# AUTOMATED VERIFICATION

# OF EFFECTFUL HIGHER-ORDER PROGRAMS

**DARIUS FOO**

*(B.Comp. (Hons.), NUS)*

**A THESIS SUBMITTED**

**FOR THE DEGREE OF DOCTOR OF PHILOSOPHY**

**DEPARTMENT OF COMPUTER SCIENCE**

**NATIONAL UNIVERSITY OF SINGAPORE**

**2026**

**Thesis Advisor** Associate Professor Chin Wei-Ngan
**Examiners** Professor Olivier Danvy
Associate Professor Ilya Sergey

## DECLARATION

I hereby declare that this thesis is my original work and has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree or examination in any other university or institution previously.

*Darius*

Darius Foo

14th July 2026

# Acknowledgments

Above all, I would like to thank my advisor, Dr. Chin Wei-Ngan, for his scientific vision and guidance. He led me into the world of formal verification and taught me the importance of principled solutions to problems. He also showed me the value of perseverance and never giving up in the face of difficulty, and was always available, no matter the time or question. I could not have made this journey without his insight, his encouragement, and most of all his conviction about where we were headed.

I am deeply grateful to my PhD committee members, Dr. Ilya Sergey and Dr. Olivier Danvy, for generously reviewing this dissertation and providing feedback. Both shaped this work long before their roles on the committee. Ilya's work in mechanised proofs was always an inspiration, and I learned a lot in his courses on program logics and distributed systems. Olivier's work is foundational to much of the content of this dissertation, and his advice and example profoundly influenced its presentation and writing.

To the teaching faculty of the School of Computing: Dr. Adi Yoga Sidi Prabawa, Dr. Boyd Anderson, Dr. Damith Rajapakse, Dr. Low Kok-Lim, Dr. Martin Henz, and Dr. Ooi Wei Tsang. Thank you for showing me, from both sides of the classroom, what it means to teach with care and passion. Hats off as well to the late Dr. Stéphane Bressan, whose Discrete Structures course gave me a first glimpse of the ideas that would come to underpin this work. When I later worked with him as a TA in his Database Systems course, I saw his commitment to precision and clarity in his teaching, which continues to inspire me.

To collaborators, colleagues, peers, and friends from NUS, including: Alex Foo, Andreea Costea, Arpita Dutta, Asankhaya Sharma, Bangjie Sun, Cristina David, Dan Baterisna, Emily Ong, George Pîrlea, Hong Jin Kang, Ivan Ho, Julia Low, Koon Wen Lee, Quang Loc Le, Matthew Saw, Shi-Jie Khor, Tristan Lim, Quang-Trung Ta, Viet-Anh Nguyen, WeiLin Low, Wenhua Li, Yahui Song, and Yasunari Watanabe. Thank you for our enriching discussions, our late-night suppers and conversations, and everything we built together. I could not have wished for better companions on this collective journey.

To The Crescendos, Acappella Anonymous, and St. Joseph's Choir: Thank you for all the adventures we shared, through concerts and festivals, weddings and carolling. You brought such colour to these years. I'm grateful for your friendship, and for the privilege of making music alongside you.

Last but not least, to my parents and sister: Thank you for your unconditional support and love, for your belief and faith, and for giving me everything, even when it wasn't easy. None of this would have been possible without you.

# Contents

# ABSTRACT

Higher-order functions and computational effects (such as mutable state, exceptions, and algebraic effects) are ubiquitous in real-world programs. However, verifying programs which use them together remains challenging: these features interact in ways that require nonlocal reasoning, necessitating handcrafted auxiliary specifications (such as invariants or protocols). This creates a conundrum, where automated verifiers provide only partial support or none at all, while interactive verifiers rely on semantic, manual approaches with no clear route to automation.

To address this problem, this dissertation introduces staged logic, a refinement-based logic for reasoning about effectful higher-order programs. Its central insight is that maintaining precision in specifications – by internalising effects in the logic rather than relying on auxiliary specifications – significantly simplifies proofs, which in turn enables automation via a syntactic proof search procedure. This is realised in Heifer, an automated SMT-based verifier for annotated OCaml programs, which has been evaluated on a range of challenging case studies. To validate its metatheory, two fragments of staged logic have also been mechanised in Rocq. Together, these efforts demonstrate that staged logic is a practical and effective foundation for the automated verification of effectful higher-order programs.

CHAPTER 1

# INTRODUCTION

## 1.1 Program verification

Bugs in software continue to plague the world, sometimes with catastrophic financial consequences. A recent example: a faulty driver deployed by the enterprise security firm CrowdStrike brought about the worst outage in history, rendering an estimated 8.5 million Windows devices across the world unable to boot [201]. The root cause was a memory error due to a parser mishandling invalid data [209]. This is just another example in a *software crisis* that has been in effect since the advent of software engineering [173], and which shows no signs of abating.

**Ensuring correctness** There is hope, however: there are well-understood ways to ensure that programs work correctly. These can be used to tame unruly software and ensure that it is free of bugs. Two families of approaches [39] in wide use are *testing* and *static analysis*.

*Testing*, in its modern forms such as fuzzing and symbolic execution, is accessible and inexpensive, scaling to practical systems. It is, however, *underapproximate* in nature, guaranteeing only that a program works correctly in known configurations that have been reached, which may cover only a small portion of an extremely large space of possibilities.

*Static analysis* seeks to prove properties of a fixed form, and do so automatically and scalably. It has seen huge success; type systems are "arguably the most widely-deployed instance of formal methods in software engineering" [154], and abstract interpretation is relied on in safety-critical domains [35]. The tradeoff is that static analyses suffer from false positives when overapproximations prove too coarse-grained [156], and false negatives when approximations are not possible [137]. In other words, they do

not provide guarantees beyond the fixed (abstract) domains they operate in, which are suitable for proving only simple properties, such as lack of memory errors or overflows.

**Verifying programs** A third family of approaches, *deductive verification*, is in a sense the only verification method which scales to arbitrarily complex programs *and* properties: hallmark examples include the OS kernels CertikOS [92] and seL4 [122], the C compiler CompCert [135], the TLS stack in Project Everest [27], and, most recently, the Amazon Web Services (AWS) authorization engine [44]. The tradeoff, however, is that deductive verification scales mainly with human effort and expertise, which is expensive [94]; automation may alleviate some of the burden of constructing proofs, but some amount of handwritten specification is fundamentally required. It is thus used in situations where failure is too costly, testing too incomplete, and approximation too imprecise.

In this work, I seek a way to broaden the applicability of deductive verification to the features found in modern languages, with the goal of lowering the cost of using it in real-world settings.

## 1.2 Staged logic

**Program logics** Deductive verification of programs is carried out using *program logics*, formal systems which view programs as mathematical objects, and are used to establish judgments that they satisfy their *specifications*, which are typically given as logical statements.

*Hoare logic* is the basis for modern program logics. Its defining feature is the *Hoare triple* $\{P\}\,e\,\{Q\}$, a ternary relation between propositions $P$ and $Q$ in some "underlying logic" and a program $e$, which means roughly that, given a state satisfying the precondition $P$, evaluation of the program $e$ in it will result in a state satisfying $Q$. Depending on the choice of logic and

the assertions used, a wide variety of properties can be proved about the behaviours of programs.

Since the inception of Hoare logic, research has focused on scaling it to the features found in real programming languages. *Separation logic*, perhaps its most influential extension, is a substructural logic of resources which allows reasoning about heap-manipulating behaviors, aliasing, fine-grained concurrency, and a myriad of other things which may be modelled as resources.

**Higher-order effectful programs** One combination of language features that has historically been challenging to support is the interaction of higher-order functions with effects. The HOPE workshop, a regular feature at ICFP since 2012, is dedicated to this problem.

While there has been much progress, support for this combination of features remains spotty in today's verifiers, which are overwhelmingly based on Hoare logic. Amongst automated tools, many do not support verification of such programs at all [134, 106], build it into their logics primitively [211], or rely on encodings for partial support [69, 186].

Amongst interactive tools, which support varying subsets of effects, there is also no consensus on a uniform means of *specification*, which ranges from invariants [33] (Section 2.1) to protocols [66] (Section 3.1), non-context-local triples [202] (Section 4.1), and monads [140, 197, 77] (Section 7.1.2).

To solve these problems, I propose rethinking both the means of specification and verification, which I argue are fundamentally what cause them. To this end, I develop a refinement-based program logic, *staged logic*. It has first-class support for the combination of higher-order functions and arbitrary effects, and provides a uniform means of specification for both. Due to its elementary nature, it is more generally applicable: in the following chapters, I demonstrate that it is well-suited to serve as the basis of both an automated

verification tool and a foundational encoding in a proof assistant.

**Staged logic** Staged logic[1] is a logic for precisely specifying programs. Syntactically, it is an untyped lambda calculus, extended with specification statements and logical connectives (Fig. 1.1). **req** and **ens** are a means of stating pre- and postconditions. Instead of being just syntactic delimiters for the components of a Hoare triple, here they are *internalised* as operations in the logic. They use propositions $\sigma$ to describe program states.

$$
\begin{array}{lll}
\varphi ::= \mathbf{req}\ \sigma \mid \mathbf{ens}\ r.\,\sigma & & \text{Specifications} \\
\quad \mid \lambda x.\,\varphi \mid \varphi_1\ \varphi_2 \mid \varphi_1; r.\ \varphi_2 \mid \ldots & & \text{Program elements} \\
\quad \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\ \varphi \mid \forall x.\ \varphi & & \text{Logical connectives}
\end{array}
$$

Fig. 1.1. Syntax of staged logic

Staged logic is intended to serve as the "underlying logic" of a Hoare triple $\{\varphi'\}\, e\, \{r.\ \varphi\}$. What would be the precondition is generalised to a *history* $\varphi'$, while the postcondition is now a *behaviour* $\varphi$. Intuitively, the triple means: $\varphi'$ is a description of some historical behaviour which occurred prior to the execution of $e$, and $\varphi$ describes the composite behaviour of $\varphi'$ and $e$. This setup is shown graphically in Fig. 1.2, where $s_0, s_1, s_2$ are program states along a single execution.

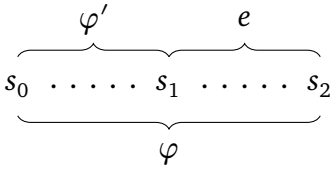


Fig. 1.2. A staged logic triple, graphically

[1]The name *staged logic* was chosen in homage to Gherghina et al. [89], where the term "staged formulae" first appeared. There, it was a means of controlling operational aspects of how verification conditions were dispatched to provers: "Apart from better sharing, this also allows verification to be carried out incrementally over multiple (smaller) stages, instead of a single (larger) stage." The process of proving a staged logic refinement also occurs piecemeal in a similar way, as the proof search decomposes a refinement according to the structure of terms. The partial evaluation around unknowns in staged logic (Section 2.3.2) is also similar to the same mechanism in multi-stage programming, though this was not the original intent behind the choice of name. In particular, it allows verification to be "staged": carried out incrementally as more information becomes available.

When the meaning is clear from the context, we use "staged logic" to refer to both the assertion language of Fig. 1.1, and its use in a program logic via triples.

Staged logic generalises Hoare logic: every Hoare logic derivation can be injected into staged logic by defining the regular Hoare triple as follows.

$$\{\, P \,\}\, e \,\{\, r.\, Q \,\} \triangleq \{\, \mathbf{ens}\ P \,\}\, e \,\{\, \mathbf{ens}\ r.Q \,\}$$

The *specification assertion* $e ::: \varphi$ is useful when there is no interesting history to speak of, which gives us another way to define the Hoare triple. In terms of Fig. 1.2, this situation is illustrated by $s_0 = s_1$ and $\varphi' = \mathbf{skip}$, where **skip** is the unit of sequencing.

$$e ::: \varphi \triangleq \{\, \mathbf{skip} \,\}\, e \,\{\, \varphi \,\}$$

$$\{\, P \,\}\, e \,\{\, r.\, Q \,\} \triangleq e ::: \mathbf{req}\ P; \mathbf{ens}\ r.Q$$

As staged logic is a program logic, it admits a consequence rule.

$$\textsc{Conseq}\quad \frac{\{\, \varphi_3 \,\}\, e \,\{\, r.\, \varphi_4 \,\} \qquad \varphi_1 \sqsubseteq \varphi_3 \qquad \varphi_4 \sqsubseteq \varphi_2}{\{\, \varphi_1 \,\}\, e \,\{\, r.\, \varphi_2 \,\}}$$

Since the staged logic assertions are behavioural, the entailment relation in the premises generalises to *refinement* $\sqsubseteq$, and is where much of the interesting work lies when constructing proofs. The use of refinement enables programs to be *represented precisely* in specifications, smoothening the discontinuity that one encounters with language features which are difficult to reason about compositionally. Proofs become not just simpler but *syntactic*, which in turn leaves the door open to automation.

An extended example justifying these claims and exploring their implications will be covered shortly, in Chapter 2.

## 1.3 Thesis statement & contributions

The thesis defended here can be stated as follows.

> **Staged logic is an effective means of automating the verification of higher-order effectful programs.**

By "effectful", I mean "using *computational effects*" [145]. This includes primitive mutable state, effect handlers, and delimited control operators.

By "effective", I mean that staged logic enables automated verification of this class of programs.

**Contributions** The main contributions of this dissertation are as follows.

I propose *staged logic*, a novel refinement-based logic for reasoning about higher-order programs with imperative and control effects, which improves upon the state of the art by enabling precise specifications without intermediate abstractions, thereby improving automation.

I investigate its expressiveness by demonstrating how it uniformly handles three distinct flavours of effectful higher-order program (Chapters 2 to 4), hitherto treated in very disparate ways.

I demonstrate its effectiveness by formulating a proof search algorithm (Section 2.3), implementing it in a new SMT-based verifier, HEIFER[2] (Section 2.3.4), and evaluating it experimentally (Sections 2.4, 3.5, and 4.4).

I provide evidence for its soundness by developing encodings for two useful fragments of staged logic in Rocq (Chapter 6) and outlining a path towards a full mechanisation.

Finally, I situate staged logic within the landscape of program verification in Chapter 7, which justifies its novelty through an extensive survey.

[2]Higher-order Effectful Imperative Function Entailments & Reasoning

## 1.4 Publications

This dissertation is based on work described in the following publications.

- Darius Foo, Yahui Song, and Wei-Ngan Chin. "Staged Specification Logic for Verifying Higher-Order Imperative Programs". In: *FM 2024: Formal Methods - 17th International Symposium on Formal Methods, Milan, Italy, Sept 9-13, 2024. Proceedings*. Lecture Notes in Computer Science. Springer, 2024

- Yahui Song, Darius Foo, and Wei-Ngan Chin. "Specification and Verification for Unrestricted Algebraic Effects and Handling". In: *Proc. ACM Program. Lang.* ICFP (2024)

- Darius Foo, Viet-Anh Nguyen, Yahui Song, and Wei-Ngan Chin, "Mostly-Automated Verification for Delimited Control" (under submission)

In the course of my PhD, I worked on a number of other projects. For posterity, publications from them are listed below in chronological order. While they are not described further in my dissertation, they directly or indirectly inspired it in many ways. In particular, the experience working on the APLAS 2022 paper made it clear to me and my collaborators that the problem of reasoning about the temporal behaviors of programs containing algebraic effects was far deeper than we had anticipated, and worthy of a more thorough treatment.

- Darius Foo and Wei-Ngan Chin, "Tracing OCaml Programs" (OCaml 2022) [81]

- Yahui Song, Darius Foo, and Wei-Ngan Chin, "Automated Temporal Verification for Algebraic Effects" (APLAS 2022) [189]

- Darius Foo, Andreea Costea, and Wei-Ngan Chin, "Protocol Conformance with Choreographic PlusCal" (TASE 2023) [82]

- Yahui Song, Darius Foo, and Wei-Ngan Chin, “Specifying and Verifying Future Conditions” (SAS 2025)

## 1.5 How to read this dissertation

For an intuitive look at the key ideas and a first pass through examples, see: Section 2.1 (higher-order functions), Section 3.1 (effect handlers), Section 4.1 (delimited continuations), and Chapter 5 (case studies).

The technical development of the logic spans Sections 1.2, 2.2, 3.2, and 4.2 and Chapter 6.

Chapter 7 provides a literature review of this corner of the field of deductive verification.

Finally, the dissertation can be read from beginning to end, for an extended presentation of the logic, interspersed with examples, with features explained as they are introduced.

CHAPTER 2

# VERIFYING HIGHER-ORDER IMPERATIVE PROGRAMS

## 2.1 Introduction

The use of effectful higher-order functions is widespread in everyday programming. It is thus important to support reasoning about such programs in automated tools, if program verification is to be applied more widely.

Yet, today, support for such function in tools is spotty – there is no straightforward answer to the question of how one can easily work with such programs in an automated verification setting.

Consider how we would verify the following OCaml program, which computes the sum of a list of integers `xs` using the higher-order function `foldl`.

```
let foldl f acc xs =
  match xs with
  | [] -> acc
  | x :: t -> foldl f (f acc x) t

foldl (fun t x -> x + t) 0 xs
```

This is a purely functional program, and so can be verified in two distinct ways by automated (SMT-based) tools.

**Approach 1** We can view `foldl` as a mathematical function, and its arguments as mathematical objects. This view allows us to embed it in a logic, such as the first-order logic of Why3, a state-of-the-art automated verifier. We can then state and prove properties of it directly, using standard relations as well as mathematical functions like `sum` (Fig. 2.1).

The statements may require a little insight; for example, generalisation of the induction hypothesis (over the accumulator `acc`). However, the proofs are practically automated: here it suffices to invoke the Why3

```
function foldl_pure
    (f:'b -> 'a -> 'b) (acc: 'b) (xs: list 'a) : 'b =
  match xs with
  | Nil -> acc
  | Cons x t -> foldl_pure f (f acc x) t
  end

goal g : forall xs:list int, acc:int.
    foldl_pure (fun t x -> x + t) acc xs = sum xs + acc
```

Fig. 2.1. A direct statement of a property of *foldl*

tactic `induction_ty_lex`, which heuristically produces an induction hypothesis (from the only structurally inductive argument and the form of the goal). An SMT solver then easily discharges the resulting proof obligations.

*Equational reasoning*, often used on functional programs, is a special case of this approach, which may in general involve other relations.

**Approach 2** We can also view `foldl` as a program and verify the call to it using Hoare logic. For that, we would construct a derivation of the Hoare triple in Fig. 2.2.

$$\{\mathit{true}\}\ \mathit{foldl}\ (\lambda t\, x.\, x + t)\ 0\ \mathit{xs}\ \{r.\, r = \mathit{sum}\ \mathit{xs}\}$$

Fig. 2.2. A Hoare triple specifying a use of *foldl*

As the top-level construct is an application, we would need a specification for `foldl`. Having written one, we would prove that the body of `foldl` satisfies it before using it to verify this call.

Looking at the body of `foldl`, the call to `f` in the recursive case immediately seems problematic: nothing is known about `f`, so we cannot proceed. We must somehow strengthen the precondition of `foldl` to say something about how `f` and the result of `foldl` are related.

```
let rec foldl (ghost inv : list 'a -> 'b -> bool)
    (f:'b -> 'a -> 'b) (acc: 'b) (xs: list 'a) : 'b
  variant { xs }
  requires { inv Nil acc }
  requires { forall acc x ys.
               inv ys acc ->
                 inv (ys ++ (Cons x Nil)) (f acc x) }
  ensures  { inv xs result }
= match xs with
  | Nil -> acc
  | Cons x t -> foldl inv f (f acc x) t
  end
```

Fig. 2.3. A specification for *foldl* in Why3, using an invariant

A standard solution is to use an invariant (Fig. 2.3), which we parameterise the specification of `foldl` over, using a Why3 ghost parameter. `inv` is a property relating the prefix of the input list `xs` thus far accumulated to the accumulator `acc`. In the precondition, we require that the invariant holds initially, when the prefix is empty and the accumulator is at its initial value `acc`, and that it is preserved across the execution of `f`: assuming that the invariant holds of the current prefix `ys` and accumulator, it holds on the extension of the prefix by the element `x` that will be given to `f`, as well as the result of `f`. Finally, the postcondition ensures that the invariant holds of the result and the entire list.

Note, crucially, that `f` appears as an argument to `inv`, *in a logical assertion* – it is treated here as a mathematical function, even though the use of Hoare logic suggests that it and `foldl` are programs.

Proving that `foldl` satisfies this specification is simple. We know that the invariant holds inductively, and since `f` preserves it, it must hold at the end of the recursive case.

With this specification, the Hoare triple from earlier can be proved. We state it in the form of a Why3 lemma (Fig. 2.4). Note that we must

provide a concrete invariant (or ghost argument) for this particular call to `foldl`, which says that the intermediate results of `foldl` on the prefixes of the input list are the sums of the elements of those lists.

```
let lemma foldl_sum (xs : list int) : int
  ensures { result = sum xs }
= foldl (fun pre r -> r = sum pre)
    (fun t x -> x + t) 0 xs
```

Fig. 2.4. An invariant for *foldl*, given to prove a Why3 lemma

**Tradeoffs** Comparing the two approaches, we see that the second is considerably less automated than the first. Not only did we have to write a specification for `foldl`, which took creativity, we had to supply an invariant, a problem-specific semantic property, which the induction hypothesis of `foldl_pure` already hinted at.

The tradeoff is that the use of a program logic enables compositional reasoning. Furthermore, Approach 2 is able to handle programs with imperative behaviours, as we will see shortly.

```
let c = ref 0 in
foldl (fun () x -> c := x + !c) () xs; !c
```

Fig. 2.5. A use of `foldl` involving a closure

**Imperative programs** Now, consider the use of `foldl` in Fig. 2.5. It computes the sum of the list by accumulation, as before, but it does so using a closure, whose imperative *effect* on a captured reference to a mutable cell `c` accumulates the numbers in the list.[1]

This is a perfectly natural OCaml program to write, but perhaps surprisingly, this small change leads to a feature cliff: it can no longer be verified by Why3.

[1]Thanks to Olivier Danvy for suggesting the imperative statement of *foldl*, which led to this example.

Why? Consider how the two approaches from before are affected. They differed in whether `foldr` was treated as a program or a mathematical function. However, in both cases, the argument to `foldr` was treated as a mathematical function and embedded directly in a first-order logic formula, giving us no way to talk about its effect.

This problem extends beyond Why3, to other mature automated verifiers [172, Section 2]: for example, Dafny also cannot verify this program for the same reasons, and there is no consistent way to verify this in Viper-based verifiers, though variants for specific languages contain extensions which can handle it (e.g., Prusti, though it targets Rust and relies on the guarantees of Rust's ownership types; and Gobra, which has primitive support for stateful closures).

### 2.1.1 Invariants in separation logic

One might wonder: how do interactive tools, which have access to more expressive logics, handle this example? As it turns out, separation logics such as Iris can be used to verify it using Approach 2.

The key insight they rely on is that the invariant `inv` is now a separation logic predicate, i.e., a value of type `(list 'a -> 'b -> hprop)`, where `hprop := (heap -> bool)`, which allows it to represent the state-changing effect of the closure `f`.

This leads to the following specification (Fig. 2.6), in Iris-like syntax.

$$\forall\, Inv\ f\ xs\ acc.$$
$$\left\{ \begin{array}{l} Inv\ [\,]\ acc\ * \\ \forall\ acc\ x\ ys.\ \{Inv\ ys\ acc\}\ f\ \ acc\ x\ \{r.\ Inv\ (ys +\!\!+ [x])\ r\} \end{array} \right\}$$
$$foldl\ f\ \ acc\ xs$$
$$\{result.\ Inv\ xs\ result\}$$

Fig. 2.6. A simple specification for *foldl* in separation logic [33]

As before, *Inv* is a parameter, represented now by a higher-order quantification. This time, the fact that $f$ preserves the invariant is represented by

a *nested Hoare triple*, which is necessary because $f$ is a program, not a mathematical function. The proof that the body of *foldl* satisfies this specification is otherwise very similar.

On to the use of *foldl*. We specify the call using a separation logic triple (Fig. 2.7), which states that initially there is some location in memory $\ell$ with value $a$. The specification guarantees that the location has its value incremented by the sum of the elements of $xs$. $[-]$ injects a first-order proposition into separation logic.

$$\forall\, \ell\, a.\ \{\, \ell \mapsto a \,\}\, foldl\ (\ldots)\ ()\ xs\ \{\, r.\ \ell \mapsto (a + sum\ xs) * [r = ()] \,\}$$

Fig. 2.7. A separation logic specification for the call to *foldl* in Fig. 2.5

To prove this triple, we would use the triple in Fig. 2.6 with the invariant $(\lambda pre\, r.\, \ell \mapsto (sum\ pre) * [r = ()])$, which is similar to the one given in Fig. 2.4.

**The one-spec-per-use problem** The use of separation logic solves the immediate problem. However, this general approach has a rather severe shortcoming, which only becomes apparent once variations of the particular use of *foldl* from above are considered. The authors of the Iris Lecture Notes [33] recognise this:

> Different clients may instantiate [*foldl*] with some very different functions, hence it can be hard to give a specification for `f` that is reasonable and general enough to support all these choices. In particular knowing when one has found a good and provable specification can be difficult in itself.

In other words, even though the specification as written is very general, it is not sufficient to verify all uses of *foldl*. There are two fundamental issues with such *intermediate specifications* which make use the use of semantic properties of the underlying logic to represent effects.

1. Consider the specification in Fig. 2.8.

$$\forall\, Inv\ P\ f\ xs\ acc.$$
$$\left\{\begin{array}{l}[all\ P\ xs] * Inv\ [\,]\ acc\ * \\ \forall\, x\ acc\, ys.\ \{[P\ x] * Inv\ ys\ acc\}\ f\ \ acc\ x\ \{r.\ Inv\ (ys +\!\!+ [x])\ r\}\end{array}\right\}$$
$$foldl\ f\ \ acc\ xs$$
$$\{result.\ Inv\ xs\ result\}$$

Fig. 2.8. A specification for *foldl* with a stronger precondition [33]

It is similar to the one in Fig. 2.6, but is additionally parametrised over a unary predicate $P$, a precondition for $f$. $P$ holds individually of each element $x$ of the list. For consistency, we may wish to propagate the precondition outwards, requiring *all P xs* holds in the precondition of *foldl*.

While this approach is natural, it is somewhat ad hoc: the choice of the (unary) form of $P$ is rather arbitrary, and constrains the kinds of properties one can use.

- Suppose one wanted to rely on a property concerning consecutive elements of the list. One would have to write a new specification, with a property parametrised over *xs*.
- What if one passed an argument such as $(\lambda x\ t.\, assert\ x \geq t; x+t)$? This would require a property relating each element and suffix-sum. Again, neither $P$ or *Inv* would be sufficient.

Making all these choices upfront would produce a specification parametrised over all sorts of extraneous predicates, which is also untenable.

The point is that this approach requires one to commit to a parametrisation of a fixed form, which clearly cannot work for all cases. I refer to this as *the one-spec-per-use problem*.

2. What if we passed the argument $(\lambda x\ t.\, raise\ E)$? This approach clearly cannot cope with this, as the invariant cannot be maintained – the effect of this function cannot be described in terms of program states.

This highlights the fact that the invariant over separation logic resources is only *a representation of a specific kind of effect*. It does not generalise to other kinds of effects because it does not faithfully describe the effects themselves.

To recap, our options today for automatically verifying effectful higher-order programs are as follows, with the problems presented in the order they were introduced. Neither is ideal.

**Approach 1** Less compositional, and ultimately inapplicable, as we are not working with pure functions.

**Approach 2** Less automated, one-spec-per-use problem, requires underlying logic that can represent effects.

**Towards a solution** Backtracking a little, suppose we allowed ourselves to imagine a logic where Approach 1 could apply. This is the motivation for staged logic. It is a simple, unifying viewpoint on the problem which combines the advantages of Approaches 1 and 2, while solving their problems:

- It is formulated as a program logic, retaining the compositionality of Approach 2.
- Like Approach 1, an intermediate specification for `foldl` is not required, and a standard induction hypothesis is used instead of an invariant.
- The one-spec-per-use problem is solved by not requiring one to commit to a parametrisation, or any other kind of representation or proxy for describing higher-order functions and effects. This is done by maintaining a precise account of effects and their ordering.
- As staged logic is itself a direct means of describing effects, the underlying logic is not used or required to represent effects. For example, to

reason about programs that use exceptions but no mutable state, there would be no need to use separation logic as the underlying logic.

In the rest of this chapter, I make these points concrete, and illustrate how staged logic does these things in the setting of programs with imperative effects (with exceptions covered later, in Section 3.1.3). Subsequent chapters (Chapters 3 and 4) generalise this to arbitrary control effects and provide a formal treatment (Chapter 6).

### 2.1.2 Imperative effects in staged logic

We start with the language of staged logic from Fig. 1.1, which allows us to both *model* and *specify* programs. In the former view, staged logic is a (higher-order) intermediate verification language (IVL) [149], where one may mix specifications and programs. In the latter, it is a logic in which one may use quantifiers and other standard connectives, assert and assume arbitrary propositions, and talk about programs using fragments of syntax.

Our approach to proving the triple in Fig. 2.7 will be as follows.

1. We model *foldl* (and its argument $f$) in staged logic, such that

$$f \;:::\; g \qquad\qquad \mathit{foldl}\ f\ \ a\ \mathit{xs} \;:::\; \mathit{foldl}\ g\ a\ \mathit{xs}$$

   both hold by construction.

2. We prove the refinement

$$\mathit{foldl}\ g\ ()\ \mathit{xs} \sqsubseteq \mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ r.\ \ell \mapsto (\mathit{sum}\ \mathit{xs} + a) * [r = ()]$$

   with the specification also given in staged logic.

3. We use the rule of consequence to piece both of these together into a proof of the assertion:

$$\mathit{foldl}\ f\ \ a\ \mathit{xs} ::: \mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ r.\ \ell \mapsto (\mathit{sum}\ \mathit{xs} + a) * [r = ()]$$

which is equivalent to the triple in Fig. 2.7.

First, we model the *foldl* program (Fig. 2.9). This is a direct translation of the constructs of the program to the corresponding connectives in the logic. We use disjunction to model the branches of the **match** and existentials for the destructured variables. When modelling programs, **ens** allows us to assume conditions on values; we use it both for the assumptions on the form of the **match** scrutinee in each branch, and to describe what the result of each branch should be, via the binder *res*. Calls to recursive and unknown functions are represented directly, in their sequential order.[2] Binding is written $\varphi_1; r.\ \varphi_2$.[3]

```
let foldl f acc xs =
  match xs with
  | [] => acc
  | h :: t =>
    foldl f (f acc h) t
```

(a) `foldl`

$$
\begin{aligned}
&\mathit{foldl}\ f\ \mathit{acc}\ \mathit{xs} \triangleq \\
&\quad \mathbf{ens}\ r.\mathit{xs}{=}[\,] \wedge r{=}\mathit{acc} \\
&\vee\ \exists h\, t.\ \mathbf{ens}\ \mathit{xs}{=}h{::}t; \\
&\qquad f\ \mathit{acc}\ h; r. \\
&\qquad \mathit{foldl}\ f\ r\ t
\end{aligned}
$$

(b) *foldl*

Fig. 2.9. *foldl* program in staged logic

Next, we prove the entailment:

$$\forall\, \ell\ a.\ \mathit{foldl}\ g\ ()\ \mathit{xs} \sqsubseteq \mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ r.\ell \mapsto (\mathit{sum}\ \mathit{xs} + a) * [r = ()]$$

Interestingly, the proof can be completely automated.

