet\,(\sigma \sqcup \{\ell \mapsto v\})}^{\gamma_{heap}} \mathrel{-\!\!*} \\ \quad \boxed{\circ\,\{\ell \mapsto v\}}^{\gamma_{heap}} \mathrel{-\!\!*} \end{array} \dot{\Rrightarrow} \left\{ \begin{array}{l} \boxed{\bullet\,(\sigma \sqcup \{\ell \mapsto w\})}^{\gamma_{heap}} \\ \boxed{\circ\,\{\ell \mapsto w\}}^{\gamma_{heap}} \end{array} \right. *$$

$$\text{(LookupHeap)} \qquad \boxed{\bullet\,\sigma}^{\gamma_{heap}} \mathrel{-\!\!*} \boxed{\circ\,\{\ell \mapsto v\}}^{\gamma_{heap}} \mathrel{-\!\!*} \{\ell \mapsto v\} \in \sigma$$

We exploit rules (ExtendHeap), (UpdateHeap), and (LookupHeap) from the position of the designers of the logic to prove the reasoning rules dealing with state (rules Alloc, Write, and Read from Subsection 2.3.2). A user of the logic does not need to be aware of these rules or the ghost cell γ_{heap} whatsoever. In Hazel, the statement of correctness of a complete program e is universally quantified by γ_{heap}, therefore, this ghost cell can indeed be seen as an abstract name. As we shall see, to extract a meta-level property about e from such a statement, one must apply the soundness theorem of Hazel (Section 2.3.3). The application of this theorem corresponds to the allocation of a concrete ghost cell γ_{heap} and the specialization of the result from the verification task to this cell.

2.3.2 Reasoning rules

We now establish a set of reasoning rules, which can be used to prove *ewp* assertions, and thereby prove properties of programs or program fragments. The main reasoning rules appear in Figure 2.4. Each inference rule should be understood as a (universally quantified) magic wand.

Rule Value expresses the idea that, at any time, a program can return a value v that satisfies the postcondition.

Rule Do reflects the idea that a program can perform an effect provided that it abides by the currently installed protocol. Indeed, the rule dictates that, to perform an effect with argument v under a context specified by Φ, the assertion $(\uparrow \Psi)\, v\, \Phi$ must hold. This rule reveals the twofold interpretation of a postcondition as both a description of the final state

$$\textsc{Value}\ \frac{\Phi(v)}{ewp\ v\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{Do}\ \frac{(\uparrow\Psi)\,v\,\Phi}{ewp\ (\texttt{do}\ v)\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{Monotonicity}\ \frac{ewp\ e\ \langle\Psi_1\rangle\{\Phi_1\} \quad \Psi_1 \sqsubseteq \Psi_2 \quad \forall w.\,\Phi_1(w) \mathrel{-\!\!*} \Phi_2(w)}{ewp\ e\ \langle\Psi_2\rangle\{\Phi_2\}}$$

$$\textsc{Bind}\ \frac{ewp\ e\ \langle\Psi\rangle\{w.\ ewp\ N[w]\ \langle\Psi\rangle\{\Phi\}\}}{ewp\ N[e]\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{BindPure}\ \frac{ewp\ e\ \langle\bot\rangle\{w.\ ewp\ K[w]\ \langle\Psi\rangle\{\Phi\}\}}{ewp\ K[e]\ \langle\Psi\rangle\{\Phi\}}$$

$$\textsc{TryWithShallow}\ \frac{ewp\ e\ \langle\Psi\rangle\{\Phi\} \quad \mathit{shallow\text{-}handler}\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}{ewp\ (\texttt{try}\ e\ \texttt{with}\ (h \mid r))\ \langle\Psi'\rangle\{\Phi'\}} \qquad \textsc{TryWithDeep}\ \frac{ewp\ e\ \langle\Psi\rangle\{\Phi\} \quad \mathit{deep\text{-}handler}\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}{ewp\ (\texttt{deep-try}\ e\ \texttt{with}\ (h \mid r))\ \langle\Psi'\rangle\{\Phi'\}}$$

$$\textsc{Read}\ \frac{\ell \mapsto v}{ewp\ (!\,\ell)\ \langle\Psi\rangle\{y.\ y = v\ *\ \ell \mapsto v\}} \qquad \textsc{Write}\ \frac{\ell \mapsto -}{ewp\ (\ell \coloneqq w)\ \langle\Psi\rangle\{\_.\ \ell \mapsto w\}}$$

$$\textsc{Alloc}\ \frac{}{ewp\ (\texttt{ref}\ v)\ \langle\Psi\rangle\{\ell.\ \ell \mapsto v\}}$$

Figure 2.4: Reasoning rules.

of a program and a specification of its surrounding environment. Rule Value endorses the first reading, while rule Bind, which we shall discuss in the following paragraphs, endorses the second reading.

Rule Monotonicity expresses the idea that, if a program has been verified with a protocol Ψ_1 and a postcondition Φ_1, then it can be used in a context requiring less strict constraints Ψ_2 and Φ_2. This rule justifies Hazel's version of the frame rule:

$$\textsc{Frame}\ \frac{R\ *\ ewp\ e\ \langle\Psi\rangle\{\Phi\}}{ewp\ e\ \langle\Psi\rangle\{w.\,R\ *\ \Phi(w)\}}$$

Indeed, rule Frame is derivable by the straightforward application of rule Monotonicity. Rule Frame states that, if a program e can be executed in a context deprived of the resources governed by R, then it is sound to assume that running e in a context extended with such resources will not compromise them.

Rule Bind allows reasoning about sequential composition. To verify a program e that evaluates under a neutral context N, one can first verify e in isolation and then verify the program, which consists of placing the output of e in N. Because there is no handler frame in N, the effects that e may perform follow the same interpretation as those performed by $N[w]$ (where w is the result of e). Therefore, both programs abide by the same protocol Ψ. In a context with handlers, one may reason according to the rule BindPure, which allows one to decouple the reasoning of a program e from an arbitrary evaluation context K, provided that e does not perform effects.

$$
\begin{array}{l}
\mathit{shallow\text{-}handler}\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\} \triangleq \\
\quad \textit{(Return branch)} \quad (\forall\, v.\ \Phi(v) \mathrel{-\!\!*} \mathit{ewp}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}) \wedge \\
\quad \textit{(Effect branch)} \quad (\forall\, v,\ k.\ (\uparrow \Psi)\, v\, (\lambda\, w.\ \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}) \mathrel{-\!\!*} \mathit{ewp}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\})
\end{array}
$$

$$
\begin{array}{l}
\mathit{deep\text{-}handler}\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\} \triangleq \\
\quad \textit{(Return branch)} \quad (\forall v.\ \Phi(v) \mathrel{-\!\!*} \mathit{ewp}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}) \wedge \\
\quad \textit{(Effect branch)} \quad \left(\forall\, v,\ k.\ \left\{ \begin{array}{l} (\uparrow \Psi)\, v\, (\lambda\, w.\ \forall\, \Psi'',\ \Phi''. \\ \quad \triangleright\, \mathit{deep\text{-}handler}\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi''\rangle\{\Phi''\} \mathrel{-\!\!*} \\ \quad\quad \mathit{ewp}\ (k\ w)\ \langle\Psi''\rangle\{\Phi''\}) \\ \quad \mathrel{-\!\!*} \\ \mathit{ewp}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\} \end{array} \right. \right)
\end{array}
$$

Figure 2.5: Definitions of the predicates *shallow-handler* and *deep-handler*.

Rule TryWithShallow allows one to reason separately about handlee and handler. This rule justifies the claim that a protocol works as a contract between handlee and handler. All the handlee needs to know is that there is an enclosing handler that implements effects according to a protocol Ψ. The handler, on the other hand, can perform effects according to another protocol Ψ', which dictates its contract with a further enclosing handler.

The predicate *shallow-handler* is the *shallow-handler judgment*. It comprises the specification of the handler branches h and r. Its definition appears in Figure 2.5. The specification of the return branch states that r should be prepared to handle any output of the handlee. The specification of the effect branch states that h should be prepared to handle any effect that the handlee may perform. The effect branch can assume that these effects are performed according to the protocol Ψ. This is expressed by the assertion

$$(\uparrow \Psi)\, v\, (\lambda\, w.\ \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}),$$

which specifies both the effect argument v and the continuation k. The handler can thus resume k with a reply that conforms to this protocol. In accordance to *HH*'s one-shot policy, the logic allows the handler to invoke the continuation at most once. This restriction is instrumented in the definition of the upward closure. Indeed, unfolding the definition of the upward closure in the above assertion yields the following one:

$$\exists\, \Phi'.\ \Psi\, v\, \Phi' \ *\ (\forall\, w.\ \Phi'(w) \mathrel{-\!\!*} \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\})$$

One can see that the specification of the continuation, the assertion

$$\forall\, w.\ \Phi'(w) \mathrel{-\!\!*} \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\},$$

is *affine*: it can be applied at most once. Iris assertions are affine by default. To circumvent this behavior, one may guard assertions by a persistently modality, but this is not the case here.

Rule TryWithDeep allows one to reason separately about a deep handler and its handlee. This rule is similar to rule TryWithShallow. It differs only in the definition of the predicate *deep-handler*.

The deep-handler judgment comprises the specification of the handler branches h and r. The first conjunct is the specification of the return branch. It states that the correctness of the return branch can be established under the assumption that the value v is the output of the handlee, thereby satisfying the handlee's postcondition Φ. The second conjunct is the specification of the effect branch. It states that the correctness of the handler branch can be established under the assumption that (1) v is the payload of an effect performed by the handlee and (2) k is a continuation representing the context under which the handlee performed this effect. This assumption is expressed by the upward closure that appears as the premise of the *ewp* assertion concerning the effect branch h. This premise is similar to the corresponding statement of the upward closure in the definition of the shallow-handler judgment. There are however two key differences: (1) the recursive occurrence of the deep-handler judgment and (2) the universally quantified terms Ψ'' and Φ''. Intuitively, this recursive occurrence of the judgment reflects the idea that, because a new instance of the handler is reinstalled, to invoke the continuation, this instance must be proven correct by establishing a new handler judgment. Because *HH* has support for mutable state, the behavior of this new instance of the handler may differ from the one that was originally installed (and which captured the continuation k). The new instance might not abide by the same protocol as the original handler does. For this reason, the specification includes some flexibility in the choice of the protocol Ψ'' and postcondition Φ'' of the new handler instance.

According to our experience, the flexibility on the choice of the protocol Ψ'' is not extremely useful. We have applied it to only one case study (which is not documented in this thesis): the verification, in a variant of Hazel, of the implementation of ML-like references using effect handlers. The flexibility on the choice of the postcondition Φ'', on the other hand, is crucial. For instance, we exploit this flexibility in the case studies from Chapters 3 and 5.

Rules Alloc, Write, and Read are the standard rules of Separation Logic for reasoning about dynamically allocated mutable state [ORY01, Rey02, Cha21]. These rules state that the operations for allocating, writing, and reading references abide by a universally quantified protocol Ψ, thereby expressing the fact that these operations do not perform effects.

2.3.3 Soundness

The *adequacy theorem* justifies that reasoning in terms of *ewp* and Hazel's reasoning rules is sound:

Theorem 2.1 (Adequacy) *Let e be a closed expression. If ewp e $\langle\bot\rangle\{\Phi\}$ holds, then e is safe.*

If one verifies e by proving a weakest precondition statement with the bottom protocol and an arbitrary postcondition Φ, then e is safe: the execution of e must either diverge or terminate with a value; it cannot crash or terminate with an unhandled effect. Because our operational semantics includes a one-shot check, the absence of crashes implies that no continuation is invoked twice.

Proof The proof of Theorem 2.1 is essentially the same as the proof of the adequacy theorem documented in [JKJ+18] (Theorem 6 of Section 6.4).

The first step is to show that the reduction of e *preserves* the weakest precondition; that is, for every nonnegative integer k, for every pair of stores σ and σ', and for every expression e', if $e \mathbin{/} \sigma$ reduces to $e' \mathbin{/} \sigma'$ in k steps, $e \mathbin{/} \sigma \rightarrow^k e' \mathbin{/} \sigma'$, then the assertion $\boxed{\bullet\, \sigma}^{\gamma_{heap}} * \mathit{ewp}\ e\ \langle\bot\rangle\{\Phi\}$ implies that the assertion $\boxed{\bullet\, \sigma'}^{\gamma_{heap}} * \mathit{ewp}\ e'\ \langle\bot\rangle\{\Phi\}$ holds after a series of k alternating logical updates and later steps:

$$\forall\, k,\, e,\, \sigma,\, e',\, \sigma'.\ (e \mathbin{/} \sigma \rightarrow^k e' \mathbin{/} \sigma') \mathrel{-\!\!*} \\ \left(\boxed{\bullet\, \sigma}^{\gamma_{heap}} * \mathit{ewp}\ e\ \langle\bot\rangle\{\Phi\}\right) \mathrel{-\!\!*} (\dot{\Rightarrow}\triangleright)^k \left(\boxed{\bullet\, \sigma'}^{\gamma_{heap}} * \mathit{ewp}\ e'\ \langle\bot\rangle\{\Phi\}\right)$$

The proof of this assertion follows by induction on k, and needs to unfold *ewp*'s definition.

The second step is to show that, for every expression e', the weakest precondition $\mathit{ewp}\ e'\ \langle\bot\rangle\{\Phi\}$ implies that e' is either a value or a reducible expression:

$$\forall\, e',\, \sigma'.\ \boxed{\bullet\, \sigma'}^{\gamma_{heap}} * \mathit{ewp}\ e'\ \langle\bot\rangle\{\Phi\} \mathrel{-\!\!*} (e' \in \mathit{Val}) \vee (e' \mathbin{/} \sigma' \rightarrow \_ \mathbin{/} \_)$$

This result follows from the unfolding of *ewp*. The bottom protocol $\bot$ excludes the case where e is an active effect.

The combination of these steps leads to the following assertion:

$$\forall\, k,\, e,\, \sigma,\, e',\, \sigma'.\ (e \mathbin{/} \sigma \rightarrow^k e' \mathbin{/} \sigma') \mathrel{-\!\!*} \\ \left(\boxed{\bullet\, \sigma}^{\gamma_{heap}} * \mathit{ewp}\ e\ \langle\bot\rangle\{\Phi\}\right) \mathrel{-\!\!*} (\dot{\Rightarrow}\triangleright)^k \left((e' \in \mathit{Val}) \vee (e' \mathbin{/} \sigma' \rightarrow \_ \mathbin{/} \_)\right) \tag{2.3}$$

The plan now is to apply Iris's soundness theorem (Theorem 5 in [JKJ+18]): for every proposition $P : \mathit{Prop}$ in the meta logic, if $(\dot{\Rightarrow}\triangleright)^k\, P$ holds in Iris, then P holds in the meta logic. One must apply this theorem to the conclusion of Assertion 2.3 to derive the following (meta-level) theorem, which is the formal statement that e is safe:

$$\forall\, e,\, \sigma,\, e',\, \sigma'.\ (e \mathbin{/} \sigma \rightarrow^* e' \mathbin{/} \sigma') \implies (e' \in \mathit{Val}) \vee (e' \mathbin{/} \sigma' \rightarrow \_ \mathbin{/} \_)$$

However, Iris's soundness theorem applies only to assertions that hold in the empty context. So, to conclude the proof, one must fulfill the premise $\boxed{\bullet\, \sigma}^{\gamma_{heap}} * \mathit{ewp}\ e\ \langle\bot\rangle\{\Phi\}$. The assertion $\mathit{ewp}\ e\ \langle\bot\rangle\{\Phi\}$ is the theorem's hypothesis, so it holds trivially. The assertion $\boxed{\bullet\, \sigma}^{\gamma_{heap}}$ can be shown to hold after the allocation of the ghost cell γ_{heap} initialized with σ (Rule Ghost-Alloc from Figure 4 of Section 2.1 in [JKJ+18]). This concludes the proof. ∎

2.4 Related work

To the extent of our knowledge, Hazel is the first Separation Logic with support for effect handlers. More broadly speaking, the work of this chapter belongs to the intersection of two research topics: (1) reasoning about effect handlers, and (2) program logics for control operators. We discuss each of these topics separately.

Reasoning about effect handlers

Denotational semantics. Plotkin and Pretnar [PP09] introduce a denotational semantics for a language with effect handlers. This semantics allows one to think of a

computation as a tree whose nodes are effectful operations, and to think of an effect handler as a *deconstructor* of such computations: an effect handler traverses the tree and substitutes each effectful operation with an implementation. More precisely, effectful operations are described by an equational theory whose free model corresponds to the semantic domain of computations. An effect handler corresponds to the application of a homomorphism from the free model to a user-defined model. This denotational semantics is defined only when handlers are correct: the handler must satisfy the equations of the algebraic theory associated to an effect. The authors also show how to adapt their previous equational logic [PP08] to account for effect handlers. This logic allows one to state and prove that two programs are equivalent. Once such a logical judgement is proven, the soundness statement of the logic implies equality between the denotational interpretations of each program.

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

Letan et al. [LRCH18, LRCH21] develop FreeSpec, a Coq library similar to ITrees. The main difference with respect to ITrees is is that the tree-like structure exposed by the FreeSpec library is an inductive definition. Therefore, the range of applications of FreeSpec is limited to terminating programs. One advantage of this restriction is that one can compute the result of operations on such inductive constructions directly in Coq, thus avoiding Coq's mechanism of code extraction.

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

Contextual equivalence. In much of the previously discussed work, the focus is on reasoning about the mathematical model of programs with handlers: Plotkin and Pretnar's equational logic allows one to establish that two programs have the same denotation; Interaction trees is a purely mathematical structure that can be the target of a denotational semantics. Another approach is to reason directly about programs with handlers through contextual equivalence: one wishes to establish when two implementations of a program can be exchanged without affecting the complete program.

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

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

Program logics for control operators

Floyd-Hoare logics. Berger [Ber09] introduces a program logic for PCF^+, an extension of PCF [Plo77] with `callcc` and `throw`. The logic is endowed of a rich assertion language allowing one to specify how control is transferred during the execution of a program. Indeed, Berger's logic suggests the interpretation of a program with higher-order control operators as the asynchronous execution of multiple processes. From this perspective, performing the operation $\texttt{throw}\, x\ \bar{v}$, for example, can be seen as pausing the current process and transferring control to the process named x, which is resumed with the vector of values $\bar{v}$. The logical assertion $\bar{x}\,\langle\bar{v}\rangle\,A$ specifies such control jumps: it states that a program so specified jumps to x with values $\bar{v}$, at which moment the assertion A holds. Berger shows that the logic enjoys *descriptive completeness* [HBY06], which intuitively means that assertions can precisely describe the behavior of programs.

Crolard and Polonowski [CP12] introduce a program logic for reasoning about terminating programs with support for non-local jumps, for `callcc`, and for mutable stack variables, but no support for dynamically allocated references. Their program logic is embedded in a dependent type theory: specifications are written as types, and reasoning rules are stated as typing rules. Even though both the language and the dependent type theory in question are quite restrictive (types can depend only on first-class values such as integers), Crolard and Polonowski [CP11] successfully apply their approach to prove the correctness of Filinski's [Fil94] encoding of `shift`/`reset` in terms of `callcc` and a *meta-continuation*. Their results are formalized in Ott [SZO+10] and Twelf [PS99].

Separation logics. Delbianco and Nanevski [DN13] conceive HTT_{cc}, a program logic embedded in Coq with support for `callcc` and `abort`. HTT_{cc} is built as an extension of HTT, a system that offers an interface to construct programs of an expressive dependent type $ST\,A\,(p, q)$. This type not only describes the type A of the program's output but also includes a Hoare-style specification with a precondition p and a postcondition q. Nanevski et al. [NAMB07, NVB10] show that it is possible to extend HTT with the usual reasoning rules from Separation Logic by restricting programs to the type $STsep$, a subset of ST where the frame rule is baked in program specifications. HTT_{cc}, however, does not support the frame rule, because programs in this variant of HTT do not necessarily

inhabit the type *STsep*. If a program happens to preserve the part of the heap that it does not know about, then this must be explicitly stated in its specification, by universally quantifying over a residual heap [DN13, Section 3].

Timany and Birkedal [TB19] develop an Iris-based Separation Logic for a calculus equipped with dynamically allocated mutable state, concurrency, and `call/cc` and `throw`. Whereas we present a unary program logic, which can prove the safety of one program, they develop a binary framework, which can be used to establish a contextual refinement assertion between two programs. Timany and Birkedal point out that "non-local control flow breaks the bind rule". They define a predicate *wp* that does not have a bind rule, but allows a certain style of low-level reasoning: the reasoning rules that describe `call/cc` and `throw` paraphrase the operational semantics. On top of this, they define a "context-local weakest-precondition" predicate *clwp*, which does enjoy a bind rule, but is restricted to expressions that have no observable control effects. In contrast, in our system, both Bind and BindPure hold: the logic reflects the fact that the expressions $K[e]$ and $\texttt{let}\ x = e\ \texttt{in}\ K[x]$ are equivalent provided that none of the effects performed by e are handled by K. We discuss both Timany and Birkedal's work and Delbianco and Nanevski's work further in Chapter 6, where we study the encoding of `callcc` and `throw` in a variant of *HH* with multi-shot continuations.

Chapter 3

Control Inversion

With the introduction of protocols and the subsequent notion of weakest precondition, we have solved Challenge 1. Indeed, a specification in Hazel is parameterized by a protocol that specifies the effects that a program may perform, thus repairing a limitation of the traditional specification language of Separation Logic. Moreover, with the introduction of Hazel's reasoning rules, we have solved Challenge 2. Indeed, Hazel includes reasoning rules for both shallow and deep handlers, thus enabling the verification of programs with delimited-control operators and continuations. In this chapter, we show that the Hazel logic is sufficiently powerful to verify `hh_invert`, a *HH* version of `invert` (Chapter 1). This verification is concisely documented in a published paper [dVP21].

3.1 Implementation

The implementation that we verify is the following translation of the OCaml code from Figure 1.1 to *HH*:

```
1  (* Implementation of control inversion in HH. *)
2  let hh_invert iter = fun () ->
3    let yield = fun x -> do x in
4    deep-try iter yield with
5    ( fun x k -> cons (x, k)
6    | fun _   -> nil
7    )
```

Figure 3.1: Translation of `invert` (Figure 1.1) to *HH*.

There are three differences between the OCaml and the *HH* versions. First, *HH* is untyped. We do not introduce the type of sequences and heads as in Figure 1.1. The lazy-sequence constructors, `cons` and `nil`, are insted encoded using binary sums:

$$\begin{aligned} \texttt{cons (x, k)} &\triangleq \texttt{inj}_1\ \texttt{(x, k)} \\ \texttt{nil} &\triangleq \texttt{inj}_2\ \texttt{()} \end{aligned}$$

Second, in *HH*, effects are unnamed. We do not introduce an effect name `Yield` as in the OCaml version; the function `yield` performs an unnamed effect "`do x`" instead. Third, in *HH*, there is no special syntax to invoke a continuation: note that there is no `continue` instruction to resume k in line 5 of the *HH* version.

Putting these differences aside, the two versions are very similar, so `hh_invert` can be briefly explained. This function installs a deep handler over the application of `iter`, an iteration method for a collection of elements, to `yield`, a function that performs an

Specification of iteration methods

$$
\begin{array}{l}
\mathit{isIter}\ \texttt{iter} \triangleq \\
\quad \square\forall\, I,\ \Psi,\ f. \\
\qquad \left(\begin{array}{l} \square\forall\, \mathit{us}, u. \\ \quad \mathit{permitted}\,(\mathit{us} +\!\!+ u) \mathrel{-\!\!*} \\ \qquad I\,\mathit{us} \mathrel{-\!\!*} \\ \qquad\quad \mathit{ewp}\ (f\ u)\ \langle\Psi\rangle\{\_.\ I(\mathit{us} +\!\!+ u)\} \end{array} \right) \mathrel{-\!\!*} \\
\qquad \mathit{canTraverse} \mathrel{-\!\!*} \\
\qquad\quad I[] \mathrel{-\!\!*} \\
\qquad\qquad \mathit{ewp}\ (\texttt{iter}\ f)\ \langle\Psi\rangle\{\_.\ \exists\, \mathit{us}.\ I\,\mathit{us} \ *\ \mathit{complete}\,\mathit{us} \ *\ \mathit{canTraverse}\}
\end{array}
$$

Specification of lazy sequences

$$
\begin{array}{l}
\mathit{isSeq}\,(k,\ \mathit{us}) \triangleq \\
\quad \mathit{ewp}\ k\ ()\ \langle\bot\rangle\{h.\ \mathit{isHead}\,(h,\ \mathit{us})\}
\end{array}
\qquad
\begin{array}{l}
\mathit{isHead}\,(h,\ \mathit{us}) \triangleq \\
\quad \mathit{match}\ h\ \mathit{with} \\
\quad \mid \texttt{nil} \Rightarrow \mathit{complete}\,\mathit{us} \ *\ \mathit{canTraverse} \\
\quad \mid \texttt{cons}\,(u, k) \Rightarrow \\
\qquad \mathit{permitted}\,(\mathit{us} +\!\!+ u) \ *\ \triangleright\, \mathit{isSeq}\,(k,\ \mathit{us} +\!\!+ u) \\
\quad \mid \_ \Rightarrow \mathsf{False}
\end{array}
$$

Specification of `hh_invert`

$$
\begin{array}{l}
\square\forall\, \texttt{iter},\ \mathit{permitted}\,,\ \mathit{complete}\,,\ \mathit{canTraverse}. \\
\quad \mathit{isIter}\ \texttt{iter} \mathrel{-\!\!*} \\
\qquad \mathit{canTraverse} \mathrel{-\!\!*} \\
\qquad\quad \mathit{ewp}\ (\texttt{hh\_invert iter})\ \langle\bot\rangle\{k.\ \mathit{isSeq}\,(k,\ [])\}
\end{array}
$$

Figure 3.2: Specification of iteration methods, lazy sequences, and `hh_invert`.

effect as soon as it sees an element of this collection. If the handler intercepts an effect thrown by `yield`, then the handler returns a `cons` head. The first component of this head is the yielded element x, and the second component is the captured continuation k. If the iteration terminates, then the handler returns a `nil` head.

3.2 Specification

The specification of `hh_invert` appears in Figure 3.2. It essentially translates the following sentence to Hazel:

> "`hh_invert` *takes an iteration method* `iter` *as input and produces a lazy sequence* k *as output.*"

The predicates *isIter* and *isSeq* that appear in the specification of `hh_invert` assign meaning to the phrases "`iter` *is an iteration method*" and "k *is a lazy sequence*", respectively.

A *data structure* t stores a certain collection of elements, that is, a (mathematical) list of values xs. If `iter` is the iteration method of such a data structure t, then the application of `iter` to an iteratee f starts a process by which f is applied to each value in xs (the elements of xs are "fed" to f). The intuitive idea behind the predicates *permitted* and *complete*, and the assertion *canTraverse*, is that they allow the expression of the behavior of `iter` without ever mentioning this data structure t, which is thus *implicit*. This idea originates from Filliâtre and Pereira's work [FP16], and is later adapted to the setting of Separation Logic by Pottier [Pot17]. Because the specification of `hh_invert` is polymorphic on these predicates, the function `hh_invert` works on any data structure that provides an iteration method `iter` fitting the interface *isIter* .

The predicate *permitted* holds of lists *us* whose elements *could* have been consumed by `iter`, in the order they appear in this list, in a partial traversal of the data structure. This permissive interpretation of *permitted* allows this predicate to describe nondeterministic traversals, where the order of elements consumed by `iter` is not known in advance. Moreover, the choice of specifying the list of elements consumed, or seen, rather than the list of elements that remain to be consumed, allows *permitted* to describe traversals over infinite data structures. For example, if the structure stores the set of nonnegative integers, then this predicate may state that *us* is a prefix of $\mathbb{Z}_{0\leq}$ (the set of nonnegative integers):

$$\mathit{permitted}_{\text{nonneg}}\, \mathit{us} \triangleq \exists\, n \in \mathbb{Z}_{0\leq}.\ \mathit{us} = [0, 1, \ldots, n]$$

The predicate *complete* holds of lists *us* whose elements could have been consumed by `iter` in a complete traversal of the data structure. For example, if the structure stores a list of values *xs*, then one may leave the order of traversal unspecified by setting this predicate to simply state that the list of consumed elements *us* and *xs* contain the same elements:

$$\mathit{complete}_{\text{set}}\, \mathit{us} \triangleq \forall\, x.\ x \in \mathit{us} \iff x \in \mathit{xs}$$

The assertion *canTraverse* is the permission to traverse the structure. Naturally, it is part of `hh_invert`'s precondition, since this function performs call `iter`. As we shall see, this permission is relinquished only when the sequence produced by `hh_invert` has been *exhausted*; that is, when the sequence produces a `nil` head. The permission to traverse an ephemeral structure is often an ephemeral assertion, so that only one traversal can happen at a time. Persistent data structures, on the other hand, might offer an interface where the assertion *canTraverse* is persistent, thus allowing multiple traversals to occur at the same time.

An example of such a persistent data structure is that of a persistent list l storing the elements *xs*. This structure is specified by the binary predicate $\mathit{isList}\, l\, \mathit{xs}$, which states that the values *xs* are layed out in the heap as a persistent linked list:

$$\begin{aligned} \mathit{isList}\, l\, (x :: \mathit{xs}) &\triangleq \exists\, l'.\ l \mapsto_{\square} \mathtt{inj}_1\, (x, l') \ *\ \mathit{isList}\, l'\, \mathit{xs} \\ \mathit{isList}\, l\, [] &\triangleq l \mapsto_{\square} \mathtt{inj}_2\, () \end{aligned}$$

The *persistent points-to predicate* [VB21] $l \mapsto_{\square} v$ states that l is a read-only location that stores v. It is a persistent assertion, and, consequently, so is the predicate *isList*.

A *HH* implementation of an iteration method for this structure could be written as follows:

```
(* Implementation of an iteration method for lists in HH. *)
let rec list_iter l f =
  match !l with (fun (x, l) -> f x; list_iter l f | fun _ -> ())
```

For pedagogical purposes, let us say that one wishes to prove that `list_iter` l is an iteration method for the structure l in the sense that `list_iter` l satisfies the predicate *isIter*. Then, in this case, one must find particular instances of the predicates and assertions *permitted*, *complete*, and *canTraverse*. The predicate *permitted* could be chosen to say that, in a partial traversal of l, a prefix of xs must have been seen, thus specifying that the order in which `list_iter` traverses l corresponds to the order of the elements in xs. The predicate *complete* could be chosen to say that, upon termination of the iteration, all the elements xs have been seen. Finally, the assertion *canTraverse* could be chosen as the specification that l contains the elements xs. Here is the complete set of definitions:

$$\begin{array}{rcl} \texttt{iter}_{\text{list}} & \triangleq & \texttt{list\_iter}\, l \\ \mathit{permitted}_{\text{list}}\, us & \triangleq & \exists\, vs.\ xs = us +\!\!+ vs \\ \mathit{complete}_{\text{list}}\, us & \triangleq & us = xs \\ \mathit{canTraverse}_{\text{list}} & \triangleq & \mathit{isList}\, l\, xs \end{array}$$

Specification of iteration methods. The predicate *isIter* , which appears in Figure 3.2, says, roughly, that, if the iteratee f can "*take one step*", then `iter` f can "*walk the entire structure*".

The position of `iter` as it walks the structure is captured by the predicate I, which holds of a list us, if the evaluation of `iter` f is in a state where the elements us have been applied to the iteratee f. This predicate is called *the loop invariant*. The meaning of the phrase *"to take one step"* can be captured in terms of this predicate I: it means to perform an update from a state where $I\,us$ holds, for a certain list us, to a state where $I(us +\!\!+ u)$ holds, where us is incremented by one element u. The iteratee f is responsible for performing such updates. This is expressed by f's specification, which appears as a premise of `iter`'s specification in Figure 3.2. Indeed, the assertion $I\,us$ appears as a precondition and the assertion $I(us +\!\!+ u)$ appears as a postcondition of the application $f\ u$, where us and u are universally quantified. Moreover, the assertion *permitted* $(us +\!\!+ u)$ also appears as a precondition of this application: if f has already consumed the elements us, then f can assume that the next element u is one such that *permitted* $(us +\!\!+ u)$ holds.

The phrase *"to walk the entire structre"* means to perform an update from a state where $I\,[]$ holds to a state where $I\,us$ holds and us is the complete list of elements of the structure. This is expressed by the specification of the application `iter` f: the precondition asserts that $I\,[]$ holds, and the postcondition asserts that both $I\,us$ and *complete us* hold. Moreover, the assertion *canTraverse* also appears as a pre- and postcondition of `iter` f: the permission to traverse the structure is necessary to call `iter`, and this permission is recoverable upon termination of `iter`.

There remains one last piece in the specification of `iter`: what are the effects performed by `iter` f? Being able to answer this question is one of the main contributions of Hazel and `iter`'s specification answers it as follows: `iter` f performs the same effects as f; in particular, `iter` f does not intercept f's effects and does not introduce new ones. Higher-order functions that satisfy this property of being oblivious to the effects that an argument function might perform are called *effect-polymorphic*. The way *isIter* expresses

$$\text{(IntroduceViews)} \qquad \mathsf{True} \mathrel{-\!\!*} \dot{\Rrightarrow} \exists \gamma.\ \mathit{iterView}_\gamma\ \mathit{us}\ *\ \mathit{handlerView}_\gamma\ \mathit{us}$$

$$\text{(ConfrontViews)} \qquad \mathit{iterView}_\gamma\ \mathit{us} \mathrel{-\!\!*} \mathit{handlerView}_\gamma\ \mathit{vs} \mathrel{-\!\!*} \mathit{us} = \mathit{vs}$$

$$\text{(UpdateViews)} \qquad \mathit{iterView}_\gamma\ \mathit{us} \mathrel{-\!\!*} \mathit{handlerView}_\gamma\ \mathit{vs} \mathrel{-\!\!*} \dot{\Rrightarrow} \begin{cases} \mathit{iterView}_\gamma\ \mathit{ws}\ * \\ \mathit{handlerView}_\gamma\ \mathit{ws} \end{cases}$$

Figure 3.3: Logical rules governing the assertions *iterView* and *handlerView*.

that `iter` is effect-polymorhphic is by universally quantifying over a protocol Ψ. The specification essentially says that, if Ψ is a correct description of f's effects, then it is also a correct description of the effects performed by the program `iter` f.

Specification of lazy sequences. Lazy sequences are specified by the combinations of predicates *isSeq* and *isHead*, both of which appear in Figure 3.2. The predicate $\mathit{isSeq}\,(k,\ \mathit{us})$ means that the elements *us* have already been produced and that k is a sequence for the elements that remain to be produced. This predicate is defined as a *ewp* assertion saying that the application of k to `()` yields a head h. Since *ewp* is an affine assertion, a sequence is an ephemeral data structure: it cannot be used more than once. This restriction is necessary, because a sequence produced by `hh_invert` is in fact a one-shot continuation. The predicate $\mathit{isHead}\,(h,\ \mathit{us})$ states that h is a value constructed either by `cons` or by `nil`. In the case of a `nil` head, the predicate asserts that the list *us* contains all the elements of the structure – the assertion *complete us* holds – and that the structure can again be traversed – the assertion *canTraverse* is recovered. In the case of a `cons` (u, k) head, the predicate asserts that the path $\mathit{us} +\!\!+ u$ is indeed permitted by the structure – the assertion $\mathit{permitted}\,(\mathit{us} +\!\!+ u)$ holds – and that k is a sequence for the remaining elements. The later modality guarding the recursive occurrence of *isSeq* ensures that this predicate is well-defined.

3.3 Verification

There are three key steps in the proof that `hh_invert` (Figure 3.1) satisfies its specification (Figure 3.2):

1. The choice of the loop invariant I to reason about the application of `iter`.
2. The choice of the protocol by which the function *yield* abides.
3. The application of the reasoning rule for deep handlers (rule TryWithDeep).

Let us postpone the choice of I and shift our attention to the choice of *yield*'s protocol. Because *yield* assumes the role of the iteratee, it must satisfy the following specification:

$$\square \forall\ \mathit{us}, u.\ \mathit{permitted}\,(\mathit{us} +\!\!+ u) \mathrel{-\!\!*} I\,\mathit{us} \mathrel{-\!\!*} \mathit{ewp}\ (\mathit{yield}\ u)\ \langle \Psi \rangle \{ \_.\ I(\mathit{us} +\!\!+ u) \} \tag{3.1}$$

Since *yield* is simply defined as a single `do` instruction, the preceding specification is already an expression of the protocol by which *yield* abides: when performing an effect, *yield* is

in a state where the assertions $\mathit{permitted}\,(\mathit{us} +\!\!+ u)$ and $I\,\mathit{us}$ hold; moreover, when *yield* resumes, it expects the assertion $I(\mathit{us} +\!\!+ u)$ to hold. Therefore, the choice of *yield*'s protocol comes naturally as a restatement of its specification:

$$\Psi \triangleq\ !\,\mathit{us}\;u\;(u)\,\{\mathit{permitted}\,(\mathit{us} +\!\!+ u)\ *\ I\,\mathit{us}\}.\,?\ ((),())\,\{I(\mathit{us} +\!\!+ u)\} \tag{3.2}$$

To show that, under this protocol, the function *yield* satisfies Specification 3.1 is a simple exercise: it suffices to apply Hazel's reasoning rule for performing an effect, rule Do.

The choice of the predicate I comes as part of our solution to the following problem: to express that handler and handlee *see* the same set of elements. This statement relies on the intuitive notion of what handler and handlee *see*. The handlee "`iter` *yield*" sees the elements of the collection that are fed to *yield*. The handler (lines 4 to 6) sees the elements of the collection that come as payload of the effects that it intercepts. To formalize this intuition, we introduce a ghost cell γ whose contents belong to the following camera:

$$\textsc{Auth}(\textsc{Ex}(\mathit{List}\ \mathit{Val}))$$

The idea is then to formalize the set of elements *vs* seen by the handler, the handler's *view*, as the ownership of the authoritative piece $\bullet\,\mathsf{ex}(\mathit{vs})$, and to formalize the set of elements *us* seen by the handlee, the handlee's *view*, as the ownership of the fragment piece $\circ\,\mathsf{ex}(\mathit{us})$. We enforce this abstraction by introducing the predicates *iterView* and *handlerView* defined as follows:

$$\mathit{iterView}_\gamma\ \mathit{us} \triangleq \boxed{\circ\,\mathsf{ex}(\mathit{us})}^{\gamma} \qquad \mathit{handlerView}_\gamma\ \mathit{vs} \triangleq \boxed{\bullet\,\mathsf{ex}(\mathit{vs})}^{\gamma}$$

The crucial property of this choice of camera is that, because elements of the exclusive camera $\textsc{Ex}(\mathit{List}\ \mathit{Val})$ do not split, the fragment piece is unique and must therefore coincide with the authoritative piece; that is, if both the assertions $\mathit{iterView}_\gamma\ \mathit{us}$ and $\mathit{handlerView}_\gamma\ \mathit{vs}$ hold, then $\mathit{us} = \mathit{vs}$. This reasoning formalizes the intuition that handler and handlee's views are in agreement.

This reasoning is captured by rule (ConfrontViews), which appears in Figure 3.3 among other logical rules. Rule (IntroduceViews) can be used to introduce the ghost cell γ at the beginning of the verification of `hh_invert`. Rule (UpdateViews) allows the update of γ's state, thus also allowing the update of handler and handlee's views.

The proof follows by choosing loop invariant I to indicate `iter`'s view:

$$I : \mathit{List}\ \mathit{Val} \to \mathit{iProp} \triangleq \mathit{iterView}_\gamma$$

Specializing `iter`'s specification (Figure 3.2) with this choice of loop invariant and with *yield* as the iteratee leads to the following specification of the application "`iter` *yield*":

$$\begin{array}{l}\mathit{canTraverse} \mathrel{-\!\!*} \\ \quad \mathit{iterView}_\gamma\ [] \mathrel{-\!\!*} \\ \quad\quad \mathit{ewp}\ (\texttt{iter}\ \mathit{yield})\ \langle\Psi\rangle\{\_.\ \exists\,\mathit{us}.\ \mathit{iterView}_\gamma\ \mathit{us}\ *\ \mathit{complete}\ \mathit{us}\ *\ \mathit{canTraverse}\}\end{array}$$

Both the assertions *canTraverse* and $\mathit{iterView}_\gamma\ []$ are transferred by `hh_invert` to `iter`. Therefore, the application `iter` *yield* is safe and abides by the protocol Ψ (Equation 3.2):

$$\mathit{ewp}\ (\texttt{iter}\ \mathit{yield})\ \langle\Psi\rangle\{\_.\ \exists\,\mathit{us}.\ \mathit{iterView}_\gamma\ \mathit{us}\ *\ \mathit{complete}\ \mathit{us}\ *\ \mathit{canTraverse}\} \tag{3.3}$$

The protocol Ψ can thus be seen as the contract established between handler and handlee from line 4. The application of rule TryWithDeep to verify this handler yields two goals: (1) the verification of the handlee, and (2) the derivation of the corresponding handler judgment. The first goal is fulfilled by Specification 3.3. The second goal has the following shape:

$$\begin{array}{l} \mathit{handlerView}_\gamma\ [] \mathrel{-\!\!*} \\ \quad \mathit{deep\text{-}handler}\ \langle\Psi\rangle\ \{\_.\ \exists \mathit{us}.\ \mathit{iterView}_\gamma\ \mathit{us} \ast \mathit{complete}\ \mathit{us} \ast \mathit{canTraverse}\} \\ \qquad\qquad (\texttt{fun x k -> cons (x, k)} \mid \texttt{fun \_ -> nil}) \\ \qquad\qquad \langle\bot\rangle\ \{h.\ \mathit{isHead}\ (h, [])\} \end{array}$$

The assertion $\mathit{handlerView}_\gamma\ []$ is transferred by `hh_invert` to the handler, justifying it to appear as an assumption of the derivation of this goal. The proof follows by Löb's induction. However, before the application of this induction principle, the statement needs to be slightly generalized:

$$\forall\ \mathit{vs}.\quad \begin{array}{l} \mathit{handlerView}_\gamma\ \mathit{vs} \mathrel{-\!\!*} \\ \quad \mathit{deep\text{-}handler}\ \langle\Psi\rangle\ \{\_.\ \exists \mathit{us}.\ \mathit{iterView}_\gamma\ \mathit{us} \ast \mathit{complete}\ \mathit{us} \ast \mathit{canTraverse}\} \\ \qquad\qquad (\texttt{fun x k -> cons (x, k)} \mid \texttt{fun \_ -> nil}) \\ \qquad\qquad \langle\bot\rangle\ \{h.\ \mathit{isHead}\ (h, \mathit{vs})\} \end{array} \tag{3.4}$$

The generalized statement says that, if the elements in the list *vs* have been yielded to the handler, then the handler is able to produce a head h such that $\mathit{isHead}\ (h, \mathit{vs})$ holds. The application of Löb's induction transforms Statement 3.4 into the following goal:

$$\triangleright\,(3.4) \implies (3.4)$$

The handler judgment in the conclusion of Statement 3.4 is the conjunction of two specifications: the specification of the effect branch (line 5) and the specification of the return branch (line 6). These specifications are established separately. The two key steps in the verification of the effect branch are the application of rule (UpdateViews), to update γ with the new element that has been yielded; and the use of the k's specification, which is given by the protocol Ψ. The specification of k has a handler judgment as part of its precondition. To satisfy this constraint, it suffices to apply the copy of Statement 3.4 introduced by Löb's induction. The key step in the verification of the return branch is the application of rule (ConfrontViews), to show that, upon termination of `iter`, all the elements have been yielded to the handler. This concludes the proof of `hh_invert`.

3.4 Related work

Filliâtre and Pereira [FP16] introduce the predicates *enumerated* and *completed* as the main ingredients of a uniform approach to specify the traversal of data structures. Let us introduce the term *iteration-predicate approach* to identify Filliâtre and Pereira's methodology.

Initially, the authors wish to apply the iteration-predicate approach to *cursors* [Cop92]. A cursor (or an *iterator* [LG01]) in Java [GJSB00] or C++ [Str95] is an object that performs the traversal of a given collection. One may accept this suggestive name and think of a

cursor c as hovering over an element of a structure. To manipulate a cursor, one has access to a function named `next`. The action of the operation `next` c is twofold: (1) it returns the element over which c hovers and (2) it advances c to the next element (if there is one). Filliâtre and Pereira's idea is then to introduce *enumerated* and *completed* as predicates on cursors to specify the state of a cursor. The assertion "*enumerated* c" expresses the conditions under which the cursor c is in a valid state, whereas the assertion "*completed* c" expresses the conditions under which the cursor c completes the traversal of the structure. These definitions vary according to the structure being traversed. Filliâtre and Pereira show many instances of such definitions. Later, the authors apply this approach to the specification of higher-order iteration methods. The straightforward way to specify an iteration method in a higher-order program logic is to quantify over the specification of the iteratee, as we do in this chapter (Figure 3.2). However, the authors study the iteration-predicate approach in the setting of Why3 [FP13], a verification tool that offers an interface for writing specifications in first-order logic. To circumvent this limitation of first-order logic, the authors propose an inventive solution in which an application of an iteration method `iter` is translated to a first-order program using a "fictional" cursor. Moreover, to verify the implementation of such an iteration method `iter`, their solution asks the verification of a specialized version of `iter`, where the iteratee is a function that adds the consumed element to a memory cell of visited elements.

The iteration-predicate approach is further studied by Pottier [Pot17], who verifies the implementation of a hash table using CFML [Cha11, Cha22]. CFML consists of a higher-order Separation Logic embedded in Coq and of a tool that links OCaml implementations to this theory by translating a program to its *characteristic formula*, a logical formula describing the semantics of this program. In the setting of a higher-order Separation Logic, Pottier applies the iteration-predicate approach to write natural specifications of higher-order iteration methods. Moreover, he introduces the specification of lazy sequences, which he calls *cascades*. The specifications of iteration methods and lazy sequences that appear in Figure 3.2 of this chapter adapt Pottier's specifications to Hazel. There are three main differences. The first one is the inclusion of protocols. Another difference is that we can avoid the existential quantification used by Pottier to define the specification of lazy sequences. He employs this existential quantification to define the corresponding version of *isSeq* as a co-inductive predicate. We can avoid this trick by exploiting Iris's native support for guarded recursion. Finally, Pottier's version of *isSeq* is a duplicable predicate, while ours is ephemeral. This aspect of our specification is crucial to enforce that a captured one-shot continuation is not invoked twice.

Chapter 4

Asynchronous Computation

Concurrency, the simultaneous execution of multiple processes, is a pervasive concept in Computer Science. It appears in applications such as network systems, where a program opens connections with multiple parties in the network, or in operating systems, where a program communicates with the outside world by writing and reading files. *Asynchronous computation* is one approach to complete such concurrent tasks. It consists of a sequential program, the *scheduler*, which orchestrates the execution of tasks. Many programming languages, such as JavaScript [MMT08, GSK10, Ecm22] and C# [Cor22], add support for asynchronous computation by incorporating new programming constructs, thus adding more effort for understanding and maintaining the language. With effect handlers, on the other hand, it is possible to implement asynchronous computation as a library. Dolan et al. [DEH+17], for instance, use effect handlers to implement an asynchronous-computation library in Multicore OCaml. In this chapter, we study a simplified version of this library. The exercise of specifying and verifying this library is yet another demonstration of the applicability of Hazel. The verification of this library is presented in a published paper [dVP21].

4.1 Implementation

Before we delve into the details of the *HH* implementation of the asynchronous-computation library that appears in Figure 4.1, let us establish the informal contract by which a user can exploit this library.

The informal contract relies on two abstractions: *fibers* and *promises*. A fiber is a programming concept related to the concepts of *lightweight threads*, *user threads*, or *green threads*. In short, fibers are processes that run *asynchronously*: at most one fiber runs at a time. Moreover, every fiber is associated to a unique object called a *promise*. A promise serves as a place holder for the result of the fiber. When a fiber is *spawned* (*i.e.*, when it begins its execution), its associated promise p is created. Initially, the promise does not store any value, and it is so characterized as being in an *unfulfilled* state. The promise upholds this status until its corresponding fiber terminates. When the fiber terminates, producing a result value y, the promise shall store this value. The promise is then characterized as being in a *fulfilled* state.

The library offers two operations for manipulating fibers and promises. The instruction `async` e (1) spawns a new fiber that executes the application e `()`, and (2) returns the promise associated to this fiber. The instruction `await` p checks the state of the promise p and then proceeds in one of two manners. If the promise is fulfilled, then the operation immediately returns with the value stored in p. If the promise is unfulfilled, then the operation blocks until p becomes fulfilled. Finally, the library offers a function `run`, the

```
let async e = do (Async e)
let await p = do (Await p)
let run main =
  let q = create_queue() in
  let next() =
    if not (is_empty q) then
      let k = take q in k()
  in
  let rec fulfill p e =
    deep-try e() with
    (* Effect branch. *)
    ( fun request k ->
        match request with
        ( fun (* Async. *) e' ->
            let p' = ref (Waiting []) in
            add (fun _ -> k p') q;
            fulfill p' e'
        | fun (* Await. *) p ->
            match !p with
            ( fun (* Waiting. *) ks ->
                p := Waiting (k :: ks);
                next()
            | fun (* Done. *) y ->
                k y
            )
        )
    (* Return branch. *)
    | fun y ->
        match !p with
        ( fun (* Waiting. *) ks ->
            p := Done y;
            list_iter ks (fun k -> add (fun _ -> k y) q);
            next()
        | fun (* Done. *) _ ->
            () () (* Unreachable! *)
        )
    )
  in
  fulfill (ref (Waiting [])) main
```

Figure 4.1: Asynchronous-computation library in *HH*.

scheduler, which orchestrates the fibers spawned during the execution of a program *main* that is passed as an argument.

Now, we wish to explain the operational behavior of an arbitrary execution of `run`. Even though the dynamic behavior of `run` is quite complex – the implementation of `run` exploits, for instance, effect handlers, dynamically allocated mutable state, and higher-order functions – it has a strikingly similarity to the staging of an exquisite theatrical play, *The Fibers' Odyssey*. So let us tell what we recall from this play in the hope that it makes the implementation of the library clear.

The *stage* of this play was divided into three sections: a first one occupying the left portion of the stage, a second one occupying the center, and a third one occupying the right portion of the stage. In the left portion, there was a *queue* of *actors*, where every actor held a *torch*. In the right portion, there were multiple *torch stands*. Every actor had the goal of placing his torch on a torch stand uniquely identified by his name. To achieve this goal, an actor in the queue must traverse the center portion of the stage. However, at most one actor could occupy this portion of the stage at a time. The *director* of the play was responsible for maintaining the order. He would allow an actor from the queue to occupy the center of the stage only when this portion was free.

During an actor's traversal to his stand, it was often the case that his torch would lose its flames. In such cases, the actor had the right to perform the following move: he would (1) memorize his position in the stage, (2) skip his way to the right portion, (3) borrow the flames from a torch in one of the stands, (4) return to the position where he left, and (5) resume his traversal. To play this move, however, the actor had to choose in advance the stand to which he intended to go. The problem is that stands were hidden in such a way that it was impossible for an actor to know whether or not the stand was already fulfilled with its promised torch. Therefore, if the actor decided to play this move, then he would have to try his chance. If he was lucky, then the actor would resume the traversal with his lightened torch. If he was unlucky, then he would have to keep waiting in the stand for the actor responsible for fulfilling it. There, he might find other waiting actors who were struck by the same lack of luck. When the actor responsible for this stand finally arrived with the promised torch, all the waiting actors were freed. They would then go to the end of the queue to wait for another chance to traverse the stage.

Another source of misfortune during an actor's traversal to his stand was exhaustion. In such cases, the actor could promote someone in the audience to become an actor. The old tired actor would then go to the end of the queue, while the director would forge a new torch and a new stand to the new actor, who would immediately occupy the stage and start his traversal to the stand.

The play ended when there were no more actors waiting in the queue.

As it might already be clear, actors represent fibers, the director represents the scheduler, a torch represents the result of a fiber, and a stand represents a promise. Like an actor, a fiber must complete a task. During the completion of the task, a fiber can either spawn a new fiber – "*to promote someone in the audience to become an actor*" – or it can wait for the completion of another fiber – "*to wait for another actor's torch*".

The implementation of the operations `await`, `async`, and `run` appears in Figure 4.1. It assumes the existence of operations for manipulating queues (`create_queue`, `add`, `take`, and `is_empty`) and operations for manipulating lists (`[]`, `_ :: _`, and `list_iter`). The

assumed formal specifications of both sets of operations appear in Figures 4.4 and 4.5. We discuss these specifications further in Section 4.2.

Informally, the instruction `create_queue()` allocates a new queue and returns its address q; the instruction `add` v q adds the element v to the queue q; the instruction `take` q removes and returns an element of the queue q, if the queue is nonempty; finally, the instruction `is_empty` q returns a Boolean indicating whether q is empty.

The sets of constructors `Waiting`/`Done` and `Async`/`Await` that appear in Figure 4.1 are encoded in *HH* using binary sums:

$$\begin{aligned} \texttt{Async}\, e &\triangleq \texttt{inj}_1\, e \\ \texttt{Await}\, p &\triangleq \texttt{inj}_2\, p \\ \texttt{Waiting}\, ks &\triangleq \texttt{inj}_1\, ks \\ \texttt{Done}\, y &\triangleq \texttt{inj}_2\, y \end{aligned}$$

The function `run`, the scheduler, executes in four steps. The first step is to allocate a queue (line 4) to store *ready* fibers, fibers that can be resumed. (In the play, this queue represented by the queue of actors.) The second step is to define the function `next` (line 5), which runs a ready fiber taken from the queue. The third step is to define the function `fulfill` (line 9), which runs a fiber e to fulfill its promise p. (This operation represented by the traversal of an actor towards his stand.) A promise p is a memory location storing either the value `Waiting` ks, where ks is a list of waiting fibers; or the value `Done` y, where y is the result of its corresponding fiber. The fourth and final step (line 39) is to call `fulfill` with the main fiber *main*.

Now, let us discuss the implementation of the function `fulfill` (line 9). The execution of the instruction `fulfill` e p begins with the application e`()` (line 10). The scheduler monitors this application by installing a handler. If the fiber e terminates, then it must be the case that p stores a list ks of waiting fibers. (Exploiting the play analogy, this argument follows from the fact that torches are unique. If the promise was already fulfilled, then it would mean that an actor finds his own stand already carrying a torch.) The fiber writes its output y to p (line 31), then the list of waiting fibers is sent to the queue (line 32), and finally the scheduler runs a fiber taken from the queue (line 33).

If, during the execution of e`()`, the instruction `async` e is performed, then the fiber is paused and reified as the continuation k. The scheduler then add this continuation to the queue (line 16), and then sets a new fiber e (pay attention to the shadowing of the name e) to run with a fresh promise (line 17).

If, during the execution of e`()`, the instruction `await` p is performed, then the scheduler checks the state of the promise p. If p is fulfilled with a value y, then the fiber is immediately resumed with this value (line 24). If p is unfulfilled, then scheduler adds k, the paused fiber, to the list of waiting fibers (line 21), and runs a fiber taken from the queue (line 22).

Deadlock. The function `run` terminates when there are no more fibers in the queue. However, there might be unfulfilled promises and there might be fibers waiting for these unfulfilled promises. This situation is called a *deadlock*. It typically arises when there is a *dependency cycle among fibers*; that is, a positive integer n and a set of fibers $\{f_i\}_{0 \leq i < n}$, such that, for every i, the fiber f_i waits for the completion of the fiber f_j, where $j = i + 1 \ (mod\ n)$. The following program induces such a behavior:

```
1  (* Example of deadlock. *)
2  let yield() = async (fun _ -> ())
3  let main() =
4    let r = ref (inj1 ()) in     (* Create channel.                *)
5    let rec f() =
6      match !r with
7      ( fun _ -> yield(); f() (* Yield control to [main].        *)
8      | fun p -> await p      (* Wait for one's own completion. *)
9      )
10   in
11   let p = async f in          (* Obtain [f]'s promise.          *)
12   r := inj2 p                 (* Send [p] through the channel.  *)
13 let _ = run main
```

Figure 4.2: Example of a client that induces a deadlock.

The fiber *main* spawns a fiber f that waits its own completion, thus creating a singleton cycle of waiting fibers. The idea is to use the store as a channel through which *main* communicates f's promise. The fiber f listens to this channel, and obtains its own promise.

In line 4, *main* allocates a reference r, initially holding a dummy value $\texttt{inj}_1\ ()$. Then, in line 11, it spawns the fiber f, thereby obtaining f's promise p. When f is spawned, the execution of *main* is suspended and control is relinquished to f (the single fiber in the scheduler's queue).

The fiber f implements a loop that checks whether r already contains its own promise p. When it first starts its execution, the reference r contains the value $\texttt{inj}_1\ ()$. Therefore, the pattern matching in line 6 causes the execution of its first branch. This branch starts with the execution of the instruction `yield`, which has the effect of (1) pausing the execution of f, (2) sending the suspended fiber f to the queue, and (3) yielding control to *main*. [1]

The fiber *main* resumes its execution from line 12, where it writes f's promise p to r. When *main* terminates, the scheduler takes f from the queue and resumes the execution of this fiber. This time, when f checks the state of r, it finds its own promise p. Therefore, the pattern matching in line 6 now causes the execution of its second branch. This branch contains the instruction `await` p, which causes f to wait for its own completion.

4.2 Specification

The formal specification of the library appears in Figure 4.3. It depends on a predicate *isPromise* and a protocol *Coop*, both of which are *abstract notions* of the library's interface: a user can ignore their definition.

The assertion $\mathit{isPromise}\, p\, \Phi$ states that, if the promise p is fulfilled, then p holds a value that satisfies the predicate Φ. The predicate *isPromise* is persistent, therefore the library allows promises to be duplicated and shared among fibers.

[1] The instruction `yield` is implemented in line 2. The meaning that we assigned to `yield` depends on the assumption that the scheduler's queue organizes elements in a first-in-first-out manner (FIFO).

$$\mathit{isPromise} : \mathit{Val} \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp} \qquad \mathit{Coop} : \mathit{Protocol}$$

$$\mathsf{persistent}(\mathit{isPromise}\, p\, \Phi)$$

$$\textsc{Async}\ \frac{\mathit{ewp}\ e\,()\ \langle \mathit{Coop}\rangle\{y.\ \Box\Phi(y)\}}{\mathit{ewp}\ (\texttt{async}\ e)\ \langle \mathit{Coop}\rangle\{p.\ \mathit{isPromise}\, p\, \Phi\}} \qquad \textsc{Await}\ \frac{\mathit{isPromise}\, p\, \Phi}{\mathit{ewp}\ (\texttt{await}\ p)\ \langle \mathit{Coop}\rangle\{y.\ \Box\Phi(y)\}}$$

$$\textsc{Run}\ \frac{\mathit{ewp}\ \mathit{main}\,()\ \langle \mathit{Coop}\rangle\{\_.\ \mathsf{True}\}}{\mathit{ewp}\ (\texttt{run}\ \mathit{main})\ \langle \bot\rangle\{\_.\ \mathsf{True}\}}$$

Figure 4.3: Specification of the asynchronous-compuation library.

$$\mathit{isList} : \mathit{Val} \to \mathit{List}\ \mathit{Val} \to \mathit{iProp}$$

$$\textsc{Nil}\ \mathit{isList}\ \texttt{[]}\ [] \qquad \textsc{Cons}\ \frac{\mathit{isList}\ l\ \mathit{vs}}{\mathit{ewp}\ (v :: l)\ \langle \bot\rangle\{l'.\ \mathit{isList}\, l'\, (v :: \mathit{vs})\}}$$

$$\textsc{ListIter}\ \frac{\mathit{isList}\ l\ \mathit{vs} \qquad I[] \qquad \Box\forall\ \mathit{us},\ u.\ I(\mathit{us}) \mathrel{-\!\!*} \mathit{ewp}\ (f\ u)\ \langle \Psi\rangle\{\_.\ I(\mathit{us} +\!\!+ u)\}}{\mathit{ewp}\ (\texttt{list\_iter}\ l\ f)\ \langle \Psi\rangle\{\_.\ \mathit{isList}\, l\, \mathit{vs}\ *\ I(\mathit{vs})\}}$$

Figure 4.4: Specification of a list library.

Specification Async states that performing the operation `async` e yields a promise p, such that $\mathit{isPromise}\, p\, \Phi$ holds. The predicate Φ is the postcondition of e. Because promises are duplicable, the library requires this predicate to describe only duplicable resources. This requirement is expressed by the persistently modality that guards e's postcondition.

Specification Await states that, if the assertion $\mathit{isPromise}\, p\, \Phi$ holds, then the operation `await` p returns a value y such that the assertion $\Box\Phi(y)$ holds.

Specification Run states that `run` correctly implements the operations of the library: (1) the protocol *Coop* specifying the application $\mathit{main}()$ means that the functionalities `async` and `await` become available to *main*, and (2) the protocol $\bot$ specifying the application application `run` *main* means that the effects performed by *main* are handled by `run`.

The correctness statement of the library is stated as follows:

Statement 4.1 (Correctness of the library) *There exists a predicate isPromise and a protocol Coop, such that isPromise is persistent and the operations* `async`*,* `await`*, and* `run` *satisfy their formal specifications (Figure 4.3).*

We present the proof of this statement in Section 4.3. The proof assumes that the operations for manipulating lists satisfy the specifications from Figure 4.5 and that the

$$isQueue : Val \to (Val \to iProp) \to Bool \to iProp$$

CreateQueue
$$ewp\ (\texttt{create\_queue}\ ())\ \langle\bot\rangle\{q.\ \forall\, I.\ isQueue\ q\ I\ \texttt{true}\}$$

IsEmpty
$$\frac{isQueue\ q\ I\ \_}{ewp\ (\texttt{is\_empty}\ q)\ \langle\bot\rangle\{b.\ isQueue\ q\ I\ b\}}$$

Take
$$\frac{isQueue\ q\ I\ \texttt{false}}{ewp\ (\texttt{take}\ q)\ \langle\bot\rangle\{v.\ I\ v\ *\ isQueue\ q\ I\ \texttt{true}\}}$$

Add
$$\frac{isQueue\ q\ I\ \_ \qquad I\ v}{ewp\ (\texttt{add}\ q\ v)\ \langle\bot\rangle\{\_.\ isQueue\ q\ I\ \texttt{false}\}}$$

Figure 4.5: Specification of a queue library.

operations for manipulating queues satisfy the specifications from Figure 4.4. Let us briefly discuss each of these logical interfaces.

The interface for manipulating lists (Figure 4.4) depends on an abstract *representation predicate* *isList*, a predicate relating a value l to a mathematical list of values vs. The specifications of the list constructors, `[]` and `_ :: _`, are standard. The specification of the list iteration method, `list_iter`, can be seen as a specialized case of the predicate *isIter*, discussed in Chapter 3. Indeed, given a value l and a list vs, Specification ListIter corresponds to the assertion *isIter* (`list_iter` l) where the iteration predicates are defined as follows:

$$\begin{array}{rcl} permitted\ \_ & \triangleq & \mathsf{True} \\ complete\ us & \triangleq & us = vs \\ canTraverse & \triangleq & isList\ l\ vs \end{array}$$

The intuitive reading of rule ListIter is that, if f can "process" single arbitrary elements of the list, then the application `list_iter` l f can "process" the entire list vs.

The interface for manipulating queues (Figure 4.5) depends on an abstract representation predicate *isQueue*, which relates a value q to a predicate I and a Boolean b. The predicate I is the *queue invariant*, it describes a property that holds of every element in the queue. The Boolean b is a partial answer to the question: is the queue empty? If b is `false`, then the queue is nonempty. However, if b is `true`, then nothing can be said.

Rule CreateQueue states that the instruction `create_queue()` returns a new queue q such that the assertion $\forall\, I.\ isQueue\ q\ I\ \texttt{true}$ holds. This assertion lets one choose the queue invariant after the creation of the queue, thus allowing I to depend on q. We need this flexibility in the verification of the library, because a fiber in a queue q may modify q once it resumes its execution. Therefore, the description of fibers in the queue depend on the queue itself. A similar situation arises in Timany and Birkedal's work [TB19, Section 5.1].

Rule IsEmpty allows one to reify the Boolean argument of a *isQueue* assertion.

$$\text{(ForgeTorch)} \qquad \mathsf{True} \mathrel{-\!\!*} \dot{\Rightarrow} \exists \gamma.\ \mathit{torch}_\gamma$$

$$\text{(ClaimUniqueness)} \quad \mathit{torch}_\gamma \mathrel{-\!\!*} \mathit{torch}_\gamma \mathrel{-\!\!*} \mathsf{False}$$

Figure 4.6: Logical rules governing the assertion *torch*.

$$\text{(IntroduceMap)} \qquad \mathsf{True} \mathrel{-\!\!*} \dot{\Rightarrow} \exists \delta.\ \mathit{promiseMap}\ \varnothing$$

$$\text{(UpdateMap)} \qquad \mathit{promiseMap}\ M \mathrel{-\!\!*} \dot{\Rightarrow} \begin{cases} \mathit{promiseMap}\ (M \sqcup \{(p,\gamma) \mapsto \Phi\}) \ * \\ \mathit{isMember}\ p\ \gamma\ \Phi \end{cases}$$

$$\text{(ClaimMembership)} \qquad \begin{array}{r} \mathit{promiseMap}\ M \mathrel{-\!\!*} \\ \mathit{isMember}\ p\ \gamma\ \Phi \mathrel{-\!\!*} \end{array} \quad \exists\, \Phi'.\ \begin{cases} \{(p,\gamma) \mapsto \Phi'\} \in M \ * \\ \triangleright \forall\, y.\ \Phi(y) =_{\mathit{iProp}} \Phi'(y) \end{cases}$$

Figure 4.7: Logical rules governing the assertions *promiseMap* and *isMember*.

Rule Take states that, if the queue q is nonempty, then the instruction `take` q returns an element v that satisfies the queue invariant. After taking one element, the information that the queue is nonempty is lost.

Rule Add states that a value v can be added to the queue if v satisfies the queue invariant. After adding an element, one learns that the queue is nonempty.

4.3 Verification

If we assume that the predicate *isPromise* is given, then the definition of the protocol *Coop* and the verification of the operations `async` and `await` are trivial. Indeed, the definition of *Coop* in terms of the predicate *isPromise* appears in Figure 4.8. It is defined as the sum of the protocols *Async* and *Await*, both of which simply rephrase the specifications Async and Await as send-receive protocols. The only difference between the protocol *Async* and the specificatioon Async is the inclusion of a later modality over the occurrence of *ewp*. This modality is used to ensure that the recursive protocol *Coop* is well-defined. It poses no problem to the verification of `async`.

The core of the verification is thus the definition of a persistent predicate *isPromise* and the proof of `run`. The key idea for both the definition of this predicate and the proof of `run` is the introduction of a ghost cell δ holding the set of promises allocated during the execution of `run`. We conceive the camera to which the contents of this cell belong in such a way that the state of the cell is *monotonic*: it only grows over time. From this monotonicity property, it follows that claiming membership to the set of promises held by this ghost cell is a persistent assertion: once a new promise has been added to the set, it cannot be removed, therefore, once a claim of membership holds, it holds always. Therefore, we define the predicate *isPromise* as the claim of membership to this set.

The ghost variable δ intuitively holding the set of promises allocated during the execution of `run` is formalized as holding an element of the following authoritative camera:

$$\textsc{Auth}(P) \quad \text{where} \quad P \triangleq (\mathit{Loc} \times \mathit{GName}) \xrightarrow{\text{fin}} \textsc{Ag}(\blacktriangleright(\mathit{Val} \to \mathit{iProp})) \tag{4.1}$$

Definition of the protocol Coop.

$$
\begin{array}{rcl}
\mathit{Coop} & \triangleq & \mathit{Async} + \mathit{Await} \\
\mathit{Async} & \triangleq & !\, e\, \Phi\, (\texttt{Async}\ e)\, \{\triangleright\, \mathit{ewp}\ (e\ ())\ \langle \mathit{Coop} \rangle \{y.\ \square \Phi(y)\}\}.\, ?\, p\, (p)\, \{\mathit{isPromise}\, p\, \Phi\} \\
\mathit{Await} & \triangleq & !\, p\, \Phi\, (\texttt{Await}\ p)\, \{\mathit{isPromise}\, p\, \Phi\}.\, ?\, y\, (y)\, \{\square \Phi(y)\}
\end{array}
$$

Definition of the predicate ready.

$$
\mathit{ready}\ q\, \Phi\, k \ \triangleq\ \forall\, y.\ \left\{ \begin{array}{l} \square \Phi(y) \mathrel{-\!\!*} \\ \quad \triangleright\, \mathit{promiseInv}\ q \mathrel{-\!\!*} \\ \quad\quad \triangleright\, \mathit{isQueue}\ q\ (\mathit{ready}\ q\ (\lambda\, y.\ y = ())\ )\ \_ \mathrel{-\!\!*} \\ \quad\quad\quad \mathit{ewp}\ (k\ y)\ \langle \bot \rangle \{\_.\ \mathsf{True}\} \end{array} \right.
$$

Definition of the predicate promiseInv.

$$
\mathit{promiseInv}\ q \ \triangleq\ \exists\, M.\ \mathit{promiseMap}\ M\ *\ \ast_{\{(p,\gamma)\mapsto\Phi\}\in M}.\ \vee \left\{ \begin{array}{ll} \exists\, y. & \left(\begin{array}{l} p \mapsto \texttt{Done}\ y\ * \\ \square \Phi(y)\ * \\ \mathit{torch}_\gamma \end{array} \right) \\ \exists\, l, ks. & \left(\begin{array}{l} p \mapsto \texttt{Waiting}\ l\ * \\ \mathit{isList}\ l\ ks\ * \\ \ast_{k\in ks}.\ \mathit{ready}\ q\, \Phi\, k \end{array} \right) \end{array} \right.
$$

Figure 4.8: Definitions used in the verification of `run`.

An element of P is a finite map from pairs of promises and ghost variables to predicates. Therefore, the variable δ can in fact be seen as holding a set of promises where each promise p is associated to a ghost name γ and a predicate Φ. The name γ is the unique token associated to the fiber that must fulfill p, and the predicate Φ is the fiber's postcondition. We introduce two assertions for describing the state of δ:

$$
\begin{array}{rcl}
\mathit{promiseMap}\ M & \triangleq & \boxed{\bullet\, M}^{\delta} \\
\mathit{isMember}\ p\, \gamma\, \Phi & \triangleq & \boxed{\circ\, \{(p,\gamma) \mapsto \Phi\}}^{\delta}
\end{array}
$$

The assertion $\mathit{promiseMap}\ M$ claims that the authoritative piece of δ is the finite map M. This assertion is a non-duplicable. The assertion $\mathit{isMember}\ p\, \gamma\, \Phi$ claims that the entry $\{(p,\gamma) \mapsto \Phi\}$ belongs to the authoritative piece of δ. This assertion is persistent.

The logical rules induced by these definitions appear in Figure 4.7. Rule (IntroduceMap) allows the introduction of the variable δ at the beginning of the execution of `run`. Naturally, the initial state of this variable is the empty map $\varnothing$. Rule (UpdateMap) lets one update the contents of δ with a new entry $\{(p,\gamma) \mapsto \Phi\}$, provided that the key pair (p,γ) does not belong to M. [2] The application of this rule not only updates δ but it also yields a *receipt* that the new entry has been effectively added to the map. Indeed, the conclusion of the rule states that the assertion $\mathit{isMember}\ p\, \gamma\, \Phi$ holds. Rule (ClaimMembership) expresses

[2] The infix operation $\_ \sqcup \_$ denotes disjoint map union.

the idea that, because the map M stored in δ only grows over time, an added entry cannot be removed from M. Indeed, if one has the receipt $\mathit{isMember}\, p\, \gamma\, \Phi$, then the pair (p, γ) is a key of M. Moreover, the predicate Φ' bound by (p, γ) in M is *almost* the predicate Φ specified by the assertion *isMember*. The assertion $\triangleright \forall\, y.\ \Phi(y) =_{\mathit{iProp}} \Phi'(y)$ means that Φ and Φ' may differ at the current step of execution, but become indistinguishable in the next step and afterwards. [3] This restriction is a consequence of the higher-order nature of the camera P (Eq. 4.1), which depends on the type of logical assertions *iProp*. Naively supporting such higher-order cameras leads to a *circularity* paradox [JKJ+18, Section 4.1]. Therefore, Iris imposes restrictions to support higher-order cameras in a sound manner. An approach to comply to Iris's restrictions is to guard occurrences of *iProp* in the definition of a camera by the *type-level later constructor* $\blacktriangleright : \mathbf{OFE} \to \mathbf{OFE}$. [4] An equality assertion between elements in the codomain of $\blacktriangleright$ is only usable after one step of execution. For this reason, the equivalence between Φ and Φ' holds only after one step of execution.

There is one more flavor of ghost state that comes up in the proof. Its purpose is to formalize the argument that the blocking operation from line 35 can never occur, because, when a fiber terminates, it cannot find its own promise already fulfilled. The key to formalize this reasoning is to introduce a logical assertion torch_γ, claiming ownership over the unique piece of a ghost variable γ. The variable γ can thus be seen as a unique token. The assertion torch_γ is forged with a fresh token γ immediately before a fiber starts running. The fiber holds possession of the assertion torch_γ until the completion of its task, then it relinquishes ownership of torch_γ to its promise.

If every fiber conforms to this discipline, then we can prove, by contradiction, that the blocking operation from line 35 does never occur. Indeed, assume that a fiber finds its promise p already fulfilled. This means that p owns torch_γ. However, the fiber itself owns the assertion torch_γ. This constitutes a contradiction, because the assertion torch_γ cannot be duplicated.

The definition of torch_γ is straightforward. We let the contents of γ range over an exclusive camera, such as

$$\textsc{Ex}(\{\bullet\}),$$

and set the assertion torch_γ to claim ownership over a piece of γ:

$$\mathit{torch}_\gamma \triangleq \boxed{\mathsf{ex}(\bullet)}^{\gamma}$$

Recall that the elements of an exclusive camera do not split, therefore γ has a unique piece, and claiming ownership over this piece suffices to ensure that torch_γ is non-duplicable. This property is formally expressed by rule (ClaimUniqueness) from Figure 4.6. The other rule from this figure, rule (ForgeTorch), states that one can always pick a fresh variable γ.

With the introduction of δ and its related assertions *promiseMap* and *isMember*, we are in position to define the predicate *isPromise* and to introduce the key invariant used

[3] The assertion $P =_{\mathit{iProp}} Q$ holds of a step-index n, if the propositions P and Q are equivalent for every step-index less than or equal to n. The definition of $\_ =_{\mathit{iProp}} \_$ follows from its interpretation given in [Tea22, Section 6] and from the definition of a step-indexed equivalence on *UPred* given in [Tea22, Section 3.3].

[4] The type **OFE** denotes the type of *ordered families of equivalences* [JKJ+18, Section 4.2]. An ordered family of equivalences is a set equipped with a step-indexed equivalence relation.

in the verification of `run`. The predicate *isPromise* is defined as follows:

$$\mathit{isPromise}\, p\, \Phi \triangleq \exists\, \gamma.\ \mathit{isMember}\, p\, \gamma\, \Phi$$

This definition captures the intuitive reading of $\mathit{isPromise}\, p\, \Phi$: that some fiber must fulfill the promise p with a value that satisfies Φ.

The proof invariant in the verification of `run` is an assertion that links the logical map M to the contents of the physical addresses of promises. The specification of a fiber depends on this assertion, because, upon termination, a fiber must have permission to write its result to its promise. In particular, a fiber in the queue relies on the proof invariant. For this reason, the proof invariant *promiseInv* q is an assertion that depends on the queue q created in line 4, and its formal definition depends on the predicate describing fibers in the queue, the queue invariant. The queue invariant is defined as a special case of the assertion $\mathit{ready}\, q\, \Phi\, k$, which states that the fiber k can resume when supplied with a value satisfying Φ. Fibers in the queue are ready to resume when applied to `()`. Therefore, the predicate $\mathit{ready}\, q\, (\lambda\, y.\ y = \texttt{()})$ must hold of every fiber in the queue. This predicate is thus the queue invariant, which we denote by I for short:

$$I \triangleq \mathit{ready}\, q\, (\lambda\, y.\ y = \texttt{()}) \tag{4.2}$$

The formal definitions of *promiseInv* and *ready* appear in Figure 4.8. The assertion *promiseInv* q states that every entry in the logical map M corresponds to a spawned fiber. Indeed, for every entry $\{(p, \gamma) \mapsto \Phi\}$ in M, the promise p is a valid memory location in one of two states. It can be either (1) fulfilled, in which case it owns the assertion torch_γ and contains a value that satisfies Φ; or (2) unfulfilled, in which case it contains a list of waiting fibers ready to resume with a value that satisfies Φ.

The definition of $\mathit{ready}\, q\, \Phi\, k$ captures the informal reading that k can be resumed when applied to a value that satisfies Φ. It also states that k performs no effects and that k relies on the permission to modify the queue and on the permission to modify the contents of promises. The permission to modify the queue is given by the assertion $\mathit{isQueue}\, q\, I\, \_$, and the permission to modify promises is given by the invariant *promiseInv* q. Because both assertions include references to *ready*, this predicate is recursively defined, and, consequently, both of these assertions must be guarded by a later modality.

Now that all the logical definitions are laied out, let us discuss the proof that `run` satisfies the specification Run (Figure 4.3). At the beginning of the execution of `run`, the queue is allocated, and the assertion $\mathit{isQueue}\, q\, I\, \_$ holds (recall the definition of the queue invariant I given in Equation 4.2). Rule (IntroduceMap) then introduces the assertion $\mathit{promiseMap}\, \varnothing$, which trivially entails the invariant *promiseInv* q. (It suffices to instantiate the existentially quantified map M with $\varnothing$.)

The next step is the definition of the function `next` (line 5), whose specification is given by the following lemma:

Lemma 4.1 (Specification of `next`) *The function* `next` *admits the following specification:*

$$\mathit{promiseInv}\, q \mathrel{-\!\!*} \mathit{isQueue}\, q\, I\, \_ \mathrel{-\!\!*} \mathit{ewp}\ \texttt{next()}\ \langle\bot\rangle\{\_.\ \mathsf{True}\}$$

The assertion *isQueue* is needed because `next` tries to remove a fiber from the queue. The invariant *promiseInv* is needed to feed the eventual fiber that comes out from q.

This lemma follows easily from the specification of the fibers in q given by the queue invariant (Eq. 4.2).

The definition of the function `fulfill` comes next. Specifying and verifying this function is the essence of the proof. The specification of `fulfill` is given by the following lemma:

Lemma 4.2 (Specification of `fulfill`) *The function* `fulfill` *admits the following specification:*

$$\forall\, e,\, p,\, \gamma,\, \Phi.\; \left\{ \begin{array}{l} \mathit{promiseInv}\; q \mathrel{-\!\!*} \\ \quad \mathit{torch}_\gamma \mathrel{-\!\!*} \\ \quad\quad \mathit{isMember}\; p\, \gamma\, \Phi \mathrel{-\!\!*} \\ \quad\quad\quad \mathit{isQueue}\; q\, I\; \_ \mathrel{-\!\!*} \\ \quad\quad\quad\quad \mathit{ewp}\; (e\; ())\; \langle \mathit{Coop} \rangle \{y.\; \square \Phi(y)\} \mathrel{-\!\!*} \\ \quad\quad\quad\quad\quad \mathit{ewp}\; (\texttt{fulfill}\; p\; e)\; \langle \bot \rangle \{\_.\mathsf{True}\} \end{array} \right.$$

This specification states that, for every fiber e conforming to the protocol *Coop* and satisfying the postcondition Φ, if p is a promise that expects to be fulfilled with a value that satisfies Φ, then the instruction `fulfill` p e is safe and performs no effects.

The assertion $\mathit{isMember}\, p\, \gamma\, \Phi$ captures the implicit link between p and Φ. The token γ is the identifier of the assertion torch_γ, whose ownership is threaded through the execution of e until termination, when it is transferred to p.

Both the assertions *isMember* and *torch* are created at the moment a promise is allocated. The following lemma formalizes this claim:

Lemma 4.3 (Promise allocation) *The allocation of a fresh promise admits the following specification:*

$$\forall\, \Phi.\; \left\{ \begin{array}{l} \mathit{promiseInv}\; q \mathrel{-\!\!*} \\ \quad \mathit{ewp}\; (\texttt{ref}\; (\texttt{Waiting}\; []))\; \langle \bot \rangle \\ \qquad\qquad\qquad\qquad \{p.\; \mathit{promiseInv}\; q \;*\; \exists\, \gamma.\; \mathit{torch}_\gamma \;*\; \mathit{isMember}\; p\, \gamma\, \Phi\} \end{array} \right.$$

The derivation of this lemma combines several logical steps. First, the allocation of a fresh promise p yields a points-to assertion $p \mapsto \texttt{Waiting}\; []$. Second, the invariant *promiseInv* is open: the leading existential quantifier is destructed, thus introducing an abstract promise map M. Third, rule (ForgeTorch) introduces the assertion torch_γ for a fresh token γ. Fourth, rule (UpdateMap) updates M with the new entry $\{(p, \gamma) \mapsto \Phi\}$ (which must be disjoint from M because p is fresh), thus introducing the assertion $\mathit{isMember}\, p\, \gamma\, \Phi$. Finally, the invariant *promiseInv* is close: the existentially quantified map is instantiated with the updated map $M \sqcup \{(p, \gamma) \mapsto \Phi\}$.

Because `fulfill` is a recursive function, it is natural that the derivation of its specification (Lemma 4.2) starts with the application of Löb's induction principle. It then follows with the application of the reasoning rule for deep handlers, rule TryWithDeep, to reason about the handler from line 10. The assertion $\mathit{ewp}\; e()\; \langle \mathit{Coop} \rangle \{y.\; \square \Phi(y)\}$, which is part of the precondition of this lemma, dispatches one of the two premises of the rule. The other premise is a deep-handler judgment assertion corresponding to the

following statement:

$$\forall\, p,\, \gamma,\, \Phi.\ \left\{\begin{array}{l} \mathit{promiseInv}\ q \mathrel{-\!\!*} \\ \quad \mathit{torch}_\gamma \mathrel{-\!\!*} \\ \quad\quad \mathit{isMember}\ p\, \gamma\, \Phi \mathrel{-\!\!*} \\ \quad\quad\quad \mathit{isQueue}\ q\, I\ \_ \mathrel{-\!\!*} \\ \quad\quad\quad\quad \mathit{deep\text{-}handler}\ \langle \mathit{Coop} \rangle\ \{y.\Box\Phi(y)\} \\ \quad\quad\quad\quad\quad\quad ((\text{lines } 12\text{–}26) \mid (\text{lines } 28\text{–}36)) \\ \quad\quad\quad\quad\quad\quad \langle \bot \rangle\ \{\_.\mathsf{True}\} \end{array}\right.$$

The proof of this statement follows by Löb's induction. The derivation of the deep-handler judgment is split into the verification of the return and the effect branches. In the verification of the return branch, one must claim uniqueness of the assertion torch_γ, through rule (ClaimUniqueness), to show that line 35 is indeed unreachable. In the verification of the effect branch, one must consider the two cases of either an `async` operation or an `await` operation.

The verification of the case of an `async` operation consists of three steps. First, the Lemma 4.3 is applied to allocate a promise (line 15) for the fiber being spawned. Second, the assumption that the program conforms to the protocol *Async* is exploited to show that the suspended fiber k satisfies the queue invariant, and can thus be added to q (line 16). Finally, Lemma 4.2 is applied to justify the recursive call to `fulfill` (line 17). This step is allowed thanks to the application of Löb's induction in the beginning of the derivation of this lemma.

The verification of the case of an `await` operation exploits the invariant *promiseInv* to justify the two possible outcomes of reading the promise p (line 29). Opening the invariant and destructing the leading existential quantification introduces an abstract map M. To prove that p belongs to M, one must exploit that the effect performed by an `await` operation abides by the protocol *Await*, which asks the assertion $\mathit{isPromise}\, p\, \Phi$ (for some predicate Φ) as a precondition to performing this effect.

The derivation of the preceding statement completes the proof of Lemma 4.2. To finish the proof of `run`, it suffices to verify the call to `fulfill` in line 39. This final step follows from the application of Lemma 4.3, which justifies the allocation of a fresh promise; and of Lemma 4.2, which justifies the call to `fulfill`.

4.4 Related work

Implementation of continuation-based concurrency. It is well-known that continuations can be used to implement concurrency abstractions. Wand [Wan80a] illustrates the implementation of many concurrency primitives, such as `fork` and *semaphores* [Han73], in Indiana Scheme 3.1 [Wan80b]. Indiana Scheme is a dialect of Scheme [sch] with support for `catch`, a construct with virtually the same semantics as `callcc`. In the same vein, Haynes et al. [HFW84] illustrate the implementation of coroutines [dMI09] in Scheme 84 [FHKW84], a dialect of Scheme with support for `callcc`.

Verification of continuation-based concurrency. The literature counts with few formal proofs of correctness of continuation-based concurrency libraries such as the one presented in this chapter. Much of the related work concerns the proof of compilers.

Timany and Birkedal [TB19] verify the correctness of two concurrency constructs, the operations `fork` and `yield`, implemented using `callcc`, `throw`, and a global queue. However, their correctness statement is a compilation-correctness result: the translation of a (closed) program exploiting the primitive operations `fork` and `yield` into a program exploiting the encoding of these primitive operations in terms of `callcc`, `throw`, and a global queue is correct. The translation is correct in the sense that the translated program is an *observational refinement* of the source program. This means that, if the translated program terminates, then the source program also would terminate. To establish observational refinement, the authors introduce a *cross-language logical relation*, an interpretation of types as a relation between programs of different languages.

Nakata and Saar [NS13] also address the verification of continuation-based concurrency as a compilation-correctness problem. They verify a compiler from a language with primitive support for concurrency constructs to a language with support for the delimited-control operations `shift` and `reset`.

Chapter 5

Automatic Differentiation

Automatic differentiation (AD) [GW08] is a set of techniques for the efficient and exact computation of derivatives of functions defined by programs. In a slightly more formal sentence than the previous one, the problem addressed by AD could be stated as follows: given a program that defines a mathematical function E, for some sense of the word *defines*, how to algorithmically construct a program that defines the derivative of E?

The set of AD algorithms providing an answer to this question can be roughly classified according to two approaches: the *forward-mode* approach and the *reverse-mode* approach. We explain each of these approaches further in Section 5.3.

Another axis under which AD algorithms can be classified is according to the interface that they expose: either as a *compiler*, which translates a program that defines a function E into a program that defines the derivative of E; or as a *library*, which is a higher-order program that takes a program that defines a function E and produces a program that defines the derivative of E. The terminology *define-then-run* and *define-by-run* has been used to specify this distinction [VS21]. The term *define-then-run* expresses the idea that, under this interface, the input program e, which defines the function E, does not need to be evaluated. A compiler is thus qualified as define-then-run, because the arithmetic operations used by the source program e to define E can be statically inferred by the compiler: it suffices to analyse the source code of e. The term *define-by-run* expresses the idea that, under this interface, the input program e, which defines the function E, must be evaluated. A library is thus qualified as define-by-run, because (in a language without the ability to reflectively inspect source code at runtime) the arithmetic operations performed by the input program e to define E can be *observed* only during the execution of this program.

Our goal in this chapter is to specify and verify a *HH* implementation of a define-by-run reverse-mode AD algorithm using effect handlers. The contents of this chapter have been presented in a submitted (and now under-revision) paper [dVP22b].

5.1 Specification

We introduced the concept of an AD algorithm as an algorithm acting on a program that *defines* a function E. However, we did not assign a precise meaning to the word *defines*.

In the formal study of define-then-run algorithms, this question is often addressed by means of a denotational semantics [KKP+22, HSV21, SMC21]: *defines* becomes *denotes*, so a program defines the mathematical function E given by its denotational interpretation.

In the formal study of define-by-run algorithms, however, the denotational-semantics approach is not well-suited. Indeed, to see the shortcomings of this approach, it suffices to suppose that the host language in which an AD library is written has support for standard imperative constructs, such as references, `if-then-else` conditionals, and `while`

loops. Then, to find the denotation of an arbitrary input program e of such a library, one must find a denotational model that is well-defined for all these imperative constructs. However, we are unaware of a denotational model interpreting programs of a language $\mathcal{L}$ as differentiable mathematical functions in the case where $\mathcal{L}$ has support for references.

In this chapter, we wish to formally study `diff`, a define-by-run AD algorithm implemented in *HH*. Therefore, we wish to formally specify this algorithm, and, to do so, we must address this question of what it means for a program e to define a function E. Our solution is to introduce a Separation Logic predicate *isExp* (Definition 5.6), such that the program e defines a function E if the assertion e *isExp* E holds.

Clearly, we have just delegated the problem of specifying the meaning of *defines* to the definition of *isExp*. However, to write the specification of `diff`, it suffices to suppose that such a predicate exists. Moreover, to understand the specification of `diff`, it suffices to understand the informal contract expressed by this predicate. The assertion e *isExp* E means that e defines the *mathematical expression* E (Definition 5.1), which can be seen as a univariate polynomial. Therefore, this predicate is narrowing two sets: (1) it is narrowing the set of mathematical functions (which we did not define) to the set of mathematical expressions, and (2) it is narrowing the set of programs e to those that define a mathematical expression E. To define a mathematical expression E can be loosely understood as evaluating the univariate polynomial E. The important remark is that the predicate *isExp* leaves unspecified how a program e implements this evaluation. Such a program e can exploit every construct of the *HH* language, including references, `if-then-else` branching, recursive functions, and effect handlers.

With the introduction of *isExp*, the specification of the define-by-run AD algorithm `diff` is easily expressible: given a program e that defines an expression E, the application `diff` e produces a program e' that defines E', the *symbolic derivative* of E (Definition 5.5). This statement is formally captured in Hazel by the following specification:

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

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

This specification of AD is strikingly simple and yet permissive. In particular, it allows the verification of programs that apply `diff` in a nested fashion such as `diff` (`diff` e). If e defines an expression E, then it is easy to show that the output of this program defines E'', the second-order derivative of E.