*Proof.* We proceed by induction on the structure of *xs*. Unfolding *foldl* and focusing on the inductive case, we get the following entailment.

$$
\begin{aligned}
&\mathbf{ens}\ \mathit{xs}{=}h{::}t; f\ ()\ h; r.\ \mathit{foldl}\ f\ r\ t \\
\sqsubseteq\ &\mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ \mathit{res}.\ell \mapsto (\mathit{sum}\ \mathit{xs} + a) * [\mathit{res} = ()]
\end{aligned}
$$

Rewriting the call to *foldl* using the induction hypothesis and unfolding

[2] For simplicity, we assume that effects only occur when functions are fully applied, i.e., partial application is assumed to be pure. It is possible to relax this by sequencing every application, at the cost of more verbosity and intermediate names.

[3] In the syntax of staged logic, binders are always followed by a period.

$f$, we get a lengthy sequence of **req** and **ens**.

$$\begin{aligned}
&\mathbf{ens}\ xs{=}h{::}t;\\
&\forall\, i.\ \mathbf{req}\ \ell \mapsto i; \underline{\mathbf{ens}\ r.(\ell \mapsto (i+h) * [r = ()])}; r.\\
&\underline{\forall\, a_1.\ \mathbf{req}\ \ell \mapsto a_1}; \mathbf{ens}\ res.\ \ell \mapsto (sum\ t + a_1) * [res = ()]\\
\sqsubseteq\ &\mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ res.\ \ell \mapsto (sum\ xs + a) * [res = ()]
\end{aligned}$$

It looks more complex than it is: a description of all the state changes to $x$ which occur along this program path. The underlined portion is notable because it can be subjected to simplification. Intuitively, when on the left side of an entailment, an **ens** can be understood as a separation logic *assumption*, *producing* a piece of heap, while **req** can be seen as an *obligation*, *consuming* a piece of heap. This view lets us "cancel" them; the mechanism which enables this is *biabduction* [42].

$$\frac{H_A * H_1 \vdash H_2 * H_F}{\mathbf{ens}\ H_1; \mathbf{req}\ H_2 \sqsubseteq \mathbf{req}\ H_A; \mathbf{ens}\ H_F}\ \textsc{Biab}$$

We can view this simplification of the formula representing the program as a kind of symbolic execution, analogous to what separation logic verifiers typically do. **req** and **ens** can be seen as an *internalisation* of the operational behaviour of heap entailment.

What remains to be proved, then, is the following.

$$\begin{aligned}
&\forall\, i.\ \mathbf{req}\ \ell \mapsto i; \mathbf{ens}\ res.\ \ell \mapsto (sum\ t + (i+h)) *\\
&\quad [res = () \wedge xs = h :: t]\\
\sqsubseteq\ &\mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ res.\ \ell \mapsto (sum\ xs + a) * [res = ()]
\end{aligned}$$

Now, we see that both sides of the goal consist of a single pair of **req** and **ens**. This allows it to be reduced to separation logic entailment via the following two rules.

$$\frac{H_2 \vdash H_1}{\mathbf{req}\ H_1 \sqsubseteq \mathbf{req}\ H_2} \qquad \frac{\forall\, v.\ Q_1\ v \vdash Q_2\ v}{\mathbf{ens}\ Q_1 \sqsubseteq \mathbf{ens}\ Q_2}$$

The two entailments follow.

$$\forall\, i.\ \ell \mapsto i \vdash \ell \mapsto a \qquad\qquad \begin{array}{c} \ell \mapsto (sum\ t + (a + h)) * \\ [res = () \wedge xs = h :: t] \\ \vdash \ell \mapsto (sum\ xs + a) * [res = ()] \end{array}$$

The one on the right can be reduced to the following first-order obligation, which is easily dispatched by an SMT solver with standard list and arithmetic theories.

$$\begin{array}{c} res = () \wedge xs = h :: t \\ \Rightarrow sum\ t + (a + h) = sum\ xs + a \wedge res = () \end{array}$$

□

### 2.1.3 Maintaining precision

What exactly enabled the automation? The crucial difference from the standard separation logic approach in Section 2.1.1 is that we were not forced to come up with an abstraction for the behaviour of *foldl* immediately, in the form of its invariant-parametrised specification. Instead, we simply left the parts which were difficult to handle *in the specification*, until we knew enough (i.e., saw the definition of the closure argument) that proving something meaningful was easy. In other words, we retained a precise specification of the essential portions – the ordering of effectful higher-order calls – while abstracting away things like the concrete state changes which occurred.

The insight from this example is simply that some language features are inherently challenging to specify modularly, because they depend on things external to the program being specified. While feature- or even problem-specific insight can be employed to write down an adequate specification, such a specification will certainly be lossy.

I argue that it is simpler and more productive to sometimes allow the extent of a modular treatment for a particularly tricky language feature to be: to let it be represented in the specification, then to handle it (in)equationally in the logic at a later point. This results in a smoothening of the discontinuity

in what a state-based logic can represent: proofs become simpler and remain syntactic, which in turn enables automation.

In the rest of this chapter, we provide a more formal treatment of what we have seen so far, by fixing a syntax and semantics for staged logic, then describing the ingredients needed for a proof search procedure which implements the automation from the previous section.

In the following chapters, we apply this methodology to the verification of more kinds of effectful programs.

For a first pass through this dissertation, it is possible to jump to Section 3.1 (effect handlers), then Section 4.1 (delimited continuations) to get a sense of the breadth of the framework.

## 2.2 Staged logic

Having introduced staged logic in Section 1.2, we now take a closer look at its workings. The definitions and rules here (and in Sections 3.2 and 4.2) are given in a form optimised for clarity and use in case studies, and are formulated differently from the mechanised versions in Chapter 6; I note the key differences as they occur.

**Syntax** The syntax of *core* staged logic is given in Fig. 2.10, along with definitions to recover the *idealised* syntax of Fig. 1.1. State formulae $\sigma$ have been specialised to separation logic *heap propositions* $H$, where $[P]$ injects a pure proposition $P$ into $H$. **req** and **ens** are not primitive, and are defined in terms of **produce** and **consume**,[4] as well as **assume** and **assert** for pure propositions; we assume their state formulae are normalised into a heap-only portion and a pure portion. **req** and **assume** have a "continuation" parameter $\varphi$ for reasons relating to how their semantics is defined. A separate constructor

[4]These are the standard operators for symbolic execution-based separation logic verification [138], also called *inhale* and *exhale* in Viper [149].

$$
\begin{array}{lll}
\varphi ::= & \textbf{produce}\ H \mid \textbf{consume}\ H & \text{Specifications} \\
\mid & \textbf{assert}\ P\,\varphi \mid \textbf{assume}\ P & \\
\mid & \lambda x.\,\varphi \mid f\ \varphi \mid \varphi_1\ \varphi_2 \mid \varphi_1; r.\,\varphi_2 \mid \textbf{ret}\ v & \text{Program elements} \\
\mid & \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\,\varphi \mid \forall x.\,\varphi & \text{Logical connectives} \\
H ::= & \mathit{emp} \mid \ell \mapsto v \mid H * H \mid [P] \mid \top \mid H \mathrel{-\!\!*} H \mid H \mathrel{-\!\!\circledast} H \mid \ldots & \text{Separation logic}
\end{array}
$$

$$
\varphi_1; \varphi_2 \triangleq \varphi_1; \_.\ \varphi_2 \qquad \textbf{req}\ (H * [P])\,\varphi \triangleq \textbf{consume}\ H; \textbf{assume}\ P\ \varphi
$$

$$
\textbf{req}\ H; \varphi \triangleq \textbf{req}\ H\,\varphi \qquad \textbf{ens}\ (H * [P]) \triangleq \textbf{produce}\ H; \textbf{assert}\ P
$$

$$
\textbf{ens}\ r.[P\ r] \triangleq \exists r.\ \textbf{ens}\ [P\ r]; \textbf{ret}\ r \qquad \textbf{skip} \triangleq \textbf{ret}\ () \qquad \bot \triangleq \textbf{assert}\ \mathit{false}
$$

$$
\top \triangleq \textbf{assume}\ \mathit{false}\ \textbf{skip}
$$

Fig. 2.10. Syntax of core staged logic

**ret** $v$ has been added for pure values. We also add applications of unknown functions $f\ \varphi$.[5] It is possible to represent them extrinsically using meta-level quantification, as we did earlier, but to discuss their semantics, we represent them intrinsically here.

**Semantics** Staged logic describes computational behaviour, so its models are conceptually traces: sequences of program states. As we work in separation logic, we identify a program state with a *heap*, a finite partial map from locations $\ell$ to values $v$. A simple model for staged logic comprises an initial heap, a final heap and a result value – the observable summary of a trace. Because $\varphi$ may contain uses of unknown functions, we augment the model with a *specification environment* $\mathcal{E}$, a finite partial map from variables to staged logic functions $\lambda x.\,\varphi$, to give them interpretations.

The semantics of a staged formula can thus be given via the *satisfies* rela-

[5] In the original papers on staged logic [83, 190], calls to unknown functions were represented as $f(v, r)$ (with a relational output parameter $r$), and sequencing was used instead of bind, with outputs existentially quantified, e.g., $\exists r.\ f(v, r); g(r, r_1)$. This encoding also works for the purposes of this chapter, but breaks down in the presence of continuations, which may produce multiple result values, whereas an existential can only be instantiated with a single value. Representing the continuations explicitly with bind solves this problem. The relational form $f(v, r); \varphi$ can be reinterpreted as $f\ v; r.\ \varphi$ (hence the choice of this syntax).

tion $\mathcal{E}, h_1, h_2, R \vDash \varphi$, defined in Fig. 2.11. It resembles a nondeterministic big-step operational semantics. $R$ is some type of *outcome*, which we specialise to values $v$ for now. SCONSUME and SPRODUCE are the standard definitions of **consume** and **produce**, adding and removing[6] a piece of heap $h_p$ satisfying $H$. SASSUME and SASSERT allow working with arbitrary propositions; the second parameter of **assume** is notably required because its semantics is defined in terms of metalogic implication, allowing the "continuation" $\varphi$ to rely on $P$. Next are the lambda calculus (SLAMBDA, SAPP) and monadic (SRET, SBIND) fragments, which are standard. The only notable addition here is SUNK, for unknown functions, and the one place at which the specification environment $\mathcal{E}$ is used; it can be seen as an interalised quantification over possibly-effectful *implementations* of $f$, through the quantification over $\mathcal{E}$. The logical fragment (SEXISTS, SFORALL, DISJOINTL, DISJOINTR, SCONJ) lifts metalogic connectives and enables nondeterminism.

### 2.2.1 Entailment

To support reasoning at the level of staged logic, with higher-level induction and rewriting principles instead of with pieces of heaps, we introduce an entailment relation $\sqsubseteq$. This is also the refinement relation, if we think of a staged logic term as a program in an IVL. As a first approximation, we define it intensionally, as a direct lifting of implication to staged logic.

**Definition 2.1** (Entailment and equivalence)**.**

$$\varphi_1 \sqsubseteq \varphi_2 \triangleq \forall\, \mathcal{E}\, h_1\, h_2\, R.\ (\mathcal{E}, h_1, h_2, R \vDash \varphi_1 \Rightarrow \mathcal{E}, h_1, h_2, R \vDash \varphi_2)$$

$$\varphi_1 \equiv \varphi_2 \triangleq \varphi_1 \sqsubseteq \varphi_2 \wedge \varphi_2 \sqsubseteq \varphi_1$$

This semantic definition is useful to justify the soundness of the entail-

[6] The original papers on staged logic [83, 190] gave the definition of **req** as $(\mathcal{E}, h_1, h_2, v \vDash \mathbf{req}\ H\ \varphi \triangleq \forall\, h_p\, h_r.\ (h_1 = h_p \circ h_r \wedge h_p \vDash H \Rightarrow \mathcal{E}, h_r, h_2, v \vDash \varphi))$. This was problematic as it could be satisfied vacuously by a heap that could not be split, which necessitated the decomposition into **produce**/**consume** and **assume**/**assert**.

$\boxed{\mathcal{E}, h, h, R \vDash \varphi}$

$$\textsc{SConsume}\ \frac{\exists h_p\, h_r.\ h_p \vDash H \wedge h_1 = h_p \circ h_r \wedge h_2 = h_r}{\mathcal{E}, h_1, h_2, () \vDash \textbf{consume}\ H} \qquad \textsc{SProduce}\ \frac{\exists h_p.\ h_p \vDash H \wedge h_2 = h_1 \circ h_p}{\mathcal{E}, h_1, h_2, () \vDash \textbf{produce}\ H}$$

$$\textsc{SAssume}\ \frac{P \Rightarrow \mathcal{E}, h_1, h_2, v \vDash \varphi}{\mathcal{E}, h_1, h_2, v \vDash \textbf{assume}\ P\ \varphi} \qquad \textsc{SAssert}\ \frac{P}{\mathcal{E}, h, h, () \vDash \textbf{assert}\ P}$$

$$\textsc{SLambda}\ \frac{}{\mathcal{E}, h, h, (\lambda x.\, \varphi) \vDash \lambda x.\, \varphi} \qquad \textsc{SApp}\ \frac{\mathcal{E}, h_1, h_3, (\lambda x.\, \varphi_b) \vDash \varphi_1 \quad \mathcal{E}, h_3, h_4, v \vDash \varphi_2 \quad \mathcal{E}, h_4, h_2, r \vDash \varphi_b[v/x]}{\mathcal{E}, h_1, h_2, r \vDash \varphi_1\ \varphi_2} \qquad \textsc{SUnk}\ \frac{\mathcal{E}(f) = \lambda x.\, \varphi \quad \mathcal{E}, h_1, h_2, r \vDash \varphi[v/x]}{\mathcal{E}, h_1, h_2, r \vDash f\ v}$$

$$\textsc{SRet}\ \frac{}{\mathcal{E}, h, h, v \vDash \textbf{ret}\ v} \qquad \textsc{SBind}\ \frac{\mathcal{E}, h_1, h_3, v_1 \vDash \varphi_1 \quad \mathcal{E}, h_3, h_2, v \vDash \varphi_2[v_1/r]}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1; r.\, \varphi_2}$$

$$\textsc{SExists}\ \frac{\exists a.\ (\mathcal{E}, h_1, h_2, v \vDash \varphi)}{\mathcal{E}, h_1, h_2, v \vDash \exists a.\, \varphi} \qquad \textsc{SForAll}\ \frac{\forall a.\ (\mathcal{E}, h_1, h_2, v \vDash \varphi)}{\mathcal{E}, h_1, h_2, v \vDash \forall a.\, \varphi}$$

$$\textsc{DisjointL}\ \frac{\mathcal{E}, h_1, h_2, v \vDash \varphi_1}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \vee \varphi_2} \qquad \textsc{DisjointR}\ \frac{\mathcal{E}, h_1, h_2, v \vDash \varphi_2}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \vee \varphi_2} \qquad \textsc{SConj}\ \frac{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \quad \mathcal{E}, h_1, h_2, v \vDash \varphi_2}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \wedge \varphi_2}$$

Fig. 2.11. Semantics of staged logic

ment rules in this section and in Sections 2.3.2 and 2.3.4.

**Properties of entailment** Entailment is a preorder (reflexive and transitive), and equivalence is an equivalence relation. Using both, we can now state properties of staged logic terms.

Fig. 2.12 shows some of the more interesting properties of the logical and monadic fragments, particularly their interactions. Within the logical fragment, many standard properties apply: for example, disjunction is associative, commutative, idempotent, and has $\bot$ as an identity element and $\top$ as annihilator; these are not shown. $\top$ and $\bot$ are the top and bottom elements

of the staged logic lattice. The logical connectives distribute over entailment as one would expect; dual ones are omitted for brevity. In the monadic fragment, **skip** is the left unit of sequencing, and the right unit only when $\varphi$ has a unit-valued result. Bind is associative and subject to eta-reduction. Disjunction distributes over sequencing. Quantifiers do as well, but only in one direction, because $\varphi_1$ may depend on $x$. When combined with sequencing, $\bot$ has its conventional interpretation as "dead code" or "unreachability", allowing anything after it to be ignored.

$$\boxed{\varphi \sqsubseteq \varphi}$$

$$\textsc{Bot}\ \frac{}{\bot \sqsubseteq \varphi} \qquad \textsc{Top}\ \frac{}{\varphi \sqsubseteq \top} \qquad \textsc{DisjL}\ \frac{\varphi_1 \sqsubseteq \varphi_3 \quad \varphi_2 \sqsubseteq \varphi_3}{\varphi_1 \vee \varphi_2 \sqsubseteq \varphi_3} \qquad \textsc{DisjR}\ \frac{\varphi_3 \sqsubseteq \varphi_1}{\varphi_3 \sqsubseteq \varphi_1 \vee \varphi_2}$$

$$\textsc{ExL}\ \frac{\forall x.\ (\varphi'\ x \sqsubseteq \varphi)}{(\exists x.\ \varphi'\ x) \sqsubseteq \varphi} \qquad \textsc{ExR}\ \frac{\varphi \sqsubseteq \varphi'\ x}{\varphi \sqsubseteq \exists x.\ \varphi'\ x} \qquad \textsc{ForAllR}\ \frac{\forall x.\ (\varphi_1 \sqsubseteq \varphi_2)}{\varphi_1 \sqsubseteq \forall x.\ \varphi_2} \qquad \textsc{ForAllL}\ \frac{\varphi_1 \sqsubseteq \varphi_2}{(\forall x.\ \varphi_1) \sqsubseteq \varphi_2}$$

$$\textsc{SeqUnitL}\ \frac{}{\mathbf{skip}; \varphi \equiv \varphi} \qquad \textsc{SeqUnitR}\ \frac{\varphi[()]}{\varphi; \mathbf{skip} \equiv \varphi} \qquad \textsc{SeqBotL}\ \frac{}{\bot; \varphi \equiv \bot} \qquad \textsc{BindEta}\ \frac{}{\varphi; x.\ \mathbf{ret}\ x \equiv \varphi}$$

$$\textsc{BindAssoc}\ \frac{}{(\varphi_1; r.\ \varphi_2); r_1.\ \varphi_3 \equiv \varphi_1; r.\ (\varphi_2; r_1.\ \varphi_3)}$$

$$\textsc{DisjSeqDistr}\ \frac{}{\varphi_1 \vee (\varphi_2; \varphi_3) \equiv (\varphi_1 \vee \varphi_2); (\varphi_1 \vee \varphi_3)} \qquad \textsc{ForAllDistrR}\ \frac{}{\varphi_1; (\forall x.\ \varphi_2) \sqsubseteq \forall x.\ \varphi_1; \varphi_2}$$

Fig. 2.12. Select entailments (logical and monadic fragment)

**Pure specifications** Fig. 2.13 shows properties of the pure specification operations **assume** and **assert**. They are pleasingly dual. **assert** is covariant while **assume** is contravariant. Furthermore, there is an adjunction between

$$\textsc{AssertIntro}\ \frac{P}{\varphi \sqsubseteq \mathbf{assert}\ P; \varphi} \qquad \textsc{AssertElim}\ \frac{}{\mathbf{assert}\ P; \varphi \sqsubseteq \varphi} \qquad \textsc{AssumeIntro}\ \frac{}{\varphi \sqsubseteq \mathbf{assume}\ P\ \varphi} \qquad \textsc{AssumeElim}\ \frac{P}{\mathbf{assume}\ P\ \varphi \sqsubseteq \varphi}$$

$$\textsc{AssertCov}\ \frac{P_1 \Rightarrow P_2 \quad \varphi_1 \sqsubseteq \varphi_2}{\mathbf{assert}\ P_1; \varphi_1 \sqsubseteq \mathbf{assert}\ P_2; \varphi_2} \qquad \textsc{AssumeContra}\ \frac{P_2 \Rightarrow P_1 \quad P_2 \Rightarrow (\varphi_1 \sqsubseteq \varphi_2)}{\mathbf{assume}\ P_1\ \varphi_1 \sqsubseteq \mathbf{assume}\ P_2\ \varphi_2}$$

$$\textsc{AssertAssumeAdj}\ \frac{}{(\mathbf{assert}\ P; \varphi_1 \sqsubseteq \varphi_2) \iff (\varphi_1 \sqsubseteq \mathbf{assume}\ P\ \varphi_2)}$$

$$\textsc{AssumeReassoc}\ \frac{}{(\mathbf{assume}\ P\ \varphi_1); \varphi_2 \sqsubseteq \mathbf{assume}\ P\ (\varphi_1; \varphi_2)}$$

Fig. 2.13. Select entailments (pure specification)

them: an **assume** on the right is equivalent to an **assert** on the left.[7] The rule is useful for verification: given a precondition as an **assume** on the right, this allows the program, which is usually on the left, to rely on it. The dual statement, where **assume** and **assert** are swapped, is also true, but trivial because both operators collapse to identity. The last rule, AssumeReassoc, allows reassociating **assume** over sequencing, showing that this "continuation" representation does not lose expressiveness: it is always possible to reassociate **assume**s into a linear sequence. The converse of AssumeReassoc may not hold as $\varphi_2$ may be dependent on $P$.

**Stateful specifications** Next are the properties for **produce** and **consume**. As operations, both technically have *preconditions*: **produce** $H$ disjointly adds a piece of heap $h$ satisfying $H$, and so requires $h$ to not already be in the heap; **consume** $H$ disjointly *removes* a H-piece, and dually requires it to be present. We thus need to generalise the notion of refinement between these operations to make it *conditional*.

[7] To understand intuitively why this statement holds, case on $P$. If $P$ is *true*, it's trivial. If $P$ is *false*, on the left, we have **assert** *false*, which has no behaviour, so the left side is vacuously true; on the right, we have **assume** *false* ..., which has all behaviours, so the right side is also vacuously true.

**Definition 2.2** (Conditional refinement)**.**

$$H \vdash \varphi_1 \sqsubseteq \varphi_2 \triangleq \forall\, h_1.\ h_1 \vDash H \Rightarrow \\ \forall\, \mathcal{E}\, h_2 R.\ (\mathcal{E}, h_1, h_2, R \vDash \varphi_1 \Rightarrow \mathcal{E}, h_1, h_2, R \vDash \varphi_2)$$

To state these preconditions generally, a modest extension to the language of heap propositions is needed. I introduce two *framing modalities* $\blacktriangle H$ ("contains $H$") and $\triangle H$ ("missing $H$").[8] They are modalities in the sense that they are unary operators on heap propositions that change how they are interpreted; in this case, they change a precise heap proposition into a *heap proposition up to an implicit frame*. $\blacktriangle H$ is satisfied by any heap containing a $H$-shaped piece; in other words, a $H$-shaped piece and an arbitrary frame. $\triangle H$ is satisfied by any heap *missing* a $H$-piece; the current heap acts as a frame around that piece. Both can be defined in terms of standard connectives; $\triangle H$ requires the relatively obscure *septraction* connective [205], the existential dual of the magic wand. Fig. 2.14 gives these definitions, and some properties of the modalities.

Every heap contains and is missing the empty heap. Both modalities are monotonic with respect to heap entailment. Containment is idempotent: containing a heap that itself contains $H$ is the same as containing $H$; nested framing does not add information. ContainsSelf allows the frame to be empty. ContainsSep shows that containing a combined $H_1 * H_2$ allows exposing one part and leaving the remainder hidden inside the remaining heap. MissingSep is analogous: having room for $H_1 * H_2$ is the same as removing $H_1$ from a heap missing $H_2$. Existentials distribute over both modalities as expected. ContainsMissingSep relates the two: a heap contains $H$ exactly when it can be split into the actual $H$ piece and some residue that is missing the same piece. Nesting the modalities can be rather unintuitive. MissingContains is true because $\triangle H$ is existential, due to the use of septrac-

[8] $\blacktriangle H$ is in Iris as the $\langle \mathsf{absorb} \rangle\, H$ modality (though it is trivial there, as Iris is affine; the separation logic we work with is not). $\triangle H$ is novel, though it is a simple extension of septraction.

tion: it suffices to exhibit some disjoint $H$-piece. From a witness for $\triangle\blacktriangle H$, we can take the piece satisfying $H$ and ignore the frame. ContainsMissing is similar: it means that "some contained heap piece is missing $H$". The canonical choice of that piece is the empty heap, which is missing $H$ exactly when $H$ is satisfiable: if some heap satisfies $H$, it is automatically disjoint from the empty heap. Two containment facts can be merged. Finally, $\blacktriangle H$ can be weakened, or framed: knowing that $H$ is contained somewhere in a heap, adding to the frame does not change this.

$\boxed{\blacktriangle H, \triangle H}$

$$
\begin{array}{rcl}
\top & \triangleq & \lambda h.\, \top \\
H_1 * H_2 & \triangleq & \lambda h.\, \exists h_1\, h_2.\, h_1 \vDash H_1 \wedge h_2 \vDash H_2 \wedge h = h_1 \circ h_2 \\
H_1 \mathrel{-\!\!\circledast} H_2 & \triangleq & \lambda h.\, \exists h_1.\, h_1 \vDash H_1 \wedge (h \circ h_1) \vDash H_2 \\
\blacktriangle H & \triangleq & H * \top \\
\triangle H & \triangleq & H \mathrel{-\!\!\circledast} \top
\end{array}
$$

$$\frac{}{\blacktriangle emp \equiv \top}\ \textsc{ContainsEmp} \qquad \frac{}{\triangle emp \equiv \top}\ \textsc{MissingEmp} \qquad \frac{H_1 \vdash H_2}{\blacktriangle H_1 \vdash \blacktriangle H_2}\ \textsc{ContainsMono} \qquad \frac{H_1 \vdash H_2}{\triangle H_1 \vdash \triangle H_2}\ \textsc{MissingMono}$$

$$\frac{}{\blacktriangle\blacktriangle H \equiv \blacktriangle H}\ \textsc{ContainsIdem} \qquad \frac{}{H \vdash \blacktriangle H}\ \textsc{ContainsSelf} \qquad \frac{}{\blacktriangle(H_1 * H_2) \equiv H_1 * \blacktriangle H_2}\ \textsc{ContainsSep}$$

$$\frac{}{\triangle(H_1 * H_2) \equiv H_1 \mathrel{-\!\!\circledast} \triangle H_2}\ \textsc{MissingSep} \qquad \frac{}{\blacktriangle(\exists x.\, H\ x) \equiv \exists x.\, \blacktriangle(H\ x)}\ \textsc{ContainsExists}$$

$$\frac{}{\triangle(\exists x.\, H\ x) \equiv \exists x.\, \triangle(H\ x)}\ \textsc{MissingExists} \qquad \frac{}{\blacktriangle H \equiv H * \triangle H}\ \textsc{ContainsMissingSep} \qquad \frac{}{\triangle\blacktriangle H \equiv \triangle H}\ \textsc{MissingContains}$$

$$\frac{}{\blacktriangle\triangle H \equiv [Sat\ H] * \top}\ \textsc{ContainsMissing} \qquad \frac{}{\blacktriangle H_1 * \blacktriangle H_2 \equiv \blacktriangle(H_1 * H_2)}\ \textsc{ContainsSepSym} \qquad \frac{}{\blacktriangle H * H_1 \vdash \blacktriangle H}\ \textsc{ContainsFrame}$$

Fig. 2.14. Framing modalities, and some of their properties

With the two framing modalities in hand, the properties of **produce** and **consume** can be finally be stated succinctly (Fig. 2.15).

First, somewhat unintuitively, on pure propositions **produce** and **con-**

**sume** are equivalent to **assert**. This is because a program using either operation must demonstrate the existence of a(n empty) piece of heap satisfying the proposition, which is tantamount to asserting/proving it. For the same reason, both are covariant as well.

Next, there is an adjunction between **produce** and **consume**, analogous to the one between **assume** and **assert**, but with additional conditions. First, $H$ must be *precise*: the heap satisfying it must be uniquely determined. This is to ensure that **produce** and **consume** act on the same piece of heap. Furthermore, framing modalities may be introduced. To see why, consider the left-to-right direction of ProduceConsumeAdj. Suppose we know **produce** $H; \varphi_1 \sqsubseteq \varphi_2$. This means $\varphi_1$ requires a pre-heap containing a $H$-piece, while $\varphi_2$ does not, and in fact *produces* such a piece, as the refinement is true. Therefore, to prove $\varphi_1 \sqsubseteq$ **consume** $H; \varphi_2$, we need a pre-heap $\blacktriangle H$, where $\varphi_1$ executes, and where we must first **consume** before we can "run" $\varphi_2$. The other three directions are analogous.

The other rules are practical ones, useful for proving entailments. We can reduce **produce** and **consume** on either side to **skip** under different conditions, and cancel them on both sides by flipping the modality, and fuse them. The final rule, Biab, enables symbolic execution, by allowing us to transpose a **produce** followed by a **consume**. $H_2$ must be precise because after producing $H_1$, there may be more than one $H_2$-shaped piece unless $H_2$ is precise.