5.2 Definitions

The previous section raised numerous questions. For instance, what is a mathematical expression E? How does a program e define an expression E? And finally, what is the definition of *isExp*? In this section, we shall answer these questions.

5.2.1 Mathematical expressions

In short, a mathematical expression E is a polynomial in one formal variable X. [1] We define the set of mathematical expressions E as an instance of the more general definition, $Exp_{\mathcal{I}}$, of expressions whose variables are indexed by a given set $\mathcal{I}$.

Definition 5.1 (Expressions) *Let $\mathcal{I}$ be a finite or infinite indexing set. Let $\imath$ range over $\mathcal{I}$. The set $Exp_{\mathcal{I}}$ of expressions whose variables are drawn from $\mathcal{I}$ is defined as follows:*

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

An expression $E \in Exp_{\mathcal{I}}$ is thus a multivariate polynomial with formal variables in the set $\{Leaf\ \imath\}_{\imath \in \mathcal{I}}$. By choosing the indexing set $\mathcal{I}$ to be the singleton set $\{X\}$, one obtains the set of mathematical expressions $Exp_{\{X\}}$, composed of univariate polynomials. Even though not immediately clear, the flexibility in the choice of indexing set is necessary. For example, it is needed in the statement of the *backward invariant* (Definition 5.13), one of the main invariants of the proof of correctness of `diff`.

It remains to define the *symbolic derivative* of a mathematical expression $E \in Exp_{\{X\}}$. We could define this notion *directly*, that is, without relying on auxiliary definitions. However, we choose again to make a detour into more general definitions that we shall meet again during the verification of `diff`. More specifically, we define the *symbolic derivative* of an expression in $Exp_{\{X\}}$ as a particular instance of the *partial derivative* of an expression in $Exp_{\mathcal{I}}$.

The partial derivative is defined in terms of the *evaluation of an expression* under *an assignment* ϱ of formal variables to values of an arbitrary *semiring*.

A semiring is a tuple $(\mathcal{R}, 0, +, 1, \times, \equiv)$, where $\mathcal{R}$ is a set and $\equiv$ is an equivalence relation on $\mathcal{R}$, for which the following equations are assumed to hold (for every a, b, and $c \in \mathcal{R}$): [2] [3]

$$\begin{aligned} a + (b + c) &\equiv (a + b) + c & \qquad a \times (b \times c) &\equiv (a \times b) \times c \\ a + b &\equiv b + a & \qquad a \times b &\equiv b \times a \\ a + 0 &\equiv a & \qquad a \times 1 &\equiv a \\ (a + b) \times c &\equiv (a \times c) + (b \times c) & \qquad a \times 0 &\equiv 0 \end{aligned}$$

We often write simply $\mathcal{R}$ to denote the semiring tuple, when the remaining components are easily inferred. Elements of $\mathcal{R}$ are called *numbers*.

The evaluation of an expression $E \in Exp_{\mathcal{I}}$ under an assignment $\varrho : \mathcal{I} \to \mathcal{R}$ is a term $[\![E]\!]_{\varrho} \in \mathcal{R}$, which results from interpreting a *node* `_ op _` in E by the corresponding

[1] We choose to work with univariate polynomials instead of multivariate polynomials, because this choice leads to a simpler implementation of AD and to a simpler correctness proof. The ideas presented in this chapter, however, can be easily adapted to a multivariate setting. We postulate that only the mechanized proof would require significant changes, even though we have not attempted to perform these changes.

[2] We require satisfiability with respect to the axioms of a semiring, rather than a ring, because the inverse of addition and its properties are not necessary to complete correctness proof.

[3] The set of axioms of a semiring may vary among different authors. Here, we follow the set of axioms defined by the *record* `semi_ring_theory`, which can be found in theory `setoid_ring.Ring_theory` of Coq's Standard Library: https://coq.inria.fr/library/Coq.setoid_ring.Ring_theory.html

arithmetic operation in $\mathcal{R}$, and from interpreting a leaf $\mathit{Leaf}\ \imath$ by the the number $\varrho(\imath)$. Here is the formal definition of $[\![E]\!]_\varrho$:

Definition 5.2 (Expression evaluation) *Let $\mathcal{R}$ be a semiring, and let $\mathcal{I}$ be an indexing set. The function $[\![\_]\!]_{(\_)} : \mathit{Exp}_\mathcal{I} \to (\mathcal{I} \to \mathcal{R}) \to \mathcal{R}$ is inductively defined as follows:*

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

The partial derivative of an expression $E \in \mathit{Exp}_\mathcal{I}$ with respect to a variable $\jmath$ under an assignment $\varrho : \mathcal{I} \to \mathcal{R}$ is a value in $\mathcal{R}$ noted $\partial E/\partial \jmath\ (\varrho)$. This value corresponds to the derivative with respect to r and at $\varrho(\jmath)$ of the function $\lambda r.\ [\![E]\!]_{\varrho[\jmath := r]}$, which evaluates E under the assignment $\varrho[\jmath := r]$ (that overwrites the value of ϱ at $\jmath$ with r). However, instead of defining a primitive notion of the derivative of a function of type $\mathcal{R} \to \mathcal{R}$, we exploit that the function we wish to differentiate is given as the evaluation of an expression E. Therefore, we can introduce an inductive definition of $\partial E/\partial \jmath\ (\varrho)$ as the result of the traversal that applies the differentiation laws of addition and multiplication to the corresponding nodes of E:

Definition 5.3 (Partial derivative) *Let $\mathcal{R}$ be a semiring, and let $\mathcal{I}$ be an indexing set. The function $\partial\_/\partial\_\ (\_) : \mathit{Exp}_\mathcal{I} \to \mathcal{I} \to (\mathcal{I} \to \mathcal{R}) \to \mathcal{R}$ is inductively defined as follows:*

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

The derivative of a univariate expression $E \in \mathit{Exp}_{\{X\}}$ can thus be defined as the partial derivative of E with respect to its single variable X. The assignment under which we consider this derivative is the function $\lambda \imath.\ \mathit{Leaf}\ \imath$, noted *Leaf* for short. The codomain of this function is the *free semiring* $\mathit{Exp}_{\{X\}}$: the semiring generated by taking $\mathit{Exp}_{\{X\}}$ as the carrier set, the function $\_\ \mathit{Add}\ \_$ as the addition operation, the function $\_\ \mathit{Mul}\ \_$ as the multiplication operation, and the constants *Zero* and *One* as the respective neutral elements. The equivalence relation $\_ \equiv_{\mathit{Exp}_{\{X\}}} \_$ of this semiring is defined inductively as the smallest relation that validates the semiring axioms for this choice of arithmetic operations. The following two definitions sum up this discussion.

Definition 5.4 (Free semiring) *The tuple $(\mathit{Exp}_{\{X\}}, \mathit{Zero}, \mathit{Add}, \mathit{One}, \mathit{Mul}, \equiv_{\mathit{Exp}_{\{X\}}})$ is a semiring, where the equivalence relation $\_ \equiv_{\mathit{Exp}_{\{X\}}} \_$ is defined as the smallest relation for which the semiring axioms hold.*

Definition 5.5 (Symbolic derivative) *Let $E \in \mathit{Exp}_{\{X\}}$ be a univariate expression. The symbolic derivative of E, written E', is the partial derivative of E with respect to X under the free-semiring assignment Leaf.*

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

One can check that this definition gives rise to the usual laws of symbolic derivation. In the case of multiplication, for instance, by combining the fourth equation from Definition 5.3 with the identity laws $[\![E_1]\!]_{Leaf} = E_1$ and $[\![E_2]\!]_{Leaf} = E_2$, one immediately obtains the familiar symbolic derivation law:

$$(E_1\ \mathit{Mul}\ E_2)' = (E_1'\ \mathit{Mul}\ E_2)\ \mathit{Add}\ (E_1\ \mathit{Mul}\ E_2').$$

5.2.2 Programmatic expressions

How to programmatically define a mathematical expression E? The immediate answer is to define E as *data*, using binary sums to distinguish among the different constructors of $\mathit{Exp}_{\{X\}}$. However, with such a representation of expressions, an AD algorithm would be constrained to offer a define-then-run interface: to differentiate an expression E, the programmer would have to first write its representation in a domain-specific language, thus revealing beforehand the arithmetic operations used in the construction of E.

To offer a define-by-run interface, a mathematical expression E should be represented as a *computation*, rather than as data. A well-known method for representing terms of an inductive definition as computations is the *Church-Böhm-Berarducci encoding* [Kis12b], also known as the *tagless final* [CKS09, Kis10] representation. According to this method, the expression E is represented by a *programmatic expression* e: a higher-order function with five arguments, $\mathit{zero}_{\mathcal{R}}$, $\mathit{one}_{\mathcal{R}}$, $\mathit{add}_{\mathcal{R}}$, $\mathit{mul}_{\mathcal{R}}$, and $x_{\mathcal{R}}$, each argument corresponding to one of the five constructors of the $\mathit{Exp}_{\{X\}}$, the constructors *Zero*, *One*, *Add*, *Mul*, and *Leaf* X.

One can see the four first arguments of a programmatic expression e ($\mathit{zero}_{\mathcal{R}}$, $\mathit{one}_{\mathcal{R}}$, $\mathit{add}_{\mathcal{R}}$, $\mathit{mul}_{\mathcal{R}}$) as a set of arithmetic operations over an abstract semiring $\mathcal{R}$; the last argument $x_{\mathcal{R}}$ as a number r of this semiring; and the expression e as computing the evaluation of E under the assignment of X to r, that is, the number $[\![E]\!]_{(\lambda X.\, r)}$. Under this perspective, it is easy to see why the behavior of e is constrained by the expression E that it defines. For example, if e defines the expression *Leaf* X, then e could be implemented as returning the value $x_{\mathcal{R}}$, or the result of multiplication $\mathit{mul}_{\mathcal{R}}\ x_{\mathcal{R}}\ \mathit{one}_{\mathcal{R}}$. However, if e was implemented by the addition $\mathit{add}_{\mathcal{R}}\ x_{\mathcal{R}}\ x_{\mathcal{R}}$, then it would define the expression (*Leaf* X) *Add* (*Leaf* X), rather than *Leaf* X. Here is another example of a programmatic expression:

```
(* [simple] defines the expression [X(X+1)]. *)
let simple = fun zeroR oneR addR mulR xR ->
  mulR xR (addR xR oneR)
```

The implementation of a programmatic expression e, however, is not confined to the arithmetic operations that it introduces as arguments. As a computation e has access to the entire set of *HH* features, including references, `if-then-else` conditionals, recursive functions, and effect handlers. An interesting example of a programmatic expression that exploits some of these features is the following program:

```
(* [monomial k] defines the expression [X^k]. *)
let monomial k = fun _ oneR _ mulR xR ->
  let res, x, k = ref oneR, ref xR, ref k in
  let rec loop() =
    if !k > 0 then
```

```
      res := mulR !res (if !k % 2 = 0 then oneR else !x);
      x := mulR !x !x;
      k := !k / 2
  in
  loop();
  !res
```

This program computes the evaluation of the monomial expression of degree k. It implements the technique of fast exponentiation by means of references, a recursive function, and a conditional.

Another interesting aspect of this representation is that, because a programmatic expression is parameterized over the set of arithmetic operations, one can evaluate a programmatic expression e with concrete arithmetic operations of an arbitrary semiring $\mathcal{R}$. For instance, one can evaluate the programs `simple` and `monomial` with integer arithmetic as follows:

```
(* A set of integer arithmetic operations. *)
let zeroI, oneI, addI, mulI, twoI =
  0, 1, (fun a b -> a + b), (fun a b -> a * b), 2

(* This program evaluates to [6]. *)
let _ = simple zeroI oneI addI mulI twoI

(* This program evaluates to [1024]. *)
let _ = (monomial 10) zeroI oneI addI mulI twoI
```

5.2.3 Relating programmatic expressions to mathematical expressions

The only ingredient in the specification of `diff` that remains to be introduced is the predicate *isExp*, which relates a programmatic expression e to a mathematical expression E. Let us not delay any further, here is the definition of *isExp*:

Definition 5.6 (*isExp*) *The predicate* $\_$ *isExp* $\_ : \mathit{Val} \to \mathit{Exp}_{\{X\}} \to \mathit{iProp}$ *is defined as follows:*

$$
\begin{aligned}
&e \mathrel{\mathit{isExp}} E \triangleq \\
&\quad \Box \forall \mathcal{R}, \Psi, \mathit{isNum}.\ \forall\, \mathit{zero}, \mathit{one}, \mathit{add}, \mathit{mul}. \\
&\qquad \mathit{isNumDict}\,(\mathit{zero}, \mathit{one}, \mathit{add}, \mathit{mul}, \mathcal{R}, \Psi, \mathit{isNum}) \mathrel{-\!\!*} \\
&\qquad\quad \forall\, x, r.\ x \mathrel{\mathit{isNum}} r \mathrel{-\!\!*} \\
&\qquad\qquad \mathit{ewp}\ (e\ \mathit{zero}\ \mathit{one}\ \mathit{add}\ \mathit{mul}\ x)\ \langle\Psi\rangle\{y.\ \exists s. \\
&\qquad\qquad\quad y \mathrel{\mathit{isNum}} s * s \equiv_{\mathcal{R}} [\![E]\!]_{(\lambda X.\, r)}\}
\end{aligned}
$$

This definition states that e defines E, if e is able to compute the evaluation of E in any given ring $\mathcal{R}$ and under any assignment of X to a number r.

The leading persistently modality means that, once e has been shown to define E, e can be used indefinitely many times as a representation of this expression.

Following this modality, there comes a series of universally quantified variables. The semiring $\mathcal{R}$ denotes the semiring in which the evaluation of E must be computed. The values *zero*, *one*, *add*, and *mul* denote the arithmetic operations to which e is going to be applied; they *model* the operations of the semiring $\mathcal{R}$ in a sense that the

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

Figure 5.1: Specification of a dictionary of arithmetic operations

assertion *isNumDict*, which we shall soon discuss, holds of these values. The values *add* and *mul* are values of *HH*, therefore they can perform effects. The protocol Ψ describes these effects. Finally, the term *isNum* is a *representation predicate*, it relates a value x to a number r, thus formalizing claims of the kind "*a value* x *represents a number* r".

The predicate *isNum* is exploited to provide a specification of the arithmetic operations *zero*, *one*, *add*, and *mul*, thereby formalizing the claim that these operations *model* the operations of the semiring $\mathcal{R}$. Indeed, the next line of the definition of *isExp* claims that the assertion *isNumDict* holds. This assertion, whose definition appears in Figure 5.1, is the conjunction of the following claims: (1) the value *zero* represents 0; (2) the value *one* represents 1; (3) for all numbers r and s, `add` computes a representation of $r + s$ when applied to representations of r and s; (4) for all numbers r and s, `mul` computes a representation of $r \times s$ when applied to representations of r and s; (5) a representation of a number is persistent.

The concluding three lines of the definition of *isExp* state that, for every value x and number r such that x represents r, the application of e to the set of arithmetic operations, *zero*, *one*, *add*, *mul*, and to x produces a value y such that y represents the evaluation of E under the assignment of X to r.

In fact, the result value y is claimed to represent a number s such that s is equivalent to $[\![E]\!]_{(\lambda X.\, r)}$. This slightly more convoluted statement exempts one from requiring that the predicate *isNum* be *compatible* with the semiring equivalence relation $\equiv_{\mathcal{R}}$:

$$
\text{"}\mathit{isNum}\ \mathit{is\ compatible\ with}\ \equiv_{\mathcal{R}}\text{"} \triangleq \forall\, x,\, r,\, s.\ x\ \mathit{isNum}\ r \mathrel{-\!\!*} r \equiv_{\mathcal{R}} s \mathrel{-\!\!*} x\ \mathit{isNum}\ s
$$

This requirement seems natural, but it is not satisfied by the particular instance of *isNum* used in the verification of `diff` (the predicate *isNode*, Definition 5.9).

Let us conclude this subsection with a couple of remarks.

First, to compute the evaluation of E in the semiring $\mathcal{R}$, the programmatic expression e has access to the value x, representing the element r; the value *zero*, representing the neutral element 0; the value *one*, representing the neutral element 1; the function *add*, which evaluates an addition node; and the function *mul*, which evaluates a multiplication node. Because $\mathcal{R}$ is abstract, e uses only these values to introduce representations of numbers. By no other means, can e produce values representing numbers. This limitation illustrates how the Church-Böhm-Berarducci encoding implements the constraint that inhabitants of an algebraic data type can only be built using the basic constructors of this type.

```
let fm_diff e = fun zeroR oneR addR mulR xR ->
  let zeroS = (zeroR, zeroR) in
  let oneS  = (oneR, zeroR) in
  let addS (av, ad) (bv, bd) =
    (addR av bv, addR ad bd) in
  let mulS (av, ad) (bv, bd) =
    (mulR av bv, addR (mulR ad bv) (mulR av bd)) in
  let xS = (xR, oneR) in
  let _, yd = e zeroS oneS addS mulS xS in
  yd
```

Figure 5.2: Forward-mode AD library in *HH*.

Second, because the protocol Ψ, which describes the effects of *add* and *mul*, is abstract, e is not allowed to install a handler over applications of these functions. Moreover, because this protocol is also used to specify the effects of e, this means that e does not introduce effects. In conclusion, e is effect-polymorphic: the only observable effects that e may perform are those introduced by calls to `add` or `mul`. The correctness of `diff` relies on this fact.

5.3 Implementation

Before presenting the program `diff`, which uses effect handlers to implement reverse-mode AD, we present the program `fm_diff`, which implements forward-mode AD, and which is easier to understand than `diff`.

The forward-mode algorithm `fm_diff` satisfies the same specification as `diff`. This shows that Specification 5.1 is agnostic as to whether the algorithm implements a reverse-mode or a forward-mode approach. Moreover, both forward-mode and reverse-mode AD share the idea of monitoring the execution of the input programmatic expression e under a *modified* set of arithmetic operations. Therefore, understanding the forward-mode algorithm `fm_diff` constitutes a good preparation for the explanation of `diff`.

5.3.1 Forward-mode AD

The implementation of `fm_diff` appears in Figure 5.2. As previously claimed, the program `fm_diff` satisfies the same specification as `diff`:

Statement 5.2 *The forward-mode AD algorithm* `fm_diff` *satisfies Specification 5.1.*

This statement means that `fm_diff` takes a programmatic expression e defining a mathematical expression E, and produces a programmatic expression e' defining the derivative of E; that is, `fm_diff` must produce a programmatic expression e' that, when applied to a set of arithmetic operations over an abstract semiring $\mathcal{R}$ and a value representation of a number r, is able to compute the number $[\![E']\!]_{(\lambda X.\, r)}$.

The key idea of the forward-mode approach is that knowing the evaluation of E in a well-chosen semiring $\mathcal{S}$ is sufficient to know the evaluation of E' in $\mathcal{R}$. In other words,

given a semiring $\mathcal{R}$ and a number $r \in \mathcal{R}$, there exists a semiring $\mathcal{S}$ and a number $s \in \mathcal{S}$, such that, if one knows $[\![E]\!]^{\mathcal{S}}_{(\lambda X.\, s)}$, then one knows $[\![E']\!]^{\mathcal{R}}_{(\lambda X.\, r)}$.[4]

Such a semiring $\mathcal{S}$ indeed exists, it is the semiring of *dual numbers*. A dual number is a pair of numbers in $\mathcal{R}$. Hence, the carrier set of $\mathcal{S}$ is $\mathcal{R} \times \mathcal{R}$. The first component of a dual number is called the *value component* and the second component is called the *tangent component*.

Addition and multiplication on dual numbers are defined as follows:

$$\begin{array}{lcl} (a, \dot{a}) +_{\mathcal{S}} (b, \dot{b}) & \triangleq & (a +_{\mathcal{R}} b, \dot{a} +_{\mathcal{R}} \dot{b}) \\ (a, \dot{a}) \times_{\mathcal{S}} (b, \dot{b}) & \triangleq & (a \times_{\mathcal{R}} b, \dot{a} \times_{\mathcal{R}} b +_{\mathcal{R}} a \times_{\mathcal{R}} \dot{b}) \end{array}$$

Now, let s be the dual number $(r, 1)$. It follows, from a simple proof by induction on E, that the evaluation of E in $\mathcal{S}$ at s satisfies the following equation:

$$[\![E]\!]^{\mathcal{S}}_{(\lambda X.\, s)} = ([\![E]\!]^{\mathcal{R}}_{(\lambda X.\, r)}, [\![E']\!]^{\mathcal{R}}_{(\lambda X.\, r)})$$

Therefore, to evaluate E' in $\mathcal{R}$, it suffices to evaluate E in $\mathcal{S}$!

The implementation of `fm_diff` is a faithful translation of this idea. In line 1, it introduces the set of arithmetic operations on $\mathcal{R}$. In lines 2 to 6, it defines the arithmetic operations on dual numbers. In line 8, it defines the dual number s. Finally, in line 9, the programmatic expression e is evaluated under dual-number arithmetic, thus producing a dual number whose tangent component is the desired result of `fm_diff`.

5.3.2 Reverse-mode AD

Reverse-mode AD follows the same basic idea as forward-mode AD: given a programmatic expression e that defines E, a set of arithmetic operations over an abstract semiring $\mathcal{R}$, and a value representation of a number r, it computes the evaluation of E' in $\mathcal{R}$ at r by computing the evaluation of E in a custom semiring $\mathcal{S}$. In reverse-mode AD, this custom semiring $\mathcal{S}$ is the free semiring $Exp_{\{X\}}$ (Definition 5.4).

The evaluation of E under the free-semiring arithmetic does not yield a value from which the derivative E can be directly extracted. The purpose of this evaluation is to record the list of arithmetic operations performed by e during its execution. This list is known in the literature as the *Wengert list* [GW08]. To compute the evaluation of E' in $\mathcal{R}$, the algorithm then processes the entries of this list in the reverse order as they were added. The execution of a reverse-mode AD algorithm is thus divided into two phases: the *forward phase*, consisting of the execution of e; and the *backward phase*, consisting of the evaluation of E'. The names *backward* and *reverse* reflect the order in which the entries of the Wengert list are treated.

The definition of `diff`, a define-by-run AD algorithm implemented in *HH*, appears in Figure 5.3. For the sake of readability, this definition employs the following abbrevia-

[4] The superscripts decorating the evaluation maps $[\![\_]\!]_{\_}$ specify the semiring in which the evaluation is computed.

```
let diff e = fun zeroR oneR addR mulR xR ->
  let zeroS = Zero in
  let oneS  = One in
  let addS  = fun a b -> do (Add (a, b)) in
  let mulS  = fun a b -> do (Mul (a, b)) in

  let mk n = Var (n, ref zeroR) in
  let get_v u =
    match u with
    ( fun (* Const. *) c ->
        match c with ( fun _ -> zeroR | fun _ -> oneR )
    | fun (* Var. *) (m, _) -> m
    )
  in
  let get_d u =
    match u with
    ( fun (* Const. *) _ -> () () (* Unreachable. *)
    | fun (* Var. *) (_, q) -> !q
    )
  in
  let update u i =
    match u with
    ( fun (* Const. *) _ -> ()
    | fun (* Var. *) (_, q) -> q := addR !q i
    )
  in

  let xS = mk xR in
  let () =
    deep-try (e zeroS oneS addS mulS xS) with
    (* Effect branch. *)
    ( fun request k ->
        match request with
        ( fun (* Add. *) (a, b) ->
            let u = mk (addR (get_v a) (get_v b)) in
            k u;
            update a (get_d u);
            update b (get_d u)
        | fun (* Mul. *) (a, b) ->
            let u = mk (mulR (get_v a) (get_v b)) in
            k u;
            update a (mulR (get_d u) (get_v b));
            update b (mulR (get_d u) (get_v a))
        )
    (* Return branch. *)
    | fun y ->
        update y oneR
    )
  in
  get_d x
```

Figure 5.3: Reverse-mode AD library in *HH*.

tions:

$$\begin{aligned} \texttt{Zero} &\triangleq \texttt{inj}_1\ (\texttt{inj}_1\ ()) \\ \texttt{One} &\triangleq \texttt{inj}_1\ (\texttt{inj}_2\ ()) \\ \texttt{Var}\ (m, \mathscr{q}) &\triangleq \texttt{inj}_2\ (m, \mathscr{q}) \\ \texttt{Add}\ (a, b) &\triangleq \texttt{inj}_1\ (a, b) \\ \texttt{Mul}\ (a, b) &\triangleq \texttt{inj}_2\ (a, b) \end{aligned}$$

In line 1, the function `diff` gains access to the following values: a programmatic expression e that defines a mathematical expression E; a set of arithmetic operations $zero_{\mathcal{R}}$, $one_{\mathcal{R}}$, $add_{\mathcal{R}}$, and $mul_{\mathcal{R}}$ implementing the operations of an abstract ring $\mathcal{R}$; and a value $x_{\mathcal{R}}$ representing a number $r \in \mathcal{R}$.

As previously mentioned, reverse-mode AD exploits the programmatic expression e to compute the evaluation of E in the free semiring $Exp_{\{X\}}$. Therefore, the function `diff` must introduce a value representation of mathematical expressions. The approach implemented by `diff` is to represent mathematical expressions as nodes in an implicit *computational graph*. A computational graph is a *directed acyclic graph* (DAG) with three types of nodes: (1) *constant* nodes, which identify the expressions *Zero* and *One*; (2) *composite* nodes, which identify the operations *Add* and *Mul*; and (3) a *variable* node, which identifies the expression *Leaf* X. Constant nodes and the variable node have no outgoing edges. A composite node has exactly two outgoing edges indicating the operands of the operation that it identifies. Every node u is associated to a mathematical expression E_u: if u is either a constant node or a variable node, then u is associated to the expression it identifies; if u is a composite node that identifies the operation op and is connected to the nodes a and b, then u is associated to the expression $E_a\ op\ E_b$. Therefore, to represent a mathematical expression as a value, it suffices to introduce a value representation of the nodes of this graph.

The constant node identifying *Zero* is represented by the value `Zero`. The constant node identifying *One* is represented by the value `One`. The variable node and composite nodes are represented by *intermediate variables*, which are values of the form `Var` $(m, \mathscr{q})$. The first component m of an intermediate variable $u = \texttt{Var}\ (m, \mathscr{q})$ is called the *value field* of u. It represents the evaluation of E_u (the expression identified by u) at r. The second component $\mathscr{q}$ is a memory location. The value stored in $\mathscr{q}$ is called the *derivative field* of u. During the forward phase, this memory location ensures that the value representation of nodes is injective: two different nodes in the graph have distinct value representations. This restriction is met by the function `mk` (line 7), which creates new nodes by allocating fresh memory locations.

The function `get_v` (line 8) computes the evaluation of the expression E_u identified by a node u. In the case of the constant nodes `Zero` and `One`, it simply returns the representation of the $\mathcal{R}$-numbers 0 and 1, respectively. In the case of an intermediate variable u, it suffices to read u's value field. The function `get_d` (line 15) reads u's derivative field. It is never applied to `Zero` or `One`.

The arithmetic operations acting on nodes of the computational graph, and, consequently, also acting on representations of mathematical expressions, are introduced in lines 2 to 5. The values $zero_{\mathcal{S}}$ (line 2) and $one_{\mathcal{S}}$ (line 3) are respectively defined as `Zero` and `One`. The functions $add_{\mathcal{S}}$ (line 4) and $mul_{\mathcal{S}}$ (line 5) are both defined by a single instruction: the action of performing an effect. In the case of $add_{\mathcal{S}}$, the effect is `Add`, whereas, in the case of `mulS`, the effect is `Mul`. The purpose of the functions $add_{\mathcal{S}}$

and $mul_{\mathcal{S}}$ is simply to send a signal to the effect handler of line 30 when e requests an arithmetic operation. The meaning of these operations is given by this handler. A call to $add_{\mathcal{S}}$, for example, with two intermediate variables a and b, triggers the creation of a new composite node represented by the intermediate variable created in line 35. A call to $mul_{\mathcal{S}}$ behaves analogously. The only difference is the operation which the node identifies. The new composite node created in line 35, for example, identifies an addition operation Add, whereas the new composite node created in line 40, identifies a multiplication operation Mul.

In line 28, the first intermediate variable $x_{\mathcal{S}}$ is introduced. This variable represents the variable node, which identifies X. Accordingly, the value field of $x_{\mathcal{S}}$ is $x_{\mathcal{R}}$, a value that represents the evaluation of X at r.

In line 30, the execution of e begins. This execution is monitored by the effect handler, whose definition spans lines 30 to 48. Every time e performs an operation $add_{\mathcal{S}}$ or `mulS`, control is handed over to the handler. In addition to creating a new composite node represented by a fresh intermediate variable u, issued by a call to `mk` (in either line 35 or 40), the handler *schedules* calls to the function `update` (lines 37 to 38, and lines 42 to 43). The function `update` is responsible for performing an incremental update to the second component of intermediate variables. We postpone the correctness argument of `diff` to the next section, but, for now, let us claim that, at the end of the backward phase, these incremental updates result in the evaluation of E'. Therefore, the creation of a new node u induces a set of update instructions that contribute to the computation of `diff`'s desired result. When the update instructions induced by a node u are executed, we say that u *has been treated*.

The handler schedules update instructions by placing them *syntactically after* the continuation resumption (lines 36 and 41). The crucial observation is that, because the handler is deep, the continuation k contains a copy of this handler as its topmost frame. Therefore, the update instructions induced by further calls to either $add_{\mathcal{R}}$ or $mul_{\mathcal{R}}$ during the execution of k are executed before those placed syntactically after the continuation resumption. Nodes are thus treated in the reverse order as they are introduced. So, even though the distinction between forward and backward phases is not clear in `diff`'s code, this distinction exists in `diff`'s operational behavior: the forward phase happens during the execution of e, where calls to $add_{\mathcal{R}}$ and $mul_{\mathcal{R}}$ create new nodes in the computational graph; whereas the backward phase happens after the execution of e, when the update instructions are called and nodes are treated in the reverse order as they were introduced. Between the forward and backward phases, there happens the execution of the single update instruction of line 47. Indeed, this instruction is executed (only once) upon e's termination. The execution of this instruction does not correspond to the treatment of a node in the graph. Its purpose is to plant the value $one_{\mathcal{R}}$ as a "seed" in the derivative field of y, the intermediate variable resulting from the execution of e. This variable is the first one to have a derivative field storing a value different from $zero_{\mathcal{R}}$. The incremental updates during the backward phase "magnify" this seed in such a way that, at the end of the backward phase, the increments accumulated in the derivative field of $x_{\mathcal{S}}$ correspond to a value representation of $[\![E']\!]_{(\lambda X.\, r)}$, the desired result of `diff`. Therefore, `diff` terminates in line 50 by looking up to $x_{\mathcal{S}}$'s derivative field.

5.4 Verification

Proof context. This section presents the formal proof of Statement 5.1: given a programmatic expression e that defines a mathematical expression E – that is, assuming the assertion e *isExp* E holds – we prove that the program `diff` e produces an expression e' that defines E' – that is, we show that the assertion e' *isExp* E' holds.

It comes as no surprise that, during the proof, the two occurrences of *isExp*, the one that appears as a premise and the one that appears as a goal, are unfolded. To avoid confusion between the two sets of universally quantified variables from each occurrence of *isExp*, the semiring quantified in e *isExp* E is noted $\mathcal{S}$, whereas the semiring quantified in e' *isExp* E' is noted $\mathcal{R}$. The semiring $\mathcal{S}$ is used as a subscript in the variables of e *isExp* E, and $\mathcal{R}$ is used as a subscript in the variables of e' *isExp* E'. In sum, the assertion e *isExp* E is unfolded as:

$$
\begin{aligned}
&e\ \mathit{isExp}\ E = \\
&\quad \Box\forall\, \mathcal{S},\ \Psi_{\mathcal{S}},\ \mathit{isNum}_{\mathcal{S}}.\ \forall\, \mathit{zero}_{\mathcal{S}},\ \mathit{one}_{\mathcal{S}},\ \mathit{add}_{\mathcal{S}},\ \mathit{mul}_{\mathcal{S}}. \\
&\qquad \mathit{isNumDict}\,(\mathit{zero}_{\mathcal{S}}, \mathit{one}_{\mathcal{S}}, \mathit{add}_{\mathcal{S}}, \mathit{mul}_{\mathcal{S}}, \mathcal{S}, \Psi_{\mathcal{S}}, \mathit{isNum}_{\mathcal{S}}) \multimap \\
&\qquad\quad \forall\, x_{\mathcal{S}},\ s \in \mathcal{S}.\ x_{\mathcal{S}}\ \mathit{isNum}_{\mathcal{S}}\ s \multimap \\
&\qquad\qquad \mathit{ewp}\ (e\ \mathit{zero}_{\mathcal{S}}\ \mathit{one}_{\mathcal{S}}\ \mathit{add}_{\mathcal{S}}\ \mathit{mul}_{\mathcal{S}}\ x_{\mathcal{S}})\ \langle\Psi_{\mathcal{S}}\rangle\{y.\ \exists\, T. \\
&\qquad\qquad\quad y\ \mathit{isNum}_{\mathcal{S}}\ T\ *\ T \equiv_{\mathcal{S}} [\![E]\!]_{(\lambda X.\ s)}\}
\end{aligned}
$$

And, the assertion e' *isExp* E' is unfolded as:

$$
\begin{aligned}
&e'\ \mathit{isExp}\ E' = \\
&\quad \Box\forall\, \mathcal{R},\ \Psi_{\mathcal{R}},\ \mathit{isNum}_{\mathcal{R}}.\ \forall\, \mathit{zero}_{\mathcal{R}},\ \mathit{one}_{\mathcal{R}},\ \mathit{add}_{\mathcal{R}},\ \mathit{mul}_{\mathcal{R}}. \\
&\qquad \mathit{isNumDict}\,(\mathit{zero}_{\mathcal{R}}, \mathit{one}_{\mathcal{R}}, \mathit{add}_{\mathcal{R}}, \mathit{mul}_{\mathcal{R}}, \mathcal{R}, \Psi_{\mathcal{R}}, \mathit{isNum}_{\mathcal{R}}) \multimap \\
&\qquad\quad \forall\, x_{\mathcal{R}},\ r \in \mathcal{R}.\ x_{\mathcal{R}}\ \mathit{isNum}_{\mathcal{R}}\ r \multimap \\
&\qquad\qquad \mathit{ewp}\ (e'\ \mathit{zero}_{\mathcal{R}}\ \mathit{one}_{\mathcal{R}}\ \mathit{add}_{\mathcal{R}}\ \mathit{mul}_{\mathcal{R}}\ x_{\mathcal{R}})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ \exists s. \\
&\qquad\qquad\quad y\ \mathit{isNum}_{\mathcal{R}}\ s\ *\ s \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}\}
\end{aligned}
$$

Because the assertion e' *isExp* E' is the goal, the universally quantified terms of this definition must be introduced. More specifically, the proof context can be divided into the following pair of hypothesis and goal:

Hypothesis 5.1 *The variables* $\mathcal{R}$, $\Psi_{\mathcal{R}}$, $\mathit{isNum}_{\mathcal{R}}$, $\mathit{zero}_{\mathcal{R}}$, $\mathit{one}_{\mathcal{R}}$, $\mathit{add}_{\mathcal{R}}$, $\mathit{mul}_{\mathcal{R}}$, $x_{\mathcal{R}}$, *and* r *are introduced, and the assertions* $\mathit{isNumDict}\,(\mathit{zero}_{\mathcal{R}}, \mathit{one}_{\mathcal{R}}, \mathit{add}_{\mathcal{R}}, \mathit{mul}_{\mathcal{R}}, \mathcal{R}, \Psi_{\mathcal{R}}, \mathit{isNum}_{\mathcal{R}})$ *and* $x_{\mathcal{R}}\ \mathit{isNum}_{\mathcal{R}}\ r$ *are assumed to hold.*

Goal 5.1 *The goal is to prove that* e' *(the result of* `diff` e*) computes the evaluation of* E' *in the semiring* $\mathcal{R}$ *at* r*, when supplied with implementations of the arithmetic operations of* $\mathcal{R}$ *and with a value representation of* r*; that is, the goal is the following ewp assertion:*

$$\mathit{ewp}\ (e'\ \mathit{zero}_{\mathcal{R}}\ \mathit{one}_{\mathcal{R}}\ \mathit{add}_{\mathcal{R}}\ \mathit{mul}_{\mathcal{R}}\ x_{\mathcal{R}})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ \exists s.\ y\ \mathit{isNum}_{\mathcal{R}}\ s\ *\ s \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}\}$$

Computational graph. In the previous section, we used an *implicit* computational graph to explain the dynamic behavior of `diff`. The main idea of the formal proof is to make this graph *explicit*; that is, to introduce a *formal representation* of a computational graph, thereby bringing this notion within reach of the program logic. Because this graph

$$\begin{array}{ll}
\text{(IntroduceContext)} & \dot{\Rrightarrow} \exists \gamma.\ \mathit{isContext}_\gamma\ [] \\
\text{(IntroduceBinding)} & \mathit{isContext}_\gamma\ K \mathrel{-\!\!*} \dot{\Rrightarrow} \left\{ \begin{array}{l} \mathit{isContext}_\gamma\ (K +\!\!+ B)\ * \\ \mathit{isBinding}_\gamma\ B \end{array} \right. \\
\text{(ClaimDefinedness)} & \mathit{isContext}_\gamma\ K \mathrel{-\!\!*} \mathit{isBinding}_\gamma\ B \mathrel{-\!\!*} B \in K \\
\text{(ShareBinding)} & \mathit{isBinding}_\gamma\ B \mathrel{-\!\!*} \Box\ \mathit{isBinding}_\gamma\ B
\end{array}$$

Figure 5.4: Logical rules governing the assertions *isBinding* and *isContext*.

evolves during the forward phase, as new nodes are added to the graph, a piece of ghost state is needed to keep track of this graph. During the backward phase, the graph is not modified, so only its formal representation is needed.

The formal representation of computational graphs relies on the notion of a *binding*. A binding B specifies a composite node u and the pair of nodes connected to u. Indeed, a binding is a quadruple of the form $\mathsf{let}\ u = a\ op\ b$, where u specifies a composite node, op is the operation identified by u, and a and b are the nodes connected to u. A graph is thus represented as a list of bindings K, also called a *context*. In conclusion, here is the formal definition of the syntax of bindings and contexts:

Definition 5.7 (Bindings; contexts) *Let u, a, b range over Val. The syntax of bindings and contexts is defined as follows:*

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

This representation ignores the constant nodes and the variable node. There is no need to include these nodes as part of a context K, because they can be easily identified. Constant nodes are represented by the fixed set of values $\mathit{zero}_\mathcal{S} = \texttt{Zero}$ (line 2) and $\mathit{one}_\mathcal{S} = \texttt{One}$ (line 3). The variable node is represented by the intermediate variable $x_\mathcal{S}$, allocated in line 28, point of the execution at which the following assertion holds:

$$x_\mathcal{S} = \texttt{Var}\ (x_\mathcal{R}, q)\ *\ q \mapsto \mathit{zero}_\mathcal{R}$$

This assertion follows a pattern that shall become common in the remainder of this proof. Therefore, we introduce the predicate *isVar*, which relates an intermediate variable u to the pair of the numbers in $\mathcal{R}$ represented by its value and derivative fields. The definition hides behind an existential quantifier the concrete value representation of numbers in $\mathcal{R}$ and the memory location q storing the derivative field of a variable:

Definition 5.8 *The predicate $\_\ \mathit{isVar}\ \_ : \mathit{Val} \to \mathcal{R} \times \mathcal{R} \to \mathit{iProp}$ is defined as follows:*

$$u\ \mathit{isVar}\ (s,\ \dot{s}) \triangleq \exists m,\ q,\ d.\ u = \texttt{Var}\ (m, q)\ *\ m\ \mathit{isNum}_\mathcal{R}\ s\ *\ q \mapsto d\ *\ d\ \mathit{isNum}_\mathcal{R}\ \dot{s}$$

Exploiting the assumption that $x_\mathcal{R}$ represents the number r and that $\mathit{zero}_\mathcal{R}$ represents 0, it is easy to show that the intermediate variable $x_\mathcal{S}$ satisfies the following specification:

$$x_\mathcal{S}\ \mathit{isVar}\ (r,\ 0) \tag{5.1}$$

Now that the computational graph admits a formal representation as a set of fixed values $zero_{\mathcal{S}}$, $one_{\mathcal{S}}$, and $x_{\mathcal{S}}$, and a series of composite nodes K, the next step is to define a binary predicate u *isNode* E_u, which formalizes the relation "*a node* u *is associated to an expression* E_u" informally introduced in the previous section. This is desirable, because `diff` relies on the execution of e under free-semiring arithmetic. Therefore, to reason about this execution, one must exploit e's specification (the assertion e *isExp* E) with suitable instances of its universally quantified variables. In particular, one must instantiate the semiring $\mathcal{S}$ with the free semiring $Exp_{\{X\}}$, and one must instantiate $isNum_{\mathcal{S}}$ with a representation predicate relating values to elements of the free semiring. However, if the relation *isNode* is to be used as the representation predicate $isNum_{\mathcal{S}}$, then *isNode* must be persistent, a requirement that comes from *isNumDict* (Figure 5.1). This restriction means, in particular, that, if one shows that a node u is associated to an expression E_u at some point in time during the execution of e, then this association must persist until the end of e's execution.

This persistence restriction is not trivially satisfiable, because the computational graph evolves during e's execution. So, one must argue that, despite the changes in the graph, the expression associated to a node remains the same. To formalize such argument, and, consequently, define the relation *isNode*, we introduce a ghost variable γ to store the set of composite nodes of the graph. In fact, this variable γ stores an element of a well-chosen camera, such that the two new assertions can be introduced: (1) the assertion $isContext_\gamma\, K$, which states that the context K corresponds to the current set of composite nodes in the graph; and (2) the assertion $isBinding_\gamma\, B$, which states that the binding B belongs to the context representing the current state of the graph. Moreover, the camera is chosen to reflect the monotonic evolution of the computational graph during the forward phase: nodes are only added to the graph; a node is never removed. This ensures that the assertion *isBinding* is persistent, and, as we shall see, that the relation *isNode* is persistent as well.

For completeness, we quickly present the particular instance of the chosen camera and the definition of the assertions *isContext* and *isBinding*, but this information is not needed for understanding the remainder of the proof. Here is the chosen camera:

$$\textsc{Auth}(\mathit{Val} \xrightarrow{\text{fin}} \textsc{Ag}\,(\mathit{Val} \times \mathit{Op} \times \mathit{Val}))$$

The assertion $isContext_\gamma\, K$ claims ownership of $toMap\, K$, a finite map built from the context K, whereas the assertion $isBinding_\gamma\,(\mathsf{let}\ u = a\ op\ b)$ claims ownership of the fragment $\{u \mapsto \mathsf{ag}(a,\ op,\ b)\}$:

$$\begin{aligned}
\mathit{isContext}_\gamma\, K &\triangleq \boxed{\bullet\, \mathit{toMap}\, K}^{\gamma} \\
\mathit{isBinding}_\gamma\,(\mathsf{let}\ u = a\ op\ b) &\triangleq \boxed{\circ\, \{u \mapsto \mathsf{ag}(a,\ op,\ b)\}}^{\gamma} \\
\mathit{toMap}\,[] &\triangleq \varnothing \\
\mathit{toMap}\,(K +\!\!+ \mathsf{let}\ u = a\ op\ b) &\triangleq (\mathit{toMap}\, K)[u \mapsto \mathsf{ag}(a,\ op,\ b)]
\end{aligned}$$

The logical rules induced by this choice of camera appear in Figure 5.4. Rule (IntroduceContext) justifies the introduction of the ghost variable γ. We apply this rule only once, immediately before the execution of e, therefore γ is unique in the scope of this proof, and it is defined since line 30. Rule (IntroduceBinding) expresses the idea

that it is sound to add new composite nodes to the graph. Adding such a node yields a receipt $\mathit{isBinding}_\gamma\, B$ stating that B is part of the context representation of the graph. Rule (ClaimDefinedness) justifies the meaning of $\mathit{isBinding}_\gamma\, B$ as claiming that B belongs to the current context K. Indeed, if both the assertions $\mathit{isContext}_\gamma\, K$ and $\mathit{isBinding}_\gamma\, B$ hold, then $B \in K$. Rule (ShareBinding) states that *isBinding* is persistent.