**Theorem 2.2.1** (Soundness of Entailment Rules)**.** *Given an entailment $\varphi_a \sqsubseteq \varphi_c$, every behaviour of the consequent is one of the antecedent, i.e. if $\mathcal{E}, h_1, h_2, R \vDash \varphi_a$, then $\mathcal{E}, h_1, h_2, R \vDash \varphi_c$.*

*Proof.* See Section 6.1. □

$$\mathit{precise}\ H \triangleq \forall\, h_1\, h_2.\ h_1 \vDash H \Rightarrow h_2 \vDash H \Rightarrow h_1 = h_2$$

ConsumePure
$$\frac{}{\textbf{assert}\ P \equiv \textbf{consume}\ [P]}$$

ProducePure
$$\frac{}{\textbf{assert}\ P \equiv \textbf{produce}\ [P]}$$

ConsumeMono
$$\frac{H_1 \vdash H_2}{\textbf{consume}\ H_1 \sqsubseteq \textbf{consume}\ H_2}$$

ProduceMono
$$\frac{H_1 \vdash H_2}{\textbf{produce}\ H_1 \sqsubseteq \textbf{produce}\ H_2}$$

ProduceConsumeAdj
$$\frac{\mathit{precise}\ H}{\textbf{produce}\ H; \varphi_1 \sqsubseteq \varphi_2 \iff \blacktriangle H \vdash \varphi_1 \sqsubseteq \textbf{consume}\ H; \varphi_2}$$

ConsumeProduceAdjunction
$$\frac{\mathit{precise}\ H}{\textbf{consume}\ H; \varphi_1 \sqsubseteq \varphi_2 \iff \triangle H \vdash \varphi_1 \sqsubseteq \textbf{produce}\ H; \varphi_2}$$

ProduceConsumeCancel
$$\frac{\mathit{precise}\ H}{\textbf{produce}\ H; \textbf{consume}\ H \sqsubseteq \textbf{skip}}$$

ConsumeProduceIntro
$$\frac{}{\blacktriangle H \vdash \textbf{skip} \sqsubseteq \textbf{consume}\ H; \textbf{produce}\ H}$$

ConsumeProduceCancel
$$\frac{\mathit{precise}\ H}{\textbf{consume}\ H; \textbf{produce}\ H \sqsubseteq \textbf{skip}}$$

ProduceConsumeIntro
$$\frac{}{\triangle H \vdash \textbf{skip} \sqsubseteq \textbf{produce}\ H; \textbf{consume}\ H}$$

ConsumeCancel
$$\frac{H_1 \vdash H_2 \qquad \triangle H \vdash \varphi_1 \sqsubseteq \varphi_2}{\blacktriangle H \vdash \textbf{consume}\ H_1; \varphi_1 \sqsubseteq \textbf{consume}\ H_2; \varphi_2}$$

ProduceCancel
$$\frac{H_1 \vdash H_2 \qquad \blacktriangle H \vdash \varphi_1 \sqsubseteq \varphi_2}{\triangle H \vdash \textbf{produce}\ H_1; \varphi_1 \sqsubseteq \textbf{produce}\ H_2; \varphi_2}$$

ProduceFuse
$$\frac{}{\textbf{produce}\ H_1; \textbf{produce}\ H_2 \equiv \textbf{produce}\ H_1 * H_2}$$

ConsumeFuse
$$\frac{}{\textbf{consume}\ H_1; \textbf{consume}\ H_2 \equiv \textbf{consume}\ H_1 * H_2}$$

Biab
$$\frac{H_A * H_1 \vdash H_2 * H_F \quad \mathit{precise}\ H_2}{\textbf{produce}\ H_1; \textbf{consume}\ H_2; \varphi \sqsubseteq \textbf{consume}\ H_A; \textbf{produce}\ H_F; \varphi}$$

Fig. 2.15. Select entailments (stateful specifications)

### 2.2.2 Reasoning about programs

In this section, we take a short detour to formalise the relation between staged logic and programs. To continue the progression towards a proof search procedure, skip ahead to Section 2.3.

$$\begin{aligned} e \ ::= \ & x \mid v \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \lambda x.\, e \mid \lambda f\, x.\, e \mid e_1\ e_2 \mid \mathit{assert}\ e \mid \\ & \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{ref}\ e \mid e_1 := e_2 \mid !e \end{aligned}$$

Fig. 2.16. Core language

**Core language** We begin with the definition of an ML-like core language (Fig. 2.16), with higher-order functions and primitive state. Expressions are assumed to be in A-normal form (ANF); sequencing and control over evaluation order may be achieved using *let*-bindings, which define immutable variables. Mutation may occur through heap-allocated *ref*s. *assert* allows proofs of program properties to be carried out at arbitrary points.

We assume a standard set of values $v$, including the unit value (), integers $i$, strings $s$, boolean values *true* and *false*, tuples $(v_1, v_2)$, lists built using constructors [] and ::, recursive and non-recursive lambda expressions $\lambda f\, x.\, e$ and $\lambda x.\, e$, and *locations* $\ell$.

A big-step reduction relation $E, h_1, e \rightsquigarrow h_2, v$ (Fig. 2.17) is used to define the semantics of programs. As before (Section 1.2), program states are represented by heaps, and *program environments* $E$ are the program analogue of specification environments, mapping variables to function values.

*let*-bindings are handled by (capture-avoiding) substitution in BIGLET, avoiding the need for a separate store. Function application is handled by BIGAPPFUN and BIGAPPFIX if functions are known; if they are unknown, i.e., a variable $f$, which would otherwise be stuck, the rule BIGAPPUNK applies. Due to the use of ANF and substitution to handle *let*-bindings, this situation only occurs when a function *without a prior definition* is applied. Hence, en-

$\boxed{E, h, e \rightsquigarrow h, v}$

BIGVAL
$$\frac{}{E, h, v \rightsquigarrow h, v}$$

BIGASSERT
$$\frac{}{E, h, (assert\ true) \rightsquigarrow h, ()}$$

BIGAPPFUN
$$\frac{E, h_1, e[v/x] \rightsquigarrow h_2, R}{E, h_1, (\lambda x.\, e)\ v \rightsquigarrow h_2, R}$$

BIGAPPUNK
$$\frac{E, h_1, (E(f)\ v) \rightsquigarrow h_2, R}{E, h_1, (f\ \ v) \rightsquigarrow h_2, R}$$

BIGAPPFIX
$$\frac{E, h_1, e[v/x][(\lambda f\, x.\, e)/f] \rightsquigarrow h_2, R}{E, h_1, (\lambda f\, x.\, e)\ v \rightsquigarrow h_2, R}$$

BIGLET
$$\frac{\begin{array}{c} E, h_1, e_1 \rightsquigarrow h_3, v \\ E, h_3, e_2[v/x] \rightsquigarrow h_2, R \end{array}}{E, h_1, (let\ x = e_1\ in\ e_2) \rightsquigarrow h_2, R}$$

BIGREF
$$\frac{\begin{array}{c} \ell \notin dom(h_1) \\ h_2 = h_1[\ell \mapsto v] \end{array}}{E, h_1, (ref\ v) \rightsquigarrow h_2, \ell}$$

BIGIFT
$$\frac{E, h_1, e_1 \rightsquigarrow h_2, R}{E, h_1, (if\ true\ then\ e_1\ else\ e_2) \rightsquigarrow h_2, R}$$

BIGDEREF
$$\frac{\ell \in dom(h)}{E, h, (!\ell) \rightsquigarrow h, h(\ell)}$$

BIGIFF
$$\frac{E, h_1, e_2 \rightsquigarrow h_2, R}{E, h_1, (if\ false\ then\ e_1\ else\ e_2) \rightsquigarrow h_2, R}$$

BIGASSIGN
$$\frac{\ell \in dom(h)}{E, h, (\ell := v) \rightsquigarrow h[\ell \mapsto v], R}$$

Fig. 2.17. Big-step semantics

vironments enable talking about *open* programs by deferring their closure to the point where an environment is provided. We say more on this subject shortly, when we discuss the soundness of staged logic.

**Specification assertions** We relate a program and a staged formula via a *specification assertion*, a binary relation $e ::: \varphi$, named by analogy to a *type assertion* $e : \tau$.

A specification assertion may be thought of as a generalization of the Hoare triple – a triple $\{ P \}\, e \,\{ Q \}$ is equivalent in meaning to $e ::: \mathbf{req}\ P; \mathbf{ens}\ Q$. However, there are staged logic specifications which cannot directly be written as Hoare triples, at least without further parametrisation at the meta-logical level; a simple example is the *open* program $e; f\ v ::: \mathbf{req}\ P; \mathbf{ens}\ Q; f\ v$, where an unknown function $f$ is called after $e$. An equivalent specification in Hoare logic would require quantifying over some invariant of $f$, or properties representing (approximations of) its pre- and postconditions, as mentioned in Section 2.1. In this view, the use of unknown functions in staged logic can be seen as internalising this quantification syntactically, as a means of giving specifications which can talk directly about program behaviour.

Fig. 2.18 contains mostly syntax-directed rules for not just reasoning about programs, but *deriving* the staged logic form of a program. Focusing on the latter reading gives a us a means of *verifying* that a program conforms to a specification, via the rule of consequence FCONSEQ, the only rule that is not syntax-directed: to verify $e ::: \varphi_2$, it suffices to derive a specification $\varphi_1$ for $e$, then check the entailment $\varphi_1 \sqsubseteq \varphi_2$.

The FVAL rule relates a value $v$ to $\mathbf{ret}\ v$. In FLET, we first derive a specification $\varphi_1$ for the expression $e_1$ being bound, and, given that *its result is* $v$ – denoted by $\varphi[v] \triangleq \forall\, \mathcal{E}\, h_1\, h_2\, v_1.\ \mathcal{E}, h_1, h_2, v_1 \vDash \varphi \Rightarrow v_1 = v$ – we substitute $v$ into $e_2$, derive its specification, then simply sequence the (effects of the) two specifications. Given the use of ANF, *let*-bindings are the only place at

$\boxed{e ::: \varphi}$

$$\textsc{FConseq}\ \frac{e ::: \varphi_1 \quad \varphi_1 \sqsubseteq \varphi_2}{e ::: \varphi_2} \qquad \textsc{FVal}\ \frac{}{v ::: \mathbf{ret}\ v} \qquad \textsc{FLet}\ \frac{e_1 ::: \varphi_1 \quad \varphi_1[v] \quad e_2[v/x] ::: \varphi_2}{(let\ x = e_1\ in\ e_2) ::: \varphi_1; \varphi_2}$$

$$\textsc{FAppFun}\ \frac{e[v/x] ::: \varphi}{(\lambda x.e)\ v ::: \varphi} \qquad \textsc{FAppFix}\ \frac{e[v/x][(\lambda f\, x.e)/f\,] ::: \varphi}{(\lambda f\, x.e)\ v ::: \varphi} \qquad \textsc{FAppUnk}\ \frac{}{f\ v ::: f\ v}$$

$$\textsc{FIf}\ \frac{e_1 ::: \varphi_1 \quad e_2 ::: \varphi_2}{\begin{array}{c}(if\ b\ then\ e_1\ else\ e_2) ::: (\mathbf{assert}\ (b = true); \varphi_1) \\ \vee (\mathbf{assert}\ (b = false); \varphi_2)\end{array}}$$

$$\textsc{FDeref}\ \frac{}{\begin{array}{c}!\ell ::: \exists y.\ \mathbf{consume}\ (\ell \mapsto y) \\ \mathbf{produce}\ (\ell \mapsto y); \mathbf{ret}\ y\end{array}} \qquad \textsc{FAssign}\ \frac{}{\begin{array}{c}(\ell := y) ::: \exists v.\ \mathbf{consume}\ (\ell \mapsto v) \\ \mathbf{produce}\ (\ell \mapsto y); \mathbf{ret}\ ()\end{array}}$$

$$\textsc{FRef}\ \frac{}{(ref\ v) ::: \exists \ell.\ \mathbf{produce}\ (\ell \mapsto v); \mathbf{ret}\ \ell} \qquad \textsc{FAssert}\ \frac{}{assert\ b ::: \mathbf{assert}\ (b = true)}$$

Fig. 2.18. Specification assertions

which effects are sequenced. The use of substitution again allows us to avoid having stores as part of the model.

The next three rules concern function application. FAPPFUN and FAPPFIX are standard rules for handling the application of possibly-recursive, *known* lambda expressions, which occur in this form due to the use of ANF and substitution. Something to note is that despite their naive-looking definitions, which appear to require verifying same function repeatedly at every call to it with different arguments – quite against the spirit of modular verification! – if a function has had a prior specification assertion $\forall\, x.\ (e ::: \varphi)$ proven (quantified over parameters occurring in *both* $e$ and $\varphi$), we may simply instantiate it and use it to satisfy the premise.

The next rule, FAPPUNK, is completely trivial, yet perhaps the most illuminating as to the design of staged logic. If there is *no known specification* for a function $f$ – indicated at different levels by the absence of a premise in the rule, and a variable being left in application position after substitution – we may simply leave it in the specification and continue. This neatly deals with the possibility that $f$ is a function parameter, unknown, and/or unspecified. Utilizing the actual specification of $f$, *if* it is eventually provided, then becomes a concern of entailment proving.

The FIF rule models conditional branching as disjunction, with assertions on the value of the conditional in both disjuncts. The FDEREF, FASSIGN, FREF rules specify heap-manipulating primitives directly in terms of *consume* and *produce*. The FASSERT rule translates to an **assert** on the value of its argument. Something a unintuitive is that *assert* $b$ ::: **skip** is true as well, just weaker: **assert** refines any term that does not change the heap and ends with a unit result, as the effect of its proposition argument is not directly observable in terms of the model.

**Semantics** In order to state and prove the soundness of the specification assertion rules, we must first give specification assertions a semantics. We begin with a basic (partial correctness) definition reminiscent of the semantics of Hoare triples, for now parameterized over program and specification environments.

**Definition 2.3** (Validity under environments)**.** *A specification assertion $E, \mathcal{E} \vdash e ::: \varphi$ is valid under environments $E$ and $\mathcal{E}$ iff for all $h_1$, $h_2$, $v$, if $e$ evaluates to $v$ under $E, h_1, h_2$, then $\varphi$ is satisfied by $\mathcal{E}, h_1, h_2, v$.*

$$E, \mathcal{E} \vdash e ::: \varphi \triangleq \forall\, h_1\, h_2\, v.$$
$$(E, h_1, e \rightsquigarrow h_2, v) \Rightarrow (\mathcal{E}, h_1, h_2, v \vDash \varphi)$$

Intuitively, this says that given an execution of a program $e$, given intensionally as a big-step semantics relating some initial and final configurations, we must be able to show that the configurations are a model of the formula $\varphi$. In other words, every behaviour of $e$ corresponds to one of $\varphi$, and the behaviors of $\varphi$ overapproximate those of $e$.

The next step is to supply appropriate environments. We first need an assumption over the contents of the input environments $E$ (mapping names to programs) and $\mathcal{E}$ (mapping names to specifications) such that executions that refer to something in an environment also correspond. The solution is to require that the environments are *compatible*, in the sense that their contents are also related by valid specification assertions.

**Definition 2.4** (Environment compatibility)**.**

$$\begin{aligned} &\mathit{compatible}(E, \mathcal{E}) \triangleq \\ &\quad \forall\, x\, e\, f\, a.\ E(f) = (\lambda x.\, e) \Rightarrow \\ &\qquad \exists\, g.\ \mathcal{E}(f) = g \wedge \left(E, \mathcal{E} \vdash e[a/x] ::: g\ a\right) \end{aligned}$$

Finally, we close the definition by quantifying over environments.

**Definition 2.5** (Specification assertion validity)**.**

$$e ::: \varphi \triangleq \forall\, E\, \mathcal{E}.\ \mathit{compatible}(E, \mathcal{E}) \Rightarrow (E, \mathcal{E} \vdash e ::: \varphi)$$

With validity defined, we can state and prove the soundness of the system of rules.

**Theorem 2.1** (Soundness of specification assertions)**.** *If* $e ::: \varphi$ *is derivable using the rules of Fig. 2.18, then it is valid.*

*Proof.* By induction over the derivation of $e ::: \varphi$. □

**History triples** The original paper on staged logic [83] used a *history triple*, a ternary relation between two staged logic terms and a program $\{ \varphi_h \} e \{ \varphi \}$, as a direct analogue to the Hoare triple. History triples add to specification assertions $e ::: \varphi$ the *history* $\varphi_h$ of a program. We describe them in this section for posterity and completeness.

The rules are given in Fig. 2.19. There are two new structural rules. HHistFrame is analogous to the frame rule from separation logic (which still applies and is used internally, within heap assertions in **req** and **ens**, but only for program states), but for *program behaviour*. HConseq is similar to FConseq, but is contravariant in the history. HAppFun and HAppFix are very similar to their earlier counterparts, simply recursing on $e$, while HVal, HAssert, HDeref, HAssign, HRef, and HAppUnk are practically the same, the only difference being that they *append* some description of their effect to the history. The only rules that are stated differently are HIf and HLet. In HIf, we instead append the constraint on the condition to the *history* of each branch as an **assert**. This is functionally the same thanks to AssertIntro and AssumeElim, but results in a more succinct rule. In HLet, we use $\varphi_1$ as the history of $e_2$ instead, leaving sequencing implicit.

**Relation to specification assertions** A specification assertion is simply a history triple with an empty history.

**Lemma 2.1** (Empty history)**.** $e ::: \varphi \iff \{ \mathbf{skip} \} e \{ \varphi \}$

$\boxed{\{\,\varphi\,\}\,e\,\{\,\varphi\,\}}$

HHistFrame
$$\frac{\{\,\varphi_h\,\}\,e\,\{\,\varphi\,\}}{\{\,\varphi_f;\varphi_h\,\}\,e\,\{\,\varphi_f;\varphi\,\}}$$

HConseq
$$\frac{\varphi_1 \sqsubseteq \varphi_3 \quad \varphi_4 \sqsubseteq \varphi_2 \quad \{\,\varphi_3\,\}\,e\,\{\,\varphi_4\,\}}{\{\,\varphi_1\,\}\,e\,\{\,\varphi_2\,\}}$$

HVal
$$\frac{}{\{\,\varphi_h\,\}\,v\,\{\,\varphi_h;\mathbf{ret}\ v\,\}}$$

HAssert
$$\frac{}{\{\,\varphi_h\,\}\,\mathit{assert}\ b\,\{\,\varphi_h;\mathbf{assert}\ (b=\mathit{true})\,\}}$$

HDeref
$$\frac{}{\{\,\varphi_h\,\}\,!\ell\,\{\,\varphi_h;\exists\, y.\ \mathbf{consume}\ (\ell\mapsto v)\ \ \mathbf{produce}\ (\ell\mapsto v);\mathbf{ret}\ v\,\}}$$

HAssign
$$\frac{}{\{\,\varphi_h\,\}\,\ell:=v\,\{\,\varphi_h;\exists\, v.\ \mathbf{consume}\ (\ell\mapsto v_0)\ \ \mathbf{produce}\ (\ell\mapsto v);\mathbf{ret}\ ()\,\}}$$

HRef
$$\frac{}{\{\,\varphi_h\,\}\,\mathit{ref}\ v\,\{\,\varphi_h;\exists\,\ell.\ \mathbf{produce}\ (\ell\mapsto v);\mathbf{ret}\ \ell\,\}}$$

HIf
$$\frac{\{\,\varphi_h;\mathbf{assert}\ (b=\mathit{true})\,\}\,e_1\,\{\,\varphi_1\,\} \quad \{\,\varphi_h;\mathbf{assert}\ (b=\mathit{false})\,\}\,e_2\,\{\,\varphi_2\,\}}{\{\,\varphi_h\,\}\,\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\,\{\,\varphi_1\vee\varphi_2\,\}}$$

HLet
$$\frac{\{\,\varphi_h\,\}\,e_1\,\{\,\varphi_1\,\} \quad \varphi_1[v] \quad \{\,\varphi_1\,\}\,e_2[v/x]\,\{\,\varphi_2\,\}}{\{\,\varphi_h\,\}\,\mathit{let}\ x=e_1\ \mathit{in}\ e_2\,\{\,\varphi_2\,\}}$$

HAppFun
$$\frac{\{\,\varphi_h\,\}\,e[v/x]\,\{\,\varphi\,\}}{\{\,\varphi_h\,\}\,(\lambda x.\,e)\ v\,\{\,\varphi\,\}}$$

HAppFix
$$\frac{\{\,\varphi_h\,\}\,e[v/x][(\lambda f\,x.\,e)/f\,]\,\{\,\varphi\,\}}{\{\,\varphi_h\,\}\,(\lambda f\,x.\,e)\ v\,\{\,\varphi\,\}}$$

HAppUnk
$$\frac{}{\{\,\varphi_h\,\}\,f\ \ v\,\{\,\varphi_h;f\ \ v\,\}}$$

Fig. 2.19. History triples

As the history describes program behaviour that has already occurred, its result does not contribute to any staged formula $\varphi$ derived from $e$, as HRES shows. This fact may be used to derive HAPPEND, which shows that every specification assertion can be turned into an append-only history triple. Soundness of history triples then directly follows from this and Theorem 2.1.

$$\text{HRES}\ \frac{\{\,\varphi_h; \mathbf{skip}\,\}\,e\,\{\,\varphi\,\}}{\{\,\varphi_h\,\}\,e\,\{\,\varphi\,\}} \qquad \text{HAPPEND}\ \frac{e ::: \varphi}{\{\,\varphi_h\,\}\,e\,\{\,\varphi_h; \varphi\,\}}$$

The remaining triples, including the structural rules, are proved sound by semantic reasoning. To that end, we give a semantics for history triples, then state the soundness theorem.

**Definition 2.6** (Validity of history triples)**.**

$$\begin{aligned}\{\,\varphi_h\,\}\,e\,\{\,\varphi\,\} \triangleq \forall\, E\, \mathcal{E}\, h_0\, h_1\, h_2\, v\, R.\\ \mathcal{E}, h_0, h_1, R \vDash \varphi_h \Rightarrow \mathit{compatible}(E, \mathcal{E}) \Rightarrow\\ (E, h_1, e \rightsquigarrow h_2, R) \Rightarrow (\mathcal{E}, h_0, h_2, R \vDash \varphi)\end{aligned}$$

To explain the definition simply, the history $\varphi_h$ constrains execution from some *pre-initial* state $h_0$ to the program's initial state $h_1$. The program then executes from $h_1$ to $h_2$. The description of the program $\varphi$ includes the history, so it constrains execution from $h_0$ to $h_2$, and the definition asserts that $e$ and $\varphi$ are related in the same way as in Definition 2.3.

**Theorem 2.2** (Soundness of history triples)**.** *Every history triple derivable using the rules of Fig. 2.19 is valid.* □

## 2.3 Entailment checking

We return in this section to the subject of automatically checking entailments $\varphi_1 \sqsubseteq \varphi_2$. We first cover the key issues and principles required for constructing proofs, then cover the ideas required for proof search in HEIFER. The presentation returns to the idealised version of staged logic from Fig. 1.1.

### 2.3.1 Flavours of proof rules

Section 2.3 presented properties of the staged logic refinement relation. Seen as proof rules, these properties can be divided into three kinds based on their function in proof search.

First are the two kinds of *rewriting rules*, which have $\sqsubseteq$ only "below the line". For example, many of the steps in Section 2.1.2 went by rewriting using rules such as BIAB. These rules rely on the refinement relation being a *(pre)congruence*: a preorder/equivalence relation that is compatible with the constructors of staged logic. A specific kind of rewriting rule is the *reduction rule*, an equivalence which follows directly from the semantics of reduction. Reduction rules which evaluate forward, e.g., BINDETA and SEQUNITL, are of particular interest because they tend to leave a simpler result. Other rewriting rules (e.g., BIAB, BINDASSOC, FORALLDISTRR) are useful but do not directly reflect reduction.

The third kind of rule is an *entailment rule*. It has at least one $\sqsubseteq$ premise, and can be seen as a transformation of the entire entailment statement, usually to lift things into the metalogic. Some examples are ASSUMECONTRA and CONSUMECANCEL, which expose pure/heap obligations in goals, and EXL and EXR, which lift quantifiers.

While in theory some entailment rules can also be used for rewriting, in practice it is useful to consider them separately. The main difference is that entailment rules apply to goals of a fixed form and do not require the congruence obligation.

Finally, rewriting and entailment rules work in tandem. For example, to lift a quantifier from staged logic into the metalogic, we must technically iterate FORALLDISTRR to move it to the left before FORALLR can lift it out.

### 2.3.2 Simplification to a normal form

Many rewriting rules can always be fruitfully applied and iterated to a fixed point to produce in a simpler goal. We call them *simplification rules*, denoted $\varphi_1 \rightsquigarrow \varphi_2$. Semantically these are just entailments, but are *oriented*, intended to be used on the antecedent in a left-to-right direction. The whole system of rules is manually ensured to be confluent and terminating.

$\boxed{\varphi \rightsquigarrow \varphi}$

$$\begin{array}{rr}
\varphi;(\forall\, x.\ \varphi_1) \rightsquigarrow \forall\, x.\ \varphi;\varphi_1 & \varphi;(\exists\, x.\ \varphi_1) \rightsquigarrow \exists\, x.\ \varphi;\varphi_1 \\
(\forall\, x.\ \varphi);\varphi_1 \rightsquigarrow \forall\, x.\ \varphi;\varphi_1 & (\exists\, x.\ \varphi);\varphi_1 \rightsquigarrow \exists\, x.\ \varphi;\varphi_1 \\
(\varphi_1;\varphi_2);\varphi_3 \rightsquigarrow \varphi_1;\varphi_2;\varphi_3 & (\mathbf{req}\ \varphi\ \varphi_1);\varphi_2 \rightsquigarrow \mathbf{req}\ \varphi\ (\varphi_1;\varphi_2) \\
\varphi;(\varphi_1 \vee \varphi_2) \rightsquigarrow \varphi;\varphi_1 \vee \varphi;\varphi_2 & (\varphi_1 \vee \varphi_2);\varphi \rightsquigarrow \varphi_1;\varphi \vee \varphi_2;\varphi \\
\mathbf{ens}\ H * [P] \rightsquigarrow \mathbf{ens}\ [P];\mathbf{ens}\ H & \mathbf{ens}\ [P_1];\mathbf{ens}\ [P_2] \rightsquigarrow \mathbf{ens}\ [P_1 \wedge P_2] \\
\mathbf{ens}\ H_1;\mathbf{ens}\ H_2 \rightsquigarrow \mathbf{ens}\ (H_1 * H_2) & \mathbf{req}\ H_1;\mathbf{req}\ H_2\ \varphi \rightsquigarrow \mathbf{req}\ (H_1 * H_2)\ \varphi
\end{array}$$

Fig. 2.20. Select simplification rules

The important rules are given in Fig. 2.20. Generally, they aim to widen the scope of quantifiers, associate sequences and **req** to the right, distribute disjunction outwards, and merge **ens** and **req** as much as possible while moving pure **ens** assumptions to the left. The goal of simplification is reach a sort of disjunctive normal form of sequences, where the head connective of each sequence in the antecedent $\varphi_a$ can guide the proof search.

$$\bigvee (\mathbf{ens}\ H;)?\varphi_a \sqsubseteq \varphi$$

Notably, the normal form has a **ens** $H$ prefix in the antecedent. This is useful to contain separation logic assertions, which cannot be directly lifted to the metalogic, by e.g., rules like AssertIntro; in this sense it is similar in role to the *spatial context* [124] in Iris. The rules are designed to manipulate and maintain this context.

The original paper on staged logic [83] identified the following stronger

normal form and used it to define an entailment checking procedure.

$$\bigvee(\mathbf{req}\ H\ \mathbf{ens}\ Q; f\ \ v;)^*\mathbf{req}\ H\ \mathbf{ens}\ Q \sqsubseteq \varphi$$

While it is achievable in the current setting, it no longer is with the more general kinds of effects described in subsequent chapters. We thus mention it here only for posterity.

### 2.3.3 Biabduction

$$\textsc{Biab}\ \frac{H_A * H_1 \vdash H_2 * H_F}{\mathbf{ens}\ H_1; \mathbf{req}\ H_2\ \varphi \sqsubseteq \mathbf{req}\ H_A(\mathbf{ens}\ H_F; \varphi)}$$

Fig. 2.21. Biabduction

A key rewriting rule is BIAB (Fig. 2.21). Intuitively, it allows an **ens** to be "pushed through" a **req**, a situation commonly encountered in the verification conditions that arise from function calls in programs – imagine $H_1$ as describing the state before a function call, and $H_2$ as the precondition of the function. This can be done by solving a *biabduction* [42] problem, a generalization of heap entailment which infers an *antiframe* $H_A$ in addition to the frame $H_F$. Example 2.1 demonstrates this, with the inferred antiframe and frame shown.

**Example 2.1** (Biabduction)**.**

$$\frac{[i=3]*(x\mapsto 3 * y\mapsto 2)\vdash (x\mapsto i)*(y\mapsto 2)}{\mathbf{ens}\ (x\mapsto 3 * y\mapsto 2); \mathbf{req}\ (x\mapsto i)\ \varphi \sqsubseteq \mathbf{req}\ [i=3](\mathbf{ens}\ y\mapsto 2; \varphi)}$$

The resulting frame and antiframe may be simpler due to the common parts (the points-to on $x$ in the example) having been eliminated. Either way, swapping the **req** and **ens** will bring them into closer proximity with other **req** and **ens**, allowing them to be merged. When iterated, this effectively

$\boxed{\varphi \sqsubseteq \varphi}$

$$\textsc{EBaseEns}\ \frac{D_1 \vdash D_2}{\mathbf{ens}\ D_1 \sqsubseteq \mathbf{ens}\ D_2} \qquad \textsc{EBaseReq}\ \frac{D_2 \vdash D_1}{\mathbf{req}\ D_1 \sqsubseteq \mathbf{req}\ D_2} \qquad \textsc{ESeq}\ \frac{\varphi_1 \sqsubseteq \varphi_3 \quad \varphi_2 \sqsubseteq \varphi_4}{\varphi_1;\varphi_2 \sqsubseteq \varphi_3;\varphi_4}$$

$$\textsc{EPureEns}\ \frac{\pi \Rightarrow (\varphi_1 \sqsubseteq \varphi_2)}{\mathbf{ens}\ \pi;\varphi_1 \sqsubseteq \varphi_2} \qquad \textsc{EPureReq}\ \frac{\pi \Rightarrow (\varphi_1 \sqsubseteq \varphi_2)}{\varphi_1 \sqsubseteq \mathbf{req}\ \pi;\varphi_2} \qquad \textsc{ECancelFn}\ \frac{x_1{=}x_2 \quad (\forall\, r.\ \varphi_1 \sqsubseteq \varphi_2)}{f\ x_1; r.\ \varphi_1 \sqsubseteq f\ x_2; r.\ \varphi_2}$$

$$\textsc{EWFInd}\ \frac{f\ x \sqsubseteq \varphi_f \qquad (\forall\, x_0.\ x_0 \leq x \Rightarrow f\ x_0 \sqsubseteq \varphi[x_0/x]) \Rightarrow \varphi_f \sqsubseteq \varphi}{f\ x \sqsubseteq \varphi} \qquad \textsc{EForAllL}\ \frac{\exists x.\ (\varphi_1 \sqsubseteq \varphi_2)}{(\forall\, x.\ \varphi_1) \sqsubseteq \varphi_2}$$

$$\textsc{EDisjL}\ \frac{\varphi_1 \sqsubseteq \varphi_3 \quad \varphi_2 \sqsubseteq \varphi_3}{(\varphi_1 \vee \varphi_2) \sqsubseteq \varphi_3} \qquad \textsc{EDisjR}\ \frac{\varphi_3 \sqsubseteq \varphi_i \quad i \in \{1,2\}}{\varphi_3 \sqsubseteq (\varphi_1 \vee \varphi_2)} \qquad \textsc{EForAllR}\ \frac{\forall x.\ (\varphi_1 \sqsubseteq \varphi_2)}{\varphi_1 \sqsubseteq (\forall\, x.\ \varphi_2)}$$

Fig. 2.22. Essential entailment rules

*symbolically executes* a sequence of separation logic state transitions.

BIAB is not an equivalence, as solutions to (bi)abduction problems are not unique [41]. In principle, this may lead to backtracking during proof search. In practice, the best solution can usually be found for symbolic heaps.

### 2.3.4 Proof search

We now have all the ingredients required to define a syntax-directed entailment checking procedure. This procedure cannot be complete, as $\varphi$ may contain arbitrary propositions. We thus focus on what is needed to handle a practical fragment.

**Algorithm 1:** Entailment proof search

```
let search () : unit tactic
  | unfold >> simpl >> reduction >> lemmas >> entl;
let entl () : unit tactic
  | base <|> lift_pure <|> biab <|> cancel <|> induction <|>
  |  quantifiers <|> disj <|> fail;
```

The approach in this section extends and generalises the one we proposed

originally [83], lifting the requirement for a normal form, and simplifying the argument for its soundness, by the use of only standard entailment and rewriting instead of a syntactic relation for propagating frames.

Algorithm 1 is structured as a pair of mutually-recursive functions in a tactic monad $'a\ tactic$, a function of type $pstate \to ('a \times pstate)\ iter$. *pstate* is the *proof state*, which contains things like assumptions, the current goal, and so on; it may be thought of as abstract for our purposes. *iter* is some abstract type of lazy list, allowing nondeterministic search. The search tree is constructed by the interleaved execution of the two functions, where a call to *search* occurs at every node, which performs simplification, unfolding of nonrecursive definitions, simplification (Section 2.3.2), and use of lemmas, followed by branching using the *alternative* constructor $<|>$; each of the *entl* cases applies one of the entailment rules (Fig. 2.22) before recursively calling *search* with a simplified goal.

The rules EBaseEns and EBaseReq serve as the base cases of this process, applying when the goal can be reduced to a (covariant or contravariant) separation logic entailment. Pure assumptions may be lifted to the context by EPureEns and EPureReq. Functions may be handled by unfolding, or in an uninterpreted manner by ECancelFn – where, if their inputs are equal, they are simply removed on both sides. If they are recursive, well-founded induction may apply (EWFInd), where an induction hypothesis is constructed and made available in the context of the new goal. Recursive functions must otherwise by handled by user-provided lemmas. ESeq is applied modulo associativity of sequencing. EDisjL, EDisjR, EForAllL, and EForAllR lift logical connectives to the metalogic; dual ones apply for conjunction and existentials. Heuristics are used to find constants for instantiating staged logic quantifiers; they mainly apply to quantifiers produced by Section 2.2.2. Quantifiers within embedded separation logic assertions are left where they are and given to the SMT solver, giving users some control over how they are

to be solved.

This proof search algorithm is implemented in a prototype tool named HEIFER,[9] a push-button verification tool which takes programs written in a subset of OCaml, annotated with specifications and lemmas, and verifies them automatically. For programs without annotations, it verifies them in a greybox manner by applying the rules of Section 2.2.2 to derive specifications. The approach is heuristic and may fail to find a proof. The automation is, however, structured in a hierarchical manner, so individual parts like, e.g., the simplification steps can be used on their own, as part of a larger, interactively-constructed proof.

## 2.4 Evaluation

To evaluate the effectiveness of HEIFER's approach, I verified a suite of programs involving higher-order functions and closures (Table 2.1). As the focus of our work is to explore a new program logic and subsumption-based verification methodology (rather than to verify existing programs), the benchmarks are small in size, and are meant to illustrate the style of specification and give a taste of the potential for automation. We thus selected programs that demonstrated interesting and varied use cases for imperative higher-order functions. A few were taken from the literature and adapted as necessary to our setting, with care taken to remain faithful to the spirit of each example. Those without references were created by us.

Table 2.1 provides an overview of the benchmark suite. The first two subcolumns show the size of each program (LoC) and the number of lines of user-provided specifications (LoS) required. The next two give the total wall-clock time taken $T$ (in seconds) to verify all functions in each program against the provided specifications, and the amount of time $T_P$ spent in external provers. Programs known to be natively inexpressible in each prover are marked with

[9] Higher-order Effectful Imperative Function Entailments and Reasoning

Table 2.1
Comparison with Cameleer and Prusti

| | Heifer | | | | Cameleer [161] | | | Prusti [211] | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | LoC | LoS | $T$ | $T_P$ | LoC | LoS | $T$ | LoC | LoS | $T$ |
| map | 13 | 11 | 0.66 | 0.58 | 10 | 45 | 1.25 | | N/A | |
| map_closure | 18 | 7 | 1.06 | 0.77 | | ✗ | | | N/A | |
| fold | 23 | 12 | 1.06 | 0.87 | 21 | 48 | 8.08 | | N/A | |
| fold_closure | 23 | 12 | 1.25 | 0.89 | | ✗ | | | N/A | |
| iter | 11 | 4 | 0.40 | 0.32 | | ✗ | | | N/A | |
| compose | 3 | 1 | 0.11 | 0.09 | 2 | 6 | 0.05 | | N/A | |
| compose_closure | 23 | 4 | 0.44 | 0.32 | | ✗ | | | ✗ | |
| closure [194] | 27 | 5 | 0.37 | 0.27 | | ✗ | | 13 | 11 | 6.75 |
| closure_list | 7 | 1 | 0.15 | 0.09 | | ✗ | | | N/A | |
| applyN | 6 | 1 | 0.19 | 0.17 | 12 | 13 | 0.37 | | N/A | |
| blameassgn [80] | 14 | 6 | 0.31 | 0.28 | | ✗ | | 13 | 9 | 6.24 |
| counter | 16 | 4 | 0.24 | 0.18 | | ✗ | | 11 | 7 | 6.37 |
| lambda | 13 | 5 | 0.25 | 0.22 | | ✗ | | | N/A | |
| | 197 | 73 | | | 45 | 112 | | 37 | 27 | |

"✗". Programs that could not be verified for technical reasons and are not known to be inexpressible are marked with "N/A".

The next column shows the same programs verified using Cameleer, a state-of-the-art deductive verifier [161]. Cameleer serves as a good baseline for several reasons: it is representative of the Hoare logic approach to automated verification, and, like HEIFER, targets a subset of OCaml. The most significant differences between Cameleer and HEIFER are that Cameleer does not support *effectful* higher-order functions and is intended to be used via the Why3 IDE in a semi-interactive way (allowing tactic-like *proof transformations*, used in the above programs) rather than in a push-button manner like HEIFER.

The last column shows results for Prusti [211]. Despite the major difference in Rust's ownership type system, we compare against Prusti because of its state-of-the-art support for mutable closures, highlighting caveats below. While we were able to reproduce the claims made in Prusti's OOPSLA 2021 artifact [144], we were not able to verify many of our own benchmark programs due to two technical reasons, namely lacking support for Rust's `impl Trait` (to return closures) and ML-like cons lists (which caused time-

outs and crashes). Support for closures is also not yet in mainline Prusti [73]. Nevertheless, we verified the programs we could use for the artifact.

**User annotations required.** Significantly less specification than code is required in Heifer, with an average LoS/LoC ratio of 0.37. This is helped by two things: the ability to specify unknown functions in specifications, and the use of biabduction, which allows the specifications of many functions to be automatically derived, requiring only auxiliary lemmas to be provided. In contrast, Cameleer's ratio is 2.49, due to the need to adequately summarise the behaviors of the function arguments and accompany these summaries with invariants and auxiliary lemmas. Two examples illustrating this are detailed in Appendix A. Prusti's ratio is 0.73, but a caveat is that in the programs for it, only closure reasoning was used, without lemmas or summarization.

**Proof automation for HEIFER** The time taken by external provers (column $T_p$ of Table 2.1) is low, indicating that proof automation is feasible for the small (but representative) higher-order programs we used in our benchmark. The remaining time $T$-$T_p$ is spent on the other proof steps required to reduce programs to staged logic terms (Section 2.2.2), simplification (Section 2.3.2), reduction to a pure verification condition, as well as the application of structural rules, induction, etc. (Section 2.3).

**Expressiveness** HEIFER is able to express many programs that Cameleer cannot, particularly closure-manipulating ones. This accounts for the ✗ rows in Table 2.1. While some of these can be verified with Prusti, unlike stages, Prusti's call descriptions do not capture ordering [69, 144]; an explicit limitation as shown by the ✗ rows in Prusti's column. Prusti is able to use history invariants and the ownership of the Rust type system, but this difference is more than mitigated in HEIFER with the adoption of an expressive staged logic with spatial heap state, more appropriate for the weaker (but more

general) type system of OCaml.

**Implementation** HEIFER is agnostic to the SMT solver used, relying on both Z3 [148] and Why3 [79] (with Z3, Alt-Ergo [136], and CVC4 [18] as back-ends) and switching between them dynamically, depending on the needs of queries. Why3 is more fully-featured, with an extensive standard library, while Z3 is faster for the theories it supports natively.

**Experimental setup** All experiments were performed on macOS using a 2.3 GHz Quad-Core Intel Core i7 CPU with 16 GB of RAM. Why3 1.7.0 was used, with SMT solvers Z3 4.12.2, CVC4 1.8, and Alt-Ergo 2.5.2. The Prusti artifact, a Docker image, was run using Moby 25.0.1.

**Data availability** All benchmark programs and the source code of HEIFER are available on GitHub.[10]

[10] https://github.com/hipsleek/Heifer

CHAPTER 3

# VERIFYING PROGRAMS WITH EFFECT HANDLERS

## 3.1 Introduction

*Algebraic effects* were introduced by Plotkin and Power [165] as a means of reasoning about computational effects *algebraically*, using equations relating effectful operations. *Effect handlers* [167] later enabled providing user-defined interpretations for effects. They proved to be not just elegant but practical, eventually finding their way into OCaml 5.0 [183] as the main means of structuring concurrency.

This chapter begins with a gentle introduction to effect handlers and related concepts. I then survey briefly how verification tools treat them today before covering how they are treated in staged logic.

```
type _ Effect.t += XCHG: int -> int Effect.t
```

```
let exchange n =
  perform (XCHG n)

let client () =
  exchange (exchange 42)
```

(a) Client

```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

(b) Handler

Fig. 3.1. `XCHG` [185, 66]

The `XCHG` synchronization primitive abstracts away the content of a mutable reference and offers clients a single operation on it: a client may update its value, receiving a reply containing the old value. Fig. 3.1 shows an implementation of it using an effect handler.

First, the extensible variant `Effect.t` is extended with a new, eponymous

constructor. At this point, the `XCHG` effect is *uninterpreted*; nothing is known about it other than that it takes an `int` and returns an `int`.

On the left is a use: a client that performs the effect twice. We would expect that it returns `42` and leaves the state of the underlying `ref` unchanged.

On the right is an effect handler which provides an interpretation for `XCHG`. It allocates a `ref` and invokes `client` under the handler. Upon the effect occurring, the handler carries out the state changes before *resuming* the *delimited continuation* `k`, which is a representation of the computation to be carried out after the incidence of the effect, delimited by the extent of the match. For example, on the first effect, it is the continuation `exchange □`, where □ is a *hole* to be filled with a value upon resumption (in this case, `old_p`). The client then performs a second effect before returning. The **`try`** syntax abbreviates a **`match`** with an identity value case, which the client's result flows out of, making the result 42.

**Zero-shot, one-shot, & multishot continuations** In this case, the continuation is only resumed once, and is thus called *one-shot*. It is of course also possible to write handlers which do not use the continuation at all – making them *zero-shot*, and essentially turning the effect into an exception – or use them multiple times, making them *multishot*. The latter are interesting because they admit both new use cases – such as nondeterminism, probabilistic programming [121], a Unix-style fork [133], or continuation-based web servers [143] – as well as new challenges, which we go into shortly.

**Deep and shallow handlers** Another notable point about this example that the handler remains installed after handling an effect, and so is in place to handle the effects which occur after each resumption (e.g., the second `XCHG`); it is called a *deep* handler. A useful mental model for it is as a fold over the computation tree of the client; it recursively handles effects which occur in subcomputations.

It is possible to instead work with *shallow handlers*, which handle only one effect and require a new handler for each resumption. This enables easier expression of some kinds of programs, including dynamic sequences of handled effects [200] and generators [104]. Semantically, while a deep handler is a fold over a computation tree of effects, a shallow handler is a case split [95]. Shallow handlers are supported in OCaml through the library functions `continue_with`, where a handler is associated with a resumption rather with a computation, and `fiber`, to lift a computation into a continuation to start the process.

In this work, I focus mostly on deep handlers, as they are the default in OCaml [1]. However, we also demonstrate that our work is applicable to shallow handlers, discussing provisions for them as needed. It is not difficult in principle to support reasoning for both, as they are inter-derivable [95], so one can be taken as primitive and the other as derived. Prior work has taken shallow handlers as primitive, as defining a deep handler in terms of a shallow one is easier [64].

**Uses of effects** Effect handlers are compelling because they are a structured means of programming with (delimited) continuations, which in turn provides a way to extend a language with user-defined computational effects. Effect handlers also allow abstraction, by decoupling an effectful program from its interpretation. One could imagine invoking an `XCHG`-using client under different handlers for e.g., testing or instrumentation.

### 3.1.1 Reasoning about effect handlers

To illustrate the current paradigm for specifying and verifying effect-handling programs, let us return to Fig. 3.1.

A client which invokes `exchange` twice should naturally get back the value they sent, causing no overall change to the state of the system. That seems

simple enough to confirm intuitively, but how can we specify and verify this in general?

Consider what we will require to verify the client modularly: eventually we will need pre- and postconditions for the **perform** of XCHG, as well as the use of **continue** in the handler.

de Vilhena and Pottier [66]'s insight is that the specifications of the continuation and the **perform** are dual, and can be expressed in the form of a *protocol*: essentially a contract for **perform**ing a particular effect which "describes functionality on which the client can rely, and which the handler must implement" [64]. Soares and Pereira [185] augment this with the idea of a *handler invariant*,[1] which a deep handler both preserves and ensures as it recursively handles effects.

```
protocol XCHG x:
  ensures !p = x && reply = old !p
  modifies p

let exchange n = perform (XCHG n)
(*@ ensures !p = n && result = old !p
    performs XCHG *)

try client () with ...
(*@ try_ensures !p = old !p && result = 42 *)
```

Fig. 3.2. Cameleer specifications for XCHG

Fig. 3.2 shows the specifications needed in GOSPEL, which Soares and Pereira use in their automated verification setting. First is the protocol, which associates a pre- and (relational) postcondition with an effect; it is then a simple matter for clients to know what **perform**ing the effect will do. exchange and client can be directly verified this way.

Next, we verify main, focusing on the effect case first. Due to the afore-

[1] de Vilhena and Pottier [66] originated the idea, but present it in more general terms as a combination of two features: (1) a guarded-recursive assumption containing the specification of a *deep-handler*, (2) with the postcondition universally quantified, allowing a user to instantiate it.

mentioned duality, the protocol's precondition tells us the initial state at the beginning of the handler body, and its postcondition is really the *precondition* of the **`continue`**. In other words, we must check that the handler body prior to the resumption actually implements the functionality specified by the protocol, that it sets the reference to the correct value, etc.

Next, what should the postcondition of the continuation be? The answer is that it should be the *handler invariant*. Seeing as the effects of the continuation are recursively handled, we can assume the handler invariant after the resumption and must re-establish it before the end of the handler body, much like the use of the postcondition of a recursive call. Because the resumption is the last thing in the handler body, this is immediate.

Finally, the (elided, identity) value case simply involves checking that the postcondition of `client` entails the handler invariant. This is also immediate, provided a suitable handler invarinant is chosen.

**Multishot continuations and context-locality** The protocol-based solution is elegant, and adequate for stock OCaml. However, OCaml may be extended [150] with multishot continuations. It has been noted that they enable some applications and make others notably simpler, but that more research is required to understand how best to support them efficiently [52].

The protocol approach (and more generally, Hoare logic) is notoriously difficult to extend to this setting. The problems may be seen in the following example from de Vilhena [64], in Fig. 3.3.

```
let call f b =
  b := 0;
  f ();
  b := !b + 1;
  assert (!b = 1)
```

Fig. 3.3. A problematic example with multishot continuations

Assuming we have a specification $\{true\}\, f\, ()\, \{r.\ r{=}()\}$ for `f`, it seems

obvious on first reading that the assertion cannot fail. The *bind rule* allows us to compose the specification of `f` with triples about the following statements to derive a triple about the whole program. The *frame rule* tells us that `f` does not have access to `b`, so the value of `b` must still be 0 after the call to `f`.

With multishot continuations, however, it is not possible to guarantee that the assertion holds. The reason is that `f` may capture its continuation and resume it multiple times, so `0`, `1`, and `2` are all possible values of `b`.

How does this conflict with the rules we normally take for granted?

One reason is that we cannot reason *context-locally* about `f`, i.e., reason about `f` and its context independently as the standard bind rule promises, at least without considering its effects. This is because f may capture its context and use it in an unrestricted way. Timany and Birkedal [202] noted this problem, which de Vilhena and Pottier [66] fixed with a weakest precondition predicate augmented with a context-local description of effects.

Another reason is that even with this fix, the frame rule also no longer applies because of how continuations can capture resources [66]. This problem has shown up many times in the literature, with different solutions. Delbianco and Nanevski [68] use a separation logic without a general frame rule; programs which preserve other parts of the heap must explicitly quantify over it in their specifications. Timany and Birkedal [202] distinguish *context-local* from *non-context-local* triples; the former allow local reasoning when there are no control operators, and the latter allow global reasoning essentially based on the operational semantics. de Vilhena [64] restricts the frame rule so that when an effect is performed, resources disappear and cannot be framed around uses of effects.

A theme common to all prior solutions is that at some point, local reasoning simply has to be restricted, with a possible fallback to global reasoning. With this in mind, the position we take is that it is fundamentally challenging to invent abstractions for modular reasoning about multishot continuations,

and we cannot truly have it without a first-class treatment of effects, i.e., *in the logic*. The solution we present next is guided by this intuition.

### 3.1.2 Effect handlers in staged logic

The difficulty of coming up with an abstraction might remind the reader of the issue addressed in the previous chapter (Section 2.1), and in a sense the problem and solution are the same.

The idea is to model effects and handlers directly in the logic, stating specifications using refinement. We may then reason about them using standard principles, such as rewriting and induction. Crucially, this retains context-locality: it is a modular approach in that one can always state a specification about a piece of code, and combine two specifications, with the tradeoff being that this statement may be without much abstraction at first. Nevertheless, abstraction can always be recovered when it is easier to do so.

To illustrate things concretely, we first explain how to verify the program in Fig. 3.3, then the one in Fig. 3.1. We end the section by concluding the narrative of Section 2.1 regarding exceptions.

First, the program in Fig. 3.3. We need a specification for `call`. For ease of presentation, we replace the call to the unknown function `f` with an effect `E`, to focus on the problematic multishot case, and define **assert** $D \triangleq$ **req** $D$; **ens** $r.\, D \wedge r{=}()$.

$$\mathit{call} \triangleq \forall\, b.\ \textbf{ens}\ b \mapsto 0; E; \forall\, v.\ \textbf{req}\ b \mapsto v; \textbf{ens}\ b \mapsto v{+}1; \textbf{assert}\ b \mapsto 1$$

The specification describes the state changes that occur on either side of the effect.

We would like to prove functional correctness: the assertion fails or does not depending on the kind of handler used and how it resumes the continuation, and if it does not fail, what the result should be.

This can be stated as the following three refinements.

$$\textbf{try}\ \mathit{call}\ \textbf{with}\ E, k \to -1 \sqsubseteq \forall\, b.\ \textbf{ens}\ r.\, b \mapsto 0 \wedge r = -1$$

$$\textbf{try}\ \mathit{call}\ \textbf{with}\ E, k \to k() \sqsubseteq \forall\, b.\ \textbf{ens}\ r.\, b \mapsto 1 \wedge r = ()$$

$$\textbf{try}\ \mathit{call}\ \textbf{with}\ E, k \to k(); k() \sqsubseteq \textbf{req}\ \mathit{false}$$

These entailments are simple enough to be automatically proved by symbolic execution. The proof for the one-shot case is as follows. Each step is annotated with (informal) justification for how to reach it from the previous step.

$$\begin{array}{lll}
 & \textbf{try}\ \mathit{call}\ \textbf{with}\ E, k \to k() \sqsubseteq \forall\, b.\ \textbf{ens}\ r.\, b \mapsto 1 \wedge r = () & \\
\sqsubseteq & \textbf{try}\,(\forall\, b.\ \textbf{ens}\ b \mapsto 0; E; \ldots)\ \textbf{with}\ E, k \to k() & \text{Unfold} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 0; \textbf{try}\,(E; \ldots)\ \textbf{with}\ E, k \to k() & \text{Float} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 0; \forall\, v.\ \textbf{req}\ b \mapsto v; \textbf{ens}\ b \mapsto v{+}1; \textbf{assert}\ b \mapsto 1 & \text{Handle} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 0{+}1; \textbf{assert}\ b \mapsto 1 & \text{Biab} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 1 \wedge r = 0 \quad \square & \text{Biab}
\end{array}$$

Float refers to the idea of letting pure prefixes of programs *float* out of handlers, to reduce their scope; Handle refers to simplification driven by the operational semantics of handlers; and Biab refers to the use of biabduction to simplify sequences of **req** and **ens**.

This example demonstrates the ability to reason simply and precisely about what happens with each kind of handler. As a protocol is not needed, there is no need to commit to an abstraction of the semantics for a particular handler, or try to abstract over all of them; the effect remains uninterpreted, as it was in the program, until a particular handler is encountered. In a similar way as in Section 2.1, *maintaining precision* has once again simplified our proof effort, opening the door to automation.

We close this section with a proof for `XCHG`, which also goes by symbolic execution.

We first model the client and handler.

$$
\begin{aligned}
\mathit{client} \triangleq\ & \mathit{XCHG}(42); r.\, \mathit{XCHG}(r) \\
\mathit{main} \triangleq\ & \forall\, p.\ \mathbf{ens}\ p \mapsto 0; \mathbf{try}\, \mathit{client}\, \mathbf{with}\, \mathit{XCHG}(n), k \to \\
& (\forall\, o.\ \mathbf{req}\ p \mapsto o; \mathbf{ens}\ p \mapsto n; k\ o)
\end{aligned}
$$

Then we prove the following entailment about their composition:

$$\mathit{main} \sqsubseteq \forall\, p.\ \mathbf{ens}\ r.\, p \mapsto 0 \wedge r{=}42$$

which is to say that the result is the original argument to the client, and the state of $p$ remains unchanged.

The proof is as follows.

$$
\begin{array}{lll}
 & \mathbf{ens}\ p \mapsto 0; \mathbf{try}\, \mathit{client}\, \mathbf{with}\, \mathit{XCHG}(n), k \to \ldots & \\
\sqsubseteq & \mathbf{ens}\ p \mapsto 0; & \text{Handle} \\
 & \mathbf{ens}\ k{=}(\lambda x.\, \mathbf{try}\, \mathit{XCHG}(x)\, \mathbf{with}\, \mathit{XCHG}(n), k \to \ldots) & \\
 & \forall\, o.\ \mathbf{req}\ p \mapsto o; \mathbf{ens}\ p \mapsto 42; k\ o & \\
\sqsubseteq & \mathbf{ens}\ k{=}(\lambda x.\, \mathbf{try}\, \mathit{XCHG}(x)\, \mathbf{with}\, \mathit{XCHG}(n), k \to \ldots); & \text{Biab} \\
 & \mathbf{ens}\ p \mapsto 42; k\ 0 & \\
\sqsubseteq & \mathbf{ens}\ p \mapsto 42; \mathbf{try}\, \mathit{XCHG}(0)\, \mathbf{with}\, \mathit{XCHG}(n), k \to \ldots & \text{Simplify} \\
\sqsubseteq & \mathbf{ens}\ p \mapsto 42; \forall\, o.\ \mathbf{req}\ p \mapsto o; \mathbf{ens}\ p \mapsto n; k\ o & \text{Handle} \\
\sqsubseteq & \mathbf{ens}\ r.\, p \mapsto 0 \wedge r{=}42 \quad \square & \text{Biab}
\end{array}
$$

The reasoning goes very operationally, but at a high level of abstraction, using lemmas to take larger steps.

### 3.1.3 Exceptions and higher-order functions

In Section 2.1.1, we saw the issues with existing approaches for reasoning about higher-order functions and effects, with separation logic invariants providing only a partial solution for imperative behaviour, without an answer for other effects. In Section 2.1.2, staged logic was presented as an alternative, but for imperative effects only. Now that we have seen how effects are treated in staged logic, we conclude the earlier thread with an example illustrating reasoning about exceptions.

Consider the `foldl` function again. Previously, we proved a functional correctness result for using it compute the sum of a list of integers. Here, we

add partiality to the mix: we assume some bound $b$ which the sum should not exceed. When exceeded, `foldl` raises an exception to stop the computation.[2]