With this piece of ghost state, we can finally introduce a persistent predicate *isNode* relating nodes to the their associated mathematical expression in the graph:

Definition 5.9 (*isNode*) *The predicate* $\mathit{isNode} : \mathit{Val} \to \mathit{Exp}_{\{X\}} \to \mathit{iProp}$ *is inductively defined as follows:*

$$
\begin{array}{rcl}
u\ \mathit{isNode}\ \mathit{Zero} & \triangleq & u = \mathit{zero}_{\mathcal{S}} \\
u\ \mathit{isNode}\ \mathit{One} & \triangleq & u = \mathit{one}_{\mathcal{S}} \\
u\ \mathit{isNode}\ (\mathit{Leaf}\ X) & \triangleq & u = x_{\mathcal{S}} \\
u\ \mathit{isNode}\ (E_a\ \mathit{op}\ E_b) & \triangleq & \exists\, a,\, b.\ \left\{ \begin{array}{l} \mathit{isBinding}_\gamma\, (\mathsf{let}\ u = a\ \mathit{op}\ b)\ * \\ a\ \mathit{isNode}\ E_a\ *\ b\ \mathit{isNode}\ E_b \end{array} \right.
\end{array}
$$

The case of constant nodes and of a variable node is straightforward: the nodes $\mathit{zero}_{\mathcal{S}}$, $\mathit{one}_{\mathcal{S}}$, and $x_{\mathcal{S}}$ are respectively associated to the expressions *Zero*, *One*, and $\mathit{Leaf}\ X$. The interesting case is that of a composite node u. Such a node is associated to the expression $E_a\ \mathit{op}\ E_b$, if there exist nodes a and b respectively associated to E_a and E_b, and if the binding $\mathsf{let}\ u = a\ \mathit{op}\ b$ is part of the context, a claim captured by the assertion $\mathit{isBinding}_\gamma\, (\mathsf{let}\ u = a\ \mathit{op}\ b)$.

Proof invariants. Having brought the notion of a computational graph to the forefront of the proof, by both introducing its formal representation as a context and introducing a ghost cell to keep track of its state, permitted the definition of the representation predicate *isNode*. As we shall see in this segment, this approach carries much further: the piece of ghost state and the notion of contexts are key in the definition of the two main proof invariants: (1) the *forward invariant* and (2) the *backward invariant*. The forward invariant specifies the meaning of the value component of intermediate variables during the forward phase, whereas the backward invariant specifies the meaning of the derivative component of intermediate variables during the backward phase. Let us discuss each of these invariants in further detail.

To assign a precise meaning to the value component of an intermediate variable u in a context K, the forward invariant exploits the definition of the *filling of* K *with* u, noted $K[u]$. The filling of K with u is the mathematical expression resulting from the suggestive interpretation of a binding B as a let-binding, and of a context K as a series of let-bindings. Here is the formalization of this definition:

Definition 5.10 (Filling) *The function* $\_\,[\_] : \mathit{Context} \to \mathit{Val} \to \mathit{Exp}_{\mathit{Val}}$ *is inductively defined as follows:*

$$
\begin{array}{rcll}
[][y] & = & \mathit{Leaf}\ y & \\
(K \mathbin{+\!\!+} \mathsf{let}\ u = a\ \mathit{op}\ b)[y] & = & (K[a])\ \mathit{op}\ (K[b]) & \mathit{if}\ u = y \\
(K \mathbin{+\!\!+} \mathsf{let}\ u = a\ \mathit{op}\ b)[y] & = & K[y] & \mathit{otherwise}
\end{array}
$$

Another way of seeing the filling of a context K with u is to think of a traversal on the computational graph denoted by K. The traversal starts in the node u and goes until the leaves, interpreting composite nodes found during the way as the operations they identify. This definition is similar to E_u, "*the mathematical expression associated to* u", a notion that we evoked in the previous section, and which is formalized by the representation predicate *isNode*. The definition of context filling then might seem superfluous, given that the predicate *isNode* is already available. This is not the case, the definition of context filling differs from this predicate in two key aspects: (1) context filling is a function in the meta logic, whereas *isNode* is a relation in *iProp*; and (2) the expression $K[u]$ has variables indexed by nodes, thereby permitting and inviting one to think about the meaning of the partial derivative of this expression with respect to a node. This second point is key to assign meaning to the derivative field of variables during the backward phase.

Here is the definition of the forward invariant:

Definition 5.11 (Forward invariant) *Let $\mathcal{R}$ be the semiring introduced in Hypothesis 5.1, and r be the $\mathcal{R}$ number analogously introduced. Let $zero_{\mathcal{S}}$, $one_{\mathcal{S}}$, and $x_{\mathcal{S}}$ be the values respectively introduced in lines 2, 3 and 28 of* `diff` *(Figure 5.3). Let γ be the ghost variable introduced by the application of rule (IntroduceContext) (Figure 5.4). The forward invariant is an assertion parameterized by a context K, noted ForwardInv K, and defined as follows:*

$$
\begin{aligned}
&\mathit{ForwardInv}\ K \triangleq \\
&\quad \mathit{isContext}_\gamma\, K\ *\ \left(\begin{array}{l} \forall u \in \{x_{\mathcal{S}}\} \cup \mathit{defs}(K). \\ \quad \mathit{leaves}(K[u]) \subseteq \{x_{\mathcal{S}}, \mathit{zero}_{\mathcal{S}}, \mathit{one}_{\mathcal{S}}\}\ * \\ \quad u\ \mathit{isVar}\ (\llbracket K[u] \rrbracket_\varrho,\ 0) \end{array}\right) \\
&\quad \text{where } \varrho : \mathit{Val} \to \mathcal{R} \triangleq (\lambda\_ .\ r)[\mathit{zero}_{\mathcal{S}} := 0][\mathit{one}_{\mathcal{S}} := 1]
\end{aligned}
$$

This definition relies on three simple definitions: (1) $\mathit{defs}(K)$, which computes the set of *variables defined* in a context K, that is, the set of variables u such that there exists a, b, and op with $\mathsf{let}\ u = a\ op\ b \in K$; (2) $\mathit{leaves}(E)$, which computes the set of leaves of E; and (3) $f[x := y]$; which overwrites the value of the function f at x with y.

The forward invariant *ForwardInv* K makes two assertions about the context K. First, it asserts that K corresponds to the current set of composite nodes in the graph – the assertion $\mathit{isContext}_\gamma\, K$ holds. Second, it asserts that K is *well-formed*: filling K with either $x_{\mathcal{S}}$ or a variable u defined in K yields an expression whose leaves are either $x_{\mathcal{S}}$, $\mathit{zero}_{\mathcal{S}}$, or $\mathit{one}_{\mathcal{S}}$. Moreover, the value component of such a variable u is the evaluation of $K[u]$ under the assignment ϱ, which assigns the leaves $\mathit{zero}_{\mathcal{S}}$, $\mathit{one}_{\mathcal{S}}$, and $x_{\mathcal{S}}$ to the numbers 0, 1, and r, respectively.

To state the backward invariant, there remains one last notion: the extension of an assignment $\varrho : \mathit{Val} \to \mathcal{R}$ with a context K, noted $\varrho\{K\}$. Here is its definition:

Definition 5.12 (Assignment extension) *The function $\_\{\_\} : (\mathit{Val} \to \mathcal{R}) \to \mathit{Context} \to (\mathit{Val} \to \mathcal{R})$ is defined as follows:*

$$\varrho\{K\} = \lambda\, u.\ \llbracket K[u] \rrbracket_\varrho$$

The assignment extension $\varrho\{K\}$ assigns the variable u to the evaluation of $K[u]$ under ϱ. It is said to extend ϱ, because, if u is not defined in K, then $\varrho\{K\}$ assigns u to $\varrho(u)$.

Now, all the machinery is in place to define the backward invariant:

Definition 5.13 (Backward invariant) *Let $\mathcal{R}$ be the semiring introduced in Hypothesis 5.1, and r be the $\mathcal{R}$ number analogously introduced. Let $zero_{\mathcal{S}}$, $one_{\mathcal{S}}$, and $x_{\mathcal{S}}$ be the values respectively introduced in lines 2, 3 and 28 of* `diff` *(Figure 5.3). The backward invariant, noted BackwardInv K_1 K_2 y, is an assertion parameterized by a pair of contexts K_1 and K_2 and by an intermediate variable y. It is defined as follows:*

$$
\begin{aligned}
&\mathit{BackwardInv}\ K_1\ K_2\ y\ \triangleq \\
&\quad \mathsf{let}\ K = K_1 +\!\!+ K_2\ \mathsf{in} \\
&\quad [\![E']\!]_{(\lambda X.\, r)} \equiv_{\mathcal{R}} \partial(K[y])/\partial x_{\mathcal{S}}\ (\varrho)\ * \\
&\quad \forall\, u \in \{x_{\mathcal{S}}\} \cup \mathit{defs}(K_1). \\
&\qquad u\ \mathit{isVar}\ (\_,\ \partial(K_2[y])/\partial u\ (\varrho\{K_1\})) \\
&\quad \textit{where}\ \varrho : \mathit{Val} \to \mathcal{R}\ \triangleq\ (\lambda \_.\, r)[\mathit{zero}_{\mathcal{S}} := 0][\mathit{one}_{\mathcal{S}} := 1]
\end{aligned}
$$

After the forward phase, the construction of the graph is complete, so its representation can be identified by a context K. During the backward phase, the nodes of K are treated in the reverse order in which they were introduced – that is, from right to left. The context K_2, by which the backward invariant is parameterized, specifies the set of treated nodes, whereas K_1 specifies the *pending* nodes (that is, the nodes that remain to be treated).

The variable y is the result of the execution of e in line 30. This variable represents the "root" node associated to the expression E. Indeed, it can be shown that filling K with y yields an expression equivalent to E modulo exchange of the leaves $zero_{\mathcal{S}}$, $one_{\mathcal{S}}$, and $x_{\mathcal{S}}$ with the expressions *Zero*, *One*, and *Leaf* X. However, the key idea of the backward invariant is to consider the filling of the *partial* context K_2 with y. The yielded expression, $K_2[y]$, has leaves indexed by variables defined in K_1, and leads to a crucial remark: the derivative field of a variable u, either defined in K_1 or equal to $x_{\mathcal{S}}$, is the partial derivative of $K_2[y]$ with respect to u. More precisely, the derivative component of such a variable u represents the number $\partial(K_2[y])/\partial u\ (\varrho\{K_1\})$, where ϱ is the assignment of the leaves $zero_{\mathcal{S}}$, $one_{\mathcal{S}}$, and $x_{\mathcal{S}}$ to the numbers 0, 1, and r.

At the end of the backward phase, K_2 refers to the complete context K and no variable is pending, so K_1 is empty. Therefore, the derivative component of $x_{\mathcal{S}}$ represents the number $\partial(K[y])/\partial x_{\mathcal{S}}\ (\varrho)$. Since `diff` terminates in line 50 by reading $x_{\mathcal{S}}$'s derivative component, this number must coincide with the evaluation of E' at r, the value that `diff` must compute (Goal 5.1). The first line of backward invariant confirms this expectation: $\partial(K[y])/\partial x_{\mathcal{S}}\ (\varrho)$ is equivalent to $[\![E']\!]_{(\lambda X.\, r)}$.

Assembling the proof. In this final segment, we discuss how to glue the presented pieces of the proof to build the complete verification of `diff`. In particular, we identify the application of the relevant logical rules.

Let us past forward the simple definitions of `diff`'s code, and start the discussion from line 28, at which point the operations $zero_{\mathcal{S}}$, $one_{\mathcal{S}}$, $add_{\mathcal{S}}$, $mul_{\mathcal{S}}$, and the variable $x_{\mathcal{S}}$ have already been defined. Then, without advancing the execution of the code, the ghost

variable γ is introduced by the application of rule (IntroduceContext). As a consequence of this application, the assertion $\mathit{isContext}_\gamma\ []$ holds at this point, intuitively meaning that the set of composite nodes of the computational graph is empty. It is easy to show that this assertion entails the forward invariant $\mathit{ForwardInv}\ []$.

The next step is then to reason about the execution of e under the handler of line 30. At this point, a glossed-over detail becomes apparent: the proof involves two protocols $\Psi_\mathcal{R}$ and $\Psi_\mathcal{S}$. The protocol $\Psi_\mathcal{R}$ specifies the effects that `diff` may perform by calling $\mathit{add}_\mathcal{R}$ or $\mathit{mul}_\mathcal{R}$. The protocol $\Psi_\mathcal{S}$ describes the effects performed by $\mathit{add}_\mathcal{S}$ and $\mathit{mul}_\mathcal{S}$, and specifies the effects that e may perform from applications to these functions.

The protocol $\Psi_\mathcal{R}$ was introduced in Hypothesis 5.1. We did not need to bother about this protocol for much of the proof, because most of Hazel reasoning rules *preserve protocols*; that is, the protocols in every occurrence of *ewp* in such a rule are the same.

The protocol $\Psi_\mathcal{S}$ is universally quantified in the assertion $e\ \mathit{isExp}\ E$, therefore its definition must be given as a requirement to reason about e. Fortunately, this task is straightforward. Since $\mathit{add}_\mathcal{S}$ and $\mathit{mul}_\mathcal{S}$ are each defined by a single effect-performing instruction, it suffices to rephrase the specifications of $\mathit{add}_\mathcal{S}$ and $\mathit{mul}_\mathcal{S}$ given by *isNumDict* (Figure 5.1) in the form of a send-receive protocol:

Definition 5.14 (Protocol $\Psi_\mathcal{S}$) *The functions $\mathit{add}_\mathcal{S}$ and $\mathit{mul}_\mathcal{S}$ abide by the protocol $\Psi_\mathcal{S}$, which is defined as follows:*

$$\begin{array}{rcll} \Psi_\mathcal{S} & \triangleq & !\ op\ a\ b\ E_a\ E_b & (op, a, b)\ \{a\ \mathit{isNode}\ E_a\ *\ b\ \mathit{isNode}\ E_b\}. \\ & & ?\ u & (u) & \{u\ \mathit{isNode}\ (E_a\ op\ E_b)\} \end{array}$$

By instantiating e's specification $e\ \mathit{isExp}\ E$ with the free semiring $\mathit{Exp}_{\{X\}}$ as $\mathcal{S}$, the above protocol $\Psi_\mathcal{S}$, and the predicate $\mathit{isNum}_\mathcal{S}$ as *isNode*, one can reason about the execution of e by means of the following *ewp* assertion (simplified by the rewriting rule $[\![E]\!]_{(\mathit{Leaf})} = E$):

$$\mathit{ewp}\ (e\ \mathit{zero}_\mathcal{S}\ \mathit{one}_\mathcal{S}\ \mathit{add}_\mathcal{S}\ \mathit{mul}_\mathcal{S}\ x_\mathcal{S})\ \langle\Psi_\mathcal{S}\rangle\{y.\ \exists T.\ y\ \mathit{isNode}\ T\ *\ T \equiv_{\mathit{Exp}_{\{X\}}} E\} \tag{5.2}$$

Now, let us reason about the expression resulting from the combination of handler and handlee (that is, the entire expression from lines 30 to 48). This expression must satisfy the following specification:

$$\begin{array}{l} \mathit{ForwardInv}\ [] \mathrel{-\!\!*} \\ \quad \mathit{ewp}\ (\texttt{deep-try}\ (e \ldots x_\mathcal{S})\ \texttt{with}\ (\text{lines 32 to 44} \mid \text{lines 46 to 48}))\ \langle\Psi_\mathcal{R}\rangle\{\_. \\ \qquad \exists K, y.\ \mathit{BackwardInv}\ []\ K\ y\} \end{array} \tag{5.3}$$

This specification states that, if the forward invariant $\mathit{ForwardInv}\ []$ initially holds, then after the execution of forward and backward phases, the backward invariant holds of variable y, resulting from the execution of e; and of complete context K. As argued in the previous segment, the assertion $\mathit{BackwardInv}\ []\ K\ y$ is sufficient to justify that the derivative component of $x_\mathcal{S}$ represents $[\![E']\!]_{(\lambda X.\ r)}$, the desired result of `diff`.

To derive Assertion 5.3, it suffices to apply rule TryWithDeep. This application yields two proof obligations. The first proof obligation asks for a specification of the

handlee, and it is easily dispatched by Assertion 5.2. The second proof obligation is the following handler judgment:

$$\begin{array}{l} \mathit{ForwardInv}\ [] \mathrel{-\!\!*} \\ \quad \mathit{deep\text{-}handler}\ \langle \Psi_{\mathcal{S}} \rangle\ \{y.\ \exists\, T.\ y\ \mathit{isNode}\ T\ *\ T \equiv_{\mathit{Exp}_{\{X\}}} E\} \\ \qquad\qquad (\text{lines } 32 \text{ to } 44 \mid \text{lines } 46 \text{ to } 48) \\ \qquad\qquad \langle \Psi_{\mathcal{R}} \rangle\ \{\_.\ \exists\, K,\ y.\ \mathit{BackwardInv}\ []\ K\ y\} \end{array} \tag{5.4}$$

To prove this judgment, we recast it under a slightly more general form:

$$\forall\, K_1.\ \left\{ \begin{array}{l} \mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \\ \quad \mathit{deep\text{-}handler}\ \langle \Psi_{\mathcal{S}} \rangle\ \{y.\ \exists\, T.\ y\ \mathit{isNode}\ T\ *\ T \equiv_{\mathit{Exp}_{\{X\}}} E\} \\ \qquad\qquad (\text{lines } 32 \text{ to } 44 \mid \text{lines } 46 \text{ to } 48) \\ \qquad\qquad \langle \Psi_{\mathcal{R}} \rangle\ \{\_.\ \exists\, K_2,\ y.\ \mathit{BackwardInv}\ K_1\ K_2\ y\} \end{array} \right. \tag{5.5}$$

As usual, the proof of Assertion 5.5 starts with the application of Löb's induction principle. Recall that the handler judgment (Figure 2.5 of Chapter 2) is the non-separating conjunction of the specifications of the effect and return branches. Therefore, both branches have access to the forward invariant.

The specification of the return branch (lines 46 to 48) unfolds to the following assertion, where K_1 and y are universally quantified and the intermediate variable y is the result of e's evaluation:

$$\mathit{ForwardInv}\ K_1 \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{update}\ y\ \mathit{one}_{\mathcal{R}})\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \exists\, K_2,\ y.\ \mathit{BackwardInv}\ K_1\ K_2\ y\}$$

The return branch takes control at the end of the forward phase, so the abstract context K_1 can be seen as the complete set of composite nodes constructed during this phase. The existentially quantified variables K_2 and y in the postcondition are thus respectively instantiated with [] with y. Choosing K_2 to be [] reflects the fact that no node has yet been treated.

The assertion $\mathit{BackwardInv}\ K_1\ []\ y$ consists of two claims.

One of these claims asserts that, for every variable u, the derivative component of u represents the number $\partial(\mathit{Leaf}\ y)/\partial u\ (\varrho\{K_1\})$. Therefore, one must consider two cases: (1) if u is equal to y, then u's derivative component must represent the number 1; and (2) if u is different from y, then u's derivative component must represent the number 0. Both of these assertions hold, because, after the execution of the return branch, the intermediate variable y is the only one whose derivative component is 1.

The other claim asserts the following equality:

$$[\![E']\!]_{(\lambda X.\, r)} \equiv_{\mathcal{R}} \partial(K_1[y])/\partial x_{\mathcal{S}}\ (\varrho) \tag{5.6}$$

The proof of this assertion exploits the fact that, according to e's postcondition (Equation 5.2), the variable y represents an expression T equivalent to E – there exists $T \in \mathit{Exp}_{\{X\}}$, such that $T \equiv_{\mathit{Exp}_{\{X\}}} E$ and the assertion $y\ \mathit{isNode}\ T$ holds. Then, by exploiting the fact that the context K_1 indeed corresponds to set of composite nodes in the graph – the assertion $\mathit{isContext}_\gamma\, K_1$ holds – one can prove that filling K_1 with y yields an expression whose partial derivative with respect to $x_{\mathcal{S}}$ is equivalent to $[\![T']\!]_{(\lambda X.\, r)}$:

$$\mathit{isContext}_\gamma\, K_1 \mathrel{-\!\!*} y\ \mathit{isNode}\ T \mathrel{-\!\!*} [\![T']\!]_{(\lambda X.\, r)} \equiv_{\mathcal{R}} \partial(K_1[y])/\partial x_{\mathcal{S}}\ (\varrho) \tag{5.7}$$

Assertion 5.7 follows by induction on T. The idea in the inductive case, when T is an operation of the form $E_a \ op \ E_b$, is to unfold the assertion $y \ isNode \ T$ (Definition 5.9) to obtain variables a and b, such that the assertions $a \ isNode \ E_a$, $b \ isNode \ E_b$, and $isBinding_\gamma \, (\mathsf{let} \ y = a \ op \ b)$ hold. One can then apply the inductive hypothesis to E_a and to E_b. Rule (ClaimDefinedness) is also useful in this proof to rewrite $K_1[y]$ as $(K_1[a]) \ op \ (K_1[b])$.

The second claim of the backward invariant (Claim 5.6) follows immediately from Assertion 5.7. It suffices to use the equivalence $T \equiv_{Exp_{\{X\}}} E$, which holds thanks to e's postcondition (Assertion 5.2) to rewrite T as E, thus completing the verification of the return branch.

Now comes the crux of the proof: the verification of the effect branch (lines 32 to 44). Indeed, as previously discussed, the forward and backward phases appear entangled in `diff`'s source code, so the proofs that the forward invariant and the backward invariant are preserved must be established in one stroke.

The effect branch handles arithmetic operations performed by e. An operation can be either *Add* or a *Mul*, so the handler further branches into these two cases. Let us concentrate on the verification of the *Add* branch (lines 35 to 38), as the verification of the *Mul* branch is analogous. The specification of the *Add* branch unfolds to the following assertion:

$$
\begin{array}{l}
ForwardInv \ K_1 \mathrel{-\!\!*} \\
\quad \forall \, a, \, b, \, E_a, \, E_b, \, k. \ a \ isNode \ E_a \mathrel{-\!\!*} b \ isNode \ E_b \mathrel{-\!\!*} \\
\quad \left(\begin{array}{l}
\quad \forall \, u, \, \Psi'', \, \Phi''. \\
\quad\quad \triangleright \, deep\text{-}handler \, \langle \Psi_{\mathcal{S}} \rangle \, \{y. \, \exists \, T. \ y \ isNode \ T \ * \ T \equiv_{Exp_{\{X\}}} E\} \ \mathrel{-\!\!*} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{(lines 32 to 44 } \mid \text{ lines 46 to 48)} \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle \Psi'' \rangle \, \{\Phi''\} \\
\qquad\quad u \ isNode \ (E_a \ Add \ E_b) \mathrel{-\!\!*} \\
\qquad\qquad ewp \ (k \ u) \ \langle \Psi'' \rangle \{\Phi''\}
\end{array} \right) \mathrel{-\!\!*} \\
\quad ewp \ (\text{lines 35 to 38}) \ \langle \Psi_{\mathcal{R}} \rangle \{\_. \, \exists \, K_2, \, y. \ BackwardInv \ K_1 \ K_2 \ y\}
\end{array}
\tag{5.8}
$$

The context K_1 and the forward invariant come from Assertion 5.5. Because the effect branch takes control during the forward phase, the context K_1 can be thought to represent an intermediate state of the graph. The universally quantified variables a, b, E_a, E_b, and k, as well as the pair of *isNode* assertions, and k's specification come from the unfolding of the protocol $\Psi_{\mathcal{S}}$. The variables a, b, and k occur free in the *Add* branch, and are thus (intentionally) captured by the universal quantifiers of Assertion 5.8.

The `mk` instruction in line 35 creates a new variable u, whose value component represents the sum of the numbers represented by the value components of a and b. At this point, the new binding

$$B \triangleq \mathsf{let} \ u = a \ Add \ b$$

must be added to the context K_1 to keep the ghost variable γ up to date. Therefore, the forward invariant $ForwardInv \ K_1$ is unfolded so the assertion $isContext_\gamma \, K_1$ is accessible. Then, by rule (IntroduceBinding), the assertion $isContext_\gamma \, K_1$ is updated to $isContext_\gamma \, (K_1 +\!\!+ B)$ and the new assertion $isBinding_\gamma \, B$ is forged.

The assertion $isContext_\gamma \, (K_1 +\!\!+ B)$ allows the proof of the invariant $ForwardInv \ (K_1 +\!\!+ B)$. The assertion $isBinding_\gamma \, B$ allows the proof that u represents the expression $E_a \ Add \ E_b$ – that is, the assertion $u \ isNode \ (E_a \ Add \ E_b)$ holds.

The next instruction is the invocation of the continuation in line 36. To reason about this instruction, we must exploit k's specification, which comes as one of the premises of Assertion 5.8. The universally quantified variables Ψ'', and Φ'' of k's specification are respectively instantiated with $\Psi_{\mathcal{R}}$ and $\lambda\ \_.\ \exists\, K_2,\ y.\ \mathit{BackwardInv}\ (K_1 +\!\!+ B)\ K_2\ y$. To fulfill the corresponding handler judgment that appears as one of the premises of k's specification, it suffices to apply Assertion 5.5, which was introduced as an inductive hypothesis by the application of Löb's induction principle. In this case, no execution step is necessary to apply this inductive hypothesis, because the handler judgment that appears in Assertion 5.8 is guarded by a "later" modality.

The application of the inductive hypothesis yields a new proof obligation: the assertion $\mathit{ForwardInv}\ (K_1 +\!\!+ B)$. This assertion and the remaining premise of k's specification, the assertion $u\ \mathit{isNode}\ (E_a\ \mathit{Add}\ E_b)$, have been shown to hold after the execution of the `mk` instruction (line 35). Therefore, every assertion in k's precondition is satisfied, and resuming the continuation with u meets the following specification:

$$\mathit{ewp}\ (k\ u)\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \exists\, K_2,\ y.\ \mathit{BackwardInv}\ (K_1 +\!\!+ B)\ K_2\ y\} \tag{5.9}$$

This specification shows that invoking the continuation is the watershed between forward and backward phases. Indeed, the effect branch takes control during the forward phase, at which point the composite nodes in the graph correspond to K_1. The contribution of this branch to the construction of the graph is the addition of the node u, thereby updating K_1 to $K_1 +\!\!+ B$. Then, control is relinquished to e, who continues the construction of the graph. In the end of the forward phase, the graph is represented by the complete context $(K_1 +\!\!+ B) +\!\!+ K_2$, where K_2 corresponds to the nodes further added by e. The backward phase begins and nodes are treated in the reverse order as they were added. Therefore, when the continuation finally terminates and transfers control back to the effect branch, only the nodes in K_2 have already been treated; the nodes $K_1 +\!\!+ B$ are pending. Indeed, as stated by the postcondition of the continuation (Assertion 5.9), the backward invariant $\mathit{BackwardInv}\ (K_1 +\!\!+ B)\ K_2\ y$ holds. Now, the goal is to show that, after the execution of the update instructions from lines 37 to 38, the node u is correctly treated; that is, the backward invariant $\mathit{BackwardInv}\ K_1\ (B :: K_2)\ y$ holds. In logical terms, this goal is expressed as follows:

$$\begin{array}{l} \mathit{BackwardInv}\ (K_1 +\!\!+ B)\ K_2\ y \mathrel{-\!\!*} \\ \quad \mathit{ewp}\ \left(\begin{array}{l} \texttt{update}\ a\ (\texttt{get\_d}\ u); \\ \texttt{update}\ b\ (\texttt{get\_d}\ u) \end{array} \right)\ \langle \Psi_{\mathcal{R}} \rangle \{\_.\ \mathit{BackwardInv}\ K_1\ (B :: K_2)\ y\} \end{array} \tag{5.10}$$

From the mechanistic perspective, these two lines of code have a simple behavior: they increment a's and b's derivative components with the derivative component of u. From the logical perspective, however, these two lines exploit an interesting mathematical fact. Indeed, while the assertion $\mathit{BackwardInv}\ (K_1 +\!\!+ B)\ K_2\ y$ describes the derivative component of intermediate variables in terms of the partial derivative of $K_2[y]$, the assertion $\mathit{BackwardInv}\ K_1\ (B :: K_2)\ y$ states this description in terms of the partial derivative of $(B :: K_2)[y]$. Therefore, these update instructions exploit a rewriting relation between the partial derivatives of the expressions $K_2[y]$ and $(B :: K_2)[y]$. This relation is formalized by the following lemma:

Lemma 5.1 (Context-filling rule) *Let $\mathcal{R}$ be a semiring. Let a, b, u, x, $y \in Val$ be intermediate variables. Let op be an operation. Let K_1 and K_2 be contexts. Let $\varrho : Val \to \mathcal{R}$ be an assignment. If $x \neq u$, then the partial derivative of $(\mathsf{let}\ u = a\ op\ b :: K_2)[y]$ with respect to x can be rewritten as follows:*

$$\begin{aligned}
&\partial(\mathsf{let}\ u = a\ op\ b :: K_2)[y]/\partial x\ (\varrho\{K_1\}) \equiv_{\mathcal{R}} \\
&\quad \partial K_2[y]/\partial x\ (\varrho\{K_1 \mathrel{+\!\!+} \mathsf{let}\ u = a\ op\ b\}) + \\
&\quad \partial K_2[y]/\partial u\ (\varrho\{K_1 \mathrel{+\!\!+} \mathsf{let}\ u = a\ op\ b\}) \times \partial(Leaf\ a)\ op\ (Leaf\ b)/\partial x\ (\varrho\{K_1\})
\end{aligned}$$

To properly discuss this lemma, let us take a quick detour and state the *chain rule* in the formalism of mathematical expressions that we have introduced:

Lemma 5.2 (Chain rule) *Let $\mathcal{R}$ be a semiring, and let $\mathcal{I}$ and $\mathcal{J}$ be indexing sets. Let $E \in Exp_{\mathcal{I}}$ be an expression with formal variables indexed by $\mathcal{I}$ and let $F : \mathcal{I} \to Exp_{\mathcal{J}}$ be an assignment of $\mathcal{I}$ to expressions with formal variables indexed by $\mathcal{J}$. Let $\vartheta : \mathcal{J} \to \mathcal{R}$ be an assignment of $\mathcal{J}$ to $\mathcal{R}$. The partial derivative of $[\![E]\!]_F$ with respect to an index $\jmath \in \mathcal{J}$ under ϑ can be rewritten as follows:*

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

The chain rule is a well-known rule of Calculus that states how to compute the derivative of the composition of differentiable functions. In our formalism of mathematical expressions, the chain rule states how to compute the partial derivative of the expression $[\![E]\!]_F$, obtained by the substitution of formal variables indexed by $\imath \in \mathcal{I}$ with $F(\imath)$. The expression $[\![E]\!]_F$ is thus the analogous of function composition in the traditional account of Calculus.

Lemma 5.1 is a specialized version of the chain rule where both indexing sets $\mathcal{I}$ and $\mathcal{J}$ are Val, and where the remaining variables E, F, ϑ, and $\jmath$ are specialized as follows:

$$\begin{aligned}
E : Exp_{Val} &\triangleq K_2[y] \\
F : Val \to Exp_{Val} &\triangleq \lambda x.\ \mathit{if}\ x = u\ \mathit{then}\ (Leaf\ a)\ op\ (Leaf\ b)\ \mathit{else}\ Leaf\ x \\
\vartheta : Val \to \mathcal{R} &\triangleq \varrho\{K_1\} \\
\jmath : Val &\triangleq x
\end{aligned}$$

The key observation, to see that this version of the chain rule coincides with Lemma 5.1, is that only two terms of the iterated sum in the chain rule are nonzero: the terms corresponding to the indices x and u. Indeed, for any u' different from both x and u, the value of $\partial F(u')/\partial x\ (\varrho\{K_1\})$ is 0. Moreover, for $u' = x$, its value is 1, and, for $u' = u$, its value is $\partial((Leaf\ a)\ op\ (Leaf\ b))/\partial x\ (\varrho\{K_1\})$. These are the terms appearing on the right-hand side of the equivalence of Lemma 5.1.

Let us resume the proof of Assertion 5.10 and consider a specialized version of Lemma 5.1 when op is Add. In this case, the term $\partial(Leaf\ a)\ Add\ (Leaf\ b)/\partial x\ (\varrho\{K_1\})$ simplifies either to 0, if $x \neq a$ and $x \neq b$; or to 1, if $a \neq b$, and $x = a$ or $x = b$; or to 2, if $x = a = b$. Therefore, the lemma states that, to compute the derivative $\partial(B :: K_2)[y]/\partial x\ (\varrho\{K_1\})$, it suffices to repeatedly increment $\partial K_2[y]/\partial x\ (\varrho\{K_1 \mathrel{+\!\!+} B\})$ with $\partial K_2[y]/\partial u\ (\varrho\{K_1 \mathrel{+\!\!+} B\})$ as many times as x appears in the list $[a, b]$. This is precisely the behavior achieved by the two update instructions of Assertion 5.10.

With the proof of Assertion 5.10, the proof of the generalised handler judgment (Assertion 5.5) is complete. (We have omitted the verification of the *Mul* branch, which is analogous to the verification of *Add* branch.) The generalised statement of the handler judgment easily entails the original one (Assertion 5.4).

To conclude the verification of `diff`, the final step is to verify the `get_v` instruction in line 50. This instruction is executed at the end of the backward phase, so the assertion $\mathit{BackwardInv}\ []\ K\ y$ holds at this point. After the execution of this line, the postcondition of `diff` must be established:

$$\mathit{BackwardInv}\ []\ K\ y \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{get\_v}\ x_{\mathcal{S}})\ \langle\Psi_{\mathcal{R}}\rangle\{y.\ \exists s.\ y\ \mathit{isNum}_{\mathcal{R}}\ s\ *\ s \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}\}$$

It follows from the backward invariant that the following assertion holds:

$$x_{\mathcal{S}}\ \mathit{isVar}\ (\_,\ \partial(K[y])/\partial x_{\mathcal{S}}\ (\varrho))\ \ *\ \ \partial(K[y])/\partial x_{\mathcal{S}}\ (\varrho) \equiv_{\mathcal{R}} [\![E']\!]_{(\lambda X.\ r)}$$

Therefore, reading $x_{\mathcal{S}}$'s derivative component yields a value representation of $[\![E']\!]_{(\lambda X.\ r)}$, thus concluding the verification of `diff`.

5.5 Related work

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

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

Sherman et al. [SMC21] focus on expressiveness: they wish to extend the applicability of automatic differentiation. They introduce λ_S, a programming language that includes higher-order functions, higher-order derivatives, and constructs for integration, root-finding and optimization. To assign meaning to λ_S programs, the authors follow a denotational approach. In doing so, they must solve the problem of interpreting programs, such as `max` and `min`, whose denotations are not differentiable everywhere. Their key idea

is to generalize the notion of derivative to *Clarke derivatives*, which are defined even at points where a function is not differentiable in the traditional sense. Therefore, automatic differentiation of λ_S programs is the computation of their Clarke derivatives. The paper discusses the implementation of a Haskell library to compute the Clarke derivatives, with arbitrary precision, of programs written in a Haskell embedding of λ_S.

An efficient implementation of effect handlers, with support for one-shot continuations only, has appeared in Multicore OCaml [SDW+21]. The *HH* implementation that we present (Figure 5.3) is inspired by Wang et al. [WR18, WZD+19] and by Sivaramakrishnan [Siv18]. However, the manner in which we package reverse-mode AD as a library, using a tagless final representation of expressions [CKS09, Kis10], seems new. This minimalist library is remarkable insofar as it offers a very simple, pure interface, yet its implementation is arguably rather subtle and involves dynamically-allocated mutable state, higher-order functions, and effect handlers. Another implementation of reverse-mode 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 several AD algorithms, including a forward-mode algorithm and a reverse-mode algorithm that performs checkpointing.

Chapter 6

A Separation Logic for Effect Handlers and Multi-Shot Continuations

In this chapter, we introduce Maze, a Separation Logic for effect handlers and multi-shot continuations; that is, a Separation Logic for reasoning about effect handlers that capture continuations that can be invoked multiple times.

As discussed in Chapter 2, the frame rule is unsound in the presence of multi-shot continuations. However, we show that Maze admits a *restricted* version of the frame rule, allowing users to apply this standard and powerful reasoning principle to fragments of the code that do not exploit effect handlers, but that may coexist with program fragments that do exploit this feature.

To assess the applicability of the logic, we consider two case studies: (1) the verification of a simple SAT solver using uses multi-shot continuations to implement backtracking, and (2) the assignment of reasoning rules to an encoding of `callcc` and `throw` using effect handlers and multi-shot continuations.

6.1 Syntax and semantics of MazeLang

In this section, we introduce MazeLang, a formal calculus for effect handlers and multi-shot continuations. Except for the one-shot policy on continuations, MazeLang follows the same design choices as *HH* does. So, apart from the discussion on "one-shot versus multi-shot", all the remarks raised in Subsection 2.1 of Chapter 2 apply to MazeLang. In particular, MazeLang has support for dynamically allocated mutable state; for unnamed effects; for shallow handlers, as a primitive construct; and for deep handlers, as a derived construct.

The syntax of MazeLang programs appears in Figure 6.1. It includes roughly the same constructs as *HH* does. There is only one new construct, which we highlight by rendering it in green. It is the construct representing first-class continuations. In *HH*, a first-class continuation assumes the form `cont` (ℓ, N), where ℓ is a memory location indicating whether the continuation has already been called. This is the device through which *HH* implements the one-shot policy. In MazeLang, we wish to lift this restriction, so first-class continuations assume the simpler form `cont` N, where the memory location ℓ is simply removed. For a discussion of the remaining constructs of the language, we refer the reader to Chapter 2.

The set of reduction rules of MazeLang appears in Figure 6.2. Again, the semantics of MazeLang follows roughly the same reduction rules as *HH* does, with the exception of

Values, expressions, and operations

$$
\begin{aligned}
\mathit{Op} \ni \odot &::= \texttt{+} \mid \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{==} \\
\mathit{Val} \ni h, r, v &::= \texttt{()} \mid b\ (\in \mathit{Bool}) \mid i\ (\in \mathit{Int}) \mid \ell\ (\in \mathit{Loc}) \mid \odot\ (\in \mathit{Op}) \\
&\quad \mid \texttt{rec}\ f\ x.\ e \mid (v, v) \mid \texttt{inj}_i\ v \mid \texttt{cont}\ N \\
\mathit{Expr} \ni e &::= v \mid x \mid e\ e \mid (e, e) \mid \texttt{proj}_i\ e \mid \texttt{inj}_i\ e \\
&\quad \mid \texttt{match}\ e\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{ref}\ e \mid \texttt{!}\, e \mid e \coloneqq e \\
&\quad \mid \texttt{do}\ e \mid \texttt{eff}\ v\ N \mid \texttt{try}\ e\ \texttt{with}\ (v \mid v)
\end{aligned}
$$

Evaluation contexts

$$
\begin{aligned}
\mathit{Ectx} \ni K &::= \bullet \mid e\ K \mid K\ v \mid (e, K) \mid (K, v) \mid \texttt{proj}_i\ K \mid \texttt{inj}_i\ K \\
&\quad \mid \texttt{match}\ K\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ K\ \texttt{then}\ e\ \texttt{else}\ e \\
&\quad \mid \texttt{ref}\ K \mid \texttt{!}\, K \mid e \coloneqq K \mid K \coloneqq v \mid \texttt{do}\ K \\
&\quad \mid \texttt{try}\ K\ \texttt{with}\ (v \mid v) \\
\mathit{Nctx} \ni N &::= \bullet \mid e\ N \mid N\ v \mid (e, N) \mid (N, v) \mid \texttt{proj}_i\ N \mid \texttt{inj}_i\ N \\
&\quad \mid \texttt{match}\ N\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ N\ \texttt{then}\ e\ \texttt{else}\ e \\
&\quad \mid \texttt{ref}\ N \mid \texttt{!}\, N \mid e \coloneqq N \mid N \coloneqq v \mid \texttt{do}\ N
\end{aligned}
$$

Figure 6.1: Syntax of MazeLang.

two rules, which, again, we highlight by rendering them in green: (1) rule MazeTryWithEffectStep, which states how evaluation contexts are captured and reified as first-class continuations; and (2) rule MazeResumeStep, which states how continuations can be invoked.

Under rule MazeTryWithEffectStep, when an active effect `eff` v N reaches a (shallow) handler, the handler's effect branch h is called with the value v and with the first-class continuation `cont` N. In contrast, under *HH*'s semantics, the context N is reified as `cont` (ℓ, N), where ℓ is a fresh memory location initially holding `false`.

Under rule MazeResumeStep, a continuation `cont` N is resumed when it is applied to a value v. In contrast, under *HH*'s semantics, when a continuation `cont` (ℓ, N) is applied to a value v, it must be the case that ℓ stores `false`, otherwise this instruction gets stuck. If ℓ stores `false`, then the continuation is resumed and the state of ℓ is updated to `true`. Therefore, whereas in *HH* continuations abide by a one-shot policy enforced by this location ℓ, the invocation of a continuation in MazeLang is unrestricted; it can happen multiple times.

For a discussion of the remaining reduction rules, we refer the reader to Chapter 2.

6.2 Program logic

The main idea of Hazel is the introduction of the notion of protocols, which, when combined with the Iris base logic, let us introduce an expressible specification language and powerful reasoning rules. Indeed, the previous chapters attest that protocols provide a natural way to reason about programs performing and handling effects. Therefore, we wish to keep protocols as the logical means by which one specifies the effects that a program may perform.

Reduction relation $\boxed{e \,/\, \sigma \;\to\; e \,/\, \sigma}$

MazeBetaStep
$(\texttt{rec}\, f\, x.\, e)\; v \,/\, \sigma \;\to\; e\{(\texttt{rec}\, f\, x.\, e)/f\}\{v/x\} \,/\, \sigma$

MazeProjStep
$\texttt{proj}_i\; (e_1, e_2) \,/\, \sigma \;\to\; e_i \,/\, \sigma$

MazeCaseStep
$\texttt{match}\; \texttt{inj}_i\; v\; \texttt{with}\; (v_1 \mid v_2) \,/\, \sigma \;\to\; v_i\; v \,/\, \sigma$

MazeIfStep
$\texttt{if}\; b\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \,/\, \sigma \;\to\; \mathit{if}\; (b = \texttt{true})\; \mathit{then}\; e_1\; \mathit{else}\; e_2 \,/\, \sigma$

MazeAllocStep
$$\frac{\ell \notin \mathit{dom}(\sigma)}{\texttt{ref}\; v \,/\, \sigma \;\to\; \ell \,/\, \sigma[\ell \mapsto v]}$$

MazeReadStep
$$\frac{\sigma(\ell) = v}{!\ell \,/\, \sigma \;\to\; v \,/\, \sigma}$$

MazeWriteStep
$$\frac{\ell \in \mathit{dom}(\sigma)}{\ell := v \,/\, \sigma \;\to\; () \,/\, \sigma[\ell \mapsto v]}$$

MazeDoStep
$\texttt{do}\; v \,/\, \sigma \;\to\; \texttt{eff}\; v\; \bullet \;/\, \sigma$

MazeEffStep
$$\frac{N_1 \neq \bullet}{N_1[\texttt{eff}\; v\; N_2] \,/\, \sigma \;\to\; \texttt{eff}\; v\; (N_1[N_2[\bullet]]) \,/\, \sigma}$$

MazeTryWithEffectStep
$\texttt{try}\; (\texttt{eff}\; v\; N)\; \texttt{with}\; (h \mid r) \,/\, \sigma \;\to\; h\; v\; (\texttt{cont}\; N) \,/\, \sigma$

MazeTryWithReturnStep
$\texttt{try}\; v\; \texttt{with}\; (h \mid r) \,/\, \sigma \;\to\; r\; v \,/\, \sigma$

MazeResumeStep
$(\texttt{cont}\; N)\; v \,/\, \sigma \;\to\; N[v] \,/\, \sigma$

MazeContextStep
$$\frac{e \,/\, \sigma \;\to\; e' \,/\, \sigma'}{K[e] \,/\, \sigma \;\to\; K[e'] \,/\, \sigma'}$$

Figure 6.2: Reduction rules of MazeLang.

However, something must change. If we simply replay the steps previously taken in the construction of *ewp*, then we would obtain the logic AwkwardHazel, which enjoys the same reasoning principles as Hazel does, but applies to programs written in MazeLang instead of *HH*. The logic AwkwardHazel would be sound, but too restrictive to reason about MazeLang programs. Indeed, whereas Hazel is well-suited for *HH*, there is a mismatch between MazeLang and AwkwardHazel: even though multi-shot continuations are allowed in MazeLang, the AwkwardHazel logic would apply only to fragments where continuations abide by a virtual one-shot policy. Indeed, since the frame rule is sound in AwkwardHazel, it would not be possible to apply the logic to programs exploiting multi-shot continuations, because multi-shot continuations break the frame rule (Chapter 2).

Recall that the upward closure is the key device by which Hazel reflects the one-shot discipline in the logic. Indeed, the following unfolding of the upward closure

$$(\uparrow \Psi)\; u\; \Phi' \;\triangleq\; \exists\, \Phi.\; \Psi\; u\; \Phi' \;*\; (\forall\, w.\; \Phi(w) \mathrel{-\!\!*} \Phi'(w))$$

Weakest precondition $\boxed{\mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\}}$

$$\mathit{ewp}\ v\ \langle\Psi\rangle\{\Phi\} \triangleq \dot{\Rightarrow}\,\Phi(v)$$

$$\mathit{ewp}\ (\texttt{eff}\ v\ N)\ \langle\Psi\rangle\{\Phi\} \triangleq (\uparrow_{\square}\Psi)\,v\,(\lambda w.\ \triangleright\,\mathit{ewp}\ N[w]\ \langle\Psi\rangle\{\Phi\})$$

$$\mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\} \triangleq \forall\sigma.\,S(\sigma) \Rrightarrow \begin{cases} \exists e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ * \\ \forall e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma' \Rrightarrow \triangleright \dot{\Rightarrow} \\ \qquad S(\sigma')\ *\ \mathit{ewp}\ e'\ \langle\Psi\rangle\{\Phi\} \end{cases}$$

Figure 6.3: Definition of Maze's *ewp*.

reveals that the "specification of the continuation", the predicate Φ', can be exploited at most once, because it appears as the conclusion of an affine assertion.

This observation hints to the idea of changing this definition to allow the predicate Φ' to be used multiple times. To achieve such a property in Separation Logic, the most natural solution is to guard the assertion $(\forall\, w.\ \Phi(w) \mathrel{-\!\!*} \Phi'(w))$ by a persistently modality. This idea works. It leads to the following alternative definition of the upward closure, named the *persistent upward closure*:

Definition 6.1 (Persistent upward closure) *The persistent upward closure of a protocol Ψ is defined as follows:*

$$\uparrow_{\square}\Psi \triangleq \lambda\, u\, \Phi'.\, \exists\, \Phi.\ \Psi\, u\, \Phi\ *\ (\square \forall\, w.\ \Phi(w) \mathrel{-\!\!*} \Phi'(w))$$

To assign the persistent upward closure a similar algebraic description as we did for the upward closure, we introduce the notion of *persistently monotonic* protocols:

Definition 6.2 (Persistently monotonic) *A protocol Ψ is persistently monotonic if the following sequent is derivable:*

$$\vdash \forall u, \Phi, \Phi'.\ (\square \forall\, w.\ \Phi(w) \mathrel{-\!\!*} \Phi'(w)) \mathrel{-\!\!*} \Psi\, u\, \Phi \mathrel{-\!\!*} \Psi\, u\, \Phi'$$

The persistent upward closure $\uparrow_{\square}\Psi$ can thus be characterized as the smallest persistently monotonic protocol that is greater than Ψ. The terms *smallest* and *greater* ought to be considered according to the protocol ordering (Definition 2.8).

The bottom protocol is persistently monotonic, and the sum of persistently monotonic protocols also is persistently monotonic. However, in general, an arbitrary send-receive protocol is not. To construct a persistently monotonic protocol in the style of send-receive protocols, we introduce *persistent send-receive protocols*:

Definition 6.3 (Persistent send-receive protocol) *Let $\vec{x}$ and $\vec{y}$ be lists of binders, let v and w be values, and let P and Q be assertions. The persistent send-receive protocol is defined as follows:*

$$\square\,!\,\vec{x}\,(v)\,\{P\}.\,?\,\vec{y}\,(w)\,\{Q\} \triangleq \lambda u\,\Phi.\,\exists\,\vec{x}.\ u = v\ *\ P\ *\ (\square \forall\,\vec{y}.\ Q \mathrel{-\!\!*} \Phi(w))$$

The persistent upward closure enjoys similar properties as the upward closure does. Here are some of the properties that are relevant to this chapter:

$$\textsc{MazeValue}\ \frac{\Phi(v)}{ewp\ v\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{MazeDo}\ \frac{(\uparrow_\square \Psi)\ v\ \Phi}{ewp\ (\texttt{do}\ v)\ \langle\Psi\rangle\{\Phi\}}$$

$$\textsc{MazeMonotonicity}\ \frac{ewp\ e\ \langle\Psi_1\rangle\{\Phi_1\} \qquad \Psi_1 \sqsubseteq \Psi_2 \qquad \square\,\forall w.\,\Phi_1(w) \mathrel{-\!\!*} \Phi_2(w)}{ewp\ e\ \langle\Psi_2\rangle\{\Phi_2\}}$$

$$\textsc{MazeMonotonicityPure}\ \frac{ewp\ e\ \langle\bot\rangle\{\Phi_1\} \qquad \forall w.\,\Phi_1(w) \mathrel{-\!\!*} \Phi_2(w)}{ewp\ e\ \langle\bot\rangle\{\Phi_2\}} \qquad \textsc{MazeBind}\ \frac{ewp\ e\ \langle\Psi\rangle\{w.\ ewp\ N[w]\ \langle\Psi\rangle\{\Phi\}\}}{ewp\ N[e]\ \langle\Psi\rangle\{\Phi\}}$$

$$\textsc{MazeBindPure}\ \frac{ewp\ e\ \langle\bot\rangle\{w.\ ewp\ K[w]\ \langle\Psi\rangle\{\Phi\}\}}{ewp\ K[e]\ \langle\Psi\rangle\{\Phi\}} \qquad \textsc{MazeTryWithShallow}\ \frac{ewp\ e\ \langle\Psi\rangle\{\Phi\} \qquad shallow\text{-}handler_M\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}{ewp\ (\texttt{try}\ e\ \texttt{with}\ (h \mid r))\ \langle\Psi'\rangle\{\Phi'\}}$$

$$\textsc{MazeTryWithDeep}\ \frac{ewp\ e\ \langle\Psi\rangle\{\Phi\} \qquad deep\text{-}handler_M\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}{ewp\ (\texttt{deep-try}\ e\ \texttt{with}\ (h \mid r))\ \langle\Psi'\rangle\{\Phi'\}}$$

Figure 6.4: Reasoning rules.

Lemma 6.1 (Properties of the persistent upward closure) *The persistent upward closure enjoys the following properties:*

1. *For every Ψ, the protocol $\uparrow_\square \Psi$ is persistently monotonic.*
2. *The persistent upward closure has no action over persistently monotonic protocols:*

$$\uparrow_\square \Psi \equiv \Psi \quad \textit{(for every pers. monotonic protocol } \Psi\textit{)}$$

3. *The persistent upward closure distributes over the protocol sum:*

$$\uparrow_\square (\Psi_1 + \Psi_2) \ \equiv\ (\uparrow_\square \Psi_1) + (\uparrow_\square \Psi_2) \quad \textit{(for every } \Psi_1 \textit{ and } \Psi_2\textit{)}$$

6.2.1 Weakest precondition and reasoning rules

Maze's notion of weakest precondition simply replaces the upward closure with the persistent upward closure. The same notation *ewp* as the one used in Hazel is used to denote Maze's weakest precondition. This overloading is bound to the scope of this chapter.

The definition of Maze's *ewp* appears in Figure 6.3. This definition differs from Hazel's definition of *ewp* in one single place (highlighted in green): the case of an active effect $\texttt{eff}\ v\ N$ uses the persistent upward closure $\uparrow_\square \Psi$ rather than $\uparrow \Psi$. Therefore, when performing an effect, a program should abide by the persistent upward closure $\uparrow_\square \Psi$.

There are two important consequences of this change to the reasoning principles of Maze.

$$\begin{array}{l}
\mathit{shallow\text{-}handler}_M\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\} \triangleq \\
\quad (\forall\, v.\ \Phi(v) \mathrel{-\!\!*} \mathit{ewp}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}) \wedge \\
\quad (\forall\, v,\ k.\ (\uparrow_{\square} \Psi)\, v\, (\lambda\, w.\ \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}) \mathrel{-\!\!*} \mathit{ewp}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\})
\end{array}$$

$$\begin{array}{l}
\mathit{deep\text{-}handler}_M\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi'\rangle\{\Phi'\} \triangleq \\
\quad (\forall v.\ \Phi(v) \mathrel{-\!\!*} \mathit{ewp}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\}) \wedge \\
\quad \left(\forall\, v,\ k.\ \left\{\begin{array}{l}
(\uparrow_{\square} \Psi)\, v\, (\lambda\, w.\ \forall\, \Psi'',\ \Phi''. \\
\quad \triangleright\, \mathit{deep\text{-}handler}_M\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle\Psi''\rangle\{\Phi''\} \mathrel{-\!\!*} \\
\qquad \mathit{ewp}\ (k\ w)\ \langle\Psi''\rangle\{\Phi''\}) \\
\quad \mathrel{-\!\!*} \\
\mathit{ewp}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\}
\end{array}\right.\right)
\end{array}$$

Figure 6.5: Definitions of the predicates $\mathit{shallow\text{-}handler}_M$ and $\mathit{deep\text{-}handler}_M$.

The first consequence is that every non-persistent assertion *vanishes* when a program performs an effect; that is, non-persistent assertions cannot be framed around programs that perform effects. In other words, non-persistent assertions that hold before a `do` v instruction, do not hold when the program fragment that performed this effect is resumed. This restriction is in agreement with the fact that, in MazeLang, when a program performs an effect, this program is susceptible of being resumed multiple times.

The second consequence is the desired reasoning principle allowing one to reason about effect handlers that invoke continuations multiple times. This expressive power is attested by the study of MazeLang reasoning rules (in particular, the reasoning rules for shallow and deep handlers), which we shall discuss next.

The reasoning rules induced by Maze's definition of *ewp* appear in Figure 6.4. As usual, we highlight in green the changes with respect to the rules of Hazel. We narrow the discussion to the rules that include such changes.

Rule MazeDo formalizes the claim that "non-persistent assertions vanish when performing effects". Indeed, after unfolding the definition of the persistent upward closure in this rule, it is easy to see that, to reason about the instruction `do` v, one must establish a goal of the form $\square \forall\, w.\ Q(w) \mathrel{-\!\!*} \Phi(w)$. Because this assertion is guarded by a persistently modality, non-persistent assertions in the proof context cannot be used to derive such a goal.

Rule MazeMonotonicity differs from rule Monotonicity (Figure 2.4 of Chapter 2) in the inclusion of a persistently modality in the weakening relation between the postconditions Φ_1 and Φ_2: $\square \forall w.\, \Phi_1(w) \mathrel{-\!\!*} \Phi_2(w)$. This modification is necessary, because, in the presence of multi-shot continuations, a program e might terminate multiple times. For each possible execution of e, this premise must be exploited to weaken e's postcondition. As a consequence, the frame rule no longer holds for an arbitrary protocol Ψ:

$$R * \mathit{ewp}\ e\ \langle\Psi\rangle\{\Phi\} \vdash \mathit{ewp}\ e\ \langle\Psi\rangle\{w.\, R * \Phi(w)\}$$

The following restricted statement of the frame rule, however, *is sound*:

$$R * \mathit{ewp}\ e\ \langle\bot\rangle\{\Phi\} \vdash \mathit{ewp}\ e\ \langle\bot\rangle\{w.\, R * \Phi(w)\}$$

The derivation of this rule follows from the straightforward application of rule MazeMonotonicityPure. Rule MazeMonotonicityPure is an alternative monotonicity

reasoning principle, which is incomparable to rule MazeMonotonicity. It applies to programs that abides by the empty protocol, thus it cannot be applied in contexts where programs perform effects. In its premise, the weakening relation between the postconditions Φ_1 and Φ_2 is not guarded by a persistently modality, therefore non-persistent assertions *can* be used to establish this relation.

Rule MazeTryWithShallow allows reasoning about shallow handlers. The rule states that, if the handlee e abides by a protocol Ψ and a postcondition Φ, then the expression resulting from installing a shallow handler over e conforms to a protocol Ψ' and a postcondition Φ'. The protocol Ψ describes e's effects, whereas the protocol Ψ' describes the effects that the shallow handler itself may perform. The claim that the answer provided by the handler to e's effects is in accordance with the agreed protocol Ψ is captured by the shallow-handler judgment *shallow-handler*$_M$. Its definition appears in Figure 6.5. As Hazel's handler judgment *shallow-handler* (Figure 2.5 of Chapter 2), it is defined as the non-separating conjunction of the specifications of the return and the effect branches. The only difference with respect to Hazel's version of the judgment is the by-now-customary use of the persistent upward closure instead of the upward closure. That is how the logic allows one to reason about multiple calls to the continuation. Indeed, to see how the logic allows reasoning about multiple continuation invocations, it suffices to unfold the definition of the persistent upward closure in the specification of the effect branch:

$$\forall\, v,\, k.\ (\uparrow_{\Box} \Psi)\, v\, (\lambda\, w.\ \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}) \mathrel{-\!\!*} \mathit{ewp}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\}$$

becomes

$$\forall\, v,\, k.\ (\exists\, Q.\ \Psi\, v\, Q\ *\ (\Box \forall\, w.\ Q(w) \mathrel{-\!\!*}\ \mathit{ewp}\ (k\ w)\ \langle\Psi\rangle\{\Phi\})) \mathrel{-\!\!*} \mathit{ewp}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\}$$

This unfolding reveals that the persistently modality introduced by the persistent upward closure guards the specification of the continuation k. Therefore, during the verification of the effect branch h, a user can exploit the specification of the continuation k multiple times, and thereby reason about multiple invocations.

Rule MazeTryWithDeep allows reasoning about deep handlers. Similar to the reasoning rule for shallow handlers, rule MazeTryWithDeep states that installing a deep handler over an expression e has the consequence of shifting the protocol Ψ and the postcondition Φ to Ψ' and Φ'. The handlee performs effects according to Ψ, and the handler must also conform to this protocol when replying to these requests. The claim that the handler conforms to Ψ is expressed by the deep-handler judgment *deep-handler*$_M$, whose definition appears in Figure 6.5. Like in the case of the shallow-handler judgment, the only difference with respect to Hazel's version of the deep-handler judgment is the use of the persistent upward closure, which allows a user of the logic to reason about multiple calls to a continuation when verifying the handler's effect branch.

6.2.2 Soundness

Here is the statement of Maze's adequacy theorem, which states that reasoning about programs in terms of *ewp* is sound:

Theorem 6.1 (Adequacy) *Let e be a closed expression. If ewp e $\langle\bot\rangle\{\Phi\}$ holds, then e is safe.*

The theorem states that, if a user of the logic establishes that the assertion $ewp\ e\ \langle\bot\rangle\{\Phi\}$ holds, then e is safe. Recall the definition of *safe* (Section 2.3.3 of Chapter 2): the execution of e must either diverge or terminate with a value; it cannot crash or terminate with an unhandled effect.

Proof By unfolding both Hazel's and Maze's definitions of *ewp*, it is easy to see that these two notions coincide when the specified protocol is $\bot$. In particular, the Maze assertion $ewp_{Maze}\ e\ \langle\bot\rangle\{\Phi\}$ entails the Hazel assertion $ewp_{Hazel}\ e\ \langle\bot\rangle\{\Phi\}$:

$$ewp_{Maze}\ e\ \langle\bot\rangle\{\Phi\} \mathrel{-\!\!*} ewp_{Hazel}\ e\ \langle\bot\rangle\{\Phi\}$$

Therefore, to complete the proof, it suffices to apply the adequacy theorem of the Hazel logic (Theorem 2.1 of Chapter 2).

6.3 Case studies

This section puts Maze into practice by studying its application to two case studies.

In the first case study, we present the verification of a simple SAT solver. Even though the solver is concisely implemented, its behavior is quite intricate. We are nonetheless capable of writing a simple specification of this program accompanied of an equally simple proof of correctness.

In the second case study, we devise reasoning rules that tame the infamous undelimited-control operators `callcc` and `throw`. We study an encoding of these operators in MazeLang. The encoding exploits a *toplevel* handler, which is assumed to be the topmost frame in the evaluation stack. Under this assumption, `callcc` and `throw` can be encoded in MazeLang as simple effect-performing instructions.

6.3.1 Verifying a simple SAT solver

Implementation

The implementation of the SAT solver appears in Figure 6.6. The solver is presented as a function `satisfy`. The informal contract by which this function abides is straightforward: when queried with a *propositional formula*, the function `satisfy` returns a Boolean indicating whether this formula is *satisfiable*.

A propositional formula can be constructed in one of three ways: it is either the conjunction of two formulas p_1 and p_2, noted $\texttt{And}\,(p_1, p_2)$; or the disjunction of two formulas p_1 and p_2, noted $\texttt{Or}\,(p_1, p_2)$; or a *literal*, which consists of a pair of a Boolean *sign* b and an index i, and which is noted $\texttt{Lit}\,(b, i)$.

An index i specifies a Boolean variable x_i, and a sign b indicates whether the truth value of the literal $\texttt{Lit}\,(b, i)$ corresponds to the value x_i or to its negation $\neg x_i$.

Propositional formulas can be represented in MazeLang using binary sums:

$$\begin{array}{rcl} \texttt{And}\,(p_1, p_2) & \triangleq & \texttt{inj}_1\ (\texttt{inj}_1\ (p_1, p_2)) \\ \texttt{Or}\,(p_1, p_2) & \triangleq & \texttt{inj}_1\ (\texttt{inj}_2\ (p_1, p_2)) \\ \texttt{Lit}\,(b, i) & \triangleq & \texttt{inj}_2\ (b, i) \end{array}$$

```
let rec interp p =
  match p with
  ( fun pair -> match pair with
      ( fun (* And. *) (p1, p2) ->
          (interp p1) and (interp p2)
      | fun (* Or.  *) (p1, p2) ->
          (interp p1) or  (interp p2)
      )
  | fun (* Lit. *) (b, i) ->
      if b then (do i) else neg (do i)
  )

let satisfy p =
  let m = create_map() in
  deep-try (interp p) with
  (* Effect branch. *)
  ( fun i k ->
      match lookup m i with
      ( fun (* Some. *) b ->
          k b
      | fun (* None. *) _ ->
          (insert m i true; k true)
             or
          (insert m i false;
             (k false)
                or
             (delete m i; false)
          )
      )
  (* Return branch. *)
  | fun b -> b
  )
```

Figure 6.6: A simple SAT solver in MazeLang.

Informally, a propositional formula p is satisfiable if there exists an *assignment* $\varphi : \mathit{Int} \xrightarrow{\text{fin}} \mathit{Bool}$ [1] (*i.e.*, a mapping from variable indices i to Booleans), such that the *interpretation of* p *under* φ is `true`. The purpose of the solver `satisfy` is thus to decide whether such an assignment exists.

The question we wish to answer is: at least from the mechanistic point of view, can we explain how does `satisfy` achieve that? Fortunately, the dynamic behavior of `satisfy` has a striking similarity with an episode that took place in an exquisite school. The director of this school, Professor Chandler, was fascinated by SAT solvers, and wished to devise a system of his own. However, to achieve this task, he did not want to rent computing hours from some fast machine; Professor Chandler knew a far cheaper currency: students' hours. The Professor would often place a student in a special desk containing a propositional formula, and would ask the student to compute the interpretation of this formula. Of course, in general, such a task cannot be realized if the formula contains unassigned literals, so the desk was equipped with two buttons: one labeled *lookup* and the other one labeled *exit*.

If the student pressed the lookup button, then the Professor would reach the student, and the student would have the right to ask the Professor the assignment of an index i. The Professor listed these indices in a private blackboard, together with the Boolean answer he provided to each index. If the Professor had already provided an answer to an index i, then he would simply reply to the student with the corresponding Boolean, and would not alter the board. If, on the other hand, the index i did not appear in the board, then the Professor would add i to the right-end of the list together with the Boolean `true`, which was the answer that he would give by default.

When the student was done interpreting the formula, thus finding a Boolean b, the student would press the exit button. If b was `true`, then the student was free and the formula was deemed satisfiable. If b was `false`, then things would go differently. Professor Chandler would first erase from the blackboard the greatest suffix of indices whose assignment was `false`. If the resulting list was empty, then the student was free and the formula was deemed unsatisfiable. Otherwise, the Professor would change the assignment of the right-most index i from `true` to `false` and would kindly ask the student two things: (1) to forget the indices that were previously in the blackboard and the old assignment of i, and (2) to restart the computation. The student had no other option, but to accept this demand, and the game would go on until the Professor had determined whether or not the formula was satisfiable.

The runtime behavior of the solver `satisfy` is indeed similar to this game. This solver calls the function `interp` (line 1), which computes the interpretation of a given formula p. During this computation, if `interp` stumbles upon a literal with index i, then it performs an effect with payload i. The student represents the function `interp`, and pressing the lookup button represents the action of performing an effect. The Professor represents the handler installed by `satisfy` in line 15 to monitor the execution of `interp`.

When the handler intercepts an effect thrown by `interp` carrying a payload i, the handler performs a lookup in a map m (line 18) assigning indices to Booleans. This

[1] The type $A \xrightarrow{\text{fin}} B$ denotes partial functions. A partial function $f : A \xrightarrow{\text{fin}} B$ has a domain $\mathit{dom}(f) \subseteq A$, such that, for every $a \in A$, the term $f(a)$ is defined if and only if $a \in \mathit{dom}(f)$. We write $f(a) = \mathit{undefined}$ to indicate that f is not defined at a.

map is allocated in line 14, before the execution of `interp`. The Professor's blackboard represents this map.

The lookup operation returns a binary sum indicating whether m contains the index i. In the affirmative case, `interp` is immediately resumed with the Boolean b (line 20) associated to i. In the negative case, the handler inserts the key-value pair $(i, \texttt{true})$ to the map, and resumes `interp` with `true`.

If `interp` evaluates to `true`, then it means that p is satisfiable, therefore the handler terminates with `true`. The specification of this returned Boolean is that the current assignment stored in m can be extended in such a way that the interpretation of p is `true`.

If `interp` evaluates to `false`, then, under the assignment of i to `true`, the map cannot be extended in such a way that the interpretation of p is `true`. However, since continuations are multi-shot, the handler can implement a *backtracking* technique by resuming `interp` a second time, but with a different answer. In line 24, the handler overwrites the map with the key-value pair $(i, \texttt{false})$ and resumes `interp` with `false`.

The result of the handler is the result of `interp`'s evaluation. If `interp` evaluates to `false`, however, the handler has also a side effect: it erases the binding $(i, \texttt{false})$ from the map.

Without the deletion of the index i, further queries performed by `interp` would be incorrectly answered by the handler with `false`. The following formula, for example, would be deemed as unsatisfiable, if this deletion instruction was not performed; that is, if lines 25 to 27 were replaced with the instruction `k false`:

$$\texttt{Or}\,(\texttt{And}\,(\texttt{Lit}\,(\texttt{true}, 0), \texttt{Lit}\,(\texttt{false}, 0)), \texttt{And}\,(\texttt{Lit}\,(\texttt{true}, 1), \texttt{Lit}\,(\texttt{false}, 0)))$$

To improve readability, let us rewrite this formula using a more natural notation:

$$(x_0 \wedge \neg x_0) \vee (x_1 \wedge \neg x_0)$$

Because the literal x_0 appears first, the first query of `interp` is the index 0. The handler answers with `true`, and writes the binding $(0, \texttt{true})$ to the map. The second query of `interp` is the index 1. After trying both possible answers, `true` and `false`, the handler backtracks to the index 0 in a state where the map contains the binding $(1, \texttt{false})$, which was the last attempt of answer to the index 1. The handler modifies the value associated to 0 from `true` to `false`, and resumes `interp` with `false`. Now, because the map still contains the binding of $(1, \texttt{false})$, further queries to the index 1 are immediately replied with `false`. As a result, the handler never explores the assignment of 0 to `false` and of 1 to `true`. So the formula is deemed unsatisfiable, even though, under this assignment, its interpretation is `true`.

Specification

Recall `satisfy`'s informal contract: `satisfy` decides whether a given a formula p is satisfiable. The formal specification of `satisfy` rephrases this informal contract in the language of the Maze logic:

Statement 6.1 (Formal specification of `satisfy`) *The specification of* `satisfy` *is expressed as follows:*

$$\mathit{ewp}\ (\texttt{satisfy}\ p)\ \langle\bot\rangle\{b.\ b = \mathit{satisfiable} \varnothing p\}$$

This statement depends on the binary predicate *satisfiable*, which relates an assignment φ to a formula p if φ can be extended with an assignment φ', such that the interpretation of p under this extended assignment is `true`. In the degenerate case where φ is the empty assignment, such as in the postcondition of `satisfy`, the assertion $\mathit{satisfiable}\,\varnothing\,p$ captures the notion of a satisfiable formula.

To introduce *satisfiable*, we must first formally define *the interpretation of formula under an assignment* and *the extension of an assignment.*

Definition 6.4 (Formula interpretation) *The interpretation of a formula is expressed by the function* $[\![\_]\!]_\_ : \mathit{Formula} \to (\mathit{Int} \xrightarrow{\mathit{fin}} \mathit{Bool}) \xrightarrow{\mathit{fin}} \mathit{Bool}$*, inductively defined as follows:*

$$\begin{array}{rcl} [\![\texttt{And}\,(p_1, p_2)]\!]_\varphi & \triangleq & [\![p_1]\!]_\varphi \wedge [\![p_1]\!]_\varphi \\ [\![\texttt{Or}\,(p_1, p_2)]\!]_\varphi & \triangleq & [\![p_1]\!]_\varphi \vee [\![p_1]\!]_\varphi \\ [\![\texttt{Lit}\,(b, i)]\!]_\varphi & \triangleq & \mathit{if}\ b\ \mathit{then}\ \varphi(i)\ \mathit{else}\ \neg\varphi(i) \end{array}$$

If a formula p contains a literal $\texttt{Lit}\,(\_, i)$*, such that $i \notin \mathit{dom}(\varphi)$, then $[\![p]\!]_\varphi$ is undefined:* $[\![p]\!]_\varphi \triangleq \mathit{undefined}$

The definition of the interpretation of a formula is straightforward. Perhaps one unusual detail is that the function $[\![\_]\!]_\_$ is partial: the interpretation of a formula p under an assignment φ is undefined when there exists at least one literal $\texttt{Lit}\,(\_, i)$ in p, such that $\varphi(i)$ is undefined. With a partial definition, the meaning of the equality $[\![p]\!]_\varphi = b$ is twofold: not only it asserts that the interpretation of p under φ is b, but it also asserts that $\mathit{dom}(\varphi)$ includes every literal in p. (This is not crucial though: the interpretation could be defined as a total function, by giving a default value to unassigned literals, and asserting, when necessary, that every literal in p is included in $\mathit{dom}(\varphi)$.)

Definition 6.5 (Assignment extension) *The extension of an assignment* $\_ +\!\!+ \_ : (\mathit{Int} \xrightarrow{\mathit{fin}} \mathit{Bool}) \to (\mathit{Int} \xrightarrow{\mathit{fin}} \mathit{Bool}) \to (\mathit{Int} \xrightarrow{\mathit{fin}} \mathit{Bool})$ *is defined as follows:*

$$(\varphi +\!\!+ \varphi')(i) \triangleq \begin{cases} \varphi(i) & \mathit{if}\ i \in \mathit{dom}(\varphi) \\ \varphi'(i) & \mathit{if}\ i \notin \mathit{dom}(\varphi)\ \mathit{and}\ i \in \mathit{dom}(\varphi') \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

The extension of φ with φ' is basically the union of these maps, with *priority* given to φ where their domains overlap; that is, if $i \in \mathit{dom}(\varphi) \cap \mathit{dom}(\varphi')$, then $(\varphi +\!\!+ \varphi')(i) = \varphi(i)$.

We are finally in position to introduce the formal definition of *satisfiable*:

Definition 6.6 (Satisfiable formula) *Let φ be an assignment. Let p be a formula. The proposition "satisfiable $\varphi\,p$" states that φ can be extended with an assignment φ', such that the interpretation of p under $\varphi +\!\!+ \varphi'$ is defined and evaluates to* `true`*:*

$$\mathit{satisfiable}\,\varphi\,p \triangleq \exists\,\varphi'.\ [\![p]\!]_{(\varphi +\!\!+ \varphi')} = \texttt{true}$$

The proposition $\mathit{satisfiable}\,\varphi\,p$ inhabits the universe of meta-level assertions *Prop*. However, the postcondition of `satisfy` (Specification 6.1) uses $\mathit{satisfiable}\,\varphi\,p$ as a Boolean. This apparent misuse is justified by the fact that *satisfiable* is *decidable*:

Lemma 6.2 (Decidability of *satisfiable*) *For every assignment φ and formula p, the proposition "satisfiable $\varphi\,p$" is decidable.*

$$isMap_{[A,B]} : Val \to (A \xrightarrow{\mathsf{fin}} B) \to iProp$$

CreateMap
$ewp\ (\texttt{create\_map()})\ \langle\bot\rangle\{m.\ isMap\, m\, \varnothing\}$

Insert
$isMap\, m\, \varphi \mathrel{-\!\!*} ewp\ (\texttt{insert}\ m\ i\ b)\ \langle\bot\rangle\{\_.\ isMap\, m\, \varphi[i \mapsto b]\}$

Delete
$isMap\, m\, \varphi \mathrel{-\!\!*} ewp\ (\texttt{delete}\ m\ i)\ \langle\bot\rangle\{\_.\ isMap\, m\, (\varphi \setminus i)\}$

Lookup
$$isMap\, m\, \varphi \mathrel{-\!\!*} ewp\ (\texttt{lookup}\ m\ i)\ \langle\bot\rangle \left\{ y.\ isMap\, m\, \varphi\ * \left(\begin{array}{l} \mathit{if}\ i \in dom(\varphi)\ \mathit{then} \\ \quad y = \texttt{inj}_1\ (\varphi(i)) \\ \mathit{else} \\ \quad y = \texttt{inj}_2\ () \end{array} \right) \right\}$$

Figure 6.7: Specification of a map library.

A meta-level proposition $P : Prop$ is decidable if either P is provable or $\neg P$ is provable. To show that *satisfiable* $\varphi\, p$ is decidable, for every φ and p, we show the existence of a *decision procedure*: a method that, for every φ and p, tells whether *satisfiable* $\varphi\, p$ is provable. The aforementioned misuse of the proposition *satisfiable* $\varphi\, p$ as a Boolean in `satisfy`'s postcondition is thus justified by an implicit coercion that applies *satisfiable* $\varphi\, p$ to its decision procedure, which produces a Boolean result. [2]

At a high-level perspective, this procedure works in three steps. First, it finds all the literals in p that do not have an assignment in φ. Because p is finite, there can only be a finite number of such literals, say n. Second, it enumerates all possible 2^n assignments of these n literals. Third, and finally, it checks whether, among these 2^n assignments, there exists at least one assignment φ', such that the interpretation of p under the extension of φ with φ' is `true`. In the affirmative case, the proposition *satisfiable* $\varphi\, p$ is provable, and, otherwise, it is not.

Intuitively, this procedure is the meta-level equivalent of the solver `satisfy`. Because it inhabits the level of mathematical functions, it does not need to care about questions of efficiency; the fact that it terminates, because it performs an exhaustive search over a finite set, is already good enough.

Verification

Let p be a formula. In this segment, we present the proof that `satisfy` p meets its formal specification (Specification 6.1): that this expression produces a Boolean b, such that $b =$ *satisfiable* $\varnothing\, p$.

[2] An alternative that avoids this implicit coercion would be to perform a case distinction on the result b of the solver as follows: *if* $b =$ `true` *then satisfiable* $\varnothing\, p$ *else* $\neg$*satisfiable* $\varnothing\, p$. However, we prefer the implicit coercion, because it leads to a concise specification and to concise assertions during the proof.

$$\text{(IntroduceBlackboard)} \qquad \mathsf{True} \mathrel{-\!\!*} \dot{\Rrightarrow} \exists \gamma.\ \mathit{blackboardSt}_\gamma\, \varnothing \ *\ \mathit{studentView}_\gamma\, \varnothing$$

$$\text{(CheckBlackboard)} \qquad \mathit{blackboardSt}_\gamma\, \varphi \mathrel{-\!\!*} \mathit{studentView}_\gamma\, \varphi' \mathrel{-\!\!*} \varphi = \varphi'$$

$$\text{(EraseBlackboard)} \qquad \begin{array}{r} \mathit{blackboardSt}_\gamma\ \_ \mathrel{-\!\!*} \\ \mathit{studentView}_\gamma\ \_ \mathrel{-\!\!*} \end{array} \quad \dot{\Rrightarrow} \left\{ \begin{array}{l} \mathit{blackboardSt}_\gamma\, \varphi\ * \\ \mathit{studentView}_\gamma\, \varphi \end{array} \right.$$

Figure 6.8: Logical rules governing the assertions *blackboardSt* and *studentView*

Map operations. The verification of `satisfy` does not depend on the internals of the map operations (`create_map`, `insert`, `delete`, `lookup`); it simply assumes that these operations can be used according to the logical interface that appears in Figure 6.7.

This interface exposes an abstract representation predicate *isMap* that relates MazeLang values to finite maps $\varphi : A \xrightarrow{\mathsf{fin}} B$, where the types A and B appear as parameters of *isMap*. In the scope of this proof, φ is always an assignment, so the types A and B are always *Int* and *Bool*, respectively. Therefore, to reduce the visual clutter, we omit these type parameters.

Let us discuss the specifications of the map operations that appear in Figure 6.7. Specification CreateMap states that calling the operation `create_map` yields a value m representing the empty map: the assertion $\mathit{isMap}\, m\, \varnothing$ holds. Specification Insert states that the instruction `insert` $m\ i\ b$ inserts the binding (i, b) to the map represented by m (previous inserted bindings are overwritten). Specification Delete states that the instruction `delete` $m\ i$ erases i from the map. Specification Lookup states that the lookup instruction `lookup` $m\ i$ produces a binary sum y: if i is in the map represented by m, then $y = \mathtt{inj}_1\ (\varphi(i))$; otherwise, $y = \mathtt{inj}_2\ ()$.

Ghost state and the *Lookup* protocol. There are two key ideas in the proof. The first one is the introduction of a piece of ghost state to ensure that both the handlee (the expression `interp` p) and handler agree on the assignment φ. The second one is the introduction of a protocol to describe `interp`'s effects.

The informal reason why the ghost state is necessary is that `interp` returns multiple times, so one cannot assign `interp` a postcondition asserting that it computes the interpretation of p under a fixed φ. Each time `interp` returns, it computes the interpretation of p under a different assignment φ'. Therefore, when writing `interp`'s postcondition, we quantify over φ' existentially. To express that φ' is equal to the handler's private assignment φ (represented by the value m allocated in line 14), we introduce a ghost cell γ that stores an assignment φ and we claim that the state of γ is φ'.

To formalize this claim, the contents of γ are taken in the following camera:

$$\textsc{Auth}(\textsc{Ex}(\mathit{Int} \xrightarrow{\mathsf{fin}} \mathit{Bool}))$$

This choice of camera lets us introduce two different assertions whose meaning is the same: they both assert what is the state of γ. One shall be used to describe the handler's view of γ, while the other shall be used to describe the handlee's view. Building on Professor Chandler's analogy, the assertion representing the handler's view is noted *blackboardSt*,

and the assertion representing the handlee's view is noted *studentView*. Here are their definitions:

$$\begin{aligned} \mathit{blackboardSt}_\gamma\,\varphi &\triangleq \boxed{\bullet\,\mathsf{ex}(\varphi)}^{\gamma} \\ \mathit{studentView}_\gamma\,\varphi &\triangleq \boxed{\circ\,\mathsf{ex}(\varphi)}^{\gamma} \end{aligned}$$

The logical rules that are induced by this choice of camera and that govern the assertions *blackboardSt* and *studentView* appear in Figure 6.8. Rule (IntroduceBlackboard) allows the introduction of γ. Initially, the state of γ is the empty assignment $\varnothing$, therefore, the assertions $\mathit{blackboardSt}_\gamma\,\varnothing$ and $\mathit{studentView}_\gamma\,\varnothing$ hold. Rule (CheckBlackboard) formalizes the claim that both *blackboardSt* and *studentView* assert what is the state of γ. Therefore, if both the assertions $\mathit{blackboardSt}_\gamma\,\varphi$ and $\mathit{studentView}_\gamma\,\varphi'$ hold, then $\varphi = \varphi'$. Rule (EraseBlackboard) allows one to update the state of γ, provided one has the possession of both the assertions *blackboardSt* and *studentView*.

The remaining ingredient to state the specification of `interp` is the definition of the protocol by which `interp` abides. Recall that `interp` performs effects to look up the assignment of literals in the formula it must interpret. During such lookup requests, the handler might update this assignment, and, consequently, might update the state of γ. Therefore, it seems natural to propose the following protocol:

Definition 6.7 (Protocol *Lookup*) *The protocol Lookup is defined as follows:*

$$\begin{aligned} \mathit{Lookup} \triangleq\ & \square\ !\,i\,\varphi\,(i)\,\{\mathit{studentView}_\gamma\,\varphi\}. \\ & \quad ?\,b\,(b)\,\{\mathit{studentView}_\gamma\,(\varphi[i \mapsto b])\ *\ (\varphi[i \mapsto b])(i) = b\} \end{aligned}$$

This protocol states that, when performing effects, `interp` transfers ownership of the assertion $\mathit{studentView}_\gamma\,\varphi$ to the handler, and, upon return, the state of γ might have been updated to $\varphi[i \mapsto b]$. The assignment $\varphi[i \mapsto b]$ is the extension of φ with the singleton assignment $\{i \mapsto b\}$; that is, $\varphi[i \mapsto b] \triangleq \varphi +\!\!+ \{i \mapsto b\}$. So, even though the handler can choose with which value b to resume the continuation, this choice is limited depending on whether or not $i \in \mathit{dom}(\varphi)$.

If $i \in \mathit{dom}(\varphi)$, then the two assignments φ and $\varphi[i \mapsto b]$ are equal. The equality $(\varphi[i \mapsto b])(i) = b$ then simplifies to $\varphi(i) = b$, thus constraining the choice of b to $\varphi(i)$. Therefore, if $i \in \mathit{dom}(\varphi)$, then the handler cannot update the assignment and must resume the continuation with $\varphi(i)$.

If $i \notin \mathit{dom}(\varphi)$, then the handler can choose any value b, because, in this case, the equality $(\varphi[i \mapsto b])(i) = b$ trivially holds. Therefore, if $i \notin \mathit{dom}(\varphi)$, then the handler can update the assignment and resume the continuation with any value b (and can do so several times).

With the introduction of this protocol, the specification of `interp` can be stated:

Lemma 6.3 (Specification of `interp`) *The function* `interp` *admits the following specification:*

$$\forall\, p.\ \mathit{studentView}_\gamma\,\varnothing \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{interp}\ p)\ \langle \mathit{Lookup}\rangle\{b.\ \exists\,\varphi'.\ \mathit{studentView}_\gamma\,\varphi'\ *\ [\![p]\!]_{\varphi'} = b\}$$

Proof Because the function `interp` is recursive, it is natural to consider a generalization of its original specification:

$$\forall\, p,\ \varphi.\ \left\{ \begin{array}{l} \mathit{studentView}_\gamma\,\varphi \mathrel{-\!\!*} \\ \quad \mathit{ewp}\ (\texttt{interp}\ p)\ \langle \mathit{Lookup}\rangle\{b.\ \exists\,\varphi'. \\ \qquad \mathit{studentView}_\gamma\,(\varphi +\!\!+ \varphi')\ *\ [\![p]\!]_{(\varphi +\!\!+ \varphi')} = b\} \end{array} \right. \tag{6.1}$$

Assertion 6.1 follows by induction on p. Since Lemma 6.3 is a special case of this assertion, the proof is complete. ∎

At first, the specification of `interp` might seem weak, because it asserts that `interp` computes the interpretation of p under *some* assignment φ', but it does not tell which assignment. As we shall see, this specification is sufficiently strong for the purposes of the proof of `satisfy`, because, the assertion $\mathit{studentView}_\gamma\,\varphi'$ ensures that φ' agrees with the state of the handler's private map. Therefore, even though φ' is existentially quantified, the handler knows that φ' is equal to the current assignment φ, for which the assertion $\mathit{blackboardSt}_\gamma\,\varphi$ holds

Assembling the proof. Now, we discuss how the verification of `satisfy` builds upon the notions previously introduced.

The execution of `satisfy` p starts with the allocation of the map m (line 14) initially representing the empty assignment; that is, the assertion $\mathit{isMap}\,m\,\varnothing$ holds. At the same time, the ghost cell γ is introduced by application of rule (IntroduceBlackboard). Therefore, both the assertions $\mathit{blackboardSt}_\gamma\,\varnothing$ and $\mathit{studentView}_\gamma\,\varnothing$ hold at this point in the code.

The next and final step of `satisfy` is the handler-monitored execution of `interp` p in line 15. To reason about this step, it suffices to apply rule MazeTryWithDeep with the following instances of the variables Ψ, Ψ', Φ, and Φ':

$$\begin{array}{rcl}
\Psi : \mathit{Protocol} & \triangleq & \mathit{Lookup} \\
\Psi' : \mathit{Protocol} & \triangleq & \bot \\
\Phi : \mathit{Val} \to \mathit{iProp} & \triangleq & \lambda\, b.\ \exists\, \varphi'.\ \mathit{studentView}_\gamma\,\varphi' \ast [\![p]\!]_{\varphi'} = b \\
\Phi' : \mathit{Val} \to \mathit{iProp} & \triangleq & \lambda\, b. \left\{ \begin{array}{l} b = \mathit{satisfiable}\,\varnothing\,p \ast \\ \left(\begin{array}{l} \mathit{if}\ b = \texttt{false}\ \mathit{then} \\ \quad \mathit{blackboardSt}_\gamma\,\varnothing \ast \mathit{studentView}_\gamma\,\varnothing \ast \mathit{isMap}\,\varnothing \end{array} \right) \end{array} \right.
\end{array}$$

The postcondition Φ' of the handler reclaims ownership of the assertions *blackboardSt*, *studentView*, and *isMap*, if b is `false`. If b is `true`, then these assertions can be discarded, because the algorithm terminates. if b is `false`, however, the algorithm might not have explored all the possible assignments, so the search continues.

The application of MazeTryWithDeep with this choice of variables yields two proof obligations.

The first proof obligation is the specification of `interp`:

$$\mathit{ewp}\ (\texttt{interp}\ p)\ \langle \mathit{Lookup} \rangle \{ b.\ \exists\, \varphi'.\ \mathit{studentView}_\gamma\,\varphi' \ast [\![p]\!]_{\varphi'} = b \}$$

This proof obligation is easily dispatched by Lemma 6.3 in combination with the assertion $\mathit{studentView}_\gamma\,\varnothing$.

The second proof obligation is the following deep-handler judgment:

$$\begin{array}{l}
\mathit{blackboardSt}_\gamma\,\varnothing \mathrel{-\!\!*} \mathit{isMap}\,m\,\varnothing \mathrel{-\!\!*} \\
\quad \mathit{deep\text{-}handler}_M\ \langle \mathit{Lookup} \rangle\ \{ b.\ \exists\, \varphi'.\ \mathit{studentView}_\gamma\,\varphi' \ast [\![p]\!]_{\varphi'} = b \} \\
\qquad\qquad (\text{lines } 17 \text{ to } 29 \mid \texttt{fun b -> b}) \\
\qquad\qquad \langle \bot \rangle \left\{ b.\ \left\{ \begin{array}{l} b = \mathit{satisfiable}\,\varnothing\,p \ast \\ \left(\begin{array}{l} \mathit{if}\ b = \texttt{false}\ \mathit{then} \\ \quad \mathit{blackboardSt}_\gamma\,\varnothing \ast \mathit{studentView}_\gamma\,\varnothing \ast \mathit{isMap}\,m\,\varnothing \end{array} \right) \end{array} \right. \right\}
\end{array}$$

This assertion is fulfilled by the application of the following lemma, which generalizes this proof obligation:

Lemma 6.4 *The following handler judgment holds:*

$$\begin{array}{l}\forall\,\varphi.\ \mathit{blackboardSt}_\gamma\,\varphi \mathrel{-\!\!*} \mathit{isMap}\,m\,\varphi \mathrel{-\!\!*} \\ \quad \mathit{deep\text{-}handler}_M\ \langle \mathit{Lookup}\rangle\ \{b.\ \exists\,\varphi'.\ \mathit{studentView}_\gamma\,\varphi' \ast [\![p]\!]_{\varphi'} = b\} \\ \qquad\qquad \text{(lines 17 to 29 | } \texttt{fun b -> b)} \\ \qquad\qquad \langle\bot\rangle \left\{ b.\ \left(\begin{array}{l} b = \mathit{satisfiable}\,\varphi\,p \ \ast \\ \mathit{if}\ b = \texttt{false}\ \mathit{then} \\ \quad \mathit{blackboardSt}_\gamma\,\varphi \ \ast \\ \quad \mathit{studentView}_\gamma\,\varphi \ \ast \\ \quad \mathit{isMap}\,m\,\varphi \end{array} \right) \right\}\end{array}$$

The statement of this lemma is the specification of the handler. It essentially says that, regardless of the state of the assignment φ, the handler correctly replies to the requests of `interp`. Moreover, the postcondition states that the handler computes the truth value of $\mathit{satisfiable}\,\varphi\,p$, and that, if this value is `false`, then it leaves no trace of the modifications it performed on φ. To meet this postcondition, the handler has access the ghost cell γ, through the assertion $\mathit{blackboardSt}_\gamma\,\varphi$; and to the physical representation of φ, the value m, through the assertion $\mathit{isMap}\,m\,\varphi$.

As usual, because the statement of Lemma 6.4 is a deep-handler judgment, the proof of this statement starts with the application of Löb's induction principle. Next, the definition of the judgment (Figure 6.5) is unfolded, revealing two simpler goals: the specification of the return branch and the specification of the effect branch.