```
foldl (fun t x ->
  let r = x + t in
  if r < b
    then r
    else raise Failed) xs 0
```

Fig. 3.4. A use of `foldl` with exceptions

For efficiency, we would like to stop as soon as $b$ is exceeded, so we invoke `foldl` as in Fig. 3.4, checking each intermediate sum. For simplicity, we assume *xs* is a list of natural numbers, and $b > 0$.

$$\{\, b > 0 \wedge \mathit{sum}\ \mathit{xs} < b \,\}\ \texttt{foldl ... 0 xs}\ \{\, r.\ r = \mathit{sum}\ \mathit{xs} \,\}$$
$$\{\, b > 0 \wedge \mathit{sum}\ \mathit{xs} \geq b \,\}\ \texttt{foldl ... 0 xs}\ \{\, r.\ \mathit{Failed}; \mathbf{req}\ \mathit{false} \,\}$$

Fig. 3.5. Two specifications for `foldl`

How can we specify this function? Two triples are given in Fig. 3.5, for the different cases of whether the bound is exceeded. *Failed* is an effect that we use to model an exception. Having it in the specification ensures that `Failed` is raised. Furthermore, we follow it with the obligation to *prove false*, to express that *Failed* itself must *guarantee false*; in other words, there can be no meaningful behaviour after it.

As we will work in staged logic, we will reuse the model of *foldl* from Fig. 2.9; crucially, we will not have to write a new intermediate specification. We will also write the final specification using *intersection*, i.e., conjunction

[2] It is of course possible to *encode* partiality using the option monad, which would allow the use of a standard logic. The tradeoff is the need for a syntactically close but ultimately different definition of `foldl` using e.g., do-notation.

lifted to staged logic. The goal we would like to prove is thus:

$$\frac{acc < b}{\begin{array}{l} \quad foldl\ g\ acc\ xs \\ \sqsubseteq \mathbf{req}\ b > 0 \wedge acc + sum\ xs < b; \mathbf{ens}\ r.\, r = acc + sum\ xs \\ \quad \wedge \mathbf{req}\ b > 0 \wedge acc + sum\ xs \geq b; Failed; \mathbf{req}\ false \end{array}}$$

where $g \triangleq \lambda x\, t.\, \mathbf{ens}\ (x + t < b); \mathbf{ret}\ (x + t) \vee \mathbf{ens}\ (x + t \geq b); Failed$

Since each step of *foldl* updates the accumulator *acc*, the statement is generalized over it. Moreover, $acc < b$ is an invariant; without it, the statement would not hold.

The proof goes by induction on *xs*.

**Base case**

$$\begin{array}{l} \quad \mathbf{ens}\ xs{=}[\,]; \mathbf{ret}\ acc \\ \sqsubseteq \mathbf{req}\ b > 0 \wedge acc + sum\ xs < b; \mathbf{ens}\ r.\, r = acc + sum\ xs \\ \quad \wedge \mathbf{req}\ b > 0 \wedge acc + sum\ xs \geq b; Failed; \mathbf{req}\ false \end{array}$$

Simplifying,

$$\begin{array}{l} \quad \mathbf{ret}\ acc \\ \sqsubseteq \mathbf{req}\ b > 0 \wedge acc < b; \mathbf{ens}\ r.\, r = acc \\ \quad \wedge \mathbf{req}\ b > 0 \wedge acc \geq b; Failed; \mathbf{req}\ false \end{array}$$

The second case is absurd by the invariant $acc < b$, and the first is immediate.

**Inductive case**

$$\begin{array}{l} \quad \exists h\, t.\ \mathbf{ens}\ xs{=}h::t; g\ acc\ h; r.\ foldl\ g\ r\ t \\ \sqsubseteq \mathbf{req}\ b > 0 \wedge acc + sum\ xs < b; \mathbf{ens}\ r.\, r = acc + sum\ xs \\ \quad \wedge \mathbf{req}\ b > 0 \wedge acc + sum\ xs \geq b; Failed; \mathbf{req}\ false \end{array}$$

Eliminating existentials and simplifying,

$$\begin{array}{l} \quad g\ acc\ h; r.\ foldl\ g\ r\ t \\ \sqsubseteq \mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) < b; \mathbf{ens}\ r.\, r = acc + sum\ (h::t) \\ \quad \wedge \mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) \geq b; Failed; \mathbf{req}\ false \end{array}$$

Unfolding $g$,

$$\begin{array}{l} \quad (\mathbf{ens}\ (acc + h < b); \mathbf{ret}\ (acc + h) \vee \mathbf{ens}\ (acc + h \geq b); Failed); r.\ foldl\ g\ r\ t \\ \sqsubseteq \mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) < b; \mathbf{ens}\ r.\, r = acc + sum\ (h::t) \\ \quad \wedge \mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) \geq b; Failed; \mathbf{req}\ false \end{array}$$

Distributing sequencing over the disjunction and simplifying (in the second

branch *Failed* absorbs its continuation, so the recursive call is discarded),

$$\begin{aligned}
&\mathbf{ens}\ (acc + h < b); foldl\ g\ (acc + h)\ t \\
&\vee\ \mathbf{ens}\ (acc + h \geq b); Failed \\
\sqsubseteq\ &\mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) < b; \mathbf{ens}\ r.\, r = acc + sum\ (h::t) \\
&\wedge\ \mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) \geq b; Failed; \mathbf{req}\ false
\end{aligned}$$

At this point we can split the goal and prove the two conjuncts independently, which requires us to say how we arrive at the two outcomes: either the sum of the list, or failure.

**Case: sum of list** First, a useful lemma:

> **Lemma 3.1.** *Prefixes preserve the bound. For a list of natural numbers,* $acc + sum\ (h::t) < b \Rightarrow acc + h < b$ *and* $(acc + h) + sum\ t < b$.

The right disjunct in the antecedent cannot contribute to this outcome, so we drop it: the goal's **req** assumes $acc + sum\ (h::t) < b$, from which $acc + h < b$ (Lemma 3.1), making the guard **ens** $(acc + h \geq b)$ unsatisfiable.

$$\begin{aligned}
&\mathbf{ens}\ (acc + h < b); foldl\ g\ (acc + h)\ t \\
\sqsubseteq\ &\mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) < b; \mathbf{ens}\ r.\, r = acc + sum\ (h::t)
\end{aligned}$$

Since $acc + h < b$, we may apply the induction hypothesis at the accumulator $acc + h$; its left conjunct discharges the goal, using Lemma 3.1.

The first conjunct of the lemma justifies dropping the failed disjunct above; the second is exactly the precondition of the induction hypothesis, and $(acc + h) + sum\ t = acc + sum\ (h::t)$ gives the result.

**Case: failure** It is possible for both disjuncts to result in failure: the current step may already exceed the bound (from $g$), or it may stay below it and a later step in the recursive call exceeds it. We thus keep both disjuncts.

$$\begin{aligned}
&\mathbf{ens}\ (acc + h < b); foldl\ g\ (acc + h)\ t \\
&\vee\ \mathbf{ens}\ (acc + h \geq b); Failed \\
\sqsubseteq\ &\mathbf{req}\ b > 0 \wedge acc + sum\ (h::t) \geq b; Failed; \mathbf{req}\ false
\end{aligned}$$

In order to know which case we are in, we need to know if $acc + h \geq b$. We thus proceed by case analysis.

**Case:** $acc + h \geq b$ Failure must have come from the current step. We thus drop the success disjunct, whose guard **ens** $(acc + h < b)$ is unsatisfiable here.

$$\begin{array}{l} \quad \textbf{ens}\ (acc + h \geq b); \mathit{Failed} \\ \sqsubseteq \textbf{req}\ b > 0 \wedge acc + \mathit{sum}\ (h::t) \geq b; \mathit{Failed}; \textbf{req}\ \mathit{false} \end{array}$$

Eliminating the guard and **req**s,

$$\begin{array}{l} \quad \mathit{Failed} \\ \sqsubseteq \mathit{Failed}; \textbf{req}\ \mathit{false} \end{array}$$

This is immediate due to the **req** *false*.

**Case:** $acc + h < b$

$$\begin{array}{l} \quad \textbf{ens}\ (acc + h < b); \mathit{foldl}\ g\ (acc + h)\ t \\ \sqsubseteq \textbf{req}\ b > 0 \wedge acc + \mathit{sum}\ (h::t) \geq b; \mathit{Failed}; \textbf{req}\ \mathit{false} \end{array}$$

The current step succeeds, so failure must have come from the recursive call; the failed disjunct is dropped, its guard **ens** $(acc + h \geq b)$ being unsatisfiable. Since $acc + h < b$, we apply the induction hypothesis at the accumulator $acc + h$.

$$\begin{array}{l} \quad \textbf{req}\ b > 0 \wedge (acc + h) + \mathit{sum}\ t < b; \textbf{ens}\ r.\, r = (acc + h) + \mathit{sum}\ t \\ \quad \wedge\ \textbf{req}\ b > 0 \wedge (acc + h) + \mathit{sum}\ t \geq b; \mathit{Failed}; \textbf{req}\ \mathit{false} \\ \sqsubseteq \textbf{req}\ b > 0 \wedge acc + \mathit{sum}\ (h::t) \geq b; \mathit{Failed}; \textbf{req}\ \mathit{false} \end{array}$$

With $(acc + h) + \mathit{sum}\ t = acc + \mathit{sum}\ (h::t)$, the goal's assumption $acc + \mathit{sum}\ (h::t) \geq b$ makes the first conjunct's **req** absurd, while the second supplies the *Failed*; **req** *false* we need. □

Next, we cover the extensions to staged logic required to realise the reasoning in the above sections. A first-time reader may wish to skip ahead to Section 4.1 to see how these techniques apply more generally to delimited continuation operators.

## 3.2 Extensions to staged logic

$$
\begin{aligned}
\varphi &::= \ldots \mid \epsilon(v) \mid \mathbf{match}^{\delta}\ \varphi\ \mathbf{with}\ \mathcal{H} \\
\delta &::= s? \\
\mathcal{H} &::= x \to \varphi \mid \epsilon(v), k \to \varphi \mid \mathcal{H}
\end{aligned}
$$

---

$$
\mathbf{try}^{\delta}\ \varphi_1\ \mathbf{with}\ \epsilon(v), k \to \varphi_2 \triangleq \mathbf{match}^{\delta}\ \varphi_1\ \mathbf{with}\ x \to \mathbf{ret}\ x \mid \epsilon(v), k \to \varphi_2
$$

Fig. 3.6. Staged logic for effect handlers

**Syntax** We first extend the syntax of staged logic to be able to represent effect handlers (Fig. 3.6). Effects $\epsilon(v)$ and handlers are imported as-is from the language of programs. The reason is that unknown functions can appear in the scrutinees of matches, and knowing nothing else about them, we may have no choice but to leave them (with their contexts) in specifications, sans abstraction, to maintain precision. Handlers may be shallow (indicated by the superscript $s$) or deep (no superscript, and the default). **try** is an abbreviation for a **match** with an identity value case.

**Semantics** Extensions to the semantics are shown in Fig. 3.7. First, we add the notion of an *outcome* $R$, inspired by the operational view of effects as a generalisation of exceptions, aka the "bubble-up" semantics [118] – a formula $\varphi$ may reduce to a value $v$, or a "unhandled" effect $\epsilon(v, \lambda x.\, \varphi_k)$, containing a payload $v$ and a continuation (represented syntactically). The latter allows us to operationalise the construction of the delimited continuation that is conceptually built up around the effect as it "unwinds the stack". SEFF shows how this process begins – an unhandled effect on its own simply stands for itself, with an intermediate identity continuation. The SBIND rule from before is now named SBINDVAL, and an alternative rule SBINDEFF discriminates on the outcome of $\varphi_1$. The latter picks up after SEFF: given an

$\boxed{\mathcal{E}, h, h, R \vDash \varphi}$

$$R \;::=\; v \mid \epsilon(v, \lambda x.\, \varphi_k)$$

SEff
$$\frac{R = \epsilon(v, \lambda r.\, \mathbf{ret}\ r)}{\mathcal{E}, h, h, R \vDash \epsilon(v)}$$

SBindVal
$$\frac{\mathcal{E}, h_1, h_3, v \vDash \varphi_1 \qquad \mathcal{E}, h_3, h_2, R \vDash \varphi_2[v/r]}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\, \varphi_2}$$

SBindEff
$$\frac{\mathcal{E}, h_1, h_2, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi_1 \qquad R = \epsilon(v, \lambda x.\, \varphi_k; r.\, \varphi_2)}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\, \varphi_2}$$

SUnhandled
$$\frac{\mathcal{E}, h_1, h_2, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi \quad \epsilon \notin \mathcal{H} \qquad R = \epsilon(v, \lambda x.\, \mathbf{match}\ \varphi_k\ \mathbf{with}\ \mathcal{H})}{\mathcal{E}, h_1, h_2, R \vDash \mathbf{match}\ \varphi\ \mathbf{with}\ \mathcal{H}}$$

SHandle
$$\frac{\begin{array}{c}\mathcal{E}, h_1, h_3, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi \\ \epsilon(x), k \to \varphi_b \in \mathcal{H} \\ \varphi_k' = \mathbf{match}\ \varphi_k\ \mathbf{with}\ \mathcal{H} \\ \mathcal{E}[k := \lambda x.\, \varphi_k'], h_3, h_2, R \vDash \varphi_b[v/x]\end{array}}{\mathcal{E}, h_1, h_2, R \vDash \mathbf{match}\ \varphi\ \mathbf{with}\ \mathcal{H}}$$

SHandleVal
$$\frac{\begin{array}{c}\mathcal{E}, h_1, h_3, v \vDash \varphi \quad x \to \varphi_b \in \mathcal{H} \\ \mathcal{E}, h_3, h_2, R \vDash \varphi_b[v/x]\end{array}}{\mathcal{E}, h_1, h_2, R \vDash \mathbf{match}\ \varphi\ \mathbf{with}\ \mathcal{H}}$$

SHandleS
$$\frac{\begin{array}{c}\mathcal{E}, h_1, h_3, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi \quad \epsilon(x), k \to \varphi_b \in \mathcal{H} \\ \mathcal{E}[k := \lambda x.\, \varphi_k'], h_3, h_2, R \vDash \varphi_b[v/x]\end{array}}{\mathcal{E}, h_1, h_2, R \vDash \mathbf{match}^s\ \varphi\ \mathbf{with}\ \mathcal{H}}$$

Fig. 3.7. Extended semantics of staged logic, on top of Fig. 2.11

unhandled effect, SBINDEFF extends its continuation with the *current* continuation, which is made explicit by the bind. The next two rules show what happens when an unhandled effect finally makes it to a handler. If the effect remains unhandled, as in SUNHANDLED, the enclosing handler continues to apply to the continuation; this is true even for shallow handlers, which vanish only after a particular effect is handled. If the effect is handled, however (SHANDLE), execution continues with the appropriate branch of the handler, with the continuation bound to the intermediate continuation of $\epsilon$, which is at this point is completely known and delimited. The continuation is placed under an identical handler, implementing the semantics of deep handlers.

**Shallow handlers** SHANDLES is a variation of SHANDLE which implements the semantics of shallow handlers directly: the only difference is that the intermediate continuation $\varphi_k$ is not placed under an identical handler.

## 3.3 Reduction

With effect handlers in the language of staged logic, situations may arise where an effect occurs immediately inside a handler, and there is nothing to do but to handle the effect symbolically, by unfolding the body of the corresponding case. We can express this using a set of *reduction rules* (Section 2.3.2).

$$\begin{aligned}
\mathbf{ret}\ v; r.\ \varphi &\rightsquigarrow \varphi[v/r] \\
\varphi; r.\ \mathbf{ret}\ r &\rightsquigarrow \varphi \\
\mathbf{match}\ \varphi_1 \vee \varphi_2\ \mathbf{with}\ \mathcal{H} &\rightsquigarrow (\mathbf{match}\ \varphi_1\ \mathbf{with}\ \mathcal{H}) \vee \\
&\qquad (\mathbf{match}\ \varphi_1\ \mathbf{with}\ \mathcal{H})
\end{aligned}$$

Fig. 3.8. New simplification rules

First, Fig. 3.8 adds a few more rules to the set in Fig. 2.20, as well as a rule relating handlers and disjunction.

$$\textbf{match}^{\delta}\ \varphi_1 \,\#\, r.\, \varphi_2\ \textbf{with}\ x \to \varphi_3 \mid \epsilon(v), k \to \varphi_4 \triangleq \textbf{match}^{\delta}\ \varphi_1\ \textbf{with}\ x \to \varphi_3 \,\#\, r.\, \varphi_2 \mid \epsilon(v), k \to \varphi_4$$

Fig. 3.9. **match#**

Next, we introduce a handler variant called **match#** (Fig. 3.9). Intuitively, all it does is expose the continuation of a handler $\varphi_2$ as a parameter. Note that the effects of $\varphi_2$ *cannot* be handled by this particular handler; **match#** is hence useful for delimiting the effects of the scrutinee which have and have not been handled by this handler. This gives us a way to keep track of the progress being made in reduction.

$\boxed{\varphi \rightsquigarrow \varphi}$

RNorm
$$\frac{x \to \varphi_b \in \mathcal{H} \qquad \varphi \setminus \mathcal{H}}{\textbf{match}\ \varphi\ \textbf{with}\ \mathcal{H} \rightsquigarrow \varphi; x.\, \varphi_b}$$

RSkip
$$\frac{\varphi_1 \setminus \mathcal{H}}{\textbf{match}\ \varphi_1; r.\, \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \varphi_1; r.\, \textbf{match}\ \varphi_2\ \textbf{with}\ \mathcal{H}}$$

RDeep
$$\frac{\textbf{match}\ \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \varphi_c}{\textbf{match}\ \varphi_1; r.\, \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \textbf{match}\ \varphi_1 \,\#\, r.\, \varphi_c\ \textbf{with}\ \mathcal{H}}$$

RShallow
$$\frac{}{\textbf{match}^{s}\ \varphi_1; r.\, \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \textbf{match}\ \varphi_1 \,\#\, r.\, \varphi_2\ \textbf{with}\ \mathcal{H}}$$

RHandle
$$\frac{\epsilon(x), k \to \varphi_b \in \mathcal{H}}{\textbf{match}\ \epsilon(v) \,\#\, r.\, \varphi_c\ \textbf{with}\ \mathcal{H} \rightsquigarrow \varphi_b[\varphi_c/k][v/x]}$$

Fig. 3.10. Reduction rules for effect handlers

We are now ready to tackle the reduction rules of Fig. 3.10. The overall strategy of reduction is to minimise the scope of handlers, by moving things out of them, and by handling as many effects as we can statically. Any effects or effectful program fragments that we cannot treat this way – i.e., when none of these rules apply and we are "stuck" – will have to be reasoned about using induction or a user-provided lemma.

The first rule, RNorm applies when $\varphi$ is *effect-free* with respect to a handler $\mathcal{H}$, denoted $\varphi \setminus \mathcal{H}$, which means that it does not perform any effects

handled by $\mathcal{H}$.

$$\varphi \setminus \mathcal{H} \triangleq \forall\, \epsilon.\ \epsilon \in \mathcal{H} \Rightarrow \mathcal{E}, h_1, h_2, R \vDash \varphi \Rightarrow \neg(\exists\, v\ x\ \varphi_k.\ R = \epsilon(v, \lambda x.\ \varphi_k))$$

In that case, we can eliminate the handler by composing its value case with $\varphi$. RSKIP relates bind and handlers – it allows moving an effect-free prefix of the scrutinee out of the handler, reducing its scope. Effects unhandled by $\mathcal{H}$ will also be taken care of here.

If RSKIP cannot be applied, i.e., $\varphi_1$ will have its effects handled by $\mathcal{H}$, we have no choice but to reason operationally; the next two rules introduce **match**#. RDEEP reduces a deep handler from *right to left*, recursively reducing $\varphi_2$. This gives us $\varphi_c$, which we move it into the continuation, as its effects will no longer be handled by this handler. Shallow handlers will only handle one effect, so in RSHALLOW, we do not recursively reduce $\varphi_2$.

This process comes to a head when we have handled everything to the right, and moved everything to the left out. If the head of scrutinee is something like an unknown function, we will be stuck and require a lemma. Otherwise, if it is an effect, we can handle it by continuing with the body of the appropriate case and binding $\varphi_c$ as the continuation in it, eliminating **match**#.

**Theorem 3.3.1** (Soundness of Reduction Rules)**.**

$$\varphi_1 \rightsquigarrow \varphi_2 \textit{ iff } \mathcal{E}, h_1, h_2, R \vDash \varphi_1 \Rightarrow \mathcal{E}, h_1, h_2, R \vDash \varphi_2.$$

*Proof.* See Section 6.1. □

## 3.4 Another route to soundness

In Section 2.2.2, we showed that staged logic was a sound way of reasoning about programs. To do this, we defined the programming language that staged logic was intended to model, then defined specification assertions and

history triples, which related programs and staged logic terms using their semantics. This was justifiable because there was some semantic distance between the two languages, with state changes modelled by **req** and **ens**, unknown functions modelled by the specification environment, etc.

However, the additions to staged logic in this chapter were direct equivalents of the corresponding program constructs, in order to be able to treat them operationally using refinement and keep specifications precise. Hence, instead of just repeating the approach for the previous chapter, we take a different route: we redefine specification assertions once and for all so that they can accommodate future operational additions to staged logic.

The idea is essentially to take staged logic as the programming language, adding new constructs concretely, possibly in an uninterpreted way, with refinement to axiomatise definitions. With this view, staged logic becomes more of an extensible IVL. Entailment then becomes the workhorse for everything, even specification assertions.

**Definition 3.1** (Concrete terms)**.** *A* concrete term *of staged logic* $\Phi^e$ *for a given language of programs* $e$ *extends* $\varphi$ *with direct equivalents for all the constructs in* $e$*. For example, given a construct if* $x$ *then* $e_1$ *else* $e_2$*, the equivalent if* $x$ *then* $\Phi^e_1$ *else* $\Phi^e_2$ *would be present in* $\Phi^e$*, with the same semantics.*

We assume a function *translate*, which maps a program $e$ to the corresponding construct in $\Phi^e$. It is sound by construction, as all of the constructs of $e$ are present in $\Phi^e$, with identical semantics. A first approximation for *translate* would take the rules of Fig. 2.18 as the definition. Alternatively, one can think of *translate* as the manual process of modelling a problem in an IVL.

**Theorem 3.1** (*translate* is sound)**.** *Given that the program executes safely in a given state, i.e.,* $h_1, e \rightsquigarrow h_2, R$*, we can construct a corresponding derivation in the semantics of translate* $e$*, i.e.,* $\mathcal{E}, h_1, h_2, R \vDash$ *translate* $e$*.*

*Proof.* By construction. □

Finally, we define specification asserts and history triples as follows.

$$\{\,\varphi_h\,\}\,e\,\{\,\varphi\,\} \triangleq \varphi_h; \mathit{translate}\ e \sqsubseteq \varphi$$

$$e ::: \varphi \triangleq \mathit{translate}\ e \sqsubseteq \varphi$$

Theorems 2.1 and 2.2 then follow easily from the soundness of entailment and *translate*.

## 3.5 Evaluation

To evaluate our proposed approach to reasoning about effect handlers, we implemented it in HEIFER and verified a suite of benchmark programs. This section reports on the results, which are shown in Table 3.1.

Program 1 is taken from the *Memory Cell with Exchange* example from de Vilhena and Pottier [66] and extended with additional stateful operations. Programs 3 and 7 are from the benchmarks of the *multicont*[3] library, which provides practical examples for multishot continuations. Program 5 revises program 3 by changing the handler to be shallow. Programs 4 and 6 demonstrate the use of lemmas for left-recursive functions in both deep and shallow handlers. Program 2 is a new example.

Table 3.1
Experimental Results

| # | Program | Ind | MultiS | ImpureC | HO | LoC | LoS | Total(s) | Z3(s) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | State monad | ✗ | ✗ | ✓ | ✗ | 126 | 16 | 8.54 | 6.21 |
| 2 | Inductive sum | ✓ | ✗ | ✓ | ✗ | 41 | 11 | 1.68 | 1.28 |
| 3 | Flip-N (Deep Right Rec) | ✓ | ✓ | ✓ | ✗ | 39 | 10 | 2.09 | 1.52 |
| 4 | Flip-N (Deep Left Rec) | ✓ | ✓ | ✓ | ✗ | 45 | 13 | 2.03 | 1.53 |
| 5 | Flip-N (Shallow Right Rec) | ✗ | ✓ | ✓ | ✗ | 37 | 11 | 5.08 | 3.18 |
| 6 | Flip-N (Shallow Left Rec) | ✓ | ✓ | ✓ | ✗ | 64 | 23 | 6.75 | 4.26 |
| 7 | McCarthy's amb operator | ✓ | ✓ | ✓ | ✓ | 109 | 45 | 7.71 | 5.34 |
| | | | | | | 461 | 129 | 33.88 | 23.32 |

In Table 3.1, columns **LoC** and **LoS** record the lines of code and lines of

[3] https://github.com/dhil/ocaml-multicont

specification, respectively. The column **Total** records the total verification time taken in seconds. These numbers include the time taken by **Z3**, which is shown separately in the next column. Although verification is mostly automated, we show that the verification is non-trivial by labeling the programs with features: **Ind** indicates whether the proof is inductive, **MultiS** indicates if the program uses multishot continuations, **ImpureC** indicates if there are impure (heap-manipulating) continuations. **HO** indicates if the example is higher-order, i.e., function inputs or outputs are of function type.

Given 461 lines of code in total, 129 lines of specification are required, resulting in an average LoS/LoC ratio of 0.27.

The average verification time is in the order of seconds, with 68.8% of verification time taken by SMT solving. These demonstrate that the approach is lightweight, and feasible for guiding the design of an automated verification tool.

CHAPTER 4

# VERIFYING PROGRAMS WITH DELIMITED CONTINUATIONS

## 4.1 Introduction

While effects handlers have attracted much recent attention due to their inclusion in OCaml 5, they represent only one set of design choices – the literature contains many more delimited continuation operators with different characteristics. In this chapter, I generalise the framework of staged logic to handle them.

We begin with a gentle introduction to the *shift* and *reset* operators and the use of refinement for reasoning about them, before moving to the other operators in the family.

```
let do_toss () =
  shift k (k true + k false)

let toss () =
  reset (if do_toss () then 1 else 0)
```

Fig. 4.1. Tossing a coin using shift/reset

Fig. 4.1 demonstrates the use of shift and reset with a classic use case: nondeterminism [61]. `do_toss` uses the **`shift`** operator to get hold of its *continuation* – the remaining computation that its result will be used in – which it resumes twice, then adds the results of. Looking at `toss` shows us the (extent of the) continuation, which is `if □ then 1 else 0`, as well as what it returns: either `1` or `0` depending on the outcome of `do_toss`. In short, `toss` models a nondeterministic coin toss and returns the number of true tosses, which is $1 + 0 = 1$.

Specifying and reasoning about programs using control operators compositionally is challenging, due to their fundamentally *non-context-local* [202]

nature: if an expression $e$ contains a control operator, reasoning about $e$ will depend on its continuation (or context), something external to $e$. This necessitates an approach such as the protocols we saw in the previous chapter, or *non-context-local* triples $\{P\}\,C[e]\,\{Q\}$ [202]. We defer a survey of approaches to Section 7.4.2.

**Answer type modification** The ability to manipulate the continuation of an expression gives rise to some subtle issues, one of which is known as *answer type modification* [60] (ATM).

```
let percent_d () =
  shift k (fun n -> k (string_of_int n))

(reset (percent_d () ^ "2")) 1
```

Fig. 4.2. *printf* using answer-type modification [58]

Consider the program in Fig. 4.2. It is an encoding of a call to the classic *printf* function with the format string `"%d2"`, here represented as a delimited program which uses **`shift`**. Its result is `"12"`.

At a glance, the program appears rather odd: the result of the **`reset`** appears to be a string, the result of a string concatenation (`^`); this is referred to as its *answer type*. However, the result of **`reset`** is applied to the integer `1`. Somehow, the types "inside" and "outside" the **`reset`** are allowed to differ, even though, as we saw in the last example, **`reset`** has no operational behaviour besides delimiting the extent of **`shift`**s, and is the identity when its body is a value.

The explanation for this phenomenon is that the type of the **`shift`** body does not necessarily have to agree with the result type of the continuation. When these are different, the program is said to have *modified the answer type* of the **`reset`**.

Initially, the answer type of `percent_d` is `string`, as it is in a context which

accepts a string. However, the result of the **shift** body is a function (**fun** n -> ...), which is said to *modify the answer type* to int -> string. This function accepts an integer before producing a string using the continuation $k$, which is then the final result of the whole expression.

As this example demonstrates, ATM is useful: being able to repeatedly change the answer type allows us to accept a variable number of arguments depending on the expressions appearing in the **reset**, among other things.

ATM complicates reasoning about programs, due to the need to know about possible answer type changes in **reset** bodies in order to verify them. Again, more approaches are covered in Section 7.4.2; we first proceed to our approach, which works even in the presence of ATM.

### 4.1.1 Reasoning with refinement

We demonstrate the use of staged logic in the verification of a recursive program with shift/reset and ATM.

```
let append_sh xs =
  match xs with
  | [] -> shift k k
  | x :: xs1 -> x :: append_sh xs1

let append_delim xs ys =
  reset (append_sh xs) ys
```

Fig. 4.3. append using shift and reset [60]

append_delim in Fig. 4.3 uses ATM to implement a *difference list* [102]: a function that builds a list by appending its argument to it. It does this using the **shift** k k idiom, which changes the answer type from a list (as the *cons* branch suggests) to a function. The continuation is in effect a difference list: it includes all the deferred operations of the recursive calls to append_sh, which will build the list once a suitable tail is provided in ys.

This program is elegant, and it is easy to convince oneself informally that it implements the standard list append function. How can we prove this formally?

First, we write the specification as a refinement.

$$\langle append\_sh(xs)\rangle; f.\ f\ ys \sqsubseteq \mathbf{ens}\ r.\ r = xs +\!\!+ ys$$

Like before, we assume a staged logic with constructs $\mathbf{shift}\ k.\ \varphi$ and $\langle\varphi\rangle$, and the ability to symbolically execute them.

A straightforward induction on the structure of $xs$ does not work. We eventually see that the inductive case requires the following generalisation, over a prefix of the input $zs$; the original goal then follows from instantiating $zs$ with $[\,]$.

$$\langle append\_sh(xs); r.\ \mathbf{ret}\ zs +\!\!+ r\rangle; f.\ f\,(ys) \sqsubseteq \mathbf{ret}\ (zs +\!\!+ xs) +\!\!+ ys$$

This lemma *can* be proved by induction on $xs$.

**Base case** $\langle append\_sh([\,]); r.\ \mathbf{ret}\ zs +\!\!+ r\rangle; f.\ f\ ys \sqsubseteq \mathbf{ret}\ (zs +\!\!+ [\,]) +\!\!+ ys$

$$\begin{array}{lll}
& \langle append_sh([\,]); r.\ \mathbf{ret}\ zs +\!\!+ r\rangle; f.\ f\ ys & \\
\sqsubseteq & \langle (\mathbf{shift}\ k.\ k); r.\ \mathbf{ret}\ zs +\!\!+ r\rangle; f.\ f\ ys & \text{Unfold} \\
\sqsubseteq & \langle \mathbf{ret}\ (\lambda r.\ \langle \mathbf{ret}\ zs +\!\!+ r\rangle)\rangle; f.\ f\ ys & \text{Reduction}^1 \\
\sqsubseteq & \mathbf{ret}\ (\lambda r.\ \mathbf{ret}\ zs +\!\!+ r); f.\ f\ ys & \text{Pure} \\
\sqsubseteq & \mathbf{ret}\ zs +\!\!+ ys & \text{Simplify} \\
\sqsubseteq & \mathbf{ret}\ (zs +\!\!+ [\,]) +\!\!+ ys \quad \square & \text{Lem. ++}
\end{array}$$

**Inductive case** $\langle append\_sh(x_1 :: xs_2); r.\ \mathbf{ret}\ zs +\!\!+ r\rangle; f.\ f\ ys \sqsubseteq \mathbf{ret}\ (zs +\!\!+ (x_1 :: xs_2)) +\!\!+ ys$

$$\begin{array}{lll}
& \langle (append_sh(xs_2); r_1.\ \mathbf{ret}\ x_1 :: r_1); r.\ \mathbf{ret}\ zs +\!\!+ r\rangle; f.\ f\ \ ys & \\
\sqsubseteq & \langle append_sh(xs_2); r_1.\ \mathbf{ret}\ zs +\!\!+ (x_1 :: r_1)\rangle; f.\ f\ \ ys & \text{Reassoc. bind} \\
\sqsubseteq & \langle append_sh(xs_2); r_1.\ \mathbf{ret}\ (zs +\!\!+ [x_1]) :: r_1\rangle; f.\ f\ \ ys & \text{Lem. cons, ++} \\
\sqsubseteq & \mathbf{ret}\ (((zs +\!\!+ [x_1]) +\!\!+ xs_2) +\!\!+ ys) & \text{Ind. hyp.} \\
\sqsubseteq & \mathbf{ret}\ ((zs +\!\!+ (x_1 :: xs_2)) +\!\!+ ys) \quad \square & \text{Lem. cons, ++}
\end{array}$$

Reduction[1] (used in the base case) eliminates **shift**s from terms via symbolic execution, similar to Handle in Section 3.1.2. The result of the shift

body is $k$, the continuation itself, which prepends $zs$ to its result.

### 4.1.2 Tossing more coins

As another demonstration of the sort of generalisation required, we extend the example in Fig. 4.1 with recursive behaviour.

```
let do_toss () =
  shift k (k true + k false)

let rec do_toss_n n =
  if n = 0 then false
  else
    let r1 = do_toss () in
    let r2 = do_toss_n (n-1) in
    r1 || r2

let toss_n n =
  assert (n >= 0);
  reset (if do_toss_n n then 1 else 0)
```

Fig. 4.4. Tossing many coins

Fig. 4.4 shows a variation, `do_toss_n`, which tosses $n$ coins and takes the disjunction of all the results, i.e., it returns `true` if at least one of the coin tosses gives `true`. Because `do_toss` *sums* its two continuation results, `toss_n` enumerates all the outcomes, returning the number of *sequences* of tosses that contain at least one `true`.

This counts every flip-sequence except the single all-`false` one, so $2^n - 1$ of the $2^n$ leaves:

$$\mathit{toss}_n\ n \sqsubseteq \mathbf{ret}\ (2^n{-}1)$$

The proof requires this lemma, itself proved by induction:

$$\begin{aligned}&\langle \mathit{do\_toss}_n\ n; r.\ (\mathbf{ens}\ [\mathit{acc}{\vee}r{=}\mathit{true}]; \mathbf{ret}\ 1 \vee \mathbf{ens}\ [\mathit{acc}{\vee}r{=}\mathit{false}]; \mathbf{ret}\ 0)\rangle \\ \sqsubseteq\ &\mathbf{ens}\ [\mathit{acc}{=}\mathit{true}]; \mathbf{ret}\ 2^n \vee \mathbf{ens}\ [\mathit{acc}{=}\mathit{false}]; \mathbf{ret}\ (2^n{-}1)\end{aligned}$$

The statement must be generalised over an accumulator $acc$ of intermediate

results. Since the continuation is applied at both *k true* and *k false*, we need to know the behaviours of both.

A stateful variant is also possible: suppose we instrument the program to count the number of continuation resumptions, using a global `ref x` (Fig. 4.5).

```
let do_toss () =
  shift k.
    x := !x+1;
    let r1 = k true in
    x := !x+1;
    let r2 = k false in
    r1 + r2

let rec toss_n n =
  reset (if do_toss_n n then 1 else 0)
```

Fig. 4.5. A stateful version of `do_toss`

This counts the number of edges in the tree, which is $2^{n+1}-2$.

The statement...

$$toss_n\ n \sqsubseteq \forall v.\ \mathbf{req}\ x \mapsto v; \mathbf{ens}\ x \mapsto (v{+}2^{n+1}{-}2); \mathbf{ret}\ (2^n{-}1)$$

...and lemma are accordingly generalised.

$$\begin{array}{l} \langle do\_toss_n\ n; r.\ (\mathbf{ens}\ [acc \vee r = true]; \mathbf{ret}\ 1 \vee \mathbf{ens}\ [acc \vee r = false]; \mathbf{ret}\ 0)\rangle \\ \sqsubseteq \forall v.\ \mathbf{req}\ x \mapsto v; \mathbf{ens}\ x \mapsto (v{+}2^{n+1}{-}2); \\ \quad (\mathbf{ens}\ [acc = true]; \mathbf{ret}\ 2^n \vee \mathbf{ens}\ [acc = false]; \mathbf{ret}\ (2^n{-}1)) \end{array}$$

but this can otherwise be proved the same way, without any new machinery.

### 4.1.3 Beyond shift and reset

Next, I generalise the framework so that it may be applied to other continuation operators, in a similar spirit as Ishio and Asai [105]. Dybvig, Peyton Jones, and Sabry [76] present a taxonomy of such operators. Their semantics are given in Fig. 4.6. On the left is the notation used by Dybvig, Peyton Jones, and Sabry, while the name of the operator used appears in each rule.

$$\langle v \rangle \Longrightarrow v$$

$$+\mathcal{F}+ \quad \langle E[\textbf{shift}\ k.\ e] \rangle \Longrightarrow \langle (\lambda k.\, e)\ (\lambda v.\, \langle E[v] \rangle) \rangle \quad [61]$$

$$-\mathcal{F}+ \quad \langle E[\textbf{shift}_0\ k.\ e] \rangle \Longrightarrow (\lambda k.\, e)\ (\lambda v.\, \langle E[v] \rangle) \quad [60]$$

$$+\mathcal{F}- \quad \langle E[\textbf{control}\ k.\ e] \rangle \Longrightarrow \langle (\lambda k.\, e)\ (\lambda v.\, E[v]) \rangle \quad [65]$$

$$-\mathcal{F}- \quad \langle E[\textbf{control}_0\ k.\ e] \rangle \Longrightarrow (\lambda k.\, e)\ (\lambda v.\, E[v]) \quad [93]$$

Fig. 4.6. Reduction semantics of various control operators [76, 55]

The operators differ operationally in whether they leave a delimiter around the continuation (indicated by + on the right) and the operator body $e$ (+ on the left). Intuitively, these differences allow for more "dynamic" [31] behaviours, along distinct axes: **control** is able to access the bodies of parallel operators, while $\textbf{shift}_0$ is able to "escape" to enclosing delimiters [142].

Assuming **control** in the logic (Section 4.2), we illustrate how reasoning with it is as natural as with shift and reset, using an interesting example originally from Biernacki and Danvy [29].