The specification of the return branch corresponds to the following assertion:

$$\begin{array}{l}\forall\,b.\ \mathit{blackboardSt}_\gamma\,\varphi \mathrel{-\!\!*} \mathit{isMap}\,m\,\varphi \mathrel{-\!\!*} (\exists\,\varphi'.\ \mathit{studentView}_\gamma\,\varphi' \ast [\![p]\!]_{\varphi'} = b) \mathrel{-\!\!*} \\ \quad \mathit{ewp}\ ((\texttt{fun b -> b})\ b)\ \langle\bot\rangle\ \{b.\ b = \mathit{satisfiable}\,\varphi\,p \ \ast\ (\mathit{if}\ b = \texttt{false}\ \mathit{then} \\ \quad \mathit{blackboardSt}_\gamma\,\varphi \ \ast\ \mathit{studentView}_\gamma\,\varphi \ \ast\ \mathit{isMap}\,m\,\varphi)\}\end{array} \tag{6.2}$$

The proof of Assertion 6.2 consists of two key reasoning steps: (1) the application of rule (CheckBlackboard), to prove that the existentially quantified assignment φ' coincides with φ; and (2) the proof of the equality $b = \mathit{satisfiable}\,p\,\varphi$. To prove this equality, one must recall the fact that $[\![\_]\!]\_$ is a partial function. Because $[\![p]\!]_\varphi$ is defined, every literal in p must be in $\mathit{dom}(\varphi)$. Therefore, the value b of $[\![p]\!]_\varphi$ determines whether $\mathit{satisfiable}\,p\,\varphi$ holds, because any extension of φ does not have an effect on the interpretation of p.

The specification of the effect branch corresponds to the following assertion (where line 17 is excluded from the concluding *ewp* clause so that the variables i and k occur free in the handler branch and are thus intentionally bound by the leading quantifiers):

$$\forall\, i,\, k,\, \varphi'.\ \mathit{blackboardSt}_\gamma\, \varphi \mathrel{-\!\!*} \mathit{studentView}_\gamma\, \varphi' \mathrel{-\!\!*} \mathit{isMap}\, m\, \varphi \mathrel{-\!\!*}$$
$$\left(\begin{array}{l} \square \forall\, b,\, \Psi'',\, \Phi''. \\ \quad \triangleright\, \mathit{deep\text{-}handler}_M\ \langle \mathit{Lookup} \rangle\ \{b.\ \exists\, \varphi'.\ \mathit{studentView}_\gamma\, \varphi' \ *\ [\![p]\!]_{\varphi'} = b\} \\ \qquad\qquad (\text{lines } 17 \text{ to } 29 \mid \texttt{fun b -> b}) \\ \qquad\qquad \langle \Psi'' \rangle\ \{\Phi''\} \quad \mathrel{-\!\!*} \\ \qquad \mathit{studentView}_\gamma\, (\varphi'[i \mapsto b]) \mathrel{-\!\!*} (\varphi[i \mapsto b])(i) = b \mathrel{-\!\!*} \\ \qquad\quad \mathit{ewp}\ (k\ b)\ \langle \Psi'' \rangle \{\Phi''\} \end{array}\right) \mathrel{-\!\!*} \qquad (6.3)$$
$$\mathit{ewp}\ (\text{lines } 18 \text{ to } 29)\ \langle \bot \rangle\ \{b.\ b = \mathit{satisfiable}\, \varphi\, p \ *\ (\mathit{if}\ b = \texttt{false}\ \mathit{then}$$
$$\mathit{blackboardSt}_\gamma\, \varphi \ *\ \mathit{studentView}_\gamma\, \varphi \ *\ \mathit{isMap}\, m\, \varphi)\}$$

This statement is, in fact, a polished version of the specification of the effect branch from the definition of the handler judgment. In particular, we have exploited that *Lookup* is persistently monotonic to simplify the persistent upward closure, and we have unfolded the definition of *Lookup*. The assignment φ', the assertion $\mathit{studentView}_\gamma\, \varphi'$, the equality $(\varphi[i \mapsto b])(i) = b$, and the specification of k result from these omitted steps of rewriting.

The proof of Assertion 6.3 begins with application of rule (CheckBlackboard) to rewrite φ' as φ.

Because the specification of k is guarded by a persistently modality, it can be applied to reason about the two invocations of k (lines 22 and 25).

Before each invocation of k with a Boolean b, Specification Insert is exploited in combination with logical rule (EraseBlackboard) to insert a binding (i, b) in the assignment φ, so that the assignment is updated both at the program level, the assertion $\mathit{isMap}\, m\, (\varphi[i \mapsto b])$ holds, and at the logical level, the assertions $\mathit{blackboardSt}_\gamma\, (\varphi[i \mapsto b])$ and $\mathit{studentView}_\gamma\, (\varphi[i \mapsto b])$ hold.

The protocol and postcondition, Ψ'' and Φ'', universally quantified in the specification of k are respectively specialized with $\bot$ and with a version of the effect-branch postcondition where φ is substituted with $\varphi[i \mapsto b]$, where b is the handler's reply with which k is resumed:

$$\Phi'' : \mathit{Val} \to \mathit{iProp} \ \triangleq\ \lambda\, b'.\ \left\{\begin{array}{l} b' = \mathit{satisfiable}\, (\varphi[i \mapsto b])\, p \ * \\ \left(\begin{array}{l} \mathit{if}\ b' = \texttt{false}\ \mathit{then} \\ \quad \mathit{blackboardSt}_\gamma\, (\varphi[i \mapsto b]) \ * \\ \quad \mathit{studentView}_\gamma\, (\varphi[i \mapsto b]) \ * \\ \quad \mathit{isMap}\, (\varphi[i \mapsto b]) \end{array}\right) \end{array}\right.$$

Therefore, the predicate Φ'' is the postcondition assigned to the expression "$k\ b$". So, the result of the continuation tells whether $\mathit{satisfiable}\, (\varphi[i \mapsto b])\, p$ holds.

If at least one invocation of k returns `true`, then the effect branch returns `true`. To show that this value satisfies the postcondition of the effect branch, it suffices to show the following (trivial) logical implication:

$$(\exists\, b.\ \mathit{satisfiable}\, (\varphi[i \mapsto b])\, p) \implies \mathit{satisfiable}\, \varphi\, p$$

If none of the two invocations of k returns `true`, then the effect branch returns `false`. The postcondition of the effect branch in this case consists of two goals: (1) the proof

that $\mathit{satisfiable}\,\varphi\,p$ does not hold, and (2) the proof that the assertions $\mathit{blackboardSt}_\gamma\,\varphi$, $\mathit{studentView}_\gamma\,\varphi$, and $\mathit{isMap}\,m\,\varphi$ hold. To establish the first goal, it suffices to show the following (trivial) logical implication:

$$(\forall\, b.\, \neg \mathit{satisfiable}\,(\varphi[i \mapsto b])\,p) \implies \neg \mathit{satisfiable}\,\varphi\,p$$

To establish the second goal, it suffices to exploit Specification Delete in combination with logical rule (EraseBlackboard) to delete i from the assignment $\varphi[i \mapsto b]$ both at the program level, so that the assertion $\mathit{isMap}\,m\,\varphi$ holds, and at the logical level, so that the assertions $\mathit{blackboardSt}_\gamma\,\varphi$ and $\mathit{studentView}_\gamma\,\varphi$ hold.

With the proof of Assertion 6.2 and Assertion 6.3, the proof of Lemma 6.4 is complete, thus concluding the verification of `satisfy`.

6.3.2 Reasoning about `callcc` and `throw`

The operator `callcc` is a programming construct introduced by the Scheme programming language [sch]. It has its roots in Landin's `J` operator [Lan98], and is now supported by languages such as Ruby [Rub] and Racket [Fla21b]. The operational behavior of `callcc` and `throw` can be described in a straightforward manner by stating the two following reduction rules:

$$\begin{array}{lrcl} \text{(CalleeStep)} & K[\texttt{callcc}\ k.\, e] & \to & K[e\{\widetilde{K}/k\}] \\ \text{(ThrowStep)} & K[\texttt{throw}\ \widetilde{K'}\ v] & \to & K'[v] \end{array}$$

The expression in the left-hand side denotes the complete program to which these reduction rules apply, so K denotes the entire evaluation context. The term $\widetilde{K}$ denotes the reification of K as a value. According to rule (CalleeStep), the instruction $\texttt{callcc}\ k.\, e$ proceeds in two steps. First, the context K is captured and reified as the value $\widetilde{K}$. Then, this value $\widetilde{K}$ is substituted for the binder k in the expression e, which assumes control. According to rule (ThrowStep), the instruction $\texttt{throw}\ \widetilde{K'}\ v$ replaces the current evaluation context K with a previously captured one K'; that is, the current context is erased and the captured one is installed.

Because `callcc` captures the entire evaluation context, instead of just a delimited fragment (as it is the case with effect handlers, for example), `callcc` and `throw` are called *undelimited-control* operators and continuations captured by `callcc` are called *undelimited continuations*. Moreover, because `throw` erases the current evaluation context, the captured continuation $\widetilde{K}$ is *non-composable* [FYFF07]. Continuations captured by effect handlers, in contrast, are *composable*: when a continuation is invoked, $K[(\texttt{cont}\ K')\ v]$, the captured context K' *composes* with the current context K: the program $K[(\texttt{cont}\ K')\ v]$ reduces to $K[K'[v]]$.

It has been extensively argued that undelimited-control operators and non-composable continuations are not good programming abstractions. Indeed, Kiselyov builds a strong argument against the adoption of `callcc` [Kis12a]. He attacks this programming construct under multiple axes: safety, efficiency, reasoning principles, expressiveness, and practicability. From the safety axis, he argues that undelimited continuations lead to memory leaks, because, in most applications, only a fragment of the evaluation context is needed, so capturing the whole evaluation context keeps more memory live than it is

necessary. From the expressiveness axis, undelimited continuations can be seen as being delimited by a fictional topmost frame enclosing the complete program. Undelimited continuations can thus be emulated by delimited continuations. Moreover, even though the converse also is true (Filinski [Fil94] shows that "`callcc` and a single mutable cell" can implement the delimited-control operators `shift` and `reset`), Kiselyov argues that this encoding is fragile: it works under the strong assumption that `shift` and `reset` are the only source of control effects. Even the use of exceptions breaks this assumption.

In this case study, we wish to investigate whether Maze can be applied to reason about `callcc` and `throw`. We show that, assuming programs execute under a distinguished *toplevel* frame delimiting the evaluation context, it is possible to implement `callcc` and `throw` in MazeLang, using effect handlers and multi-shot continuations. We then show that it is indeed possible to ascribe Maze reasoning rules to these constructs. One can thus apply all of Maze reasoning principles to `callcc` and `throw`. Surprisingly, this shows that `callcc` and `throw` are compatible with *context-local reasoning*: it is sound to apply the bind rule in the presence of these constructs. The main limitation to reasoning is the unsoundness of the frame and monotonicity rules, a limitation inherited from Maze due to multi-shot continuations. This seems to contradict Timany and Birkedal [TB19], who claim that the bind rule is unsound in the presence of `callcc`. We compare our work to Timany and Birkedal's at the end of this case study, where we explain that there is no contradiction.

Programming with `callcc` and `throw`

Before we present the implementation of `callcc` and `throw` in MazeLang and their reasoning rules, let us build some experience of programming with these constructs. Leroy [Ler18] makes the suggestive claim that "`callcc` *is the* `goto` *of lambda calculus*". Madore [Mad02] makes a similar claim: he suggests that `callcc` is a "*dynamic* `goto`". These claims rely on the intuition that an evaluation context $K[\bullet]$ can be seen as a pointer to the *position* of the hole $\bullet$. From this perspective, the operation `callcc` $k.\, e$ allows the expression e to gain access to its own position via the binder k, and the operation `throw` $k\ x$ allows a program to return to the position denoted by k.

A typical application of `callcc` building on this intuition is to simulate exceptions. When searching for the first element x in a data structure that satisfies a predicate p, for example, one could use `callcc` to stop the search as soon as this element has been found. The following example implements this strategy:

```
let list_find xs p = callcc k.
  list_iter xs (fun x -> if p x then throw k (Some x)); None
```

The function `list_find` uses `callcc` to record its *call site*, that is, the position where it has been called. Then, it begins the search for an element in xs that satisfies the predicate p. It performs this search by calling the higher-order iteration method `list_iter` (Chapter 3) with a function that tests whether a given element x satisfies p. In the affirmative case, the operation `throw` $k\ x$ is performed, thus stopping the iteration and forcing `list_find` to immediately return this element.

A more advanced application of `callcc` and `throw` than the one previously discussed is to implement the conversion of eager iteration methods into lazy sequences; that is, to implement the function `invert` from Chapter 1:

```
let invertcc iter = fun () -> callcc kc.
  let r = ref kc in
  let yield x =
    callcc kp. throw !r (cons (x, fun () ->
      callcc kc. r := kc; throw kp ()))
  in
  iter yield; throw !r nil
```

The main novelty of this example is the introduction of a reference r to store continuations. This is a typical idiom when programming with `callcc` and `throw`, because `throw` discards the current evaluation context K. So, to avoid the complete loss of K, it is natural to store the reification of K in a reference, before performing a `throw` operation.

To understand the implementation of `invertcc`, it is useful to think in terms of two communicating agents: the *consumer* and the *producer*.

The consumer calls `invertcc`: it is a context expecting a *sequence head*; that is, either a `Nil` value, or a `Cons` pair containing an element x and a sequence of the remaining elements. At the beginning of `invertcc`'s execution, a `callcc` instruction is used to obtain the position of the consumer *kc*, with which the reference r is initialized.

The producer represents the function `invertcc`. It calls `iter` with a function *yield* that jumps back to the consumer every time *yield* stumbles upon a new element x. However, just before performing this jump, *yield* uses `callcc` to obtain its own position *kp*. The second component of the `Cons` pair sent to the consumer is a closure representing the remaining elements of the sequence. This closure is thus meant to be called by the consumer when the consumer demands a new element. This closure uses the position *kp* to resume the iteration from where it stopped. Moreover, it also uses `callcc` to obtain the position *kc* from which it was called by the consumer. It then writes *kc* to r, so the producer knows where to jump the next time the producer stumbles upon a new element.

Finally, when the iteration terminates, the producer performs a final jump to the consumer with the value `Nil` to indicate that all elements have been produced.

Implementation

Figure 6.9 shows how `callcc` and `throw` can be macro-expressed as effect-performing MazeLang programs. Their interpretation is given by the handler `toplevel`, which is also defined in this figure. Supposedly, this handler corresponds to the topmost frame in the execution stack. Under this assumption, it is easy to check that this encoding of `callcc` and `throw` *respects* the reduction rules (CallccStep) and (ThrowStep); that is, for a suitable definition of $\widetilde{K}$, the implementations of `callcc` and `throw` that appear in Figure 6.9 satisfy the following reduction rules:

$$\begin{array}{lcl} \texttt{toplevel}\, K[\texttt{callcc}\, k.\, e] & \to^* & \texttt{toplevel}\, K[e\{\widetilde{K}/k\}] \\ \texttt{toplevel}\, K[\texttt{throw}\, \widetilde{K'}\, v] & \to^* & \texttt{toplevel}\, K'[v] \end{array}$$

Indeed, the preceding rules are easily derivable with the following definition of $\widetilde{K}$:

$$\widetilde{K} \triangleq \texttt{cont}\,(\texttt{toplevel}\, K[\bullet\ ()]) \tag{6.4}$$

Let us explain why this encoding works. Recall that, when a program performs an effect, the evaluation context up to the innermost enclosing handler is (1) *reified* and (2)

```
callcc k. e ≜ (do (inj1 (fun k -> e))) ()
throw  k' x ≜  do (inj2 (k', x))
toplevel e  ≜
  deep-try e with
  (* Effect branch. *)
  ( fun arg k ->
      match arg with
      ( fun (* Callcc. *) t ->
          k  (fun _ -> t k)
      | fun (* Throw. *) (k', x) ->
          k' (fun _ -> x)
      )
  (* Return branch. *)
  | fun y -> y
  )
```

Figure 6.9: Implementation of `callcc` and `throw` in MazeLang.

replaced by the effect branch of the handler:

$$\texttt{try}\ (N[\texttt{do}\ v])\ \texttt{with}\ (h \mid r)\ \to^*\ h\ v\ k$$

When the operation `callcc` $k.\,e$ is performed, on the other hand, the enclosing evaluation context is only reified, but not replaced: control is kept by the expression e. Therefore, the idea is to encode `callcc` as an effect whose handler immediately invokes the continuation to which it gains access. So, even though the context surrounding `callcc` is replaced by `callcc`'s handler, this context is immediately restored.

Ideally, `callcc`'s handler should resume the captured continuation k with the expression $e\{k/k'\}$. However, continuations can be resumed only with values. The solution is thus to resume k with the thunk $\lambda\_.\ (\lambda k'.\ e)\ k$, which must be forced at the effect-call site. Therefore, the complete encoding of `callcc` $k'.\,e$ is a MazeLang program that proceeds as follows: first, it performs the effect `do` $(\texttt{inj}_1\ (\lambda k'.\ e))$, then, after it receives a thunk f, it immediately forces f through its application to `()`.

The left sum in the encoding of `callcc` is needed to let the handler distinguish between `callcc` and `throw`, which is also encoded as an effect. Indeed, `throw` $k'\ x$ is simply implemented as the instruction `do` $(\texttt{inj}_2\ (k', x))$. The handler of this effect is equally simple: all it does is to resume k' with the thunk $\lambda\_.\ x$. In particular, the captured continuation k is discarded: the context surrounding the expression `throw` $k'\ x$ is thus effectively replaced with the one denoted by k'.

Reasoning rules

The Maze reasoning rules for `callcc` and `throw` appear in Figure 6.10. They are stated in terms of an abstract predicate *isCont* and an abstract protocol CT, for `callcc` and `throw`.

The protocol CT describes the control effects that arise from the encoding of `callcc` and `throw`. This protocol is introduced by the application of rule MazeToplevel, which expresses the idea that, if the expression e is enclosed by a `toplevel` instruction, then e

$$isCont : Val \to (Val \to iProp) \to iProp \qquad CT : Protocol$$

IsContPersistent
$\mathsf{persistent}(isCont\, k\, \Phi)$

IsContWeakening
$$\frac{\Box \forall\, w.\ \Phi'(w) \mathrel{-\!\!*} \Phi(w)}{isCont\, k\, \Phi \mathrel{-\!\!*} isCont\, k\, \Phi'}$$

MazeToplevel
$$\frac{ewp\ e\ \langle CT \rangle \{\_.\ \mathsf{True}\}}{ewp\ (\texttt{toplevel}\, e)\ \langle \bot \rangle \{\_.\ \mathsf{True}\}}$$

MazeCallcc
$$\frac{\forall\, k.\ isCont\, k\, \Phi \mathrel{-\!\!*} ewp\ e\ \langle CT \rangle \{\Phi\}}{ewp\ (\texttt{callcc}\ k.\, e)\ \langle CT \rangle \{\Phi\}}$$

MazeThrow
$$\frac{isCont\, k\, \Phi \qquad \Phi(x)}{ewp\ (\texttt{throw}\, k\ x)\ \langle CT \rangle \{\_.\ \mathsf{False}\}}$$

Figure 6.10: Reasoning rules for `callcc`, `throw`, and `toplevel`.

can exploit the instructions `callcc` and `throw`. This rule also states that the complete program `toplevel` e abides by the protocol $\bot$; that is, this program performs no observable effect.

The binary predicate *isCont* relates a value k, representing an evaluation context K, to a predicate Φ. Intuitively, if the assertion $isCont\, k\, \Phi$ holds, then, for every value x satisfying Φ, the context K can be filled with x. Such assertions are introduced by applications of rule MazeCallcc, which states that, to reason about the operation $\texttt{callcc}\ k.\, e$ in a context that expects a value satisfying Φ, it suffices to reason about e and prove that e produces a value satisfying Φ under the assumption that $isCont\, k\, \Phi$ holds. In the verification of e, one can exploit the assertion $isCont\, k\, \Phi$ multiple times, because *isCont* is persistent (rule IsContPersistent). Moreover, one can weaken the assertion $isCont\, k\, \Phi$ by replacing Φ with a stronger predicate Φ' (rule IsContWeakening).

Rule MazeThrow states that, to perform the operation $\texttt{throw}\, k\ x$, the value k must have been introduced by `callcc`, that is, the assertion $isCont\, k\, \Phi$ must hold for some Φ; and x must satisfy Φ. The postcondition of $\texttt{throw}\, k\ x$ is the empty predicate, $\lambda\ \_.\ \mathsf{False}$. This predicate means that `throw` does not produce a return value, and can thus be used in any context. This logical description is in agreement with the dynamic behavior of `throw`, which does not return a value, but rather *escapes* its own evaluation context by resuming k.

The definition of the predicate *isCont* and of the protocol CT appears in Figure 6.11. The definition of $isCont\, k\, \Phi$ captures the idea that k is the reification of some evaluation context K, therefore k must assume the form $\widetilde{K} = \texttt{cont}\ (\texttt{toplevel}\, K[\bullet\ ()])$ (Equation 6.4). More precisely, the assertion $isCont\, k\, \Phi$ states that k expects a thunk f, which, when applied to `()`, produces a value that satisfies Φ. Moreover, the expression $k\ f$ abides by the empty protocol and the postcondition $\lambda\ \_.\ \mathsf{True}$.

The protocol CT is defined as the sum of two protocols: the protocol *Callcc*, which describes the effect performed by `callcc`; and *Throw*, which describes the effect performed by `throw`. The protocol *Throw* is a persistent send-receive protocol that simply rephrases

Definition of the predicate isCont.

$$\mathit{isCont}\, k\, \Phi \triangleq \Box \forall f.\ \mathit{ewp}\ (f\ ())\ \langle \bot \rangle \{\Phi\} \mathrel{-\!\!*} \mathit{ewp}\ (k\ f)\ \langle \bot \rangle \{\_.\ \mathsf{True}\}$$

Definition of the protocol CT.

$$\begin{aligned} \mathit{CT} &\triangleq \mathit{Callcc} + \mathit{Throw} \\ \mathit{Callcc} &\triangleq \Box\ !\, t\, \Phi\, (\mathtt{inj}_1\ t)\, \{\forall k.\ \mathit{isCont}\, k\, \Phi \mathrel{-\!\!*} \triangleright\, \mathit{ewp}\ (t\ k)\ \langle \mathit{CT} \rangle \{\Phi\}\}. \\ &\qquad ?\, f\, (f)\, \{\triangleright\, \mathit{ewp}\ (f\ ())\ \langle \mathit{CT} \rangle \{\Phi\}\} \\ \mathit{Throw} &\triangleq \Box\, !\, k'\, x\, \Phi\, (\mathtt{inj}_2\ (k', x))\, \{\mathit{isCont}\, k'\, \Phi\ *\ \Phi(x)\}.\, ?\, y\, (y)\, \{\mathsf{False}\} \end{aligned}$$

Figure 6.11: Definition of *isCont* and *CT*.

rule MazeThrow. The main difference between the definition of the protocol *Callcc* and the statement of rule MazeCallcc is the presence of later modalities. These modalities are necessary to guard the occurrences of the protocol *CT*, which make *Callcc* a recursive definition.

Comparison with Delbianco and Nanevski's HTT_{cc}. We note that the introduction of reasoning rules for `callcc` and `throw` in Separation Logic is not novel. As discussed in Chapter 2, Delbianco and Nanevski [DN13] devise HTT_{cc}, an extension of *HTT* [NAMB07, NVB10] with support for `callcc` and `abort` (a construct as expressive as `throw`).

Recall that *HTT* is a program logic exploiting Coq's rich dependent type theory to introduce the type of programs that satisfy a given specification. Reasoning rules in *HTT* are thus stated as typing rules. The reasoning rule for `callcc` could be translated to Maze as follows:

$$\frac{\forall k.\ \mathit{isCont}\, k\, \Phi \mathrel{-\!\!*} \mathit{ewp}\ e\ \langle \mathit{CT} \rangle \{\Phi'\}}{\mathit{ewp}\ (\mathtt{callcc}\ k.\, e)\ \langle \mathit{CT} \rangle \{y.\ \Phi(y) \vee \Phi'(y)\}}\ \textsc{HTTccCallcc}$$

Rule HTTccCallcc expresses the idea that the result of `callcc` $k.\, e$ comes either from the normal execution of e or from the execution of an instruction `throw` $k\ x$. It differs from rule MazeCallcc in that the postcondition of `callcc` $k.\, e$ is stated as a disjunction rather than an arbitrary predicate.

Rule HTTccCallcc is derivable in Maze. Indeed, it suffices to apply rule MazeCallcc followed by (1) the application of rule isContWeakening, to pass from the assertion $\mathit{isCont}\, k\, (\lambda y.\ \Phi(y) \vee \Phi'(y))$ to $\mathit{isCont}\, k\, \Phi$; (2) the application of rule MazeMonotonicity, to pass from a goal of the form $\mathit{ewp}\ e\ \langle \mathit{CT} \rangle \{y.\ \Phi(y) \vee \Phi'(y)\}$ to a goal of the form $\mathit{ewp}\ e\ \langle \mathit{CT} \rangle \{\Phi'\}$; and (3) the application of the premise $\forall k.\ \mathit{isCont}\, k\, \Phi \mathrel{-\!\!*} \mathit{ewp}\ e\ \langle \mathit{CT} \rangle \{\Phi'\}$, to complete the proof.

Rule MazeCallcc is derivable from rule HTTccCallcc. Indeed, it suffices to apply rule MazeMonotonicity, to pass from a goal of the form $\mathit{ewp}\ e\ \langle \mathit{CT} \rangle \{\Phi\}$ to a goal of the form $\mathit{ewp}\ e\ \langle \mathit{CT} \rangle \{y.\ \Phi(y) \vee \Phi(y)\}$, and then apply rule HTTccCallcc to complete the proof.

One of the examples studied by Delbianco and Nanevski is the function `inc3`, which uses `callcc` to twist the control flow in such a way that a reference x, passed as an argument, is incremented three times, even though only two increment operations are visible in `inc3`'s source code. Here is the translation of `inc3` to MazeLang:

```
let inc3 x =
  let c = callcc k.
    fun _ -> x := !x + 1; throw k (fun _ -> ())
  in
  x := !x + 1; c()
```

This function provides an interesting opportunity to exercise Maze's reasoning rules. Indeed, the derivation of the following statement is straightforward:

Statement 6.2 *The function* `inc3` *admits the following specification:*

$$\forall\, R,\, x,\, n.\; R \mathrel{-\!\!*} x \mapsto n \mathrel{-\!\!*} \mathit{ewp}\ (\texttt{inc3}\ x)\ \langle CT\rangle\{\_.\; R * x \mapsto (n+3)\}$$

This specification is adapted from Delbianco and Nanevski's version written in HTT_{cc}. Both versions employ a similar trick, which is to compensate for the inadmissibility of the frame rule by universally quantifying over a *residual heap*, here represented by the proposition R.

As written in Section 2.4 of Chapter 2, when we compared the Hazel logic to HTT_{cc}, one of the limitations of HTT_{cc} is the inability to reason about *dynamically allocated higher-order store*; that is, dynamically allocation of values such as first-class functions or first-class continuations. Like Hazel, the Maze logic is built on top of Iris, which has support for reasoning about dynamically allocated higher-order store, therefore Maze does not have this limitation. Moreover, Maze inherits all the Iris verification machinery, such as higher-order ghost state and invariants.

Comparison with Timany and Birkedal's work. To conclude this case study, let us clarify a seeming contradiction. Timany and Birkedal [TB19] claim that the bind rule is inadmissible in the presence of `callcc` and `throw`. This claim seems to contradict the results of this case study, which introduces reasoning rules for `callcc` and `throw` in Maze, a logic that enjoys the bind rule (MazeBind). However, there is no contradiction: Timany and Birkedal's claim applies to the *usual* notion of weakest precondition; that is, a predicate *wp* for which the assertion $\mathit{wp}\ e\ \{\Phi\}$ has the usual meaning that e either diverges or evaluates to a value that satisfies the postcondition Φ. Our notion of weakest precondition, *ewp*, does not conform to this reading. For instance, *ewp* does ***not*** validate the following reasoning rules from [TB19], which rephrase the operational behavior of `callcc` and `throw`:

$$\textsc{CallccWP}\ \frac{\triangleright\, \mathit{wp}\ (e\{\widetilde{K}/k\})\ \{\Phi\}}{\mathit{wp}\ (K[\texttt{callcc}\ k.\, e])\ \{\Phi\}} \qquad \textsc{ThrowWP}\ \frac{\triangleright\, \mathit{wp}\ (K'[v])\ \{\Phi\}}{\mathit{wp}\ (K[\texttt{throw}\ \widetilde{K'}\ v])\ \{\Phi\}}$$

If one replaces the weakest precondition $\mathit{wp}\ \_\ \{\_\}$ with $\mathit{ewp}\ \_\ \langle CT\rangle\{\_\}$, then the resulting logic would be unsound. Indeed, one could easily apply the resulting logic to derive a specification stating that the program `inc3` x increments the reference x twice.

The informal reason why Maze does not allow the operational reasoning about `callcc` and `throw` permitted by the rules CallccWP and ThrowWP is that these constructs are implemented as effect-performing MazeLang programs. The only way, in Maze, to reason about an effect isolated from its handler is by means of the protocol currently installed, which, in this case, is CT. Therefore, the only way to reason about `callcc` and `throw` in terms of the predicate $ewp\ \_\ \langle CT \rangle \{\_\}$ is by means of the rules MazeCallcc and MazeThrow.

We believe that rules MazeCallcc and MazeThrow endorse more abstract reasoning principles than those permitted by rules CallccWP and ThrowWP. For instance, whereas CallccWP compels one to think about the captured continuation k as the reification of some evaluation context, rule MazeCallcc invites one to think about k as a *position* to which a program can jump with a value, provided that this value satisfies the predicate Φ specified by the assertion $isCont\ k\ \Phi$. Moreover, the admissibility of the bind rule in Maze allows context-local reasoning in the presence of non-local control flow.

Chapter 7

A Type System for Effect Handlers with Dynamic Labels

In this chapter, we introduce Tes, a type system for a language with support for effect handlers, multiple named effects, and *dynamic allocation of effect labels*. As we shall see, this construct allows several programming styles, including but not restricted to *lexically scoped handlers* [ZM19, BPPS20], where the generation of a new name and the installation of a handler are tied together.

The main achievement of Tes is a set of simple typing and subtyping rules. Indeed, we believe that Tes is the first system, not restricted to lexically scoped handlers, that allows both the extension and the permutation of *rows*. This is possible due to a novel reading of a row, which is interpreted not only as a permission to perform effects with certain names and types but also as a requirement that these names be pairwise distinct.

To prove that Tes is *sound*, we follow the *semantic approach*: we present the interpretation of Tes into a novel Hazel-inspired Iris-embedded Separation Logic for effect handlers and named effects. We believe that we are the first authors to prove *effect safety* by means of a semantic interpretation in Iris. This chapter adapts the contents of a draft paper [dVP22a]. The results of this chapter are formalized in Coq.

7.1 Introduction

A type system consists of a set of *types* and a set of *typing rules*. A type is a concise description of a set of values. A *typing judgment* relates a program fragment to a type representing the set of values to which the result of this program belongs. A program related to some type by a typing judgment is a *well-typed program*. A typing rule establishes that a program is well-typed assuming that its *components* are well-typed; that is, from the assumption that certain program fragments are well-typed, a typing rule establishes that their composition by means of one of the constructs of the language is equally well-typed.

Type systems are thus similar to program logics in the sense that they allow the programmer to think about programs *abstractly* (in terms of types) and *modularly* (by means of typing rules).

The main practical purpose of a type system is to ensure that well-typed programs enjoy a set of guarantees. This set of guarantees depends on the type system, but one of the most common guarantees is *type safety*: every operation performed by a program has a well-defined meaning; in particular, every function call specifies a well-defined function and has the correct number of arguments. In the presence of effect handlers, a commonly desired guarantee is *effect safety*: that every effect performed by a program is handled.

In this chapter, we introduce Tes, a type system that ensures both type and effect safety. Although this goal has already been achieved by many authors [BP14, HL16, BPPS18, BPPS19, BPPS20, ZM19], we believe that Tes is the first type system that (1) attains type and effect safety for a programming language with *dynamic allocation of effect labels*; (2) has support for an expressive set of types, which includes *effect-polymorphic types*; and (3) has a simple set of typing rules, which includes the *extension* and the *permutation* of *rows*.

Let us further clarify the contributions of Tes by separating the discussion into two topics. In the first topic, *Operational semantics*, we explain what is *dynamic allocation of effect labels* and why this is a desired programming feature. In the second topic, *Static semantics*, we explain the concepts of effect polymorphism, rows, extension of rows, and permutation of rows; and we explain why these are desired features of a type system. We then conclude the discussion in a third and final subsection, *Overview*, where we recast the contributions of Tes and give an overview of this chapter.

7.1.1 Operational semantics

Multiple named effects. So far, both languages we have studied (*HH* and MazeLang) have support for unnamed effects: a handler installed over an expression e intercepts all the effects performed by e. However, a program often performs effects of different nature, such as I/O instructions and mutable state. Therefore, it is natural to extend the language with the ability to specify the effects that a handler can intercept. Effect names offer this additional expressive power. More precisely, we would like to introduce the instruction `perform` $s\ v$ to perform an effect identified by the effect name s and carrying payload v. Accordingly, we would like an effect handler to indicate which effects it wishes to handle, based on the name s. Thus, in the new effect-handling construct `handle` e `with` $(s : h \mid r)$, we would like the effect branch h to be invoked when the expression e performs an effect named s, and we would like the return branch r to be invoked if e terminates normally.

Dynamic allocation of effect labels. Dynamic allocation of effect labels is the ability to introduce fresh effect labels. We would like to the language to have a construct of the form `effect` s `in` e, which binds the *effect name* s to a freshly generated *effect label*. This distinction between effect names and effect labels is similar to the distinction between variables and memory locations that is traditionally used in the operational semantics of mutable references [Pie02]. An argument in favor of this feature is that dynamic allocation of effect labels can serve as a tool to defend against *accidental handling* [ZM19]. It is also worth noting that dynamic allocation of exception labels has existed for a long time in Standard ML [MTHM97] and in OCaml [LDF+19].

Accidental handling. Accidental handling is an *informal* concept that can be described roughly as follows: accidental handling occurs when an effect handler intercepts an effect that it was not meant to handle. For example, consider the function `bad_counter`, defined as follows:

```
bad_counter ff f ≜
  let g = λx. perform tick (); f x in
  (handle (ff g) with (tick : λ_. λk. λn. k () (n + 1) | λy. λn. (y, n))) 0
```

This example and most of the following ones in this section are written in TesLang, a calculus whose syntax and semantics are introduced in Section 7.2.

The function `bad_counter` uses an effect named *tick* to implement a memory cell in state-passing style. The intended result of the application `bad_counter` *ff* is a *version* of the second-order function *ff* that counts the number of times *ff* calls its argument. However, because of accidental handling, the function `bad_counter` may not behave as intended. For example, consider the following program:

$$\texttt{bad\_counter}\ (\texttt{bad\_counter}\ (\lambda f.\ f\,\texttt{()}))\ (\lambda \_.\ \texttt{()})$$

The expected result of this program is $((\texttt{()}, 1), 1)$, because the function $(\lambda f.\ f\,\texttt{()})$ calls its argument once, and its wrapped version, produced by the inner application of `bad_counter`, should preserve this property. However, the actual result is the value $((\texttt{()}, 2), 0)$. To understand why this value is produced, let us draw a loose picture of the runtime behavior of this program. Each application of `bad_counter` contributes to one *tick* handler and one *tick* effect. Because both handlers specify the same effect name *tick*, one of them intercepts the two *tick* effects. The handler that came from the inner application of `bad_counter` is the one that appears the most deeply nested in the evaluation stack, so this is the handler that intercepts both *tick* effects, thus leading to the result value.

To avoid accidental handling, precautions can be taken at several levels: the syntax and dynamic semantics of the programming language can be altered so as to make programmer mistakes less likely; and/or a static type discipline can be imposed. At the language design level, the literature describes two mechanisms that are intended to help protect against accidental handling: (1) *lexically scoped handlers* [BS17, BPPS20, BSO20a] and (2) *effect coercions* [BPPS18].

Lexically scoped handlers. The characteristic intended feature of a "*lexically scoped handler*" is the fact that one can statically tell which handler is invoked when an effect is performed [SBMO22]. This can be achieved by installing a handler for a freshly generated effect label, that is, a label generated immediately before the handler is installed. In other words, lexically scoped handlers can be simulated using ordinary effect handlers and dynamic generation of effect labels. The encoding is as follows:

$$\begin{array}{l}\texttt{lex-handle}\ e\ \texttt{with}\ (h \mid r) \triangleq \\ \quad \texttt{effect}\ s\ \texttt{in} \\ \quad \texttt{handle}\ (e\ (\lambda x.\ \texttt{perform}\ s\ x))\ \texttt{with}\ (s : h \mid r)\end{array} \tag{7.1}$$

This code first generates a fresh effect label and binds the name s to this label. Then, it installs a handler, which monitors the application of the expression e to the function $\lambda x.\ \texttt{perform}\ s\ x$.

Coming back to the example of the function `bad_counter`, one can employ a lexically scoped handler to correct the code as follows:

$$\begin{array}{l}\texttt{counter}\ \mathit{ff}\ f \triangleq \\ \quad \texttt{lex-handle}\ (\lambda \mathit{tick}.\ \mathit{ff}\ (\lambda x.\ \mathit{tick}\ \texttt{()}\,;\ f\ x))\ \texttt{with} \\ \quad (\lambda \_\ k.\ \lambda n.\ k\ \texttt{()}\ (n + 1) \mid \lambda y.\ \lambda n.\ (y, n))\ 0\end{array}$$

The variable *tick* now stands for a function that performs an effect named s. Every time `counter` is invoked, a fresh label ℓ is allocated, and the local name s is bound to this label. Therefore, the program

$$\texttt{counter}\ (\texttt{counter}\ (\lambda f.\ f\,\texttt{()}))\ (\lambda\_.\ \texttt{()})$$

reduces to the value $((\texttt{()}, 1), 1)$, as desired, because the two applications of `counter` install two handlers for two distinct dynamic labels.

Effect coercions. An effect coercion is an operation that modifies the manner in which a client that performs an effect is matched with one of the enclosing handlers.

Perhaps the most illustrative example is that of the `lift` coercion [BPPS18, BPPS19]. Usually, performing an effect named s transfers control to the innermost enclosing handler that selects the name s. In the presence of a `lift` coercion, however, control is transferred instead to the *second* closest handler. To this end, the `perform` instruction must be wrapped in a `lift` coercion: `lift` s (`perform` $s\ v$). Here is how one could employ such a coercion to correct the accidental-handling behavior from the function `bad_counter`:

$$\begin{array}{l}\texttt{lift\_counter}\ \mathit{ff}\ f\ \triangleq \\ \quad \texttt{let}\ g = \lambda x.\ \texttt{perform}\ \mathit{tick}\ \texttt{();}\ \texttt{lift}\ \mathit{tick}\ (f\ x)\ \texttt{in} \\ \quad (\texttt{handle}\ (\mathit{ff}\ g)\ \texttt{with}\ (\mathit{tick} : \lambda\_.\ \lambda k.\ \lambda n.\ k\ \texttt{()}\ (n + 1) \mid \lambda y.\ \lambda n.\ (y, n)))\ 0\end{array}$$

As desired, the program

$$\texttt{lift\_counter}\ (\texttt{lift\_counter}\ (\lambda f.\ f\,\texttt{()}))\ (\lambda\_.\ \texttt{()})$$

reduces to the value $((\texttt{()}, 1), 1)$, because among the two *tick* effects, one *tick* effect is intercepted by the innermost handler, whereas the other is intercepted by the outermost handler thanks to a `lift` coercion.

Between these two mechanisms of protection against accidental handling, we argue in favor of lexically scoped handlers. However, we do not defend lexically scoped handlers as the only effect-handling construct of the language. We rather advocate for ordinary handlers and the dynamic allocation of effect labels, which, as we have seen, are sufficient to express lexically scoped handlers. In sum, we argue *against* restricting the programming language to lexically scoped handlers only, and *against* effect coercions.

Our argument against effect coercions is unwarranted complexity. Effect coercions complicate the dynamic semantics of the language, and are potentially difficult to explain to programmers. Dynamic allocation of effect labels allows avoiding accidental handling, while preserving a simple, standard dynamic semantics.

Argument against the restriction to lexically scoped handlers. Our argument against a restriction to lexically scoped handlers is threefold:

1. **Unusual style.** Programming languages with support for exceptions, such as OCaml, Python, and Java, propose a set of globally defined exception names, to which every program has access. Programmers agree to throw each of these exceptions in specific situations – for example, when the task of searching for an element terminates unsuccessfully, or when an arithmetic computation causes a

division by zero. In a language with support for effect handlers, one can imagine a similar convention. Every program can have access to a set of globally defined effect names, and programmers can agree to perform each of these effects in specific situations. For example, when programming with *coroutines* [dMI09], programmers might agree to perform a globally defined *yield* effect to produce elements. For example, the following `filter` function iterates over a list xs and "yields" the elements that satisfy a certain predicate f. By convention, an element is "yielded" by performing a *yield* effect.

$$\texttt{filter}\ xs\ f \triangleq \texttt{iter}\ xs\ (\lambda x.\ \texttt{if}\ f\ x\ \texttt{then perform}\ \mathit{yield}\ x) \tag{7.2}$$

We assume that `iter` is a higher-order iteration combinator, which applies a function in succession to every element of a list. Another function, `reassemble`, constructs a list of the elements yielded by a function g:

$$\texttt{reassemble}\, g \triangleq \texttt{handle}\ (g\,\texttt{()})\ \texttt{with}\ (\mathit{yield} : \lambda x\, k.\ x :: k\,\texttt{()} \mid \lambda \_.\ \texttt{[]})$$

Filtering a list to obtain a new list is just a matter of combining the previous two functions:

$$\texttt{reassemble}\ (\lambda \_.\ \texttt{filter}\ xs\ f) \tag{7.3}$$

With lexically scoped handlers, one can write similar programs, albeit in an unusual style. Since an effect label can be allocated only when the handler is installed, a client has to take an extra parameter representing the function that performs this effect. For example, here is how one could define `filter` in a language that is restricted to lexically scoped handlers:

$$\texttt{filter}'\ xs\ f\ \mathit{yield} \triangleq \texttt{iter}\ xs\ (\lambda x.\ \texttt{if}\ f\ x\ \texttt{then}\ \mathit{yield}\ x)$$

And here is how one could define `reassemble` using a lexically scoped handler:

$$\begin{array}{l}\texttt{reassemble}'\, g \triangleq \\ \quad \texttt{lex-handle}\ g\ \texttt{with}\ (\lambda x\, k.\ x :: k\ \texttt{()} \mid \lambda \_.\ \texttt{[]})\end{array}$$

This style becomes more cumbersome when a program performs multiple named effects: a client needs to take one extra parameter for each effect name.

An argument in favor of this style would be that `reassemble′` protects against accidental handling. Indeed, consider the following program:

$$\texttt{reassemble}'\ (\texttt{filter}'\ xs\ f) \tag{7.4}$$

Because `reassemble′` installs a lexically scoped handler, we are assured that this handler will not intercept the effects that f might perform. On the other hand, the handler installed by `reassemble` (Program 7.3) might accidentally intercept a *yield* effect that f might perform. Our rebuttal is that, to protect against accidental handling in Program 7.3, one can specify, as an assumption to the application of `filter`, that f must not perform *yield* effects. As we shall see in Section 7.3, the programmer can express such a specification using the type system introduced in this chapter. Our system ensures that, in every application of `filter`, the function f does not perform *yield* effects.

2. **Not backwards compatible.** A restriction to lexically scoped handlers runs contrary to the design choices made in OCaml 5 [SDW+21], which does not impose such a restriction. Indeed, OCaml supports unrestricted dynamic allocation of effect labels.

3. **Theoretically unsatisfying.** Because a lexically scoped handler is simply the combination of the allocation of a fresh label and the installation of a handler for this label, it is theoretically unsatisfying to have a language that supports only lexically scoped handlers. With unrestricted dynamic allocation of effect labels, the programmer can choose between allocating effect labels globally (at the top level of the program) or locally.

7.1.2 Static semantics

Having decided in which semantic features we are interested, namely multiple named effects and dynamic allocation of effect labels, we can now address the question of designing a type system. Let us discuss what type system features and what static guarantees are desired.

Desired features.

1. **Annotated arrows.** In addition to an argument type τ and a return type κ, an annotated arrow $\tau \xrightarrow{\rho} \kappa$ includes a *row* ρ [Ré89, Lei14, HL16], which specifies the effects that a function might perform. Annotated arrows are a desired feature, because it is essentially impossible to achieve effect safety without them.

2. **Effect polymorphism.** Effect polymorphism is the possibility to reuse a program component in several different contexts that are prepared to perform or handle different effects. A typical example is that of the higher-order iteration combinator `iter`, which applies its argument f to the elements of a collection. Effect polymorphism is a desired feature, because we would like to express that `iter` is insensitive to the effects f might perform. More precisely, if the elements have type α, then we would like `iter` to have the type $\forall\theta.\ (\tau \xrightarrow{\theta} \texttt{()}) \xrightarrow{\theta} \texttt{()}$, which is universally quantified over the effect of the function f.