```
let visit xs =
  match xs with
  | [] -> []
  | x :: xs1 -> visit (shift k (x :: k xs))

let traverse xs = reset (visit xs)
```

Fig. 4.7. `traverse` [29, 28, 30]

The function `traverse` (Fig. 4.7) copies a given list. Interestingly, if the **control** operator were instead used in `visit`, `traverse` *reverses* the list.

How does this happen? Let us first consider how the programs execute on a small input to for some intuition (Fig. 4.8). The key difference is in whether and how the control operators access the cons operations; each **shift** does not access them, and so extends the list in "well-behaved" manner, whereas **control** does access them and nests them one level deeper, resulting in the reversal.

Now, to prove these two facts, we start with specifications (Fig. 4.9): pure

```
reset (visit [1; 2; 3])
= reset (visit (shift k (1 :: k [2; 3])))
= reset (1 :: reset (visit [2; 3]))
= reset (1 :: reset (visit (shift k (2 :: k [3]))))
= reset (1 :: reset (2 :: reset (visit [3])))
= reset (1 :: reset (2 :: reset (visit (shift k (3 :: k [])))))
= reset (1 :: reset (2 :: reset (3 :: reset (visit []))))
= reset (1 :: reset (2 :: reset (3 :: reset ([]))))

reset (visit' [1; 2; 3])
= reset (visit' (control k (1 :: k [2; 3])))
= reset (1 :: visit' [2; 3])
= reset (1 :: visit' (control k (2 :: k [3])))
= reset (2 :: (fun x -> 1 :: visit' x) [3])
= reset (2 :: 1 :: visit' [3])
= reset (2 :: 1 :: visit' (control k (3 :: k [])))
= reset (3 :: (fun x -> 2 :: 1 :: visit' x) [])
= reset (3 :: 2 :: 1 :: visit' [])
= reset (3 :: 2 :: 1 :: [])
```

Fig. 4.8. shift to traverse, and control to reverse

functions *copy* and *reverse*. The latter is implemented using an accumulator.

$$
\begin{aligned}
&\mathit{copy}\ \mathit{xs} = \\
&\quad \mathbf{ens}\ \mathit{xs}{=}[\,];\mathbf{ret}\ [\,] \\
&\quad \vee\ \exists\, x\, \mathit{xs}_1.\ \mathbf{ens}\ \mathit{xs}{=}x :: \mathit{xs}_1; \mathit{copy}\ \mathit{xs}_1; r_1.\ \mathbf{ret}\ x :: r_1
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rev}\ \mathit{xs}\ \mathit{acc} = \\
&\quad \mathbf{ens}\ \mathit{xs}{=}[\,];\mathbf{ret}\ \mathit{acc} \\
&\quad \vee\ \exists\, x\, \mathit{xs}_1.\ \mathbf{ens}\ \mathit{xs}{=}x :: \mathit{xs}_1; \mathit{rev}\ \mathit{xs}_1\ (x :: \mathit{acc})
\end{aligned}
$$

Fig. 4.9. Specifications for *copy* and *rev*

That using **shift** copies the list is easy to prove by induction.

$$\langle \mathit{visit}\ \mathit{xs} \rangle \sqsubseteq \mathit{copy}(\mathit{xs})$$

The base case is trivial (both sides are identical after unfolding).

**Inductive case** $\langle visit\ (x :: xs_1)\rangle \sqsubseteq copy\ (x :: xs_1)$

$$
\begin{array}{lll}
& \langle visit\ (x :: xs_1)\rangle & \\
\sqsubseteq & \langle (\mathbf{shift}\ k.\ k\ xs_1; r.\ \mathbf{ret}\ x :: r); r_1.\ visit\ r_1\rangle & \text{Unfold} \\
\sqsubseteq & \mathbf{ens}\ k{=}(\lambda r_1.\ \langle visit\ r_1\rangle); \langle k\ xs_1; r.\ \mathbf{ret}\ x :: r\rangle & \text{Reduction}^2 \\
\sqsubseteq & \langle \langle visit\ xs_1\rangle; r.\ \mathbf{ret}\ x :: r\rangle & \text{Simplify} \\
\sqsubseteq & \langle copy\ xs_1; r.\ \mathbf{ret}\ x :: r\rangle & \text{Ind. hyp.} \\
\sqsubseteq & copy\ xs_1; r.\ \mathbf{ret}\ x :: r \quad \square & \text{Pure}
\end{array}
$$

The statement using **control** is trickier. $visit'$ is the same as *visit* except for using **control** instead of **shift**.

$$\langle visit'\ xs\rangle \sqsubseteq rev(xs, [\,])$$

In the inductive case, we eventually get stuck in the following state, where we can't appeal to the induction hypothesis.

$$\langle visit'\ xs_1; r.\ \mathbf{ret}\ x :: r\rangle$$

The key idea is that we must generalise the entailment over the *continuation* of $visit'$. In general, nontrivial generalisation may be required [96].

Suppose we did that for the left side of the entailment. Now, what is the appropriate generalisation of *rev* over $k$?

$$\langle visit'\ xs; r.\ k(r)\rangle \sqsubseteq \ldots$$

A naive idea would be to apply the CPS transformation to *rev*, resulting in a definition with signature $rev'\ xs\ acc\ k$. While this works just as well in staged logic as it does for programs, it does not help us: looking at the state where we got stuck, we see that the continuation that we are generalising over is involved in building the reversed list. In other words, we need a version of *rev* which performs the accumulation *using* the continuation, i.e., $rev_1\ xs\ k$. Coming up with this reformulation is not straightforward, as it is further complicated by **control**.

The good news is that this definition can be *synthesised* using just a small

extension of the machinery we have presented so far.

**Lemma synthesis** The technique of lemma synthesis we propose is an exploratory one which discovers intermediate lemmas to prove a goal.

Suppose we have an entailment $\varphi \sqsubseteq \varphi_c$ to be proved. We find that direct inductive proof does not work, and that we need to come up with a lemma $\varphi'$ as an intermediate step towards the goal. We know that this lemma is a generalisation of the goal and can write down its signature, but do not know its definition, nor the two proofs demonstrating that it is indeed an intermediate step. This situation is shown as the following incomplete proof tree.

$$\dfrac{\dfrac{\boxed{\ldots}}{\varphi \sqsubseteq \boxed{\varphi'}} \qquad \dfrac{\vdots}{\varphi' \sqsubseteq \varphi_c}}{\varphi \sqsubseteq \varphi_c}$$

The idea is that we can discover a candidate for $\varphi'$ by taking it as uninterpreted, and searching for a proof of $\varphi \sqsubseteq \varphi'$ as per normal. When we inevitably get stuck, we proceed by interpreting the pending goal(s) as constraints on the definition of $\varphi'$, making $\varphi \sqsubseteq \varphi'$ true by construction. Having synthesised $\varphi'$, we may then recursively continue trying to prove the goal $\varphi_c$.

We illustrate this technique by using it to solve the *visit*′ example. The statement we use to direct the search is as follows, where $rev_1$ is uninterpreted.

$$\langle visit'\ xs; r.\ k\ r\rangle \sqsubseteq rev_1\ xs\ k$$

**Base case** $\langle visit'\ [\,]; r.\ k\ r\rangle \sqsubseteq rev_1\ [\,]\ k$

$$\begin{array}{lll} & \langle visit'\ [\,]; r.\ k\ r\rangle & \\ \sqsubseteq & \langle \mathbf{ret}\ [\,]; r.\ k\ r\rangle & \text{Unfold} \\ \sqsubseteq & k\ [\,] \quad \square & \text{Simplify} \end{array}$$

**Inductive case** $\langle visit'\ x :: xs_1; r.\ k\ r\rangle \sqsubseteq rev_1\ x :: xs_1\ k$

$$
\begin{array}{lll}
& \langle visit'\ x :: xs_1; r.\ k\ r\rangle & \\
\sqsubseteq & \langle (\mathbf{control}\ k_1.\ k_1(xs_1); r.\ \mathbf{ret}\ x :: r); r_1.\ visit'\ r_1; r.\ k\ r\rangle & \text{Unfold} \\
\sqsubseteq & \mathbf{ens}\ k_1{=}(\lambda r_1. visit'\ r_1;; r.\ k\ r); \langle k_1(xs_1); r.\ \mathbf{ret}\ x :: r\rangle & \text{Reduction}^3 \\
\sqsubseteq & \langle visit'\ xs_1; r_1.\ k\ r_1; r.\ \mathbf{ret}\ x :: r\rangle & \text{Simplify} \\
\sqsubseteq & rev_1\ xs_1\ \lambda r_1. k(r_1); r.\ \mathbf{ret}\ x :: r \quad \square & \text{Ind. hyp.}
\end{array}
$$

This is as far as we can get in both proofs. Notably, the proof search is exactly what would have been done given a goal with an interpretation that we could direct the search towards. Unlike Reduction$^2$, Reduction$^3$ does not leave a delimiter in the continuation, according to the semantics of **control** (Fig. 4.6).

The final unproved goals together give us an inductive definition for $rev_1$ (Fig. 4.10), by assembling together the two cases under the assumptions on $xs$ in each case.

$$
\begin{array}{l}
rev_1\ xs\ k = \\
\quad \mathbf{ens}\ xs{=}[]; k\ [] \\
\quad \vee\ \exists\, x\, xs_1.\ \mathbf{ens}\ xs{=}x :: xs_1; rev_1\ xs_1\ (\lambda r_1. k\ r_1; r.\ \mathbf{ret}\ x :: r)
\end{array}
$$

Fig. 4.10. Synthesised definition for $rev_1$

Interestingly, due to the use of reduction, both **control** and delimiter have been eliminated, resulting in a *continuation-composing* program that by construction (over)approximates $visit'$. This program is not strictly in CPS (as the continuation is not invoked in tail position, fixing its answer type to a list) and is not derivable through only the CPS transformation. However, it *is* the definition, and the intermediate step, that we need.

Now, it remains to prove the final subgoal:

$$rev_1\ xs\ k \sqsubseteq rev\ xs\ acc$$

Our next problem is that there is no relation between $k$ and $acc$; the induction step will thus give us no information about how $acc$ changes across it. We need another generalisation, guided by intuition as to what $acc$ is.

*acc* is the suffix of the reversed list that is extended as *rev* traverses the original list. The continuation of $rev_1$ builds a similar suffix as it descends, but in closures, with $k$ finally being applied to the empty list at the end. The generalisation is thus the following statement: given that the continuation is finally applied to the empty list, *acc* is everything that will be *prepended* to the empty list, i.e., the suffix.

$$rev_1\ xs\ \lambda y.\, acc +\!\!+ y \sqsubseteq rev\ xs\ acc$$

This statement can now be proved automatically.

**Base case** $rev\ [\,]\ (\lambda y.\, \mathbf{ret}\ acc +\!\!+ y) \sqsubseteq rev\ xs\ acc$

$$\begin{array}{lll} & rev\ [\,](\lambda y.\, \mathbf{ret}\ acc +\!\!+ y) & \text{Unfold} \\ \sqsubseteq & \mathbf{ens}\ xs{=}[\,]; \mathbf{ret}\ acc & \text{Simplify} \\ \sqsubseteq & rev\ xs\ acc \quad \square & \text{Fold} \end{array}$$

**Inductive case** $rev\ (x :: xs_1)\ (\lambda y.\, \mathbf{ret}\ acc +\!\!+ y) \sqsubseteq rev\ xs\ acc$

$$\begin{array}{lll} & rev\ x :: xs_1\ (\lambda y.\, \mathbf{ret}\ acc +\!\!+ y) & \\ \sqsubseteq & rev\ xs_1\ (\lambda y.\, \mathbf{ret}\ acc +\!\!+ y; r_1.\ \mathbf{ret}\ x :: r_1) & \text{Unfold} \\ \sqsubseteq & rev\ xs_1\ (\lambda y.\, \mathbf{ret}\ x :: (acc +\!\!+ y)) & \text{Simplify} \\ \sqsubseteq & rev\ xs_1\ (\lambda y.\, \mathbf{ret}\ (x :: acc) +\!\!+ y) & \text{Lem. cons, ++} \\ \sqsubseteq & rev\ (x :: xs_1)\ acc \quad \square & \text{Ind. hyp.} \end{array}$$

Taking a step back, *how* did the proof work? When we generalised $visit'$ over the continuation, lemma synthesis took us from a **control**-using program into a CPS program, which we then related to the final direct-style program via a second generalisation – a surprisingly systematic approach given its heuristic nature. Interestingly, it did not go through the standard CPS transformation, which for **control** is more involved [111].

Finally, given the proofs for both *copy* and *rev*, where does the difference in behaviour that we saw in Fig. 4.8 show up? Consider the steps *after* Reduction$^2$ and Reduction$^3$. The $\mathbf{ret}\ x :: r$ portion is the deferred operation that may or may not be accessed, depending on which control operator is used. In the case of Reduction$^2$, it is not accessed, reflected by how we can

simply rewrite (a delimited) *visit* on its own. In Reduction$^3$, however, we had to generalise to be able to talk about what follows *visit'*, *or what it accesses*. Moreover, this became part of the inductive argument, pointing to how *visit'* *repeatedly* accesses the deferred operation, to nest it deeper and reverse the list.

These subtleties illustrate the need for tools to support carrying out such reasoning. As Biernacki, Danvy, and Millikin [30] put it:

> In our earlier work [31], we argued that dynamic delimited continuations need examples, reasoning tools, and meaning-preserving program transformations, not only new variations, new formalizations, or new implementations.

This chapter takes a step towards such a reasoning tool, starting in the next section with the extensions to staged logic required.

## 4.2 Extensions to staged logic

We follow the technical development of Chapter 3, extending staged logic with delimited continuation operators. As before, we define their semantics directly, instead of encoding them as effects, for a simpler and more direct treatment.

$$
\begin{aligned}
\varphi &::= \ldots \mid \mathbf{shift}^c_n\ k.\ \varphi\ (\lambda x.\, \varphi_k) \mid \mathbf{control}^c_n\ k.\ \varphi\ (\lambda x.\, \varphi_k) \mid \langle \varphi \rangle \\
n &::= 0? \\
\mathbf{id} &\triangleq \lambda x.\, \mathbf{ret}\ x \\
\mathbf{shift}_n\ k.\ \varphi &\triangleq \mathbf{shift}^c_n\ k.\ \varphi\ \mathbf{id} \\
\mathbf{control}_n\ k.\ \varphi &\triangleq \mathbf{control}^c_n\ k.\ \varphi\ \mathbf{id}
\end{aligned}
$$

Fig. 4.11. Staged logic with delimited control operators

**Syntax** We add both **shift** and **control**, with both zero and nonzero variants, and a single delimiter (Fig. 4.11). We define the control operators in terms of more primitive constructors $\mathbf{shift}^c$ and $\mathbf{control}^c$, which carry an

additional argument for an *intermediate continuation*, which is the identity continuation **id** by default. This is similar to the continuation we used in the **match**# in Section 3.3, but here it is associated with the control operator rather than the delimiter. This allows us to perform reduction in a more intuitive "forward" manner, driven by the control operator rather than the delimiter. It is also possible to associate the continuation with the delimiter by giving it a value case, i.e., the *dollar* operator [142], and in that case we would take the same approach to reduction as in the previous chapter.

$\boxed{\mathcal{E}, h, h, R \vDash \varphi}$

$$R \ ::= \ v \mid \mathbf{shift}_n(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \mid \mathbf{control}_n(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k)$$

SSHIFT
$$\frac{R = \mathbf{shift}_n(\lambda k.\, \varphi_b, \mathbf{id})}{\mathcal{E}, h, h, R \vDash \mathbf{shift}_n\ k.\ \varphi_b}$$

SCONTROL
$$\frac{R = \mathbf{control}_n(\lambda k.\, \varphi_b, \mathbf{id})}{\mathcal{E}, h, h, R \vDash \mathbf{control}_n\ k.\ \varphi_b}$$

SBINDVAL
$$\frac{\mathcal{E}, h_1, h_3, v \vDash \varphi_1 \quad \mathcal{E}, h_3, h_2, R \vDash \varphi_2[v/r]}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\ \varphi_2}$$

SBINDSHIFT
$$\frac{\mathcal{E}, h_1, h_2, \mathbf{shift}_n(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \vDash \varphi_1 \quad R = \mathbf{shift}_n(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k; r.\ \varphi_2)}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\ \varphi_2}$$

SBINDCONTROL
$$\frac{\mathcal{E}, h_1, h_2, \mathbf{control}_n(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \vDash \varphi_1 \quad R = \mathbf{control}_n(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k; r.\ \varphi_2)}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\ \varphi_2}$$

SDELIMVAL
$$\frac{\mathcal{E}, h_1, h_2, v \vDash \varphi}{\mathcal{E}, h_1, h_2, v \vDash \langle \varphi \rangle}$$

SDELIMSHIFT
$$\frac{\mathcal{E}, h_1, h_3, \mathbf{shift}(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \vDash \varphi \quad \mathcal{E}[k := \lambda x.\, \langle \varphi_k \rangle], h_3, h_2, r \vDash \langle \varphi_b \rangle}{\mathcal{E}, h_1, h_2, R \vDash \langle \varphi \rangle}$$

SDELIMSHIFT0
$$\frac{\mathcal{E}, h_1, h_3, \mathbf{shift}_0(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \vDash \varphi \quad \mathcal{E}[k := \lambda x.\, \langle \varphi_k \rangle], h_3, h_2, r \vDash \varphi_b}{\mathcal{E}, h_1, h_2, R \vDash \langle \varphi \rangle}$$

SDELIMCONTROL
$$\frac{R_0 = \mathbf{control}(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \quad \mathcal{E}, h_1, h_3, R_0 \vDash \varphi \quad \mathcal{E}[k := \lambda x.\, \varphi_k], h_3, h_2, r \vDash \langle \varphi_b \rangle}{\mathcal{E}, h_1, h_2, R \vDash \langle \varphi \rangle}$$

SDELIMCONTROL0
$$\frac{R_0 = \mathbf{control}_0(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \quad \mathcal{E}, h_1, h_3, R_0 \vDash \varphi \quad \mathcal{E}[k := \lambda x.\, \varphi_k], h_3, h_2, r \vDash \varphi_b}{\mathcal{E}, h_1, h_2, R \vDash \langle \varphi \rangle}$$

Fig. 4.12. Semantics of control operators in staged logic, extending Fig. 3.7

**Semantics** Like for effects in Section 3.2, we use the "bubble-up" semantics "bubble-up semantics" of resumable exceptions proposed by Kiselyov [118].

Again, the construction of the delimited continuation is operationalised in it.

The semantics (Fig. 4.12) starts with a definition of outcomes $R$: values, or "unhandled" control operators; the latter directly correspond to **shift**$^c$ and **control**$^c$. The rules for them, SSHIFT and SCONTROL, serve as base cases. Unhandled control operators are technically ill-formed programs, but we treat them as we do effects, giving them a semantics. Practical systems often also make concessions here: for example, Racket uses an implicit top-level delimiter with an identity continuation [119].

SBINDVAL is the same as in Fig. 3.7. SBINDSHIFT and SBINDCONTROL extend the continuation, pushing the "current continuation" into the outcome. SDELIMVAL serves as a base case for when $\varphi$ is a value, in which case the delimiter is removed. SDELIMSHIFT applies when $\varphi$ reduces to a shift; reduction continues with the shift body, with the the continuation appropriately bound, and delimiters around both. The remaining rules are similar, directly implementing the semantics of each operator by varying where the delimiters are placed, as in Fig. 4.6.

## 4.3 Reduction

Control operators can be subject to the same semantics-directed *reduction* introduced in Section 2.3.2, and adapted to effects in Section 3.3. In this section, we focus first on **shift**, as it gives the strongest reasoning rules, noting along the way where other operators differ. We begin with a few auxiliary definitions.

**Definition 4.1** (Shift-free)**.** *$\varphi$ is shift-free, denoted $\varphi^{\#}$, iff it can never result in a shift outcome.*

$$\varphi^{\#} \triangleq \mathcal{E}_1, h_1, h_2, R \vDash \varphi \Rightarrow \neg\exists\, k\, \varphi_b\, x\, \varphi_k.\ R = \mathbf{\mathit{shift}}(\lambda k.\ \varphi_b, \lambda x.\ \varphi_k)$$

Shift-freedom can be proved compositionally (Fig. 4.13).

$\boxed{\varphi^{\#}}$

$$\textsc{SFEns}\ \frac{}{(\mathbf{ens}\ Q)^{\#}} \qquad \textsc{SFReq}\ \frac{\varphi^{\#}}{(\mathbf{req}\ H\ \varphi)^{\#}} \qquad \textsc{SFSeq}\ \frac{\varphi_1^{\#} \quad \varphi_2^{\#}}{(\varphi_1; \varphi_2)^{\#}} \qquad \textsc{SFReset}\ \frac{}{\langle\varphi\rangle^{\#}}$$

Fig. 4.13. Proving shift-freedom (select rules)

**ens** is always shift-free, **req** and sequencing are shift-free if their subexpressions are, and most importantly, a reset is always shift-free; unlike an effect handler, it is not possible for a shift to "escape" an enclosing reset. Intuitively, this is because the shift body is always placed under another reset. Crucially, this last rule is not true for the zero operators.

$\boxed{\varphi \rightsquigarrow \varphi}$

$$\textsc{RNormal}\ \frac{\varphi^{\#}}{\langle\varphi\rangle \rightsquigarrow \varphi} \qquad \textsc{RSkip}\ \frac{\varphi_1^{\#}}{\langle\varphi_1; \varphi_2\rangle \rightsquigarrow \varphi_1; \langle\varphi_2\rangle} \qquad \textsc{RInit}\ \frac{}{\mathbf{shift}\ k.\ \varphi_b \rightsquigarrow \mathbf{shift}^c\ k.\ \varphi_b\ \mathbf{id}}$$

$$\textsc{RExtend}\ \frac{}{(\mathbf{shift}^c\ k.\ \varphi_b\ \lambda x.\varphi_k); r.\ \varphi \rightsquigarrow \mathbf{shift}^c\ k.\ \varphi_b\ (\lambda x.\varphi_k; r.\ \varphi)} \qquad \textsc{RShiftElim}\ \frac{}{\langle\mathbf{shift}^c\ k.\ \varphi_b\ (\lambda x.\varphi_k)\rangle \rightsquigarrow \mathbf{ens}\ [k{=}\lambda x.\varphi_k]; \varphi_b}$$

Fig. 4.14. Shift/reset reduction

The reduction rules are given in Fig. 4.14. Again, the goal is to minimise the scopes of resets, trying to move things "out and to the left" if they are shift-free, before eliminating resets by matching against a shift in its body. RNormal serves as a base case: if $\varphi$ is shift-free, we can simply remove an enclosing reset. RSkip allows moving a shift-free prefix $\varphi_1$ out from under a reset. The next two rules work in tandem to determine the extent of the continuation. RInit begins the process, introducing a $\mathbf{shift}^c$ with an identity continuation; because of the definition of **shift**, this is just an unfolding. RExtend picks up after this: when iterated, it extends the continuation $\varphi_k$ with any $\varphi$ after it. When a $\mathbf{shift}^c$ is the only thing remaining, the delimited

continuation is completely determined. RSHIFTELIM then ends the process, eliminating the shift and replacing it with its body, and also defining the continuation for use inside it.

The rules are written as entailments, and are thus sound by construction.

## 4.4 Evaluation

To demonstrate the effectiveness and applicability of our proposed methodology, we extended HEIFER with support for the shift and reset operators, as well as the reduction rules described in Section 4.3.

We then used HEIFER to verify a suite of benchmark programs involving nontrivial uses of shift and reset, mostly taken from the literature on delimited continuations. As our focus is broadening the applicability of automated verification and scaling it to handle more language features, we selected small programs using language features which are difficult for existing automated verifiers to handle, such stateful higher-order functions and multishot continuations. To our knowledge, none of the programs have been verified automatically.

Table 4.1
Experimental results

| **Program** | **Desc.** | **LoC** | **LoS** | **Total(s)** | **SMT(s)** |
|---|---|---|---|---|---|
| State monad [10] | HO | 21 | 2 | 4.84 | 0.02 |
| Alice-Cat [142] | Multi | 3 | 1 | 0.33 | 0.02 |
| Printf [10] | ATM | 30 | 6 | 2.20 | 0.05 |
| Flip [61] | Multi | 9 | 1 | 3.89 | 0.04 |
| Toss [190] | Imp | 8 | 1 | 6.50 | 0.08 |
| HTTcc-inc3 [68] | Imp | 5 | 1 | 5.05 | 0.05 |
| Times [10] | Ind | 17 | 5 | 0.88 | 0.11 |
| Total | | 93 | 17 | 23.69 | 0.37 |

Table 4.1 lists the benchmark programs, followed by a list of the interesting features of each program: whether it is higher-order, involves multi-shot continuations, whether it uses imperative state, whether induction is needed,

and whether it uses answer-type modification (ATM), its size (lines of code and lines of specification), and finally the total time taken to verify the whole program and the amount of time spent in SMT solvers (both in seconds). All experiments were performed on Ubuntu 24.04.2 Linux using a 3201 MHz 6-Core AMD Ryzen 5 CPU with 16 GB of RAM. Z3 4.13.0 was used.

**Insights** These experiments are small in scale, so mainly serve to validate the claims that staged logic can feasibly automate the verification of programs containing delimited continuations. This is demonstrated in the lines of specifications required being significantly less than the LoC (with a spec-to-code ratio of 0.18), owing to the use of biabduction; often only the specification needs to be given, and an induction hypothesis (where a loop invariant would normally be required in other systems) is inferred based on the structure of the verification condition.

## 5.1 Landin's Knot

*Landin's Knot* [126] is a classic example of a higher-order, effectful program. It illustrates how the addition of state to a terminating language leads to the ability to write nonterminating programs, via cycles through the heap.

A faithful, semantic account of Landin's Knot is rather involved. Typing the heap into which mutable references point historically required a step-indexed, possible-worlds model [3]. A modern specification of the Knot uses impredicative invariants [33, Section 9].

To a reader, however, the Knot is not a very complex program: it is easy to see informally how the recursive behaviour arises, by mentally executing it a few steps until the cycle becomes apparent. In this section I present a simple, syntactic proof in staged logic that follows this pattern of reasoning.

First, some definitions.

$$\mathit{read}\ \ell \triangleq \forall v.\ \mathbf{req}\ \ell \mapsto v; \mathbf{ens}\ \ell \mapsto v; \mathbf{ret}\ v$$

$$\mathit{write}\ \ell\ v \triangleq \forall v_0.\ \mathbf{req}\ \ell \mapsto v_0; \mathbf{ens}\ \ell \mapsto v$$

$$\begin{array}{l}\mathit{landin\_rec}\ f \triangleq \\ \quad \exists\, \ell\ \mathit{knot}. \\ \qquad \mathbf{ens}\ \ell \mapsto (); \\ \qquad \mathbf{ens}\ \mathit{knot} = (\lambda n.\, \mathit{read}\ \ell; f_1.\ f\ \ f_1\ n); \\ \qquad \mathit{write}\ \ell\ \mathit{knot}; \\ \qquad \mathbf{ret}\ \mathit{knot}\end{array}$$

*landin_rec* uses the eponymous technique, presented as a fixed-point combinator. It allocates a reference $\ell$ (with a dummy initial value, ()) and writes a function value to it. This function in turn reads from $\ell$ and calls $f$ with whatever is there. We take this description of *landin_rec* as its specification and focus on how to verify a use of it. Following Birkedal and Bizjak [33],

we verify a recursive factorial function.

$$\begin{aligned} &\mathit{fact}_f\ \mathit{self}, n \triangleq \\ &\quad \mathbf{ens}\ r.\, n \le 0 \land r = 1 \\ &\quad \lor\ \mathbf{ens}\ n > 0; \mathit{self}\ (n-1); r_1.\ \mathbf{ret}\ n * r_1 \end{aligned}$$

$$\mathit{let}\ \mathit{fact}\ n \triangleq \mathit{if}\ n \le 0\ \mathit{then}\ 1\ \mathit{else}\ n * \mathit{fact}\ (n-1)$$

We give an open-recursive version $\mathit{fact}_f$ in staged logic, as well as a pure, mathematical version *fact* to serve as a specification.

We then prove the following entailment, which uses *landin_rec* to tie a knot before invoking it on $n$.

$$\begin{aligned} &\mathit{landin\_rec}\ \mathit{fact}_f; f.\ f\ \ n \\ \sqsubseteq\ &\exists \ell v.\ \mathbf{ens}\ \ell \mapsto v; \mathbf{ret}\ (\mathit{fact}\ n) \end{aligned}$$

The fact that a reference is allocated in the process of tying the knot is intentionally exposed. We quantify away the exact value, so it doesn't otherwise affect the proof.

The idea of the proof is to symbolically execute the knot until we surface how it is tied, at which point we can proceed by induction on $n$. This example has been mechanically verified[1] against the simple model of Section 6.1.

*Proof.* We start with the left side of the entailment, and unfold and simplify it until we have a direct use of the knot.

$$\begin{aligned} &\mathit{landin\_rec}\ \mathit{fact}_f; f.\ f\ \ n \\ \sqsubseteq\ &\exists \ell\ \mathit{knot}.\ \mathbf{ens}\ \ell \mapsto (); \mathbf{ens}\ \mathit{knot} = (\lambda n.\, \mathit{read}\ \ell; f_1.\ f\ \ f_1\ n); \\ &\quad \mathit{write}\ \ell\ \ \mathit{knot}; \mathbf{ret}\ \mathit{knot}; f.\ f\ \ n \\ \sqsubseteq\ &\exists \ell\ \mathit{knot}.\ \mathbf{ens}\ \mathit{knot} = (\lambda n.\, \mathit{read}\ \ell; f_1.\ f\ \ f_1\ n); \cancel{\mathbf{ens}\ \ell \mapsto ()}; \\ &\quad \cancel{\forall v_0.\ \mathbf{req}\ \ell \mapsto v_0}; \mathbf{ens}\ \ell \mapsto \mathit{knot}; \mathbf{ret}\ \mathit{knot}; f.\ f\ \ n \\ \sqsubseteq\ &\exists \ell\ \mathit{knot}.\ \mathbf{ens}\ \mathit{knot} = (\lambda n.\, \mathit{read}\ \ell; f_1.\ f\ \ f_1\ n); \mathbf{ens}\ \ell \mapsto \mathit{knot}; \mathbf{ret}\ \mathit{knot}; f.\ f\ \ n \\ \sqsubseteq\ &\exists \ell\ \mathit{knot}.\ \mathbf{ens}\ \mathit{knot} = (\lambda n.\, \mathit{read}\ \ell; f_1.\ f\ \ f_1\ n); \mathbf{ens}\ \ell \mapsto \mathit{knot}; \mathit{knot}\ n \end{aligned}$$

We then continue by induction on $n$.

[1] https://github.com/dariusf/staged/blob/master/shiftreset/more_examples/Landin.v

**Base case** $\quad \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; knot\ 0$
$\sqsubseteq \mathbf{ens}\ \ell \mapsto knot; \mathbf{ret}\ (fact\ 0)$

$\quad \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; knot\ 0$
$\sqsubseteq \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; read\ \ell; f_1.\, fact_f\ f_1\ 0$
$\sqsubseteq \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; fact_f\ knot\ 0$
$\sqsubseteq \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; \mathbf{ens}\ r.\, n \leq 0 \wedge r = 1$
$\sqsubseteq \mathbf{ens}\ \ell \mapsto knot; \mathbf{ret}\ (fact\ 0)$ $\square$

**Inductive case** $\quad \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; knot\ (n_0+1)$
$\sqsubseteq \mathbf{ens}\ \ell \mapsto knot; \mathbf{ret}\ (fact\ (n_0+1))$

$\quad \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; knot\ (n_0+1)$
$\sqsubseteq \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot; fact_f\ knot\ (n_0+1)$
$\sqsubseteq \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot;$
$\quad \mathbf{ens}\ n_0+1>0; knot\ ((n_0+1)-1); r_1.\ \mathbf{ret}\ (n_0+1)*r_1$
$\sqsubseteq \mathbf{ens}\ n_0+1>0;$
$\quad \mathbf{ens}\ knot = (\lambda n.\, read\ \ell; f_1.\ f\ f_1\ n); \mathbf{ens}\ \ell \mapsto knot;$
$\quad knot\ n_0; r_1.\ \mathbf{ret}\ (n_0+1)*r_1$
$\sqsubseteq \mathbf{ens}\ n_0+1>0; \mathbf{ens}\ \ell \mapsto knot; \mathbf{ret}\ fact\ n_0; r_1.\ \mathbf{ret}\ ((n_0+1)*r_1)$
$\sqsubseteq \mathbf{ens}\ n_0+1>0; \mathbf{ens}\ \ell \mapsto knot; \mathbf{ret}\ ((n_0+1)*fact\ n_0)$
$\sqsubseteq \mathbf{ens}\ n_0+1>0; \mathbf{ens}\ \ell \mapsto knot; fact\ (n_0+1)$
$\sqsubseteq \mathbf{ens}\ \ell \mapsto knot; fact\ (n_0+1)$ $\square$

$\square$

**Why is the proof so simple?** The crux of the proof is that we need a specification for the contents of the location, as the program dereferences it and runs whatever is there. We refer to this as the location's *invariant*, since it holds across the execution of the program. The proof given here differs from the one in Birkedal and Bizjak [33, Section 9] in how this invariant is formulated.

Here, the heap stores the *name* of a function, which the specification environment associates upfront with a *syntactic representation* of a specification, to be interpreted on later use. Therefore:

1. The location's invariant is first-order: it says only that the location stores a name (which points into the specification environment).

2. Reading the location thus gives us a usable specification immediately, because behaviour and specification are unified in staged logic.

3. There is recursion in this specification (as it dereferences the location, thus referring to itself), but this does not cause problems, as it is represented as syntax and unfolded only when the function is applied.

4. This can be seen as *defunctionalization*, as the set of functions stored in the heap is finite and statically known, and available syntactically. The interpretation through the specification environment is analogous to the dispatch function, and the name stored in the location plays the role of the tag.

In contrast, in Birkedal and Bizjak, the heap stores a *function*. Therefore:

1. The location's invariant is *higher-order*: it constrains the *behaviour* of the stored function and is self-referential.

2. Reading the location gives us a function. To use it in the proof, we need a specification/invariant, as (program) behaviour and specifications are distinct in Iris' program logic. This invariant is nontrivial to state, as it is self-referential: the function reads the location and calls whatever is there, which happens to be the function itself.

3. The recursion is now in the self-reference in the *definition* of the invariant, which is not well-founded as stated.

4. The fix is *step-indexing*: the function only needs to satisfy its specification for a smaller number of future steps. This makes the definition of the invariant well-founded. However, significantly more machinery is required, as mentioned at the beginning of this section.

## 5.2 Interpreting regular expressions

In this section, we verify a series of interpreters for a simple language of regular expressions, defined in OCaml as follows.