3. **Permutation and extension of rows.** Permutation and extension of rows are *subtyping rules*. A subtyping rule defines a *subtyping relation*: intuitively, a type τ is a *subtype* of κ, if terms of type τ can be used in a context expecting terms of type κ. Therefore, subtyping rules add flexibility to the type system; the laxer the subtyping relation is, the greater the set of well-typed programs is. In this chapter, we wish to study the subtyping relation on annotated arrow types. We argue that two desirable subtyping rules are the permutation and the extension of rows; that is, a type $\tau \xrightarrow{\rho} \kappa$ should be a subtype of $\tau \xrightarrow{\rho'} \kappa$, if ρ specifies less effects than ρ' does, regardless of the order of the effects in each of these rows. Being insensitive to the order of the effects in a row ρ essentially means that, when writing a function f of type $\tau \xrightarrow{\rho} \kappa$, the programmer needs not to reason about the order in which the effect handlers appear in the stack during the evaluation of f. The

permutation of rows is thus an important rule that alleviates cognitive load on the programmer by abstracting the order of handlers. Admitting the type $\tau \xrightarrow{\rho} \kappa$ to be a subtype of $\tau \xrightarrow{\rho'} \kappa$, if ρ specifies less effects than ρ', essentially means that f can be used in a context with more handlers than f "needs". This subtyping rule is especially important in the presence of effect-polymorphic types. Indeed, it is often the case that one wishes to call an effect-polymorphic function of type $\tau \xrightarrow{\theta} \kappa$ under a handler for a fresh effect name s. This is the case of the function `counter` (Eq. 7.1.1), whose type-checking we shall present in Section 7.3. In these cases, the type $\tau \xrightarrow{\theta} \kappa$ must be a subtype of $\tau \xrightarrow{(s\,:\,\_)\cdot\theta} \kappa$, where the row $(s : \_) \cdot \theta$ specifies the effect s in addition to those specified by θ.

Desired static guarantees. An important desired static guarantee is effect safety: that, during the execution of a complete (closed) program, no effect is left unhandled. In other words, performing an effect named s outside of the scope of a handler for this name must be statically forbidden.

The "absence of accidental handling", sometimes also referred to as "abstraction safety", might also be considered a desirable property. However, this property is only loosely defined in the literature. Zhang and Myers [ZM19] suggests that it is connected in some sense with *parametricity of effect polymorphism*. However, parametricity itself is loosely defined. Parametricity requires a universal quantifier in the syntax to be interpreted by a meta-level universal quantification over some universe of semantic types. However, *which* universe of semantic types is chosen matters, and this creates a tension between conflicting goals: while a larger universe allows establishing more contextual equivalence laws, a smaller universe allows type-checking more programming language constructs, such as `dynamic-wind` [FYFF07]. We continue this discussion later in Section 7.5. For now, let us state that "absence of accidental handling" is not a well-defined goal. The existence of a sound semantic interpretation of types, and the ability to exploit parametricity to establish strong program equivalence laws, are more clearly defined goals. In this chapter, we define a sound semantic model for Tes, but leave the study of equivalence laws to future work.

7.1.3 Overview

The main contribution of this chapter is the introduction of Tes, a type system for a language with support for effect handlers, multiple named effects, dynamic allocation of effect labels, and general references. The type system supports effect polymorphism and ensures the absence of unhandled effects. One key novelty of Tes is the meaning assigned to arrow types. An arrow type implicitly expresses the *requirement* that the effect labels in its row are pairwise distinct. In Section 7.3, we show how to exploit this feature in a number of examples. Moreover, Tes offers relaxed subsumption rules that allow, among other properties, the extension and permutation of rows. With simple yet flexible typing rules, Tes is the first system, not restricted to lexically scoped handlers, that accepts the program `counter` (Eq. 7.1.1). We give a more detailed comparison of Tes with previous work in Section 7.5. As a second contribution, in Section 7.4, we introduce a novel Separation Logic for reasoning about multiple named effects and multi-shot continuations.

Variables and effect identifiers

$$\mathit{Var} \ni f, x, s \qquad \mathit{EffId} \ni n ::= s\ (\in \mathit{Var}) \mid \ell\ (\in \mathit{Loc})$$

Values, expressions, and operations

$$\begin{aligned}
\mathit{Op} \ni \odot &::= \texttt{+} \mid \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{==} \\
\mathit{Val} \ni v &::= \texttt{()} \mid b\ (\in \mathit{Bool}) \mid i\ (\in \mathit{Int}) \mid \ell\ (\in \mathit{Loc}) \mid \odot\ (\in \mathit{Op}) \\
&\quad \mid \texttt{rec}\ f\ x.\ e \mid (v, v) \mid \texttt{inj}_i\ v \mid v :: v \mid \texttt{[]} \mid \texttt{cont}\ K \\
\mathit{Expr} \ni e &::= v \mid x \mid e\ e \mid e :: e \mid (e, e) \mid \texttt{proj}_i\ e \mid \texttt{inj}_i\ e \\
&\quad \mid \texttt{match}\ e\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{ref}\ e \mid \texttt{!}e \mid e \coloneqq e \\
&\quad \mid \texttt{effect}\ s\ \texttt{in}\ e \mid \texttt{perform}\ n\ e \mid \texttt{eff}\ (\ell, v)\ K \mid \texttt{handle}\ e\ \texttt{with}\ (n\ \texttt{:}\ v \mid v)
\end{aligned}$$

Evaluation contexts

$$\begin{aligned}
\mathit{Ectx} \ni K ::= {} & \bullet \mid e\ K \mid K\ v \mid K :: v \mid e :: K \mid (e, K) \mid (K, v) \mid \texttt{proj}_i\ K \mid \texttt{inj}_i\ K \\
& \mid \texttt{match}\ K\ \texttt{with}\ (v \mid v) \mid \texttt{if}\ K\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{ref}\ K \mid \texttt{!}K \mid e \coloneqq K \mid K \coloneqq v \\
& \mid \texttt{perform}\ \ell\ K \mid \texttt{handle}\ K\ \texttt{with}\ (\ell\ \texttt{:}\ v \mid v)
\end{aligned}$$

Notation

Figure 7.1: Syntax of TesLang.

We use this logic as to offer a semantic interpretation of Tes and to prove that Tes ensures effect safety. As a third contribution, we study the combination of effects, mutable state, and polymorphism, and point out that this combination (if unrestricted) is unsound, because the universal quantifier does not commute with the update modality. In Tes, typing judgments carry *purity attributes*, which indicate whether a program interacts with the store. A program that is considered *pure* by the type system satisfies a specification that does not involve an update modality; therefore, its type can be generalized.

7.2 Syntax and semantics of TesLang

We introduce TesLang, a calculus that formalizes the semantics of dynamic allocation of effect labels. Moreover, this calculus includes mutable state, effect handlers, multiple named effects, and multi-shot continuations. Even though one-shot continuations are more principled programming constructs than multi-shot continuations (see Paragraph 2.1 of Chapter 2), we choose to support multi-shot continuations, because, to statically rule out multiple calls to a continuation, a *substructural type system* [Wal05] would be needed. We leave the extension of Tes to ensure that continuations abide by a one-shot discipline to future work.

Reduction relation $\boxed{e \,/\, \sigma \;\to\; e \,/\, \sigma}$

$$\textsc{EffectStep}\quad \frac{\ell \notin \mathit{dom}(\sigma)}{\texttt{effect}\ s\ \texttt{in}\ e \,/\, \sigma \;\to\; e\{\ell/s\} \,/\, \sigma[\ell \mapsto ()]}$$

$$\textsc{PerformStep}\quad \texttt{perform}\ \ell\ v \,/\, \sigma \;\to\; \texttt{eff}\ (\ell, v)\ \bullet \,/\, \sigma$$

$$\textsc{GobbleUpAppLCtxStep}\quad (\texttt{eff}\ (\ell, v_1)\ K)\ v_2 \,/\, \sigma \;\to\; \texttt{eff}\ (\ell, v_1)\ (K\ v_2) \,/\, \sigma$$

$$\textsc{GobbleUpAppRCtxStep}\quad e_1\ (\texttt{eff}\ (\ell, v_2)\ K) \,/\, \sigma \;\to\; \texttt{eff}\ (\ell, v_2)\ (e_1\ K) \,/\, \sigma$$

$$\textsc{GobbleUpHandleCtxStep}\quad \frac{\ell_1 \neq \ell_2}{\texttt{handle}\ (\texttt{eff}\ (\ell_2, v)\ K)\ \texttt{with}\ (\ell_1 : h \mid r) \,/\, \sigma \;\to\; \texttt{eff}\ (\ell_2, v)\ (\texttt{handle}\ K\ \texttt{with}\ (\ell_1 : h \mid r)) \,/\, \sigma}$$

$$\textsc{HandleEffectStep}\quad \texttt{handle}\ (\texttt{eff}\ (\ell, v)\ K)\ \texttt{with}\ (\ell : h \mid r) \,/\, \sigma \;\to\; h\ v\ (\texttt{cont}\ (\texttt{handle}\ K\ \texttt{with}\ (\ell : h \mid r))) \,/\, \sigma$$

$$\textsc{HandleReturnStep}\quad \texttt{handle}\ v\ \texttt{with}\ (\ell : h \mid r) \,/\, \sigma \;\to\; r\ v \,/\, \sigma$$

$$\textsc{InvokeStep}\quad (\texttt{cont}\ K)\ v \,/\, \sigma \;\to\; K[v] \,/\, \sigma$$

Figure 7.2: Selected reduction rules of TesLang.

7.2.1 Syntax

The syntax of values, expressions, and evaluation contexts is shown in Figure 7.1. Most of the constructs are standard. They include recursive functions, binary products, binary sums, binary and unary operations, lists, and references. Non-standard constructs include first-class continuations, `cont` K, and instructions to perform and handle effects: the operations `perform` n e and `handle` e `with` $(n : h \mid r)$, respectively. The argument e of the instruction to perform effects is called the effect *payload*. Both of the instructions to perform and to handle effects carry an effect identifier n, which is either a variable s, an *effect name*, or a memory location ℓ, an *effect label*. Even though source programs use only effect names as identifiers, the syntax of expressions must allow an effect identifier to be a location so that the syntax is closed by the reduction relation. These memory locations correspond to fresh effect labels introduced by the construct `effect` s `in` e. Finally, the syntax includes active effects, `eff` (ℓ, v) K, which are also not part of source programs, but which play a role in the definition of the operational semantics as we shall explain in the next subsection.

7.2.2 Semantics

The small-step operational semantics of TesLang is defined as a reduction relation acting on pairs of an expression e and a store σ. A store is a finite map from memory locations to values. An excerpt of the reduction rules appears in Figure 7.2. The omitted reduction

rules, such as β-reduction, and the rules for allocating, reading and writing references, are standard. Invoking a continuation `cont` K with a value v reduces to $K[v]$, the term obtained by filling the hole of K with v. The remaining reduction rules that appear in Figure 7.2 illustrate the subtle aspects of the calculus: introducing a fresh effect label, and performing and handling effects.

Introducing a fresh effect label. Fresh effect labels are introduced using the store: the instruction `effect` s `in` e allocates a memory location ℓ, initialized with the value `()`, and performs the substitution of ℓ for the variable s in the expression e.

Performing and handling effects. Performing an effect transfers control to the handler. Indeed, the instruction `perform` ℓ v reduces to the active effect `eff` (ℓ, v) $\bullet$, which start the mechanism of capture of the evaluation context. An active effect swallows the evaluation context, frame by frame, until it reaches a handler. If the handler selects an effect label other than ℓ, then it continues the capture mechanism and swallows the handler. If the handler includes the effect label ℓ, then it transfers control to the effect branch h of handler. The effect branch h receives both the argument v with which the effect was performed, and the reification of the captured evaluation context K as a first-class continuation `cont` K. Notice that the reified continuation includes the effect handler. Recall that this corresponds to a deep-handler semantics: resuming the continuation reinstalls the handler.

7.3 Definition of Tes

In this section, we document the definition of Tes. First, we present the syntax of types. Second, we introduce the set of typing and subtyping rules. Finally, we illustrate the strength and the subtleties of the system through a number of examples.

7.3.1 Syntax of types, rows, and signatures

Figure 7.3 shows the syntax of types, rows, and signatures. The set *TypeVar* is an infinite set of type variables, ranged over by α, β, and γ, and *RowVar* is an infinite set of row variables, ranged over by θ. We also introduce the set *Attribute* of *purity attributes*. A purity attribute is either I for "impure" – to indicate that a program can interact with the store by introducing fresh effect labels, or by allocating, updating, or reading references – or P for "pure" – to indicate that a program cannot interact with the store. As we shall explain in the next subsection, the introduction of purity attributes is related to the introduction of polymorphic types and the *value restriction* [Wri95, Gar04].

Most types are standard; they include the unit type `()`, top and bottom types, noted as $\top$ and $\bot$, respectively, the type of Booleans `bool`, the type of integers `int`, sum and product types, the type of lists, and reference types. The syntax also includes value-polymorphic types to universally quantify over type variables α. The two remaining types, annotated arrow types and effect-polymorphic types, are the most interesting, and we discuss them separately.

Type variables, row variables, and purity attributes

$$\mathit{TypeVar} \ni \alpha, \beta, \gamma \qquad \mathit{RowVar} \ni \theta \qquad \mathit{Attribute} \ni a ::= P \mid I$$

Types, rows, and signatures

$$\begin{aligned} \mathit{Type} \ni \tau, \iota, \kappa ::= {} & \texttt{()} \mid \texttt{bool} \mid \texttt{int} \mid \bot \mid \top \\ & \mid \alpha \mid \tau\ \texttt{list} \mid \tau\ \texttt{*}\ \tau \mid \tau\ \texttt{+}\ \tau \mid \tau\ \texttt{ref} \\ & \mid \tau \xrightarrow{\rho}_a \tau \mid \forall\alpha.\ \tau \mid \forall\theta.\ \tau \end{aligned} \qquad \begin{aligned} \mathit{Row} \ni \rho ::= {} & \langle\rangle \mid \sigma \cdot \rho \\ \mathit{Sig} \ni \sigma ::= {} & (s : \tau \Rightarrow \tau) \mid \theta \end{aligned}$$

Notation

$$\begin{aligned} (s : \texttt{abs}) &\triangleq (s : \bot \Rightarrow \top) \\ \tau \to_a \kappa &\triangleq \tau \xrightarrow{\langle\rangle}_a \kappa \\ \tau \xrightarrow{\rho} \kappa &\triangleq \tau \xrightarrow{\rho}_I \kappa \end{aligned}$$

Figure 7.3: Syntax of types, rows, and signatures.

Annotated arrow types. In addition to an argument type ι and a return type τ, an arrow type $\iota \xrightarrow{\rho}_a \tau$ includes a purity attribute a and a *row* ρ. A row is a list of *signatures* σ. (The empty row $\langle\rangle$ denotes the empty list, and the the row composition $\_ \cdot \_$ denotes list cons.) A signature, in its turn, is either a *concrete signature* $(s : \iota' \Rightarrow \tau')$ or an *abstract signature* θ. A concrete signature $(s : \iota' \Rightarrow \tau')$ indicates that performing the effect s is analogous to calling a function of argument type ι' and return type τ'. According to this reading, the signature $(s : \bot \Rightarrow \top)$, noted $(s : \texttt{abs})$, forbids f from performing s. We call this signature the *absence signature*. An abstract signature θ is simply a row variable: it stands for a row ρ' possibly containing multiple concrete and abstract signatures. Intuitively, a program f of type $\iota \xrightarrow{\rho}_a \tau$ is a function that, when applied to an argument of type ι, will either return a result of type τ or perform an effect whose signature appears in the row ρ. Moreover, if a is the impure attribute I, then this function can interact with the store during its evaluation; otherwise it cannot do so. Moreover, Tes introduces a novel aspect to the reading of an arrow: *Tes includes the requirement that the effect labels bound by signatures in ρ are pairwise distinct.* In other words, the program f can assume that the dynamic instances of the effects in ρ are distinct from one another. We say that a row ρ is *dynamically distinct* if it satisfies this separation requirement. The syntax of rows includes rows that are not dynamically distinct (for example, a row can have two occurrences of the same signature $(s : \_ \Rightarrow \_)$), however, if a function carries such a row, then this function cannot be called, because the separation requirement does not hold. Finally, we remark that, when the purity attribute is omitted from an arrow, the attribute I is chosen by default.

Effect-polymorphic types. A row might contain one or more row variables. An effect-polymorphic type adds the ability to quantify universally over such variables. Intuitively, a program of type $\forall\theta.\ \tau$ has a behavior that does not depend on the set of effects abstracted by θ. Recall the typical example of the effect-polymorphic `iter` function discussed in

Section 7.1: a higher-order iteration method whose behavior is insensitive to the set of effects performed by its iteratee. An example of such an iteration method is the function `iter`, defined as follows:

$$\texttt{iter} \triangleq \texttt{rec}\ \mathit{iter}\ \mathit{xs}\ \mathit{f}.\ \texttt{match}\ \mathit{xs}\ \texttt{with}\ (\lambda \mathit{x}\ \mathit{xs}.\ \mathit{f}\ \mathit{x}\texttt{;}\ \mathit{iter}\ \mathit{xs}\ \mathit{f} \mid \lambda\_.\ \texttt{()}) \tag{7.5}$$

With effect (and value) polymorphism, we can assign the following type to `iter`:

$$\texttt{iter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \to (\alpha \xrightarrow{\theta} \texttt{()}) \xrightarrow{\theta} \texttt{()}$$

This type tells that `iter` is independent both of the representation of the elements of xs and of the set of effects that f might perform. In particular, the function `iter` does not perform any effects and does not intercept the effects performed by f.

7.3.2 Typing and subtyping judgment

A typing judgment in Tes assumes the form $\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau$. It depends on three environments: (1) a *row- and type-variable context* Ξ, which is a set of row and type variables θ and α; (2) an *effect-name context* Δ, which is a set of effect names; and (3) a *value context* Γ, which is a map from variables x to types τ. A typing judgment relates an expression e to a purity attribute a, a row ρ, and a type τ. Intuitively, such a judgment asserts that, during the evaluation of e, this program may perform effects according to ρ, interact with the store according to a, and produce an output of type τ. Moreover, the program e can assume that ρ is dynamically distinct.

A selection of the typing rules appears in Figure 7.4. These rules depend on the *operation-typing judgment*, which associates an operation to a pair of an argument type ι and a return type τ. We write this pair as $\iota \to \tau$. The derivation rules of the operation-typing judgment appear in Figure 7.5.

We divide the remainder of this subsection into three parts: (1) purity attributes and the value restriction, (2) an overview of the main typing rules, and (3) the monotonicity rule and the subsumption relations.

Purity attributes and the value restriction

It is well-known that, in the presence of mutable state, the unrestricted introduction of polymorphic types is unsound [Tof90]. One solution is the value restriction [Wri95, Gar04]: to restrict polymorphism to values. With purity attributes, we adopt a slightly more general solution: we restrict the introduction of polymorphic types to *pure* expressions, that is, to expressions whose typing judgment carries the attribute P. Indeed, rules TLamTyped and RLamTyped allow the generalization of both row and type variables of a pure expression e. This solution is more general than the value restriction, because every value is a pure expression. The only impure constructs are reading, writing, and allocating references, and allocation of effect labels. Therefore, even constructs such as handlers and effects can be considered pure expressions and can thus have their row and type variables generalized. Kammar and Pretnar study a similar system [KP17]. They show that, in the absence of references and allocation of effect labels, the unrestricted generalization of type variables is sound. The soundness of Tes subsumes this result. Finally, we remark that the

Typing judgment $\boxed{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau}$

UnitTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a () : \rho : ()$$

BoolTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a b : \rho : \texttt{bool}$$

IntTyped
$$\Xi \mid \Delta \mid \Gamma \vdash_a i : \rho : \texttt{int}$$

OpTyped
$$\frac{\vdash_{Op} \odot : \iota \to \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a \odot : \rho : \iota \xrightarrow{\rho}_a \tau}$$

VarTyped
$$\frac{\Gamma(x) = \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a x : \rho : \tau}$$

TLamTyped
$$\frac{\alpha \notin \Xi, \Gamma, \rho \qquad \alpha, \Xi \mid \Delta \mid \Gamma \vdash_P e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \alpha.\, \tau}$$

RLamTyped
$$\frac{\theta \notin \Xi, \Gamma, \rho \qquad \theta, \Xi \mid \Delta \mid \Gamma \vdash_P e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \theta.\, \tau}$$

ReadTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\,\texttt{ref}}{\Xi \mid \Delta \mid \Gamma \vdash_I !e : \rho : \tau}$$

TAppTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \alpha.\, \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\{\tau'/\alpha\}}$$

RAppTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall \theta.\, \tau}{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\{\rho'/\theta\}}$$

WriteTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau\,\texttt{ref} \qquad \Xi \mid \Delta \mid \Gamma \vdash_{a'} e' : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I e \coloneqq e' : \rho : ()}$$

LetEffTyped
$$\frac{s \notin \Gamma, \rho, \tau \qquad \Xi \mid s, \Delta \mid \Gamma \vdash_a e : (s : \texttt{abs}) \cdot \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{effect}\, s\, \texttt{in}\, e : \rho : \tau}$$

AllocTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{ref}\, e : \rho : \tau\,\texttt{ref}}$$

PerformTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \iota \qquad (s : \iota \Rightarrow \tau) \in \rho \qquad s \in \Delta}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{perform}\, s\, e : \rho : \tau}$$

RecTyped
$$\frac{\Xi \mid \Delta \mid \Gamma, f : \iota \xrightarrow{\rho}_a \tau, x : \iota \vdash_a e : \rho : \tau}{\Xi \mid \Delta \mid \Gamma \vdash_{a'} \texttt{rec}\, f\, x.\, e : \langle\rangle : \iota \xrightarrow{\rho}_a \tau}$$

HandleTyped
$$\frac{\sigma = (s : \iota \Rightarrow \tau) \qquad \sigma' = (s : \iota' \Rightarrow \tau') \qquad \rho' = \sigma' \cdot \rho \qquad s \in \Delta \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e : \sigma \cdot \rho : \kappa \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho' : \iota \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a r : \rho' : \kappa \xrightarrow{\rho'}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{handle}\, e\, \texttt{with}\, (s : h \mid r) : \rho' : \kappa'}$$

MonotonicityTyped
$$\frac{a \leq_A a' \qquad \vdash_b \rho \leq_R \rho' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau \qquad \vdash \tau \leq_T \tau'}{\Xi \mid \Delta \mid \Gamma \vdash_{a'} e : \rho' : \tau'}$$

AppTyped
$$\frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \iota \xrightarrow{\rho}_a \tau \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e' : \rho : \iota}{\Xi \mid \Delta \mid \Gamma \vdash_a e\, e' : \rho : \tau}$$

Figure 7.4: Typing rules.

Operation-typing judgment $\boxed{\vdash_{Op} \odot : \iota \to \tau}$

$$\vdash_{Op} \texttt{+} : \texttt{int} * \texttt{int} \to \texttt{int} \qquad \vdash_{Op} \texttt{not} : \texttt{bool} \to \texttt{bool}$$

$$\frac{\odot \in \{\texttt{and}, \texttt{or}\}}{\vdash_{Op} \odot : \texttt{bool} * \texttt{bool} \to \texttt{bool}} \qquad \frac{\tau \in \{(), \texttt{bool}, \texttt{int}\}}{\vdash_{Op} \texttt{==} : \tau * \tau \to \texttt{bool}}$$

Figure 7.5: Operation-typing rules.

reason why rule LetEffTyped carries an impure marker I is mainly technical: since the allocation of an effect label is implemented using memory allocation, such an expression interacts with the store. We believe that marking it as pure would not break the system's soundness. (Kammar and Pretnar [KP17] note on Paragraph 1 of the Conclusion Section that a proof of this statement remains an open problem.)

Overview of the main typing rules

We discuss the rules for allocating effect labels (LetEffTyped), performing effects (PerformTyped), and handling effects (HandleTyped).

Rule PerformTyped states that to perform an effect under the signature $(s : \iota \Rightarrow \tau)$, a program must produce a value of type ι, and, in return, it can expect a value of type τ. This typing rule confirms the intuitive idea that performing an effect is like calling a function of argument type ι and return type τ.

If we read rule LetEffTyped in the backwards direction (from the conclusion to the premise), then this rule states that allocating s gives e the ability to use s as an effect name, which is initially attached to the absence signature. Because a row is supposed to be dynamically distinct, the introduction of s to e's row, ρ, means that e can suppose s binds a fresh effect label, that, in particular, does not clash with a label bound by an effect name in ρ.

Rule HandleTyped, for installing a handler, expresses the idea that a handler establishes a boundary between the client e, which performs s effects according to σ, and the outer context in which the handler appears, where s effects abide by σ'.

Both the effect branch h and the return branch r can perform s effects according to σ'. Moreover, the continuation corresponds to h's second argument, whose type is $\tau \xrightarrow{\rho'}_a \kappa'$. It is interesting to remark that (1) the continuation has an arrow type, (2) this arrow is annotated by the row ρ', and (3) the return type of the continuation is the same as the one assigned to the handler. The first remark justifies that a delimited continuation can be seen as a function. The explanation for second and third remarks comes from the semantics of deep handlers: because the handler is reinstalled as the top frame of the continuation, s effects performed by the interrupted client are intercepted by the handler, who can introduce effects according to σ'.

Subsumption relation on types $\boxed{D \vdash \tau \leq_T \tau}$

$$\textsc{Bot}\quad D \vdash \bot \leq_T \tau \qquad \textsc{TypeRefl}\quad D \vdash \tau \leq_T \tau \qquad \textsc{Top}\quad D \vdash \tau \leq_T \top$$

$$\textsc{TypeTrans}\quad \frac{D \vdash \tau \leq_T \tau' \qquad D \vdash \tau' \leq_T \tau''}{D \vdash \tau \leq_T \tau''}$$

$$\textsc{Arrow}\quad \frac{D' = \langle \rho' \rangle \cup D \qquad a \leq_A a' \qquad D' \vdash \iota' \leq_T \iota \qquad D' \vdash \tau \leq_T \tau' \qquad D' \vdash_b \rho \leq_R \rho'}{D \vdash \iota \xrightarrow{\rho}_a \tau \leq_T \iota' \xrightarrow{\rho'}_{a'} \tau'}$$

Subsumption relation on signatures $\boxed{D \vdash \sigma \leq_S \sigma}$

$$\textsc{SigRefl}\quad D \vdash \sigma \leq_S \sigma$$

$$\textsc{SigConc}\quad \frac{D \vdash \iota \leq_T \iota' \qquad D \vdash \tau' \leq_T \tau}{D \vdash (s : \iota \Rightarrow \tau) \leq_S (s : \iota' \Rightarrow \tau')}$$

Subsumption relation on rows $\boxed{D \vdash_b \rho \leq_R \rho}$

$$\textsc{Empty}\quad D \vdash_b \langle\rangle \leq_R \langle\rangle \qquad \textsc{RowCons}\quad D \vdash_b \rho \leq_R \sigma \cdot \rho$$

$$\textsc{Skip}\quad \frac{D \vdash \sigma \leq_S \sigma' \qquad D \vdash_{\mathit{false}} \rho \leq_R \rho'}{D \vdash_b \sigma \cdot \rho \leq_R \sigma' \cdot \rho'}$$

$$\textsc{Swap}\quad D \vdash_b \sigma \cdot \sigma' \cdot \rho \leq_R \sigma' \cdot \sigma \cdot \rho$$

$$\textsc{Erase}\quad \frac{D \vdash s \notin \rho}{D \vdash_{\mathit{true}} (s : \mathtt{abs}) \cdot \rho \leq_R \rho} \qquad \textsc{RowTrans}\quad \frac{D \vdash_b \rho \leq_R \rho' \qquad D \vdash_b \rho' \leq_R \rho''}{D \vdash_b \rho \leq_R \rho''}$$

Figure 7.6: Subsumption relations on types, signatures, and rows.

Monotonicity rule and the subsumption relation.

Rule MonotonicityTyped states the conditions under which a typing judgment subsumes another. These conditions are written in terms of subsumption relations on attributes, types, rows, and signatures. If two terms are related by a subsumption relation, then we say the term in the left-hand side is stronger than the one in the right-hand side.

The subsumption relation on attributes, noted $\_ \leq_A \_$, is the unique total order on *Attribute* where P is less than I. This relation can be defined as follows:

$$a \leq_A a' \triangleq (a = I \implies a' = I)$$

Figure 7.6 shows the definition of the remaining subsumption relations. Before we present these relations, let us introduce a notion on which they all depend: the notion of a *disjointness context*. A disjointness context D maps an effect name to a pair of a multiset

of effect names S and a set of row variables V. Informally, a disjointness context D stores disjointness information. In particular, if D maps an effect name s to the pair (S, V), then the dynamic label bound by s is distinct from the dynamic labels bound by S and by V. We formally capture this description in Section 7.4, where we introduce the *semantic interpretation* of a disjointness context.

Why are disjointness contexts necessary? We introduce disjointness contexts to allow the sound *erasure of absence signatures* of a row. The erasure of an absence signature corresponds to the claim that $\sigma \cdot \rho \leq_R \rho$, where σ stands for $(s : \texttt{abs})$. However, this claim is not true in general. Indeed, if such subsumption relation was permitted, then the relation $\sigma \cdot \sigma \leq_R \sigma$ would be derivable. Intuitively, this relation says that a function f with row $\sigma \cdot \sigma$ can be seen as a function with row σ. However, the row $\sigma \cdot \sigma$ is not dynamically distinct, whereas the singleton row σ trivially is. Therefore, f has a false precondition, whereas a function with row σ does not. Assigning f the row σ would erase an unsatisfiable constraint, and thus lead to the acceptance of unsafe programs. An example of such an unsafe program is the following one:

```
1  effect s in
2  handle
3    handle (perform s ()) with (s : λx _. not x | λ_. true)
4  with (s : λ_ _. () | λ_. ())
```

Under the assumption that the relation $\sigma \cdot \sigma \leq_R \sigma$ is derivable, it is possible to show that this program is assigned the empty row and type `()`. The type derivation starts with the application of rule LetEffTyped, which introduces σ to the empty row. Then, the application of rule MonotonicityTyped, with the relation $\sigma \cdot \sigma \leq_R \sigma$ as the instance of the subsumption relation on rows, allows the type derivation of the program between lines 2–4 to be completed under the row $\sigma \cdot \sigma$. Because the row has two signatures for the same name s, we can install two handlers for the same effect s. The handler on line 2 allows its client – the program on line 3 – to perform s effects according to the signature $(s : () \Rightarrow ())$. The handler on line 3 allows its client to perform s effects according to the signature $(s : \texttt{bool} \Rightarrow ())$. Because the client of the handler on line 3 is also in the scope of the handler on line 2, it can perform s effects according to either one of these signatures. It then deviously chooses to perform an s effect according to the signature handled by the outermost handler. The innermost handler intercepts this effect and the mismatch of signatures leads to a runtime error due to the execution of the expression `not ()`.

With disjointness contexts, the erasure of absence signatures can be soundly permitted. One can allow the erasure of s in $(s : \texttt{abs}) \cdot \rho$ under a disjointness context D, provided D guarantees that the dynamic label of s is different from the dynamic labels in ρ. We explain this idea in more detail when presenting the subsumption relation on rows.

Subsumption relation on types. The subsumption relation on types, noted $\_ \vdash \_ \leq_T \_$, is a relation parameterized by a disjointness context. The rules in Figure 7.6 state that this relation is reflexive and transitive and that it admits $\bot$ and $\top$ as bottom and top elements, respectively. Moreover, the relation is contravariant on the argument type of an

arrow and covariant on the result type. The rule ARROW enriches the disjointness context. Intuitively, the rule exploits the assumption that ρ' is dynamically distinct to enrich the disjointness information stored in the current context D. The non-aliasing information learnt from the well-formedness of ρ' is represented by the disjointness context $\langle \rho' \rangle$. Its definition is written in terms of the functions *conc* and *abst*. The function *conc* computes the multiset of effect names of a row, whereas the function *abst* computes the set of row variables of a row. They are inductively defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{conc}\,(\langle\rangle) & \triangleq & \{\} & \mathit{abst}\,(\langle\rangle) & \triangleq & \{\} \\ \mathit{conc}\,((s : \_) \cdot \rho) & \triangleq & \{s\} \cup \mathit{conc}\,(\rho) & \mathit{abst}\,((s : \_) \cdot \rho) & \triangleq & \mathit{abst}\,(\rho) \\ \mathit{conc}\,(\theta \cdot \rho) & \triangleq & \mathit{conc}\,(\rho) & \mathit{abst}\,(\theta \cdot \rho) & \triangleq & \{\theta\} \cup \mathit{abst}\,(\rho) \end{array}$$

Notice that, because *conc* computes a multiset, the union, singleton set, and empty set have different meaning in each of the previous definitions. In the definition of *conc*, they are interpreted as multiset constructs, whereas, in the definition of *abst*, they are interpreted as set constructs.

The disjointness context $\langle \rho' \rangle$ can thus be defined as follows:

$$\langle \rho' \rangle \triangleq \bigcup_{s \in \mathit{conc}\,(\rho')} \{s \mapsto (\mathit{conc}\,(\rho') \setminus \{s\}, \mathit{abst}\,(\rho'))\}$$

The context $\langle \rho' \rangle$ maps every effect name s in ρ' to the pair of the multiset of effect names in ρ', excluding one occurrence of s, and the set of row variables in ρ'. The construction of this context exploits the assumption that ρ' is dynamically distinct: the dynamic label bound by s is distinct from the dynamic labels bound by $\mathit{conc}\,(\rho') \setminus \{s\}$ and by $\mathit{abst}\,(\rho')$.

To update a context D with $\langle \rho' \rangle$, it suffices to perform the *union* of these two contexts. The union of contexts D_1 and D_2 is defined as follows:

$$(D_1 \cup D_2)\,(s) \triangleq \begin{cases} D_1\,(s) \cup D_2\,(s) & \text{if } s \in \mathit{dom}(D_1) \cap \mathit{dom}(D_2) \\ D_i\,(s) & \text{if } s \in \mathit{dom}(D_i) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The union of pairs $D_1(s)$ and $D_2(s)$ is defined as the pairwise union.

Subsumption relation on signatures. The subsumption relation on signatures, noted $\_ \vdash \_ \leq_S \_$, is parameterized by a disjointness context. The rules in Figure 7.6 state that this relation is reflexive, and that, unlike an arrow constructor, the signature constructor $(\_ : \_ \Rightarrow \_)$ is covariant in its domain and contravariant in its range. (Transitivity is derivable.) The seemingly inverted variance in the case of a signature constructor matches the intuition that an effect signature $(s : \iota \Rightarrow \tau)$ is an arrow in the value context: a program that carries such a signature can perform s effects in the same way as it calls a function of type $\iota \to \tau$.

Subsumption relation on rows. The subsumption relation on rows, noted $\_ \vdash \_ \; \_ \leq_R \_$, is parameterized by a disjointness context and a Boolean flag b. When true, this flag represents a permission to erase absence signatures. From the combination of rules in Figure 7.6, it follows that this relation is reflexive and transitive. Moreover, it follows

that $D \vdash_b \rho \leq_R \rho'$ is derivable for any row ρ' that includes ρ, regardless of the order of the signatures in ρ'. In particular, ρ is stronger than any of its permutations. The ability to freely permute entries in a row is present in systems that impose the restriction to lexically scoped handlers [BPPS20, ZM19], and in Links's type system [HL16], which, by means of a syntactic criteria, does not allow repeated occurrences of a label in a row. The systems studied in [BPPS19, BPPS18, Lei17], on the other hand, do not support this feature.

Rule Erase depends on the following assertion:

$$D \vdash s \notin \rho \triangleq s \in dom(D) \wedge (conc(\rho), abst(\rho)) \subseteq D(s)$$

This assertion claims that, from the disjointness information stored in D, one can derive that the label bound by s is distinct from the labels in ρ. Rule Erase also asks for the flag to be true: one must have the permission to erase signatures. This permission is lost when comparing rows of the form $\sigma \cdot \rho$ and $\sigma' \cdot \rho'$ through rule Skip. Intuitively, this restriction is necessary, because otherwise one would be able to erase a signature s from ρ without checking that the labels bound by s and σ are distinct. We omit the technical arguments for the sake of space, but it is possible to exhibit an unsafe program if we consider the subsumption rules without flags.

7.3.3 Examples

Now, we present a number of examples that illustrate the subtleties of the system: how to exploit the implicit assumption that rows are dynamically distinct in order to restrict the effects a program can perform, and how to exploit the subsumption rules.

Filter

As the first example, let us consider the function `filter` from Section 7.1. The goal is to show that `filter` is a well-typed program in Tes and to understand what restrictions are enforced by its type. In particular, we want to require that, during any application `filter` xs f, the effects performed by f do not include *yield* effects. Therefore, a *yield* handler will not accidentally intercept effects performed by f.

Recall the definition of `filter` (Eq. 7.2), which applies the predicate f to each element of xs, and *yields* those elements for which f returns `true`:

$$\begin{array}{l}\texttt{filter}\ xs\ f \triangleq \\ \quad \texttt{let}\ g = (\lambda x.\ \texttt{if}\ f\ x\ \texttt{then perform}\ \mathit{yield}\ x)\ \texttt{in iter}\ xs\ g\end{array}$$

The definition depends on the higher-order iteration method `iter` (Eq. 7.5), which applies a user-provided function to each element of a list. Tes accepts `iter` and assigns to it the type

$$\texttt{iter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \to (\alpha \xrightarrow{\theta} ()) \xrightarrow{\theta} ().$$

Tes also accepts `filter` and assigns to it the following type:

$$\texttt{filter} : \forall\alpha.\ \forall\theta.\ \alpha\ \texttt{list} \to (\alpha \xrightarrow{\theta} \texttt{bool}) \xrightarrow{(\mathit{yield} : \alpha \Rightarrow ()) \cdot \theta} ()$$

Since the body of `filter` is defined as an application of `iter`, to type check `filter`, it suffices to show that the arguments of this application respect `iter`'s type. The only nontrivial step in this task is to show that g has type $\alpha \xrightarrow{Y(\alpha)\cdot\theta} ()$, where α and θ are the variables quantified by `filter`'s type and that are introduced in the first steps of type checking, and where $Y(\alpha)$ is an abbreviation for the signature $(\mathit{yield} : \alpha \Rightarrow ())$. This step is nontrivial because f's row θ does not exactly match the row $Y(\alpha) \cdot \theta$ under which f is called and which must include the signature $Y(\alpha)$ since g might perform this effect. But this step is not a problem for Tes, it suffices to apply rule MonotonicityTyped, which allows the extension of rows: in particular, the relation $D \vdash_b \theta \leq_R Y(\alpha) \cdot \theta$ is derivable under any context D and flag b.

What does `filter`'s type mean? This type means `filter`'s behavior is independent on both the representation of the list elements and the set of effects that f might perform. Moreover, the row $Y(\alpha)\cdot \rho$ tells that the program `filter` xs f performs either *yield* effects or θ effects (introduced by f). Finally, `filter`*'s type prevents* f *from performing yield effects*. At first, it might seem strange that `filter`'s type is polymorphic on θ: what stops one from specializing θ to the singleton row $Y(\alpha)$, thus allowing f to perform *yield*? The answer is: nothing. However, such a specialized version of `filter` would be useless. Implicitly, the type of `filter` imposes the requirement that the row $Y(\alpha)\cdot \theta$ is dynamically distinct. This requirement appears when `filter` is fully applied. So, although one could specialize `filter` with the row $Y(\alpha)$, one would not be able to invoke this version of `filter` because the separation requirement for such a row would not hold.

Counter

In Section 7.1, we introduced the function `counter` (Eq. 7.1.1), which receives a second-order function *ff* as an argument and produces a version of this function that counts the number of times *ff* calls its argument function f. Type-checking `counter` depends on the guarantee that the effects performed by f are not intercepted by the handler installed by `counter`. Although, this behavior is clear from an operational-semantics point of view – the handler targets a fresh dynamic label – to statically ensure it is challenging. Previous type systems that accepted this program [ZM19, BPPS20] were designed for languages restricted to lexically scoped handlers. Therefore, type-safety of `counter` in such systems is a weaker result than the one obtained in Tes, because the language considered in such systems is less expressive than TesLang, which supports traditional handlers and the unrestricted allocation of effect labels. We believe Tes is the only system that supports such a language and accepts `counter`. The closest related previous work of which we are aware is the system λ^{HEL}, devised by Biernacki et al. [BPPS19]. They consider a calculus that also supports unrestricted allocation of effect labels, and where the function `counter` could be similarly defined. However, in λ^{HEL}, the function `counter` is ill-typed. To satisfy the type checker, the programmer would have to place a `lift` coercion around the function call f `()`. [1]

In Tes, the program `counter` can be assigned the following type:

$$\texttt{counter} : \forall\alpha\,\beta\,\gamma.\ (\forall\theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta} \gamma) \to\ (\forall\theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta} (\gamma\ \texttt{*}\ \texttt{int}))$$

[1] This claim was confirmed by the authors via personal communication.

This type means that `counter` works with any effect-polymorphic second-order function *ff* and that `counter` *ff* produces a function whose type is similar to *ff*'s type – the only difference is the return type γ `*` `int`, which is the product of *ff*'s return type and the type of integers. In particular, like the function *ff*, the result of `counter` *ff* is effect-polymorphic: it can be further applied to a function f regardless of the effects performed by this function.

Lexically scoped handlers

As the last example, let us consider the typing rule for lexically scoped handlers. This example illustrates an interesting application of the subsumption rule to erase an absence signature. Recall the definition of a lexically scoped handler (Eq. 7.1) in TesLang as a program that (1) generates a fresh effect label bound by s, and (2) installs a s handler over the application of e to a function that performs this effect:

$$\texttt{lex-handle}_s\ e\ \texttt{with}\ (h \mid r) = \texttt{effect}\ s\ \texttt{in}\ \texttt{handle}\ (e\ (\lambda x.\ \texttt{perform}\ s\ x))\ \texttt{with}\ (s : h \mid r)$$

Tes admits the following derived typing rule for this construct:

$$\textsc{LexHandleTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \forall\theta.\ (\iota \xrightarrow{\theta}_a \tau) \xrightarrow{\theta\cdot\rho}_a \kappa \qquad s \notin \Gamma, \rho, \iota, \tau, \kappa, \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \iota \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho}_a \kappa' \qquad \Xi \mid \Delta \mid \Gamma \vdash_a r : \rho : \kappa \xrightarrow{\rho}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_I \texttt{lex-handle}_s\ e\ \texttt{with}\ (h \mid r) : \rho : \kappa'}$$

(This rule is not original; it is similar to the rule for handlers presented in Figure 3 of the paper [BPPS20].) The expression e must be polymorphic in the effect θ: it must work independently of the label that is bound by s. The first step in the derivation of this rule consists of the application of rule LetEffTyped. This application is the reason that the conclusion of rule LexHandleTyped is marked as impure, and it is also the origin of the freshness condition on s that one sees in this same rule. It is unpleasant to have such a condition polluting the rule, but this condition can be easily satisfied, provided one is willing to rename s to a fresh s'. The second (and last) step in the derivation of LexHandleTyped is the application of rule HandleTyped. The main step to dispatch the premises of rule HandleTyped is to prove that one can introduce s in h's type:

$$\frac{\rho' = (s : \texttt{abs}) \cdot \rho \qquad \Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \iota \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho}_a \kappa'}{\Xi \mid \Delta \mid \Gamma \vdash_a h : \rho : \iota \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa'}$$

To establish this implication, we apply rule MonotonicityTyped. Then, it suffices to show the following chain of subsumption relations:

$$\iota \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho}_a \kappa' \leq_T \iota \to_a (\tau \xrightarrow{\rho}_a \kappa') \xrightarrow{\rho'}_a \kappa' \leq_T \iota \to_a (\tau \xrightarrow{\rho'}_a \kappa') \xrightarrow{\rho'}_a \kappa'$$

The first step follows trivially from the ability to extend rows – by rule RowCons, the relation $\vdash_b \rho \leq_R \rho'$ holds for any b – and the second step follows from the ability to erase absence signatures – by rule Erase, the relation $\langle\rho'\rangle \vdash_{true} \rho' \leq_R \rho$ holds, and, since the right-most arrow includes the row ρ', we can satisfy the separation requirement stored in $\langle\rho'\rangle$.