```
type regex =
  | Emp
  | Atom of int
  | Seq of regex * regex
  | Disj of regex * regex
```

The regular expression 1|(23) would be represented as `Disj (Atom 1) (Seq (Atom 2) (Atom 3))`. The strings matching regular expressions are assumed to be lists of integers. Emp matches the empty string, an atom matches a string consisting of a single integer, and regular expressions may be sequenced and may use disjunction.

```
let rec matches_prefix r ns : int list option =
  match r with
  | Emp -> Some ns
  | Atom a ->
    (match ns with
    | b :: ns1 when a = b -> Some ns1
    | _ -> None)
  | Seq (r1, r2) ->
    (match matches_prefix r1 ns with
    | None -> None
    | Some rest -> matches_prefix r2 rest)
  | Disj (r1, r2) ->
    (match matches_prefix r1 ns with (* broken! *)
    | none => matches_prefix r2 ns
    | some r => some r)

let matches r ns : bool =
  match matches_prefix r ns with
  | Some [] -> true
  | _ -> false
```

Fig. 5.1. A naive (and incorrect) regular expression interpreter

A first attempt at an interpreter for it is shown in Fig. 5.1. `matches` is defined in terms of an auxiliary function, `matches_prefix`, which evaluates a

regular expression `r` against a string of numbers `ns`. Its result is the suffix of the input remaining after removing the numbers satisfying `r`, or `None` if there is no match. `matches` then simply checks if the entire string is matched, i.e., if there is a match and the string remaining is empty.

The first three cases are easy to define. `Emp` always matches and leaves the string changed. `Atom a` matches only if the input string is nonempty and the first element is exactly `a`. `Seq` matches if the string remaining after matching `r1` can be used to match `r2`, and the result is the string remaining afterward.

The case for disjunction is trickier. Consider how we would interpret the example from earlier, `Disj (Atom 1) (Seq (Atom 2) (Atom 3))`. At the top-level `Disj` we can return two possible suffixes, depending on which disjunct will eventually lead to a successful match. However, the present return type does not provide a way to do this. Doing something naive, as in Fig. 5.1, renders the matcher incorrect: consider the input list `[1; 3]`, which will not be matched. The reason is that, at the failure in the first `Seq` argument, we have no way to backtrack to the disjunction and "try again" from there.

A simple solution is to generalise the type of the function to `(int list) list`, where we produce every possible choice, aka a list monad. We consider instead two continuation-based solutions.

`matches_prefix_cps` (Fig. 5.2) generalises `maches_prefix` so that it takes a continuation `k`, with answer type as annotated.[2] The first three cases now use the continuation to produce a result; the case for `Seq` uses continuations to receive intermediate results, with `k` getting only the final result. Disjunction can now be implemented by using the continuation twice, and examining the result of matching `r1` for failure (making the interpreter continuation-composing).

[2]The option type in the codomain of the function is technically unnecessary; it is possible to instead represent the failure to match implicitly, as the *absence* of an invocation of the continuation [62]. This interpreter is used here as it is structurally more similar to the next one, making the presentation slightly simpler. Its equvialence to the one in Danvy and Nielsen [62] is proved formally later.

```
let rec matches_prefix_cps r ns k : int list option =
  match r with
  | Emp -> k (Some ns)
  | Atom a ->
    k (match ns with
       | b :: ns1 when a = b -> Some ns1
       | _ -> None)
  | Seq (r1, r2) ->
    matches_prefix_cps r1 ns (function
      | None -> None
      | Some rest -> matches_prefix_cps r2 rest k)
  | Disj (r1, r2) ->
    (match matches_prefix_cps r1 ns k with
    | None -> matches_prefix_cps r2 ns k
    | r -> r)

let matches_cps r ns : bool =
  match matches_prefix_cps r ns (fun x -> x) with
  | Some [] -> true
  | _ -> false
```

Fig. 5.2. A continuation-composing regular expression interpreter

A solution using shift/reset is shown in Fig. 5.3. The beauty of it is that it all the cases which do not rely on manipulating continuations are identical to the version in Fig. 5.1. However, in this version, disjunction can be implemented without any other changes, by using shift to access the continuation of that branch, peek into its outcome, and "restart" execution with the second disjunct if the first leads to failure. The only other change is the delimiter around the scrutinee of the match in `matches_sr`.

**Verification** Let us consider what it means to have a correct matcher. First, we define the denotation of a regular expression.

**Definition 5.1** (Denotation of a regex)**.** $[\![r]\!]$ *denotes the set of strings that* $r$

```
let rec matches_prefix_sr r ns =
  match r with
  | Emp -> Some ns
  | Atom a ->
    (match ns with
    | b :: ns1 when a = b -> Some ns1
    | _ -> None)
  | Seq (r1, r2) ->
    let v = matches_prefix_sr r1 ns in
    (match v with
    | None -> None
    | Some rest -> matches_prefix_sr r2 rest)
  | Disj (r1, r2) ->
    shift (fun k ->
      match k (matches_prefix_sr r1 ns) with
      | None -> k (matches_prefix_sr r2 ns)
      | v -> v)

let matches_sr r ns : bool =
  match reset (matches_prefix_sr r ns) with
  | Some [] -> true
  | _ -> false
```

Fig. 5.3. A regular expression interpreter using shift/reset [61]

*matches. It is defined inductively as follows:*

$$\begin{aligned}
[\![Emp]\!] &= \{[\,]\} \\
[\![Atom\ a]\!] &= \{[a]\} \\
[\![Seq(r_1, r_2)]\!] &= \{ns_1 +\!\!+ ns_2 \mid ns_1 \in [\![r_1]\!],\ ns_2 \in [\![r_2]\!]\} \\
[\![Disj(r_1, r_2)]\!] &= [\![r_1]\!] \cup [\![r_2]\!]
\end{aligned}$$

**Definition 5.2** (Correct matcher)**.** *A correct matcher $M$ returns a true result iff $ns$ is in the denotation of $r$, i.e. $M\ r\ ns = true \iff ns \in [\![r]\!]$.*

A correct matcher satisfies a number of properties: sequencing is associative and distributes over disjunction, disjunction is commutative, etc.

$$M\ Emp\ ns = \top \iff ns = [\,] \qquad M\ (Atom\ a)\ ns = \top \iff ns = [a]$$

$$M\ (Seq(r_1, (Seq(r_2, r_3))))\ ns = M\ Seq((Seq(r_1, r_2)), r_3)\ ns$$

$$M\ (Seq(Emp, r))\ ns = M\ r\ ns \qquad M\ (Seq(r, Emp))\ ns = M\ r\ ns$$

$$M\ (Disj(r, r))\ ns = M\ r\ ns$$

$$M\ (Seq(Disj(r_1, r_2), r))\ ns = M\ (Disj(Seq(r_1, r), Seq(r_2, r)))\ ns$$

These all follow from the denotation. Thus, given a matcher, it suffices to show that it is correct (i.e., prove Definition 5.2) to be able to benefit from these properties. I take this approach for the continuation-composing interpreter (Fig. 5.2) and a number of others [62].[3]

Relating an interpreter directly to its denotation can, however, be tedious and involved. For example, the continuing-composing interpreter requires coming up with the lemmas in Fig. 5.4 to describe its behaviour at specific arguments and continuations. The proofs of both require induction.

An often simpler approach is to take advantage of the transitivity of refinement and, given an interpreter known to be correct, prove that a new interpreter refines it, and therefore refines the denotation. This works especially well when the programs are structurally similar.

Since the final interpreter uses shift and reset, we can no longer treat it as a mathematical function. Hence, we take the refinement approach in staged

[3] https://github.com/dariusf/regex-interpreter

$$\frac{k\ None = None \qquad matches\_prefix\_cps\ r\ ns\ k = Some\ rs}{\exists\, ns_1\ ns_2.\ ns = ns_1 +\!\!+ ns_2 \wedge ns_1 \in [\![ r ]\!] \wedge k\ (Some\ ns_2) = Some\ rs}$$

$$\frac{k\ None = None \qquad \forall\, x\ rs.\ k\ x = Some\ rs \Rightarrow rs = [\,]}{\begin{array}{l} matches\_prefix\_cps\ r\ ns\ k = Some\ [\,] \Longleftrightarrow \\ \exists\, ns_1\ ns_2.\ ns = ns_1 +\!\!+ ns_2 \wedge ns_1 \in [\![ r ]\!] \wedge k\ (Some\ ns_2) = Some\ [\,] \end{array}}$$

Fig. 5.4. Lemmas required to prove the correctness of Fig. 5.2

logic.

The most structurally-similar interpreter is the continuation-composing one in Fig. 5.2, so we relate the last interpreter it using the lemma in Fig. 5.5. In this way, *matches_prefix_cps* serves as a specification of *matches_prefix_sr*. We generalise over the (shift-free) continuation of the reset in order to relate it to the explicit continuation, a technique first used in Section 4.1.3.

$$\frac{k^{\#}}{\langle matches\_prefix\_sr\ r\ ns; r_1.\ k\ r_1 \rangle \sqsubseteq matches\_prefix\_cps\ r\ ns\ k}$$

Fig. 5.5. Relating the interpreters fo Fig. 5.3 and Fig. 5.2

We first model both programs in staged logic, in Fig. 5.6. We then prove the following entailment.

$$\begin{array}{l} \langle matches\_prefix\_sr\ r\ ns \rangle; r_1.\ k\ r_1 \\ \sqsubseteq matches\_prefix\_cps\ r\ ns\ k \end{array}$$

The proof goes by induction on $r$. We give a detailed account of the proof below, but it is otherwise routine. The point is that it is simplified by the use of staged logic, as there is no need to invent abstractions for the uses of shift.

**Case** $r{=}Emp$ $\quad \langle \mathbf{ens}\ r{=}Emp; \mathbf{ret}\ Some\ ns; r_1.\ k\ r_1 \rangle$
$\sqsubseteq \mathbf{ens}\ r{=}Emp; k\ (Some\ ns)$

$$\begin{array}{lll} & \langle \mathbf{ens}\ r{=}Emp; \mathbf{ret}\ Some\ ns; r_1.\ k\ r_1 \rangle & \\ \sqsubseteq & \langle \mathbf{ens}\ r{=}Emp; k\ (Some\ ns) \rangle & \text{Simplify} \\ \sqsubseteq & \mathbf{ens}\ r{=}Emp; k\ (Some\ ns)\ \square & \text{Pure} \end{array}$$

$$
\begin{array}{l}
\mathit{matches\_prefix\_sr}\ r, ns \triangleq \\
\quad\quad \mathbf{ens}\ r{=}\mathit{Emp}; \mathbf{ret}\ \mathit{Some}\ ns \\
\quad \vee \exists a.\ \mathbf{ens}\ r{=}(\mathit{Atom}\ a); \\
\quad\quad (\exists\, b\, \mathit{ns}_1.\ \mathbf{ens}\ \mathit{ns}{=}b::\mathit{ns}_1 \wedge a{=}b; \mathbf{ret}\ (\mathit{Some}\ \mathit{ns}_1) \vee \mathbf{ret}\ \mathit{None}) \\
\quad \vee \exists\, r_1\, r_2.\ \mathbf{ens}\ r{=}(\mathit{Seq}(r_1, r_2)); \\
\quad\quad \mathit{matches\_prefix\_sr}\ r_1\ \mathit{ns}; v. \\
\quad\quad (\mathbf{ens}\ v{=}\mathit{None}; \mathbf{ret}\ \mathit{None} \vee \\
\quad\quad\quad (\exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \mathit{matches\_prefix\_sr}\ r_2\ \mathit{rest})) \\
\quad \vee \exists\, r_1\, r_2.\ \mathbf{ens}\ r{=}(\mathit{Disj}(r_1, r_2)); \\
\quad\quad \mathbf{shift}\ k.\ \mathit{matches\_prefix\_sr}\ r_1\ \mathit{ns}; v_1.\ k\ v1; r_3. \\
\quad\quad\quad (\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_sr}\ r_2\ \mathit{ns}; a.\ k\ a) \vee \\
\quad\quad\quad\quad (\exists a.\ \mathbf{ens}\ r_3{=}(\mathit{Some}\ a); \mathbf{ret}\ r_3)
\end{array}
$$

$$
\begin{array}{l}
\mathit{matches\_prefix\_cps}\ r, ns, k \triangleq \\
\quad\quad \mathbf{ens}\ r{=}\mathit{Emp}; k\ \mathit{Some}\ ns \\
\quad \vee \exists a.\ \mathbf{ens}\ r{=}(\mathit{Atom}\ a); \\
\quad\quad (\exists\, b\, \mathit{ns}_1.\ \mathbf{ens}\ \mathit{ns}{=}b::\mathit{ns}_1 \wedge a{=}b; k\ \mathit{Some}\ \mathit{ns}_1 \vee k\ \mathit{None}) \\
\quad \vee \exists\, r_1\, r_2.\ \mathbf{ens}\ r{=}(\mathit{Seq}(r_1, r_2)); \\
\quad\quad \mathit{matches\_prefix\_cps}\ r_1 ns(\lambda v. \\
\quad\quad (\mathbf{ens}\ v{=}\mathit{None}; k\ \mathit{None} \vee \\
\quad\quad\quad (\exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \mathit{matches\_prefix\_cps}\ r_2\ \mathit{rest}\ k))) \\
\quad \vee \exists\, r_1\, r_2.\ \mathbf{ens}\ r{=}(\mathit{Disj}(r_1, r_2)); \\
\quad\quad \mathit{matches\_prefix\_cps}\ r_1\ \mathit{ns}\ k; r_3. \\
\quad\quad\quad (\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_cps}\ r_2\ \mathit{ns}\ k \vee r_3)
\end{array}
$$

Fig. 5.6. *matches_prefix_cps* and *matches_prefix_sr* in staged logic

**Case** $r{=}Atom$ ...
$$\begin{aligned}
&\langle \exists\, a.\ \textbf{ens}\ r{=}Atom\ a;\\
&\quad ((\exists\, b\, ns_1.\ \textbf{ens}\ ns{=}b{::}ns_1 \wedge a{=}b; \textbf{ret}\ Some\ ns_1) \vee\\
&\qquad \textbf{ret}\ None)\rangle; r_1.\ k\ r_1\\
\sqsubseteq\ &\exists\, a.\ \textbf{ens}\ r{=}Atom\ a;\\
&\quad \exists\, b\, ns_1.\ \textbf{ens}\ ns{=}b{::}ns_1 \wedge a{=}b; k\ (Some\ ns_1) \vee\\
&\qquad k\ None
\end{aligned}$$

We eliminate the existentials and **ens** proof obligations, and match the disjuncts. This pattern is followed in the other cases.

$$\begin{aligned}
&\langle \textbf{ret}\ (Some\ ns_1) \vee \textbf{ret}\ None; r_1.\ k\ r_1\rangle\\
\sqsubseteq\ &k\ (Some\ ns) \vee k\ None
\end{aligned}$$

The goal then follows from simplification and the fact that $k$ is shift-free.

$$\begin{aligned}
&\langle \textbf{ret}\ (Some\ ns_1) \vee \textbf{ret}\ None; r_1.\ k\ r_1\rangle && \\
\sqsubseteq\ &\langle k\ (Some\ ns_1) \vee k\ None\rangle && \text{Simplify}\\
\sqsubseteq\ &k\ (Some\ ns) \vee k\ None \quad \square && \text{Pure}
\end{aligned}$$

**Case** $r{=}Seq$ ...
$$\begin{aligned}
&\langle \exists\, r_1\, r_2.\ \textbf{ens}\ r{=}Seq(r_1, r_2);\\
&\quad matches_prefix_sr\ r_1\ ns; v.\ (\textbf{ens}\ v{=}None; \textbf{ret}\ None \vee\\
&\quad (\exists\, rest.\ \textbf{ens}\ v{=}(Some\ rest);\\
&\qquad matches_prefix_sr\ r_2\ rest)); r_3.\ k\ r_3\rangle\\
\sqsubseteq\ &\exists\, r_1\, r_2.\ \textbf{ens}\ r{=}Seq(r_1, r_2);\\
&\quad matches_prefix_cps\ r_1\ ns\ (\lambda v.\textbf{ens}\ v{=}None; k\ None \vee\\
&\quad (\exists\, rest.\ \textbf{ens}\ v{=}(Some\ rest);\\
&\qquad matches_prefix_cps\ r_2\ rest\ k))
\end{aligned}$$

The induction hypotheses are

$$\forall\, ns\, k.\ \langle matches\_prefix\_sr\ r_i\ ns; r_3.\ k\ r_3\rangle \sqsubseteq matches\_prefix\_cps\ r_i\ ns\ k$$

for $i \in \{1, 2\}$, assuming $k$ is shift-free.

We first eliminate the existentials and cancel **ens** on both sides.

$$\begin{aligned}
&\langle matches_prefix_sr\ r_1\ ns; v.\ (\textbf{ens}\ v{=}None; \textbf{ret}\ None \vee\\
&\quad (\exists\, rest.\ \textbf{ens}\ v{=}(Some\ rest);\\
&\qquad matches_prefix_sr\ r_2\ rest)); r_3.\ k\ r_3\rangle\\
\sqsubseteq\ &matches_prefix_cps\ r_1\ ns\ (\lambda v.\textbf{ens}\ v{=}None; k\ None \vee\\
&\quad (\exists\, rest.\ \textbf{ens}\ v{=}(Some\ rest);\\
&\qquad matches_prefix_cps\ r_2\ rest\ k))
\end{aligned}$$

We need the lemma [110, Section 3.2]

**Lemma 5.1** (Reset under bind). $\langle \varphi_1; r.\ \varphi_2 \rangle \equiv \langle \varphi_1; r.\ \langle \varphi_2 \rangle \rangle$

We then successively simplify the left side.

$$
\begin{aligned}
&\langle \mathit{matches\_prefix\_sr}\ r_1\ \mathit{ns}; v.\ (\mathbf{ens}\ v{=}\mathit{None}; \mathbf{ret}\ \mathit{None} \lor \\
&\quad (\exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \\
&\quad \mathit{matches\_prefix\_sr}\ r_2\ \mathit{rest})); r_3.\ k\ r_3 \rangle \\
\sqsubseteq\ &\langle \mathit{matches\_prefix\_sr}\ r_1\ \mathit{ns}; v.\ \langle (\mathbf{ens}\ v{=}\mathit{None}; \mathbf{ret}\ \mathit{None} \lor \\
&\quad (\exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \\
&\quad \mathit{matches\_prefix\_sr}\ r_2\ \mathit{rest})); r_3.\ k\ r_3 \rangle \rangle && \text{Lemma 5.1} \\
\sqsubseteq\ &\mathit{matches\_prefix\_cps}\ r_1\ \mathit{ns}\ (\lambda v. \langle (\mathbf{ens}\ v{=}\mathit{None}; \mathbf{ret}\ \mathit{None} \lor \\
&\quad (\exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \\
&\quad \mathit{matches\_prefix\_sr}\ r_2\ \mathit{rest})); r_3.\ k\ r_3 \rangle) && \text{Ind. hyp. 1} \\
\sqsubseteq\ &\mathit{matches\_prefix\_cps}\ r_1\ \mathit{ns}\ (\lambda v. \\
&\quad (\langle \mathbf{ens}\ v{=}\mathit{None}; \mathbf{ret}\ \mathit{None}; r_3.\ k\ r_3 \rangle \lor \\
&\quad \langle \exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \\
&\quad \mathit{matches\_prefix\_sr}\ r_2\ \mathit{rest}; r_3.\ k\ r_3 \rangle))
\end{aligned}
$$

The application of the induction hypothesis requires us to prove that what we instantiate $k$ is with shift-free, which it is due to the reset. The last step follows from distributing disjunction over sequencing, and reset over disjunction.

Because $\sqsubseteq$ is a precongruence with respect to functions, it suffices to prove the following two entailments.

$$
\begin{aligned}
&\langle \mathbf{ens}\ v{=}\mathit{None}; \mathbf{ret}\ \mathit{None}; r_3.\ k\ r_3 \rangle \\
\sqsubseteq\ &\mathbf{ens}\ v{=}\mathit{None}; k\ \mathit{None}
\end{aligned}
$$

This can be proved by simplification.

$$
\begin{aligned}
&\langle \exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \mathit{matches\_prefix\_sr}\ r_2\ \mathit{rest}; r_3.\ k\ r_3 \rangle \\
\sqsubseteq\ &\exists \mathit{rest}.\ \mathbf{ens}\ v{=}(\mathit{Some}\ \mathit{rest}); \mathit{matches\_prefix\_cps}\ r_2\ \mathit{rest}\ k
\end{aligned}
$$

This is exactly induction hypothesis 2 after simplification, and $k$ is shift-free. □

**Case** $r{=}\mathit{Disj}$ ...

$$\begin{aligned}
&\langle \exists\, r_1\, r_2.\ \mathbf{ens}\ r{=}\mathit{Disj}(r_1, r_2); \mathbf{shift}\ k_1.\\
&\quad \mathit{matches_prefix_sr}\ r_1\ \mathit{ns}; v_1.\ k_1(v_1); r_3.\\
&\quad\quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_sr}\ r_2\ \mathit{ns}; a.\ k_1(a))\\
&\quad\quad \vee (\exists\, a.\ \mathbf{ens}\ r_3{=}(\mathit{Some}\ a); \mathbf{ret}\ r_3)); r_4.\ k\ r_4\rangle\\
&\sqsubseteq \exists\, r_1\, r_2.\ \mathbf{ens}\ r{=}\mathit{Disj}(r_1, r_2); \mathit{matches_prefix_cps}\ r_1\ \mathit{ns}\ k; r_3.\\
&\quad (\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_cps}\ r_2\ \mathit{ns}\ k \vee r_3)
\end{aligned}$$

The induction hypotheses are

$$\forall\, \mathit{ns}\, k.\ \langle \mathit{matches\_prefix\_sr}\ r_i\ \mathit{ns}; r_3.\ k\ r_3\rangle \sqsubseteq \mathit{matches\_prefix\_cps}\ r_i\ \mathit{ns}\ k$$

for $i \in \{1, 2\}$, assuming $k$ is shift-free.

First we eliminate existentials and proof obligations.

$$\begin{aligned}
&\langle \mathbf{shift}\ k_1.\\
&\quad \mathit{matches_prefix_sr}\ r_1\ \mathit{ns}; v_1.\ k_1(v_1); r_3.\\
&\quad\quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_sr}\ r_2\ \mathit{ns}; a.\ k_1\ a)\\
&\quad\quad \vee (\exists\, a.\ \mathbf{ens}\ r_3{=}(\mathit{Some}\ a); \mathbf{ret}\ r_3)); r_4.\ k\ r_4\rangle\\
&\sqsubseteq \mathit{matches_prefix_cps}\ r_1\ \mathit{ns}\ k; r_3.\\
&\quad (\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_cps}\ r_2\ \mathit{ns}\ k \vee r_3)
\end{aligned}$$

Given the shift immediately inside the reset, we can perform shift/reset reduction. We then delimit the bind body with Lemma 5.1 and apply the induction hypothesis, as in the previous case.

$$\begin{aligned}
&\langle \mathbf{shift}\ k_1.\\
&\quad \mathit{matches_prefix_sr}\ r_1\ \mathit{ns}; v_1.\ k_1\ v_1; r_3.\\
&\quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_sr}\ r_2\ \mathit{ns}; a.\ k_1\ a)\\
&\quad \vee (\exists\, a.\ \mathbf{ens}\ r_3{=}(\mathit{Some}\ a); \mathbf{ret}\ r_3)); r_4.\ k\ r_4\rangle && \text{SR reduction}\\
&\sqsubseteq \langle \mathit{matches_prefix_sr}\ r_1\ \mathit{ns}; v_1.\ \langle k\ v_1\rangle; r_3.\\
&\quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_sr}\ r_2\ \mathit{ns}; a.\ \langle k\ a\rangle)\\
&\quad \vee (\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3))\rangle && \text{Lemma 5.1}\\
&\sqsubseteq \langle \mathit{matches_prefix_sr}\ r_1\ \mathit{ns}; v_1.\ \langle\langle k\ v_1\rangle; r_3.\\
&\quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_sr}\ r_2\ \mathit{ns}; a.\ \langle k\ a\rangle)\\
&\quad \vee (\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3))\rangle\rangle && \text{Ind. hyp. 1}\\
&\sqsubseteq \mathit{matches_prefix_cps}\ r_1\ \mathit{ns}\ (\lambda v_1.\\
&\quad \langle\langle k\ v_1\rangle; r_3.\\
&\quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches_prefix_sr}\ r_2\ \mathit{ns}; a.\ \langle k\ a\rangle)\\
&\quad \vee (\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3))\rangle)
\end{aligned}$$

The application of the induction hypothesis requires the continuation we instantiate $k_1$ with to be shift-free, which it is due to the reset. To match the

right side, we factor out a part of the continuation using the Lemma 5.2.

**Lemma 5.2** (*matches_prefix_cps* is continuation-composing)**.**

$$\begin{aligned} & \mathit{matches\_prefix\_cps}\ r\ ns\ (\lambda v.\, k\ v; r'.\ f\ \ r') \\ \sqsubseteq\ & \mathit{matches\_prefix\_cps}\ r\ ns\ k; r'.\ f\ \ r' \end{aligned}$$

This is a property of *matches_prefix_cps* itself and is proved by induction on $r$: every recursive call threads $k$ linearly in tail position, so the trailing computation $f$ may be pulled out of the continuation and run after.

Using the fact that $\sqsubseteq$ is a precongruence, it then suffices to prove the following.

$$\begin{aligned} & \langle\langle k\ v_1\rangle; r_3. \\ & \quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_sr}\ r_2\ ns; a.\ \langle k\ a\rangle) \\ & \quad \vee\ (\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3))\rangle \\ \sqsubseteq\ & k\ v_1; r_3. \\ & \quad (\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_cps}\ r_2\ ns\ k \vee r_3) \end{aligned}$$

We simplify the left side, using the fact that $k$ is shift-free.

$$\begin{aligned} & \langle\langle k\ v_1\rangle; r_3. \\ & \quad ((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_sr}\ r_2\ ns; a.\ \langle k\ a\rangle) \\ & \quad \vee\ (\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3))\rangle && \text{SR reduction} \\ \sqsubseteq\ & k\ v_1; r_3. \\ & \quad \langle((\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_sr}\ r_2\ ns; a.\ \langle k\ a\rangle) \\ & \quad \vee\ (\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3))\rangle \end{aligned}$$

By precongruence, it remains to relate the continuation bodies. Distributing reset over the disjunction and matching disjuncts,

$$\begin{aligned} & \langle\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_sr}\ r_2\ ns; a.\ \langle k\ a\rangle\rangle \\ & \quad \vee\ \langle\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3\rangle \\ \sqsubseteq\ & (\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_cps}\ r_2\ ns\ k) \vee r_3 \end{aligned}$$

the second is immediate.

$$\langle\exists\, a.\ \mathbf{ens}\ r_3{=}\mathit{Some}\ a; \mathbf{ret}\ r_3\rangle \sqsubseteq r_3$$

In the first,

$$\begin{aligned} & \langle\mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_sr}\ r_2\ ns; a.\ \langle k\ a\rangle\rangle \\ \sqsubseteq\ & \mathbf{ens}\ r_3{=}\mathit{None}; \mathit{matches\_prefix\_cps}\ r_2\ ns\ k \end{aligned}$$

cancelling **ens** on both sides,

$$\begin{aligned} &\langle \mathit{matches\_prefix\_sr}\ r_2\ ns; a.\ \langle k\ a\rangle\rangle \\ \sqsubseteq\ &\mathit{matches\_prefix\_cps}\ r_2\ ns\ k \end{aligned}$$

$k$ is shift-free, so we finish with induction hypothesis 2. □

## 5.3 Relational verification

In this section, I explore encodings of various relational verification problems in staged logic. This amounts to reducing them to refinement, itself a relational property.

I consider two such problems: program equivalence over fused loops, which requires aligning them, and the relation between stateful *foldl* and *foldr*, which requires hyperproperties like commutativity and associativity.

### 5.3.1 Loop fusion and alignment

```
int doubleSquare1(int x)
{
  int y = 0;
  int z = 2 * x;
  while (z > 0) {
    y = y + x;
    z--;
  }
  return y;
}
```

```
int doubleSquare2(int x)
{
  int y = 0;
  int z = x;
  while (z > 0) {
    y = y + x;
    z--;
  }
  return 2 * y;
}
```

Fig. 5.7. Two programs which, given $x$, compute $2x^2$ [182]

Consider the two programs in Fig. 5.7. `doubleSquare2` is a faster program, with half the number of loop iterations. We would like to prove `doubleSquare1` $\equiv$ `doubleSquare2`, e.g., to validate a compiler optimisation. As the two objects we are proving equivalence of are (stateful) programs, we can view this as a

relational verification problem of a $\forall\forall$ property, of the form

$$\forall s_1 \forall s_2.\ s_1 \sim s_2 \Rightarrow [\![\texttt{doubleSquare1}]\!]\ s_1 \sim [\![\texttt{doubleSquare2}]\!]\ s_2$$

where $\sim$ is some kind of equivalence relation, and $s_1$ and $s_2$ are program states.

There are two families of approaches to proving relational properties of this form. The first, "product programs", reduces the relational verification problem to one in standard Hoare logic by constructing a new program such that its verification soundly implies that of the original program. The second is to use a relational Hoare logic [25], a system of rules for composing and constructing *quadruples* of the form $\{ P \}\ e_1 \mid e_2\ \{ Q \}$.

I describe the first approach and briefly comment on the second, to illustrate the key issue of *alignment*. After that, I show in detail how staged logic can be used to carry out similar equivalence proofs.

**Product programs** The simplest way to construct a product program for this situation is append them together, and modify them to act on disjoint state (Fig. 5.8).

We have to provide two loop invariants which "align" the two programs: to say that one iteration of one program corresponds to two iterations in another, by relating their intermediate states. This lets us prove that the states at the end are equal.

We can construct a better alignment by interleaving the statements of the two programs (Fig. 5.9). Note that we have to explicitly unroll the loop of `doubleSquare1` once. The loop invariant then becomes simpler.[4]

**Relational Hoare logic** In the second approach, using a relational Hoare logic, the issue of alignment is internalised in its proof rules. For example,

[4] Both programs were inspired by Dickerson, Mukherjee, and Delaware [70] and are verified with Dafny.