Weakest precondition $\boxed{wp_a\ e\ \langle E\rangle\{\Phi\}}$

$$wp_a\ e\ \langle E\rangle\{\Phi\} \triangleq \mathit{ValidDistinct}\, E.1 \mathrel{-\!\!*} ewp_a\ e\ \langle E\rangle\{\Phi\}$$

Basic weakest precondition $\boxed{ewp_a\ e\ \langle E\rangle\{\Phi\}}$

$$ewp_a\ v\ \langle E\rangle\{\Phi\} \triangleq \mathit{if}\ a = P\ \mathit{then}\ \Phi(v)\ \mathit{else}\ \dot{\Rightarrow}\Phi(v)$$

$$ewp_a\ (\texttt{eff}\ (\ell, v)\ K)\ \langle E\rangle\{\Phi\} \triangleq \exists\,\Psi.\ (\ell,\Psi)\in E\ *\ (\uparrow_{\square}\Psi)\,v\,(\lambda\,w.\ \triangleright\, ewp_a\ K[w]\ \langle E\rangle\{\Phi\})$$

$$ewp_I\ e\ \langle E\rangle\{\Phi\} \triangleq \forall\sigma.\, S(\sigma)\ {}^{\top}\!\Rrightarrow^{\emptyset} \begin{cases} \exists e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ * \\ \forall e', \sigma'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma'\ {}^{\emptyset}\!\Rrightarrow^{\emptyset} \triangleright\, {}^{\emptyset}\!\Rightarrow^{\top} \\ \qquad S(\sigma')\ *\ ewp_I\ e'\ \langle E\rangle\{\Phi\} \end{cases}$$

$$ewp_P\ e\ \langle E\rangle\{\Phi\} \triangleq \forall\sigma.\, \begin{cases} \exists e'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma\ * \\ \forall e'.\ e\ /\ \sigma \longrightarrow e'\ /\ \sigma \mathrel{-\!\!*} \triangleright\, ewp_P\ e'\ \langle E\rangle\{\Phi\} \end{cases}$$

Figure 7.7: Definition of *wp* and *ewp*.

7.4 Metatheory

In this section, we prove *strong type soundness* of Tes. Strong type soundness states that, if a complete program is accepted by the system, then this program enjoys type and effect safety: it can thus be safely executed, and no effect is left unhandled. To prove this statement, we interpret Tes typing judgments as specifications written in TesLogic, a novel Separation Logic [ORY01] for reasoning about TesLang programs. This approach to prove the soundness of a type system is known as the *semantic approach*[AFM05, KJSB17, KTB17]. This section is structured as follows: first, we present TesLogic; second, we introduce the interpretation of typing judgments; finally, we state and prove the soundness of Tes.

7.4.1 Program logic

Specification language

We have seen in the previous chapters that the notion of protocol works quite well as a logical description of the interaction between handlee and handler. Therefore, it seems natural to keep this notion as the way to specify the effects that a program may perform. The main challenge now is that a TesLang program performs labeled effects to interact with different labeled handlers. Each labeled handler may interpret effects of different nature. For example, the label *exit* may be used to implement an exception and the label *ask* may be used to implement a read-only memory cell containing the value 42. These handlers conform to different disciplines: the *exit* handler, for instance, discards the captured continuation, whereas the *ask* handler immediately resumes the paused computation with 42. Instead of specifying a program that performs both of these

effects, *exit* and *ask*, with a single protocol Ψ, the approach proposed by TesLogic is pretty simple: to specify a program by a list of pairs of labels and protocols.

Indeed, TesLogic's notion of weakest precondition assumes the form $wp_a\ e\ \langle E \rangle \{\Phi\}$, where E is a *protocol list* E, a list of pairs of an effect label and a protocol. The protocol list E maps each effect label ℓ to a protocol describing the ℓ-labeled effects that e performs during its execution.

Figure 7.7 shows the definition of *wp*, which is written in terms of the *basic weakest precondition ewp*. The basic weakest precondition is (recursively) defined by case distinction on a and e. (Recursive calls are guarded by a later modality, thus ensuring that *ewp* is well-defined.) When a is P, neither the update modality nor the state interpretation S [2] occur in the definition of *ewp*. As for the the case distinction on e, we have to consider the three following cases:

1. *Value.* When e is a value v, this value must satisfy the postcondition Φ.
2. *Effect.* When e is an active effect $\texttt{eff}\ (\ell, v)\ K$, the label ℓ must be associated to a protocol Ψ, such that, for every possible answer w ascribed by Ψ to the request v, it is safe to resume K with w. This property is stated in terms of the persistent upward closure (Def. 6.1): $(\uparrow_\square \Psi)\, v\, \Phi$. We use the persistent version of the upward closure, because TesLang allows multi-shot continuations and because Tes does not rule out programs that exploit this feature.
3. *Reducible expression.* When e is neither a value nor an effect, then e must be a reducible expression and every possible reduction e' must be safe.

The weakest precondition *wp* is defined on top of *ewp* by adding the assumption that the list of labels in E is *valid and distinct*: these labels have been allocated and there is no aliasing among them. We write $E.1$ for the list of effect labels in E, that is, the projection of the first component of every pair in E. The valid-and-distinct property is captured by the predicate *ValidDistinct* :

$$\mathit{ValidDistinct}\ L \triangleq \mathit{NoDup}\ L \wedge (\forall \ell \in L.\ \ell \mapsto_\square ())$$

The assertion $\mathit{NoDup}\ L$ captures the non-aliasing claim: it states that there are no duplicates in the list L. The assertion $\ell \mapsto_\square ()$ is the claim that ℓ has been allocated. Because both of these assertions are persistent, so is the assertion $\mathit{ValidDistinct}\ L$. This is a desirable property, because, as we shall discuss in Subsection 7.4.1, the frame rule is unsound in TesLogic. Therefore, working with non-persistent assertions in TesLogic is problematic.

Reasoning rules

Figure 7.8 shows a subset of the reasoning rules of TesLogic. These are the most relevant rules to the soundness proof of Tes. We briefly discuss each one of them. We say that a reasoning rule *justifies* a typing rule to mean that this is the key rule to prove that the typing rule preserves the semantic interpretation of judgments.

[2] The definition of S here is the same as in Chapter 2: the assertion $S(\sigma)$ states that the authoritative piece of γ_{heap} is σ; that is, $S(\sigma) \triangleq \boxed{\bullet\, \sigma}^{\gamma_{heap}}$.

$$\textsc{TesValue}\ \frac{\Phi(v)}{wp_a\ v\ \langle E\rangle\{\Phi\}} \qquad \textsc{TesMonotonicity}\ \frac{a \leq_A a' \qquad \Box\forall w.\, \Phi(w) \mathrel{-\!\!*} \Phi'(w) \qquad wp_a\ e\ \langle E\rangle\{\Phi\} \qquad E \leq_L E'}{wp_{a'}\ e\ \langle E'\rangle\{\Phi'\}}$$

$$\textsc{TesBind}\ \frac{K\ \textit{is neutral} \qquad wp_a\ e\ \langle E\rangle\{v.\, wp_a\ K[v]\ \langle E\rangle\{\Phi\}\}}{wp_a\ K[e]\ \langle E\rangle\{\Phi\}}$$

$$\textsc{TesPerform}\ \frac{(\ell, \Psi) \in E \qquad \Psi\ \textit{allows}\ \texttt{eff}\ v\ \{\Phi\}}{wp_a\ (\texttt{perform}\ \ell\ v)\ \langle E\rangle\{\Phi\}} \qquad \textsc{TesLetEff}\ \frac{\forall \ell.\, wp_a\ (e\,[\ell/s])\ \langle(\ell, \bot) :: E\rangle\{\Phi\}}{wp_I\ (\texttt{effect}\ s\ \texttt{in}\ e)\ \langle E'\rangle\{\Phi'\}}$$

$$\textsc{TesHandle}\ \frac{\textit{Handler}_a\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle E'\rangle\{\Phi'\} \qquad E' = (\ell, \Psi') :: E \qquad wp_a\ e\ \langle(\ell, \Psi) :: E\rangle\{\Phi\}}{wp_a\ (\texttt{handle}\ e\ \texttt{with}\ (\ell\ \texttt{:}\ h \mid r))\ \langle E'\rangle\{\Phi'\}}$$

$$\textsc{TesInfinitaryConjunction}\ \frac{E\ \textit{is pure} \qquad \Box\forall x.\ wp_P\ e\ \langle E\rangle\{y.\, \Phi(x, y)\}}{wp_a\ e\ \langle E\rangle\{y.\, \forall x.\, \Phi(x, y)\}}$$

Figure 7.8: Reasoning rules

Rule TesValue expresses the idea that a program can terminate by returning a value v that satisfies the postcondition. This rule justifies the typing rules of values (rules UnitTyped, BoolTyped, and IntTyped).

Rule TesMonotonicity expresses the idea that, if a program e satisfies the specification $ewp_a\ e\ \langle E\rangle\{\Phi\}$, then it also satisfies a specification $ewp_{a'}\ e\ \langle E'\rangle\{\Phi'\}$, where a', E', and Φ' are weaker than than a, E, and Φ, respectively. The predicate Φ' is weaker than Φ if $\Phi(v)$ implies $\Phi'(v)$ for every v. The premise of rule TesMonotonicity stating this implication is covered by a persistence modality, because, in the presence of multi-shot continuations, a program may terminate multiple times. This persistence modality is the reason why *the frame rule does not hold in TesLogic*; only a restricted version holds: one can apply the frame rule under the empty protocol list []. Intuitively, the empty protocol list states the absence of effects, and, under this restriction, a program can never be part of the context captured by a multi-shot continuation. The protocol list E' is weaker than E, noted $E \leq_L E'$, if $E.1$ is valid and distinct, under the assumption that $E'.1$ is valid and distinct, and if for every Ψ in E there is a protocol Ψ' in E' that describes the same effect ℓ as Ψ and that is weaker than Ψ:

$$E \leq_L E' \ \triangleq\ \begin{array}{l}(\mathit{ValidDistinct}\ E'.1 \mathrel{-\!\!*} \mathit{ValidDistinct}\ E.1) \wedge \\ (\Box\forall(\ell, \Psi) \in E, v, \Phi.\ (\uparrow_\Box \Psi)\ v\ \Phi \mathrel{-\!\!*} \exists\Psi'.\ (\ell, \Psi') \in E' \wedge (\uparrow_\Box \Psi')\ v\ \Phi)\end{array}$$

The property of Ψ' being weaker than Ψ is the assertion that every pair of a request v and a predicate Φ that is allowed by Ψ is also allowed by Ψ'.

Rule TesBind allows *context-local reasoning*: one can reason about the execution of a program independently of its surrounding evaluation context. This rule is surprising given the non-local nature of effects and handlers. There is however a side condition: the context K must be *neutral*, that is, K cannot contain a frame of the form $\texttt{handle}\ \_\ \texttt{with}\ (\ell\ \texttt{:}\ h \mid r)$.

To reason about non-neutral contexts, one can employ rule TesHandle. This rule expresses the idea that to install a handler for a label ℓ around a program e, it suffices to know by which protocol e abides when performing effects labeled ℓ. Indeed, the rule asks

for the verification of e according to a protocol list that associates ℓ to Ψ. The verification of r and h is delegated to the *handler judgment*, noted $\mathit{Handler}$, which compresses the specifications of r and h into a single assertion:

$$\begin{array}{l}\mathit{Handler}_a\ \langle\Psi\rangle\{\Phi\}\ (h \mid r)\ \langle E'\rangle\{\Phi'\} \triangleq \\ \quad (\Box\forall v.\ \Phi(v) \mathrel{-\!\!*} wp_a\ (r\ v)\ \langle E'\rangle\{\Phi'\}) \quad \wedge \\ \quad \left(\begin{array}{ll} \Box\forall v,\ k. & (\uparrow_\Box \Psi)\ v\ (\lambda w.\ wp_a\ (k\ w)\ \langle E'\rangle\{\Phi'\}) \mathrel{-\!\!*} \\ & wp_a\ (h\ v\ k)\ \langle E'\rangle\{\Phi'\} \end{array}\right)\end{array}$$

Indeed, the handler judgment is the conjunction of r's and h's specifications. The specification of r assumes that v satisfies e's postcondition and claims that r satisfies the postcondition Φ' and protocol list E'. The specification of h requires h to satisfy postcondition Φ' and protocol list E' under two assumptions: (1) that v corresponds to the payload of a effect labeled ℓ performed by a program that abides by Ψ, and (2) that k represents this suspended program.

Rule TesPerform tells that one can reason about performing an effect as calling a function. A protocol Ψ dictates which answer a client can expect in exchange for a value v in a similar way as a program specification dictates which result one can expect from a function call.

Rule TesLetEff justifies the typing rule LetEffTyped. The protocol $\bot$, to which the label ℓ is initially associated, is the *empty protocol*. It is defined as $\lambda\_\ \_.\ \mathsf{False}$. A program that abides by this protocol performs no effect, because the assertion $\bot$ *allows* `eff _ {_}` never holds. The proof of rule TesLetEff is essentially the proof that, after the allocation of a fresh effect label, the list $E' = (\ell, \bot) :: E$ is valid and distinct, given that E is valid and distinct. The key step in this proof is the introduction of a full points-to assertion $\ell \mapsto ()$, which we exploit to prove that ℓ is not an alias of a previously allocated label, and which we update to a persistent points-to assertion $\ell \mapsto_\Box ()$ to complete the proof that E' is valid and distinct.

Rule TesInfinitaryConjunction justifies the rules for introducing polymorphic types, namely rules TLamTyped and RLamTyped. This rule is atypical, because the unrestricted infinitary conjunction rule does not hold in Separation Logic [O'H07]. From a technical point of view, the unrestricted infinitary conjunction rule does not hold in the particular case of Iris, because a universal quantification does not commute with the update modality. Here, we are able to prove a restricted version of the infinitary conjunction rule: the premise includes the attribute P, which excludes the update modality from the definition of wp_P. Another restriction is that the protocol list E must be *pure*. The list E is pure, if every protocol in E is *pure*. A protocol Ψ is pure if the following assertion holds:

$$\forall v.\ \exists Q.\ \forall Q'.\ \Psi\ v\ Q' \mathrel{-\!\!*} (\Psi\ v\ Q\ *\ (\forall w.\ Q\ w \mathrel{-\!\!*} \Box Q'\ w))$$

This is a sufficient condition for proving the rule TesInfinitaryConjunction. Intuitively, the definition of a pure protocol Ψ asserts that Ψ admits a minimal set of answers described by a predicate Q, and that this predicate is persistent, hence a *pure* protocol. To see why this condition is useful in the proof of TesInfinitaryConjunction, one must consider the case where e is an active effect and unfold the definition of the persistent upward closure (Def. 6.1 of Chapter 6). In this case, the premise of this rule is an assertion of the form "$\Box\forall x \ldots \exists Q' \ldots$", whereas the goal assumes the form "$\exists Q' \ldots \forall x \ldots$". The

existentially quantified predicate Q' comes from the unfolding of the upward closure. The existence of a minimal predicate Q means that both occurrences of Q' are in essence the same predicate; that is, Q' does not depend on x.

Soundness of TesLogic

TesLogic's adequacy theorem is stated as follows:

Theorem 7.1 (Adequacy) *If* $wp_a\ e\ \langle[]\rangle\{\Phi\}$ *holds then* e *is safe.*

Recall that the adequacy theorem of a program logic justifies reasoning in terms of weakest preconditions: Theorem 7.1 states that the TesLogic assertion $wp_a\ e\ \langle[]\rangle\{\Phi\}$ implies the meta-level assertion that e is safe; that is, the execution of e either diverges or terminates with a value, but it does not crash, nor does it perform an unhandled effect.

7.4.2 Semantic interpretation

Semantic interpretation of typing judgments

The semantic interpretation of judgments translates Tes typing judgments to TesLogic specifications: it maps $\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau$ to a *semantic judgment* $\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau$. The definition of a semantic judgment is written in terms of the *semantic interpretation of types, signatures, and rows*. Types are interpreted as *semantic types* that inhabit $\mathit{SemType} \triangleq \{P : \mathit{Val} \to \mathit{iProp} \mid \forall v.\ P(v)\ \mathit{is\ persistent}\}$. Therefore, semantic types are *persistent predicates*, which can be seen as sets of values. The persistence requirement reflects the lack of a substructural discipline in Tes, that is, Tes is not an affine type system. Signatures are interpreted as *semantic signatures* that inhabit $\mathit{SemSig} \triangleq \{E : \mathit{List}\,(\mathit{Loc} \times \mathit{Protocol}) \mid E\ \mathit{is\ pure}\}$. Therefore, semantic signatures are pure protocol lists. Rows are interpreted as *semantic rows* that inhabit *SemRow*, which coincide with semantic signatures, that is, semantic rows are also pure protocol lists.

The interpretation of types, signatures, and rows appears in Figure 7.9. It has two parameters: (1) a row- and type-variable map η, which maps row variables to semantic rows and type variables to semantic types; and (2) an effect-name map δ, which maps effect names s to effect labels.

The interpretation of types follows the standard approach for the definition of *unary logical relations* in Iris [KTB17]. The only case where our interpretation deviates from previous works is in the interpretation of arrow types. The interpretation of an arrow type $\kappa \xrightarrow{\rho}_a \tau$ contains values v that satisfy the specification $\Box\forall w.\, \mathcal{V}[\![\kappa]\!]^\delta_\eta(w) \mathrel{-\!\!*} wp_a\ (v\ w)\ \langle \mathcal{R}[\![\rho]\!]^\delta_\eta \rangle \{y.\mathcal{V}[\![\tau]\!]^\delta_\eta(y)\}$. This specification states that v produces values in the interpretation of τ when applied to a value in the interpretation of κ. The novel aspect of this interpretation is the use of our notion of the weakest precondition, *wp*, which describes the effects that v might perform. The description of these effects is given by the interpretation of the row ρ: it is the list concatenation of the interpretation of the signatures in ρ. A row variable θ is interpreted as the semantic row $\eta(\theta)$, and a concrete signature $(s : \iota' \Rightarrow \tau')$ is interpreted as the (singleton list containing the) pair of the dynamic label to which s is bound and the protocol dictating that the answer to a value in the interpretation of ι' is a value in the interpretation of τ'. Therefore, v's specification states that v performs effects according to the interpretation of ρ. Moreover,

Interpretation of types. $\boxed{\mathcal{V}[\![\tau]\!]^\delta_\eta : \mathit{SemType}}$

$$
\begin{array}{rcl}
\mathcal{V}[\![\bot]\!]^\delta_\eta(\_) & \triangleq & \mathsf{False} \\
\mathcal{V}[\![\top]\!]^\delta_\eta(\_) & \triangleq & \mathsf{True} \\
\mathcal{V}[\![()]\!]^\delta_\eta(v) & \triangleq & (v = ()) \\
\mathcal{V}[\![\mathtt{bool}]\!]^\delta_\eta(v). & \triangleq & \exists b.\, v = b \\
\mathcal{V}[\![\mathtt{int}]\!]^\delta_\eta(v) & \triangleq & \exists i.\, v = i \\
\mathcal{V}[\![\alpha]\!]^\delta_\eta(v) & \triangleq & \eta(\alpha)(v) \\
\mathcal{V}[\![\tau\ \mathtt{list}]\!]^\delta_\eta(v) & \triangleq & v = [] \lor \exists u, us.\, v = u :: us \ast \mathcal{V}[\![\tau]\!]^\delta_\eta(u) \ast \triangleright \mathcal{V}[\![\tau\ \mathtt{list}]\!]^\delta_\eta(us) \\
\mathcal{V}[\![\kappa * \tau]\!]^\delta_\eta(v) & \triangleq & \exists w, w'.\, v = (w, w') \ast \mathcal{V}[\![\kappa]\!]^\delta_\eta(w) \ast \mathcal{V}[\![\tau]\!]^\delta_\eta(w') \\
\mathcal{V}[\![\kappa + \tau]\!]^\delta_\eta(v) & \triangleq & \exists w.\, (v = \mathtt{inj}_1\ w \ast \mathcal{V}[\![\kappa]\!]^\delta_\eta(w)) \lor (v = \mathtt{inj}_2\ w \ast \mathcal{V}[\![\tau]\!]^\delta_\eta(w')) \\
\mathcal{V}[\![\tau\ \mathtt{ref}]\!]^\delta_\eta(v) & \triangleq & \exists \ell.\, v = \ell \ast \boxed{\exists w.\, \ell \mapsto w \ast \mathcal{V}[\![\tau]\!]^\delta_\eta(w)}^{\mathcal{N}.\ell} \\
\mathcal{V}[\![\kappa \xrightarrow{\rho}_a \tau]\!]^\delta_\eta(v) & \triangleq & \Box \forall w.\, \mathcal{V}[\![\kappa]\!]^\delta_\eta(w) \mathrel{-\!\!*} wp_a\ (v\ w)\ \langle \mathcal{R}[\![\rho]\!]^\delta_\eta \rangle \{y. \mathcal{V}[\![\tau]\!]^\delta_\eta(y)\} \\
\mathcal{V}[\![\forall \alpha.\, \tau]\!]^\delta_\eta(v) & \triangleq & \forall A.\, \mathcal{V}[\![\tau]\!]^\delta_{\eta, \alpha \mapsto A}(v) \\
\mathcal{V}[\![\forall \theta.\, \tau]\!]^\delta_\eta(v) & \triangleq & \forall E.\, \mathcal{V}[\![\tau]\!]^\delta_{\eta, \theta \mapsto E}(v)
\end{array}
$$

Interpretation of rows. $\boxed{\mathcal{R}[\![\rho]\!]^\delta_\eta : \mathit{SemRow}}$

$$\mathcal{R}[\![\rho]\!]^\delta_\eta \quad \triangleq \quad \bigcup_{\sigma \in \rho} \mathcal{S}[\![\sigma]\!]^\delta_\eta$$

Interpretation of signatures. $\boxed{\mathcal{S}[\![\sigma]\!]^\delta_\eta : \mathit{SemSig}}$

$$
\begin{array}{rcl}
\mathcal{S}[\![(s : \iota \Rightarrow \tau)]\!]^\delta_\eta & \triangleq & (\delta(s),\ \lambda v\, Q.\, \mathcal{V}[\![\iota]\!]^\delta_\eta(v) \ast \Box \forall w.\, \mathcal{V}[\![\tau]\!]^\delta_\eta(w) \mathrel{-\!\!*} Q(w)) \\
\mathcal{S}[\![\theta]\!]^\delta_\eta & \triangleq & \eta(\theta)
\end{array}
$$

Interpretation of typing judgments. $\boxed{\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau}$

$$
\begin{array}{l}
\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau \triangleq \\
\quad \forall \eta, \delta, vs.\, (\forall \{x \mapsto \kappa\} \subseteq \Gamma.\, \mathcal{V}[\![\kappa]\!]^\delta_\eta(vs(x))) \mathrel{-\!\!*} wp_a\ (e[vs][\delta])\ \langle \mathcal{R}[\![\rho]\!]^\delta_\eta \rangle \{y.\, \mathcal{V}[\![\tau]\!]^\delta_\eta(y)\}
\end{array}
$$

Figure 7.9: Interpretation of types, rows, signatures, and typing judgments.

Interpretation of disjointness contexts. $\boxed{[\![D]\!]^{\delta}_{\eta} : iProp}$

$$[\![D]\!]^{\delta}_{\eta} \triangleq (\forall s \in dom(D).\ \delta(s) \mapsto_{\square} ()) \wedge (\forall \{s \mapsto (S, V)\} \subseteq D.\ \delta(s) \notin \delta(S) \wedge \delta(s) \notin \eta(V).1)$$

Interpretation of the subsumption relation on types. $\boxed{D \vDash \tau \leq_T \tau}$

$$D \vDash \iota \leq_T \tau \triangleq \square\forall \eta, \delta.\ [\![D]\!]^{\delta}_{\eta} \mathrel{-\!\!*} \forall v.\ \mathcal{V}[\![\iota]\!]^{\delta}_{\eta}(v) \mathrel{-\!\!*} \mathcal{V}[\![\tau]\!]^{\delta}_{\eta}(v)$$

Interpretation of the subsumption relation on rows. $\boxed{D \vDash_b \rho \leq_R \rho}$

$$D \vDash_b \rho \leq_R \rho' \triangleq \square\forall \eta, \delta.\ [\![D]\!]^{\delta}_{\eta} \mathrel{-\!\!*} (\mathcal{R}[\![\rho]\!]^{\delta}_{\eta} \leq_L \mathcal{R}[\![\rho']\!]^{\delta}_{\eta} \wedge (b = \mathit{false} \mathrel{-\!\!*} \mathcal{R}[\![\rho]\!]^{\delta}_{\eta}.1 \subseteq_m \mathcal{R}[\![\rho']\!]^{\delta}_{\eta}.1))$$

Interpretation of the subsumption relation on signatures. $\boxed{D \vDash \sigma \leq_S \sigma'}$

$$D \vDash \sigma \leq_S \sigma' \triangleq \square\forall \eta, \delta.\ [\![D]\!]^{\delta}_{\eta} \mathrel{-\!\!*} (\mathcal{S}[\![\sigma]\!]^{\delta}_{\eta} \leq_L \mathcal{S}[\![\sigma']\!]^{\delta}_{\eta} \wedge \mathcal{S}[\![\sigma]\!]^{\delta}_{\eta}.1 = \mathcal{S}[\![\sigma']\!]^{\delta}_{\eta}.1)$$

Figure 7.10: Interpretation of disjointness contexts and of the subsumption relation.

the implicit valid-and-distinct property in the definition of *wp* unfolds as the assertion $\mathit{ValidDistinct}\, \mathcal{R}[\![\rho]\!]^{\delta}_{\eta}.1$. This assertion means that the list of dynamic labels bound by the effect names in ρ is valid and distinct, or, using the terminology introduced in Section 7.3, that ρ is dynamically distinct.

The meaning of the semantic judgment $\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau$ can be given as follows: for every row- and type-variable map η, effect-name map δ, and for every substitution vs of variables x in $dom(\Gamma)$ with values in the interpretation of $\Gamma(x)$, the complete program $e[vs][\delta]$ produces a value in the semantic type $\mathcal{V}[\![\tau]\!]^{\delta}_{\eta}$ and performs effects according to the semantic row $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$.

Semantic interpretation of the subsumption relation

The *semantic interpretation of a subsumption relation on types, signatures, or rows* is a TesLogic relation on the interpretation of types, signatures, or rows, respectively. Like the semantic judgment, the semantic interpretation of a subsumption relation employs a double turnstile symbol $\vDash$ to differentiate it from the syntactic subsumption relation. Its definition appears in Figure 7.10. This definition depends on the *interpretation of disjointness contexts*.

The interpretation of a disjointness context D asserts, for every effect name s in the domain of D, that s has been allocated, the assertion $\delta(s) \mapsto_{\square} ()$ holds, and that D maps s to a pair of a set of names S and a set of row variables V, such that there is no aliasing between s and S, $\delta(s)$ does not belong to the image of δ over S, nor between s and V, $\delta(s)$ does not belong to the image of η over V.

The interpretation of the relation $D \vdash \kappa \leq_T \tau$ asserts that, under the non-aliasing assumption $[\![D]\!]^{\delta}_{\eta}$, every value in the interpretation of κ belongs to the interpretation of τ. The interpretation of the relation $D \vdash_b \rho \leq_R \rho'$ asserts that the interpretation of ρ' is weaker than the interpretation of ρ (according to the order $\_ \leq_L \_$, defined in

Subsection 7.4.1). Moreover, it also asserts that, if the permission b to erase absence signatures is off, then every label appears in $\mathcal{R}[\![\rho]\!]^{\delta}_{\eta}$ with multiplicity equal to, or less than, its multiplicity in $\mathcal{R}[\![\rho']\!]^{\delta}_{\eta}$. We express this property by implicitly regarding lists as multisets and by exploiting the subset relation on multisets $\_ \subseteq_m \_$. The interpretation of the relation $D \vdash \sigma \leq_S \sigma'$ asserts that (1) the interpretation of σ' is weaker than the interpretation of σ and that (2) the list of labels in $\mathcal{S}[\![\sigma]\!]^{\delta}_{\eta}.1$ is equal to $\mathcal{S}[\![\sigma']\!]^{\delta}_{\eta}.1$. This equality expresses the fact that the subsumption relation allows one to weaken/strengthen the types that appear in a signature, but not their effect names.

7.4.3 Soundness of Tes

The *soundness theorem* states that, if Tes accepts a program e, then e is safe (that is, e either diverges or terminates with a value):

Theorem 7.2 (Soundness) *If the judgment* $\emptyset \mid \emptyset \mid \emptyset \vdash_a e : \langle\rangle : ()$ *is derivable, then* e *is safe.*

To prove this theorem, we first establish the *fundamental theorem*:

Theorem 7.3 (Fundamental Theorem) *If the syntactic typing judgment* $\Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau$ *is derivable, then its semantic interpretation* $\Xi \mid \Delta \mid \Gamma \vDash_a e : \rho : \tau$ *holds.*

By exploiting Theorem 7.3, it follows that, if Tes assigns a program e the typing judgment $\emptyset \mid \emptyset \mid \emptyset \vdash_a e : \langle\rangle : ()$, then e enjoys the following specification: $wp_a\ e\ \langle \mathcal{R}[\![\langle\rangle]\!]^{\emptyset}_{\emptyset} \rangle \{ \mathcal{V}[\![()]\!]^{\emptyset}_{\emptyset} \}$.

To complete the proof of Theorem 7.2, it suffices to simplify $\mathcal{R}[\![\langle\rangle]\!]^{\emptyset}_{\emptyset}$ as [], and to apply Theorem 2.1.

The proof of Theorem 7.3 proceeds by induction on the derivation of typing judgments. Each typing rule gives rise to the proof obligation that the interpretation of judgments in the premise implies the interpretation of the judgment in the conclusion. For each typing rule, there is a well-suited reasoning rule allowing this proof obligation to be completed inside the TesLogic logic. The case of the typing rule MonotonicityTyped relies on the *Fundamental Theorem of The Subsumption Relation*:

Theorem 7.4 (Fundamental Theorem of The Subsumption Relation) *(1) If* $D \vdash \kappa \leq_T \tau$ *is derivable then* $D \vDash \kappa \leq_T \tau$ *holds; (2) If* $D \vdash \sigma \leq_S \sigma'$ *is derivable then* $D \vDash \sigma \leq_S \sigma'$ *holds; (3) If* $D \vdash_b \rho \leq_R \rho'$ *is derivable then* $D \vDash_b \rho \leq_R \rho'$ *holds.*

Each of the statements in Theorem 7.4 is proved by induction on the subsumption derivation.

7.5 Related work

Hillerström and Lindley [HL16] study a core formal calculus that serves as a model of Links [CLWY06], a functional programming language for web applications that the authors extend with support for effect handlers. Taking advantage of Links's existing row-based approach to type-checking records, the authors consider a similar approach to

type-checking effectful programs: they annotate an arrow with a row of labels denoting the effects that a function might perform. Moreover, the system has support for row polymorphism following Rémy's discipline [Ré89]: the kind system, in combination with a syntactic well-formedness criteria, ensures that labels in a row are pairwise distinct. Links does not support the dynamic generation of effect labels.

Bauer and Pretnar [BP14] present the theoretical foundation of *Eff*, a programming language with support for handlers, and dynamic generation of effect labels. However, the authors consider a subset of *Eff* that excludes the generation of effect labels. They endow this core *Eff* calculus with a row-based type system that ensures strong type safety. Later [BP15], the same authors extend this core calculus with support for dynamic generation of effect labels. However, in this extended calculus, strong type safety no longer holds: well-typed programs can perform unhandled effects. The system has support for value-polymorphic types, whose introduction applies only to programs that do not perform effects.

Leijen [Lei17] formalizes a subset of the Koka language [Lei20]. This formalization consists of a calculus with support for handlers and globally defined effects, of a type system with support for both value and effect polymorphism, and of a compilation strategy for explicitly typed programs. This strategy relies on a *selective CPS transformation* [Nie01], which Leijen extends with support for effect-polymorphic programs. As in Tes, an arrow type in Leijen's type system is annotated with a row of effects. Unlike Tes, a row in Leijen's system is *univariate*: it can have at most one row variable. In Tes, the ability to have multiple variables in a row is important, for example, in the statement of the typing rule of a lexically scoped handler. Indeed, the client of such a handler has an effect-polymorphic type $\forall\theta.\ (\alpha \xrightarrow{\theta} \beta) \xrightarrow{\theta \cdot \rho} \tau$, where θ stands for the effect that is allocated by the handler. If rows were univariate, then the row ρ could contain no variables, because, otherwise, the row $\theta \cdot \rho$ would not be univariate. Tes does not impose such a restriction. Another difference is the introduction of polymorphic types: in Tes, the generalization of type and row variables applies to programs that perform effects, whereas Leijen argues that, in a possible extension with primitive references, generalization would apply only to *total programs*: programs type-checked under the empty row, and, consequently, that do not perform effects. Tes shows that this restriction is unnecessary: even in the presence of references, it is sound to generalize the type of a program that performs effects, provided that it does not interact with the store, that is, it neither uses references nor allocates labels.

A notable omission from Leijen's formalization is Koka's `inject` [Lei14], a construct that works as a `lift` coercion. Biernacki et al. [BPPS18] are the first authors to provide a formal treatment of a calculus with such a construct. In addition to its formal operational semantics, they design a row-based type system (with support for effect polymorphism through univariate rows) for this calculus. They conceive the first binary logical relations for handlers, and they apply these relations to prove that their system is sound. In a later paper [BPPS19], the same authors introduce λ^{HEL}, a calculus with support for both the dynamic allocation of effect labels and effect coercions. In addition to the `lift` coercion, they consider (1) the `swap` coercion, which exchanges two effects in a row, (2) the `cons` coercion, which rearranges effects deep in a row, and (3) the composition of coercions. These coercions do not add expressiveness to the language: they can all be written in

terms of `lift`. Still, they facilitate how the programmer can modify the way in which an effect dynamically searches for a handler. They equip this calculus with a type system with support for effect-polymorphic, value-polymorphic, and existential types. Although the function `counter`, discussed in Sections 7.1 and 7.3, is expressible in λ^{HEL}, the type system designed by the authors does not accept this program. (This has been confirmed by the authors via personal communication.) The technical reason why this program is rejected is that the subsumption rules of their system are not flexible enough: a call to a function f with an abstract row θ cannot occur in a context whose row is not exactly θ. (It is not trivial how to overcome this issue, because, in their system, the interpretation of a signature depends on the signature's position in the row.) In Tes, such a function call can occur in any context whose row includes the variable θ.

Zhang and Myers [ZM19] introduce the *tunneling semantics* as a novel operational semantics that avoids accidental handling by construction. This semantics, however, is presented only informally as a program transformation. This transformation is not presented in the setting of $\lambda_{\Downarrow\Uparrow}$, a formal calculus introduced by the authors. Furthermore, as noted by Biernacki et al. [BPPS20], there is a discrepancy between the paper presentation of $\lambda_{\Downarrow\Uparrow}$ and its Coq formalization. In the article, there is no dynamic generation of fresh labels, whereas, in the Coq formalization, one finds a calculus with support for this feature through a construct that corresponds to a lexically scoped handler: it introduces a fresh effect label and installs a handler for this label. Therefore, if we take the Coq formalization as the main reference, then Zhang and Myers's work consists of the introduction of a calculus for lexically scoped handlers, of a type system for this calculus with support for effect polymorphism, and the soundness proof of this system using binary logical relations. They apply these binary logical relations to show interesting program equivalences. One of these equivalences, for example, shows that an effect-polymorphic function cannot intercept effects abstracted by a row variable. This property seems aligned with the intuitive idea of *absence of accidental handling*, but a formal definition of this term still does not exist. Zhang and Myers and other authors [BSO20b] suggest that the absence of accidental handling is equivalent to the parametricity of effect polymorphism. However, this definition is unsatisfactory, because there are systems both with parametric polymorphism and that allow programs to intercept effects abstracted by row variables. One example is the system Tes+Wind obtained by extending TesLang with `dynamic-wind` [FYFF07] construct, `dynamic-wind` $p\ e\ q$, which monitors the execution of e by invoking the thunk p when control enters e (at the beginning of e's execution and every time e is resumed) and by invoking the thunk q when control leaves e (at the end of e's execution and every time e performs an unhandled effect). Then, we extend Tes with the following typing rule:

$$\textsc{DynamicWindTyped}\quad \frac{\Xi \mid \Delta \mid \Gamma \vdash_a p : \rho : () \to_a () \qquad \Xi \mid \Delta \mid \Gamma \vdash_a e : \rho : \tau \qquad \Xi \mid \Delta \mid \Gamma \vdash_a q : \rho : () \to_a ()}{\Xi \mid \Delta \mid \Gamma \vdash_a \texttt{dynamic-wind}\ p\ e\ q : \rho : \tau}$$

This rule preserves the interpretation presented in Section 7.4. Therefore, Tes+Wind has parametric effect polymorphism, because an effect-polymorphic type is interpreted by a universal quantifier, but it accepts the program `dynamic-wind` $p\ e\ q$, which breaks Zhang and Myers's equivalences and is thus an example of accidental handling.

Despite of their previous formal study of effect coercions, in the paper [BPPS20], Biernacki et al. argue against these constructs, which they deem as impractical for real-world programming, and conceive a type system for a language with support for lexically scoped handlers only. The authors present two semantics for this language: (1) the *open semantics*, where effect names are not substituted with labels, and evaluation is defined among open terms in a capture-avoiding way, and (2) the *generative semantics*, where, as in TesLang, effect names are substituted with dynamic labels. By means of binary logical relations, the authors show that the system is sound and that these semantics are equivalent.

Schuster et al. [SBMO22] show that well-typed lexically scoped handlers admit a type-preserving CPS translation to System F. This translation erases the dynamic generation of effect labels: the correspondence between effect-invocation sites and handlers is thus statically inferred. This translation depends on the information gathered by their type system, which includes a notion of *regions* [TT97], to denote the scope of a handler, and a subsumption relation on regions to represent nested scopes.

Brachthäuser, Schuster, and Ostermann [BSO20b] develop a library in Scala for programming with effect handlers. The library implemented using multi-prompt delimited continuations. The operational behavior obtained through this encoding is similar to lexically scoped handlers: installing a handler marks the stack with a fresh label; performing an effect dynamically searches for the nearest matching label. The authors exploit Scala's support for path-dependent types and intersection types to encode effect polymorphism and to guarantee effect safety.

Kammar and Pretnar [KP17] show that a handler calculus without references and without allocation of effect labels admits a type system with unrestricted introduction of value-polymorphic types. In particular, the generalization of type variables applies to a program that handles and performs effects. Kammar and Pretnar establish the soundness of their system through the syntactic approach [WF94], whereas we establish Tes's soundness through the semantic approach: we provide a semantic interpretation of judgments as specifications in a program logic. This change in perspective leads to a key observation: from a semantic point of view, combining polymorphism with general references is akin to commuting a universal quantifier with an update modality.

Bibliography

[AFM05] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *International Conference on Functional Programming (ICFP)*, pages 78–91, September 2005.

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

[BB18] Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic. Lectures notes, December 2018.

[BD04] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In *Logic Based Program Synthesis and Transformation*, pages 143–159. Springer, 2004.

[Ber09] Martin Berger. Program logics for sequential higher-order control. In *Fundamentals of Software Engineering*, volume 5961 of *Lecture Notes in Computer Science*, pages 194–211. Springer, April 2009.

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

[BP14] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.

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

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

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

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

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

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

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

[BS17] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: extensible algebraic effects in Scala. In *Symposium on Scala*, pages 67–72, October 2017.

[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.

[Cha11] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *International Conference on Functional Programming (ICFP)*, pages 418–430, September 2011.

[Cha21] Arthur Charguéraud. *Separation Logic Foundations*, volume 6 of *Software Foundations*. 2021. http://softwarefoundations.cis.upenn.edu.

[Cha22] Arthur Charguéraud. The CFML tool and library. http://www.charglueraud.org/softs/cfml/, 2022.

[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.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, November 2006.

[Cop92] James Coplien. *Advanced C++ Programming Styles And Idioms*. Addison-Wesley, 1992.

[Coq20] Coq Development Team. *The Coq Proof Assistant*, 2020.

[Cor22] Microsoft Corporation. C# documentation, 2022. https://docs.microsoft.com/en-us/dotnet/csharp/.

[CP11] Tristan Crolard and Emmanuel Polonowski. A program logic for higher-order procedural variables and non-local jumps. December 2011.

[CP12] Tristan Crolard and Emmanuel Polonowski. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *Journal of Logical and Algebraic Methods in Programming*, 81(3):181–208, 2012.

[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.

[dMI09] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):1–31, February 2009.

[DN13] Germán Andrés Delbianco and Aleksandar Nanevski. Hoare-style reasoning with (algebraic) continuations. In *International Conference on Functional Programming (ICFP)*, pages 363–376, September 2013.

[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. https://hal.inria.fr/hal-03049514/document.

[dVP22a] Paulo Emílio de Vilhena and François Pottier. A type system for effect handlers and dynamic effect labels. 2022.

[dVP22b] Paulo Emílio de Vilhena and François Pottier. Verifying an effect-handler-based define-by-run reverse-mode AD library. 2022. https://arxiv.org/abs/2112.07292.pdf.

[Ecm22] Ecma. ECMAScript 2022 Language Specification, 2022. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[FHKW84] Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker, and Mitchell Wand. The Scheme 84 Reference Manual. Technical report no. 154, Indiana University, Computer Science Department, March 1984.

[Fil94] Andrzej Filinski. Representing monads. In *Principles of Programming Languages (POPL)*, pages 446–457, February 1994.

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In *Logic in Computer Science (LICS)*, pages 442–451, July 2018.

[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.

[Fla21a] Matthew Flatt. The Racket Reference, 2021. https://docs.racket-lang.org/reference/.

[Fla21b] Matthew Flatt. The Racket Reference – Continuations, 2021. https://docs.racket-lang.org/reference/cont.html.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[FP16] Jean-Christophe Filliâtre and Mário Pereira. A modular way to reason about iteration. In *NASA Formal Methods (NFM)*, volume 9690 of *Lecture Notes in Computer Science*, pages 322–336. Springer, June 2016.

[FYFF07] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *International Conference on Functional Programming (ICFP)*, pages 165–176, October 2007.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, November 2007.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150. Springer, 2010.

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

[Han73] Per Brinch Hansen. *Operating System Principles*. Prentice Hall, 1973.

[HBY06] Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *Automata, Languages and Programming*, pages 360–371, Berlin, Heidelberg, 2006. Springer.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 293—-298, 1984.

[HL16] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *International Workshop on Type-Driven Development (TyDe@ICFP)*, pages 15–27, September 2016.

[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.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018.

[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.

[KcSFS05] Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *International Conference on Functional Programming (ICFP)*, pages 192—-203, September 2005.

[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.

[Kis12a] Oleg Kiselyov. An argument against call/cc. https://okmij.org/ftp/continuations/against-callcc.html, December 2012.

[Kis12b] 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.

[KJSB17] Mortern Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of type-and-effects in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*, pages 218–231, January 2017.

[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.

[KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *International Conference on Functional Programming (ICFP)*, pages 145–158, September 2013.

[KP17] Ohad Kammar and Matija Pretnar. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming*, 27:e7, 2017.

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*, January 2017.

[Lan98] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125—-143, 1998.

[LDF+19] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system: documentation and user's manual, September 2019.

[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.

[Lei17] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Principles of Programming Languages (POPL)*, pages 486–499, January 2017.

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

[Ler18] Xavier Leroy. You've got to decide either way or the other! – classical logic, continuations, and control operators. https://xavierleroy.org/CdF/2018-2019/4.pdf, 2018.

[LG01] Barbara Liskov and John V. Guttag. *Program Development in Java – Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2001.

[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.

[Mad02] David Madore. A page about call/cc. http://www.madore.org/~david/computers/callcc.html, 2002.

[MJ84] Francis L. Morris and Clifford B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 307–325. Springer, 2008.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML – Revised*. MIT Press, May 1997.

[NAMB07] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, March 2007.

[Nie01] Lasse R. Nielsen. A selective CPS transformation. *Electronic Notes in Theoretical Computer Science*, 45:311–331, November 2001.

[NS13] Keiko Nakata and Andri Saar. Compiling cooperative task management to continuations. In *Fundamentals of Software Engineering*, pages 95–110. Springer, April 2013.

[NVB10] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Principles of Programming Languages (POPL)*, pages 261–274, January 2010.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, September 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):225–255, 1977.

[Pot17] François Pottier. Verifying a hash table and its iterators in higher-order separation logic. In *Certified Programs and Proofs (CPP)*, pages 3–16, January 2017.

[PP04] Gordon D. Plotkin and A. John Power. Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science*, 73:149–163, 2004.

[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.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *International Conference on Automated Deduction (CADE)*, pages 202–206. Springer, 1999.

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

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

[Rub] Ruby Community. Class: Continuation. https://ruby-doc.org/core-2.5.0/Continuation.html.

[Ré89] Didier Rémy. Type checking records and variants in a natural extension of ML. In *Principles of Programming Languages (POPL)*, pages 77–88, 1989.

[SBMO22] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. A typed continuation-passing translation for lexical effect handlers. In *Programming Language Design and Implementation (PLDI)*, pages 566–579, June 2022.

[sch] Scheme. https://www.scheme.org/.

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

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

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

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

[Str95] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1995.

[SZO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

[TB19] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proceedings of the ACM on Programming Languages*, 3(ICFP):105:1–105:28, July 2019.

[Tea22] The Iris Team. The Iris documentation. https://iris-project.org/, 2022.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Tur49] Alan Turing. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[VB21] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*, pages 76–90, January 2021.

[VS21] Matthijs Vákár and Tom Smeding. CHAD: Combinatory homomorphic automatic differentiation. 2021.

[Wal05] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, 2005.

[Wan80a] Mitchell Wand. Continuation-based multiprocessing. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 19—-28, 1980.

[Wan80b] Mitchell Wand. SCHEME Version 3.1 Reference Manual. Technical report no. 93, Indiana University, Computer Science Department, June 1980.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

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

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[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.