```
method doubleSquareSequenced (x1:int, x2:int)
  returns (y1: int, y2: int)
  requires x1 >= 0 && x2 >= 0
  requires x1 == x2
  ensures y1 == y2
{
  y1 := 0;
  var z1 := 2 * x1;
  while (z1 > 0)
    invariant y1 == x1 * (2 * x1 - z1)
    invariant z1 >= 0
  {
    y1 := y1 + x1;
    z1 := z1 - 1;
  }
  y2 := 0;
  var z2 := x2;
  while (z2 > 0)
    invariant y2 == x2 * (x2 - z2)
    invariant z2 >= 0
  {
    y2 := y2 + x2;
    z2 := z2 - 1;
  }
  y2 := 2 * y2;
}
```

Fig. 5.8. Alignment of programs by sequencing

```
method doubleSquareInterleaved (x1:int, x2:int)
  returns (y1:int, y2:int)
  requires x1 >= 0 && x2 >= 0
  requires x1 == x2
  ensures y1 == y2
{
  y1 := 0;
  y2 := 0;
  var z1 := 2 * x1;
  var z2 := x2;
  while (z1 > 0 && z2 > 0)
    invariant y1 == 2 * y2 // simpler
  {
    y1 := y1 + x1;
    y2 := y2 + x2;
    if (z1 > 0) { // unroll
      y1 := y1 + x1;
      z1 := z1 - 1;
    }
    z1 := z1 - 1;
    z2 := z2 - 1;
  }
  y2 := y2 * 2;
}
```

Fig. 5.9. Alignment of programs by interleaving statements

in Relational Separation Logic [213], the WHILE rule only applies to two programs with top-level while loops, similar to the second example above.

$$\frac{Inv \Rightarrow (B_1 \iff B_2) \qquad \{ Inv \wedge B_1 \}\, C_1 \mid C_2 \,\{ Inv \}}{\{ Inv \}\, \mathit{while}\ B_1\ \mathit{do}\ C_1 \mid \mathit{while}\ B_2\ \mathit{do}\ C_2 \,\{ Inv \wedge \neg B_1 \}}\ \text{WHILE}$$

In other words, the user can only verify programs with similar control flow structure, and have to get programs into the right form either by rewriting them or using things like one-sided rules.

**Staged logic** As staged logic is able to model programs faithfully, even stateful and effectful ones, equivalence can be directly stated and proved. The issue of alignment persists.

This section shows two proofs using staged logic, one using a functional presentation, the other using a more faithful imperative one.

**Functional** One view of the loop is as a tail-recursive functional program. We model it in staged logic in Fig. 5.10. For simplicity, $z$ is assumed to be a natural number.

$$\begin{aligned} &\mathit{loop}_1\ x\ y\ z \triangleq \\ &\qquad \textbf{ens}\ z{>}0; \mathit{loop}_1\ x\ (y{+}x)(z{-}1)) \\ &\quad \vee\ \textbf{ens}\ r.\, z{\leq}0 \wedge r{=}y \end{aligned}$$

$$\begin{aligned} &\mathit{doubleSquare}_1\ x \triangleq \mathit{loop}_1\ x\ 0\ (2 * x) \\ &\mathit{doubleSquare}_2\ x \triangleq \mathit{loop}_1\ x\ 0\ x; r_1.\ \textbf{ens}\ r.\, r = r_1 * 2 \end{aligned}$$

Fig. 5.10. Functional presentation of the loop

We would like to prove $\forall\, x.\ \mathit{doubleSquare}_1\ x \equiv \mathit{doubleSquare}_2\ x$. To do this, we first prove a more general lemma:

**Lemma 5.3** (Alignment between loop iterations)**.**

$$\mathit{loop}_1\ x\ (y + y)\ (2 * z) \equiv \mathit{loop}_1\ x\ y\ z; r_1.\ \textbf{ens}\ r.\, r = 2 * r_1$$

This is where the alignment occurs: saying how one iteration of the loop corresponds to two. The proof goes by induction on $z$.

**Base case** $z = 0$

$$loop_1\ x\ (y+y)\ (2*0) \equiv loop_1\ x\ y\ 0; r_1.\ \textbf{ens}\ r.r = 2{*}r_1$$

Simplifying,

$$\textbf{ens}\ r.r = y+y \equiv \textbf{ens}\ r.r = 2*y \quad \square$$

**Inductive case** $z = z_0 + 1$

$$loop_1\ x\ (y+y)\ (2*(z_0+1)) \equiv loop_1\ x\ y\ (z_0+1); r_1.\textbf{ens}\ r.r = 2*r_1$$

$$\begin{aligned}
& loop_1\ x\ (y+y)\ (2*(z_0+1)) \\
\equiv\ & loop_1\ x\ (y+y)\ (2+2*z_0) \\
\equiv\ & loop_1\ x\ (y+y+x+x)\ (2*z_0) \\
\equiv\ & loop_1\ x\ ((x+y)+(x+y))\ (2*z_)0 \\
\equiv\ & loop_1\ x\ (x+y)\ z_0; r_1.\textbf{ens}\ r.r = 2*r_1 \qquad \text{Ind. hyp.} \\
\equiv\ & loop_1\ x\ y\ (1+z_0); r_1.\textbf{ens}\ r.r = 2*r_1 \quad \square
\end{aligned}$$

The original goal follows easily from this lemma.

$$\begin{aligned}
& doubleSquare_1(x) \\
\equiv\ & loop_1\ x\ 0\ (2*x) \\
\equiv\ & loop_1\ x\ (0+0)\ (2*x) \\
\equiv\ & loop_1\ x\ 0\ x; r_1.\ \textbf{ens}\ r.r = 2*r_1 \quad \text{Lemma 5.3} \\
\equiv\ & doubleSquare_2\ x \quad \square
\end{aligned}$$

**Imperative** The imperative behaviours of the loop can also be modelled imperatively, and an advantage of staged logic is that the same proof methods can be used.

In Fig. 5.11, we first define stateful operations in terms of **req** and **ens**. $loop_2$ is defined using these, and used in the two programs below. This time, the *value* of $z$ is assumed to be a natural number; note that it is this value which decreases in $loop_2$, whose arguments otherwise remain unchanged.

This time, it is difficult to align the programs by the number of loop iterations, as that is not exposed as one of its parameters. Instead, we align

$$
\begin{aligned}
\mathit{incr}\ \ell\ n &\triangleq \forall a.\ \mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ \ell \mapsto a + n \\
\mathit{decr}\ \ell\ n &\triangleq \forall a.\ \mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ \ell \mapsto a - n \\
\mathit{read}\ \ell &\triangleq \forall a.\ \mathbf{req}\ \ell \mapsto a; \mathbf{ens}\ r.\ell \mapsto a * [r = a]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{loop}_2\ x\ y\ z \triangleq \forall a.\ & \\
& \mathbf{req}\ z \mapsto a \wedge a > 0; \mathbf{ens}\ z \mapsto a; \\
& \quad \mathit{incr}\ y\ x; \mathit{decr}\ z\ 1; \mathit{loop}_2\ x\ y\ z \\
& \wedge\ \mathbf{req}\ z \mapsto a \wedge a \leq 0; \mathbf{ens}\ z \mapsto a; \mathbf{ret}\ ()
\end{aligned}
$$

$$
\begin{aligned}
\mathit{doubleSquare}_1\ x &\triangleq \exists y\, z.\ \mathbf{ens}\ y \mapsto 0 * z \mapsto 2 * x; \mathit{loop}_2\ x\ y\ z; \mathit{read}\ y \\
\mathit{doubleSquare}_2\ x &\triangleq \exists y\, z.\ \mathbf{ens}\ y \mapsto 0 * z \mapsto x; \mathit{loop}_2\ x\ y\ z; \mathit{read}\ y; r.\ \mathbf{ret}\ r * 2
\end{aligned}
$$

Fig. 5.11. Imperative presentation of the loop

them by loop structure, by writing a lemma that describes the net effect of the loop. It increments the value of $y$ by $z * x$ and sets $z$ to 0. Hence, the following specification for it:

**Lemma 5.4** (Net effect of imperative loop)**.**

$$
\mathit{loop}_2\ x\ y\ z \sqsubseteq \forall a\ b.\ \mathbf{req}\ y \mapsto a * z \mapsto b; \mathbf{ens}\ y \mapsto (a + b * x) * z \mapsto 0
$$

Let $b_0$ be the initial value of $z$. The lemma can be proved by well-founded induction on $b_0$.

The induction hypothesis is:

$$
\begin{aligned}
\forall\, b.\ b < b0 \Rightarrow & \\
& \mathit{loop}_2\ x\ y\ z \\
\sqsubseteq\ & \forall a.\ \mathbf{req}\ y \mapsto a * z \mapsto b; \mathbf{ens}\ y \mapsto (a + b * x) * z \mapsto 0
\end{aligned}
$$

Unfolding and simplifying quantifiers,

$$
\begin{aligned}
& \mathbf{req}\ z \mapsto b_0 \wedge b_0 > 0; \mathbf{ens}\ z \mapsto b_0; \\
& \quad \mathit{incr}\ y\ x; \mathit{decr}\ z\ 1; \mathit{loop}_2\ x\ y\ z \\
& \wedge\ \mathbf{req}\ z \mapsto b_0 \wedge b_0 \leq 0; \mathbf{ens}\ z \mapsto b_0; \mathbf{ret}\ () \\
\sqsubseteq\ & \mathbf{req}\ y \mapsto a * z \mapsto b_0; \mathbf{ens}\ y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}
$$

To use one of the two conjuncts, we proceed by case analysis on $b_0$.

**Case** $b_0 > 0$

$$\begin{aligned}
&\textbf{req } z \mapsto b_0; \textbf{ens } z \mapsto b_0; \\
&\quad \mathit{incr}\ y\ x; \mathit{decr}\ z\ 1; \mathit{loop}_2\ x\ y\ z \\
\sqsubseteq\ &\textbf{req } y \mapsto a * z \mapsto b_0; \textbf{ens } y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}$$

The first step is symbolically execute the stateful operations. We do this by moving the **req**s to the left,

$$\begin{aligned}
&\textbf{ens } y \mapsto a * z \mapsto b_0; \\
&\textbf{req } z \mapsto b_0; \textbf{ens } z \mapsto b_0; \\
&\quad \mathit{incr}\ y\ x; \mathit{decr}\ z\ 1; \mathit{loop}_2\ x\ y\ z \\
\sqsubseteq\ &\textbf{ens } y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}$$

then using biabduction:

$$\begin{aligned}
&\textbf{ens } y \mapsto a; \\
&\textbf{ens } z \mapsto b_0; \\
&\quad \mathit{incr}\ y\ x; \mathit{decr}\ z\ 1; \mathit{loop}_2\ x\ y\ z \\
\sqsubseteq\ &\textbf{ens } y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}$$

We can repeat this to symbolically execute the stateful operations in the obvious way.

$$\begin{aligned}
&\textbf{ens } y \mapsto (a + x); \\
&\textbf{ens } z \mapsto (b_0 - 1); \\
&\quad \mathit{loop}_2\ x\ y\ z \\
\sqsubseteq\ &\textbf{ens } y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}$$

We can then use the induction hypothesis for $b_0 - 1$.

$$\begin{aligned}
&\textbf{ens } y \mapsto (a + x); \\
&\textbf{ens } z \mapsto (b_0 - 1); \\
&\quad \forall\, a'.\ \textbf{req } y \mapsto a' * z \mapsto (b_0 - 1); \textbf{ens } y \mapsto (a' + (b_0 - 1) * x) * z \mapsto 0 \\
\sqsubseteq\ &\textbf{ens } y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}$$

Letting $a'$ be $a + x$, we can simplify this to the following entailment, which follows from reflexivity.

$$\begin{aligned}
&\textbf{ens } y \mapsto ((a + x) + (b_0 - 1) * x) * z \mapsto 0 \\
\sqsubseteq\ &\textbf{ens } y \mapsto (a + b_0 * x) * z \mapsto 0
\end{aligned}$$

**Case** $b_0 \leq 0$

$$\begin{aligned} & \mathbf{req}\ z \mapsto b_0; \mathbf{ens}\ z \mapsto b_0; \mathbf{ret}\ () \\ \sqsubseteq\ & \mathbf{req}\ y \mapsto a * z \mapsto b_0; \mathbf{ens}\ y \mapsto (a + b_0 * x) * z \mapsto 0 \end{aligned}$$

We use the same idea to move **req** to the left.

$$\begin{aligned} & \mathbf{ens}\ y \mapsto a * z \mapsto b_0; \mathbf{req}\ z \mapsto b_0; \mathbf{ens}\ z \mapsto b_0; \mathbf{ret}\ () \\ \sqsubseteq\ & \mathbf{ens}\ y \mapsto (a + b_0 * x) * z \mapsto 0 \end{aligned}$$

Using biabduction,

$$\begin{aligned} & \mathbf{ens}\ y \mapsto a * z \mapsto b_0; \mathbf{ret}\ () \\ \sqsubseteq\ & \mathbf{ens}\ y \mapsto (a + b_0 * x) * z \mapsto 0 \end{aligned}$$

Since $b_0$ is a natural number, $b_0 * x = 0$. □

Having proved Lemma 5.4, we can use it to show the equivalence of $doubleSquare_1$ and $doubleSquare_2$.

$$\begin{aligned} & doubleSquare_1\ x \\ \sqsubseteq\ & \exists\, y\, z.\ \mathbf{ens}\ y \mapsto 0 * z \mapsto 2 * x; loop_2\ x\ y\ z; read\ y \\ \sqsubseteq\ & \mathbf{ens}\ y \mapsto 0 * z \mapsto 2 * x; \\ & \quad \forall\, a\, b.\ \mathbf{req}\ y \mapsto a * z \mapsto b; \mathbf{ens}\ y \mapsto (a + b * x) * z \mapsto 0; read\ y \\ \sqsubseteq\ & \mathbf{ens}\ y \mapsto (0 + (2 * x) * x) * z \mapsto 0; read\ y \\ \sqsubseteq\ & \mathbf{ret}\ 2 * (x * x) \end{aligned}$$

$$\begin{aligned} & doubleSquare_2\ x \\ \sqsubseteq\ & \exists\, y\, z.\ \mathbf{ens}\ y \mapsto 0 * z \mapsto x; loop_2\ x\ y\ z; read\ y; r.\ \mathbf{ret}\ r * 2 \\ \sqsubseteq\ & \mathbf{ens}\ y \mapsto 0 * z \mapsto x; \\ & \quad \forall\, a\, b.\ \mathbf{req}\ y \mapsto a * z \mapsto b; \mathbf{ens}\ y \mapsto (a + b * x) * z \mapsto 0; \\ & \quad read\ y; r.\ \mathbf{ret}\ r * 2 \\ \sqsubseteq\ & \mathbf{ens}\ y \mapsto (0 + x * x) * z \mapsto 0; \\ & \quad read\ y; r.\ \mathbf{ret}\ r * 2 \\ \sqsubseteq\ & \mathbf{ret}\ 2 * (x * x) \end{aligned}$$

□

**Conclusion** Staged logic is very much geared towards proofs of refinement and equivalence, so it handles this particular kind of relational property well. Nontrivial generalisation may in general be required to come up with the alignment, but there is some flexibility in how to do so, as shown by the

different routes we took for the functional and imperative versions of the proofs.

### 5.3.2 Folding left and right

The second problem we look at is the relation between *foldl* and *foldr*.[5] Unlike the previous example, where two programs were unconditionally related, *foldl* and *foldr* are related only under assumptions on the function $f$. For example, we might require that $f$ is commutative and associative; because $f$ is effectful, these are technically *hyperproperties* [56]: relations between multiple executions.

$$
\begin{array}{ll}
\mathit{foldl}\ f\ \mathit{acc}\ \mathit{xs} \triangleq & \mathit{foldr}\ f\ \mathit{xs}\ \mathit{init} \triangleq \\
\quad \textbf{ens}\ \mathit{xs}{=}\mathit{nil}; \mathit{acc} \vee & \quad \textbf{ens}\ \mathit{xs}{=}\mathit{nil}; \mathit{init} \vee \\
\quad \exists h\, t.\ \textbf{ens}\ \mathit{xs}{=}h{::}t; & \quad \exists h\, t.\ \textbf{ens}\ \mathit{xs}{=}h{::}t; \\
\quad\quad f\ \mathit{acc}\ h; r.\ \mathit{foldl}\ f\ r\ t & \quad\quad \mathit{foldr}\ f\ t\ \mathit{init}; r.\ f\ h\ r
\end{array}
$$

Fig. 5.12. *foldl* and *foldr*

We model the two folds as recursive definitions over lists *xs* in Fig. 5.12. They differ in the computational processes they generate: *foldr* is structurally recursive, while *foldl* is tail-recursive. This difference appears syntactically as a reordering of the recursive call and the call to $f$, and in the role of the *acc*/*init* parameter, which works as an accumulator in *foldl* but does not change in *foldr*. We call this value the *seed* when it does not matter which fold we are referring to. Note also that the order of the arguments to $f$ is different across both folds, following OCaml's `fold_left` and `fold_right`.[6]

There are a number of ways to give the conditions on $f$ needed for both folds to be equivalent. Bird and Wadler [32] give two in their first two *duality theorems*. The first is to require the type of the elements of *xs*, together with $f$ and the seed, to form a monoid, i.e., $f$ is associative and the seed is the

[5] In answer to a question raised by Olivier Danvy at my thesis proposal presentation.
[6] These differences in argument order have a rich history [57, Appendix 1].

identity element on both sides. Requiring this forces both arguments of $f$ to have the same type. The second is more general, and more easily given infix: if $x \oplus (y \otimes z) = (x \oplus y) \otimes z$ and $x \oplus a = a \otimes x$, then $foldl \otimes acc\ xs = foldr \oplus init\ xs$. For simplicity, we focus on the first theorem, replacing the identity condition with commutativity (a stronger condition).

$$f\ x\ y \sqsubseteq f\ y\ x \qquad\qquad f\ x\ y; r.\ f\ r\ z \sqsubseteq f\ y\ z; r.\ f\ x\ r$$

Under these assumptions, the right fold refines the left fold, and vice versa.

**Theorem 5.1** (Folds agree)**.** *If $f$ is commutative and associative, then*

$$foldr\ f\ a\ xs \sqsubseteq foldl\ f\ a\ xs$$

The proof is by induction on $xs$. The base case is immediate. After applying the induction hypothesis in the recursive case, the goal is

$$foldl\ f\ a\ t; r.\ f\ x\ r \sqsubseteq foldl\ f\ a\ (x :: t)$$

with $f$ on the *right* of *foldl*. To make progress, we need a lemma that "pushes" this trailing application into the accumulator on the left:

**Lemma 5.5** (Pushing an application past a left fold)**.**

$$foldl\ f\ a\ xs; r.\ f\ x\ r \sqsubseteq f\ a\ x; r.\ foldl\ f\ r\ xs$$

This can be seen as an alignment step, analogous to Lemma 5.3: it says how $f$ can move across *foldl*. It is itself proved by induction on $xs$. The base case uses commutativity, and the recursive case uses both commutativity and associativity to re-bracket applications of $f$.

The converse of Theorem 5.1 is proved in a symmetric way, with a similar "pushing" lemma for *foldr*.

**Conclusion** What is interesting about this example is that it demonstates the equivalence of the two folds for an *arbitrarily effectful* $f$. The properties

of $f$ required were stated naturally; in contrast, a traditional program logic would require an extension such as hyper-triples [56]. The issue of alignment also remains fundamental.

# MECHANISING STAGED LOGIC

Chapters 2 to 4 presented the workings of staged logic using an intuitive, idealised model. While this was convenient for exposition (and sufficient for the implementation of an automated verifier), it is not the full story, especially for the purpose of mechanising the logic in a foundational proof assistant.

This chapter completes the narrative with models for two fragments of the logic shown in Chapters 2 and 4. The first model follows the presentation thus far closely, but eventually proves inadequate in ways which the second model addresses, though at the cost of considerably greater complexity.

## 6.1 A simple model using behavioural refinement

The first mechanisation tackles the fragment of staged logic shown in Fig. 6.1. It was originally written in Rocq,[1] though some parts (notably, the definition of **req**) have been superseded by a new version in Lean.[2]

$$
\begin{aligned}
\varphi ::= \ & \mathbf{produce}\ H \mid \mathbf{consume}\ H \\
\mid \ & \mathbf{assert}\ P\,\varphi \mid \mathbf{assume}\ P \\
\mid \ & \cancel{\lambda x.\varphi} \mid f\ \varphi \mid \varphi_1\ \varphi_2 \mid \varphi_1; r.\ \varphi_2 \mid \mathbf{ret}\ v \\
\mid \ & \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\ \varphi \mid \forall x.\ \varphi \\
\mid \ & \mathbf{defun}\ f\ (\lambda x.\ \varphi) \mid \mathbf{discard}\ f \\
\mid \ & \mathbf{shift}^c\ k.\ \varphi\ \mathbf{id} \mid \langle f \rangle
\end{aligned}
$$

Fig. 6.1. A fragment of staged logic, sans lambda

The most notable change is that it is not truly higher-order: lambda is not a term of the language. The hallmark feature of staged logic, unknown functions, is still present; these functions are identified by name. **defun** and **discard** operations allow dynamically (de)allocating function names, associating them with statically-known bodies to recover some of what lambda

[1] https://github.com/dariusf/staged
[2] https://github.com/dariusf/staged-lean

would offer. This can be seen as a form of *defunctionalization* (Section 5.1). shift and reset are also present.

```
Inductive val : Type :=
  | vfptr : var → val
  | ...
with flow : Type :=
  | assert_ : Prop → flow | assume : Prop → flow
  | produce : hprop → flow | consume : hprop → flow
  | unk : var → val → flow
  | ret : val → flow | bind : flow → (val → flow) → flow
  | fex : forall (A:Type), (A → flow) → flow
  | fall : forall (A:Type), (A → flow) → flow
  | defun : var → (val → flow) → flow | discard : var → flow.
  | shc : var → flow → (val → flow) → flow | rs : flow → flow
  | ...
```

Fig. 6.2. Mixed embedding of the syntax of staged logic

**Syntax and semantics** Staged logic terms are represented using the inductive datatype `flow` (Fig. 6.2). The embedding is mixed: the overall structure is deeply embedded, with shallowly embedded binders, using higher-order abstract syntax (HOAS). Names are values (`vfptr`). An oddity is that the shift body binder in `shc` is *deeply embedded*, while the continuation is shallowly embedded. The reason for this is that shift allows its argument to "escape" via a `vfptr` value, while the continuation is only built up during reduction and its argument is never inspected.

The underlying separation logic is shallowly embedded as well, with propositions `hprop` represented in a standard way [47] as functions of type `heap → Prop`.

The semantics of staged logic is defined inductively via a separate interpretation relation, following the presentation in Fig. 2.11 and 4.12 closely. Compared to the earlier presentation, this semantics has an additional specification environment to accommodate updates. The new cases for `defun` are shown in Fig. 6.3 to give a flavour for how things are encoded. `defun` adds a

```
Inductive satisfies : senv → senv →
  heap → heap → result → flow → Prop :=
  | s_defun s1 s2 h1 x uf :
    ~ Fmap.indom s1 x →
    s2 = Fmap.update s1 x uf →
    satisfies s1 s2 h1 h1 (norm vunit) (defun x uf)
  | s_discard s1 s2 h (f:var) :
    s2 = Fmap.remove s1 f →
    satisfies s1 s2 h h (norm vunit) (discard f)
  | ...
```

Fig. 6.3. Semantics of staged logic (select cases)

new name to the specification environment, while `discard` removes it. The name must be new, so that `defun` followed by `discard` is the identity. Due to the `Inductive` interpretation of the semantics, it only includes terminating behaviours, if staged logic is seen as an abstract programming language. If it is seen as a logic, this means that derivations are finite: it is not possible to compromise soundness by creating an infinite derivation.

**Why this design and fragment?** Suppose we attempted a simple shallow embedding, resembling the standard one for separation logic:

```
Definition senv := var → flow.
Definition flow := senv → senv → heap → heap → val → Prop.
```

These definitions are cyclic and would not be accepted by Rocq.

A natural fix would be to deeply embed the syntax and define a separate interpretation function. This almost works, but breaks down in a key place: the `unk` case is not structurally recursive, as it invokes an arbitrary function from the environment. For this reason, we end up with an interpretation *relation*, as in Fig. 6.3.

With the syntax deeply embedded, consider how we can add lambda.

```
Inductive val : Type :=
  | vfun : (val → flow) → val
with flow : Type :=
  | ret : val → flow
  | ...
```

This definition is also rejected by Rocq, due to non-strict-positivity – the type `val` being defined occurs to the left of an arrow. The cause is the use of HOAS; a first-order version, like `vfun : var → flow → val`, would be accepted. However, the first-order version comes with a major downside: it requires both `hprop` and `Prop` to be deeply embedded. This closes the assertion language, disallowing quantification over arbitrary types and preventing us from importing facts already proved in Rocq. It also saddles us with many more issues and obligations involving binders – substitution, capture-avoidance, alpha-equivalence... Finally, it is still prone to a major problem we'll encounter at the end of this section.

The design we propose is a sweet spot for these reasons.

Another way to view it is as a freer monad. This is a common approach in the literature (Section 7.1.2), where we deeply embed the *effects* of our language and reuse the binding structure of Rocq, as we do with the continuation of `bind`. The main effects we have are unknown functions and shift, though we embed the other constructors as well so they can be inspected.

**Behavioural refinement** A simple, extensional account of refinement is used, following Section 2.3.

$$\varphi_1 \sqsubseteq \varphi_2 \triangleq \forall\, \mathcal{E}\, h_1\, h_2\, R.\ (\mathcal{E}, h_1, h_2, R \vDash \varphi_1 \Rightarrow \mathcal{E}, h_1, h_2, R \vDash \varphi_2)$$

The benefits of this are twofold: (1) all the various entailment rules (Fig. 2.12, 2.13, 2.15, 2.20, 2.22, 3.10, and 4.14) are sound by construction, and (2) interpreting entailment directly in terms of Rocq's implication lets us reuse its proof context and associated machinery for interactive proofs. Properties such as the contravariance of **assume** also come for free.

**Theorem 2.2.1** (Soundness of Entailment Rules)**.** *Given an entailment $\varphi_a \sqsubseteq \varphi_c$, every behaviour of the consequent is one of the antecedent, i.e. if $\mathcal{E}, h_1, h_2, R \vDash \varphi_a$, then $\mathcal{E}, h_1, h_2, R \vDash \varphi_c$.*

*Proof.* By construction. Each rule is proved against the semantics directly. □

**Rewriting** Staged logic proofs use rewriting extensively. The mechanisation makes heavy use of Rocq's *generalized rewriting* [192] to support this, which allows rewriting to be performed not just with equalities, but with user-defined equivalence relations or *preorders* (which need not be symmetric).

The entailment relation $\sqsubseteq$ is such a preorder, and may be used for rewriting given supporting lemmas demonstrating that it is a precongruence: that each constructor of staged logic is *monotonic* [15] or *proper* [192] with respect to it. These lemmas are provided as instances of a `Proper` typeclass. For example, the following key lemma (left), which states that entailment is monotonic with respect to itself, is given as the typeclass instance on the right. This allows a goal $\varphi_1 \sqsubseteq \varphi_2$ to be replaced with $\varphi_3 \sqsubseteq \varphi_4$, allowing the antecedent to be weakened and the consequent to be strengthened.

PROPERENTAILS

$$\frac{\varphi_3 \sqsupseteq \varphi_1 \quad \varphi_4 \sqsubseteq \varphi_2 \quad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1 \sqsubseteq \varphi_2}$$

```
Instance Proper_entails :
  Proper (flip entails ==>
    entails ==> impl) entails.
```

For another example, suppose the antecedent is $C[\varphi]$ for some context $C$. To rewrite it to $C[\varphi']$, we would need a `Proper` instance for every enclosing staged logic constructor in $C$, demonstrating that all of them preserve the entailment relation, or that they respect the context they are rewriting under. Examples of these lemmas are given in Fig. 6.4.

Depending on $C$, Rocq's rewriting machinery uses typeclasses to piece together a proof using these lemmas to descend into it, justifying a rewrite all the way down.

**Simplification and reduction** With the ability to rewrite inside arbitrary contexts, we are able to state the reduction rules of Section 4.3 simply as

$$\textsc{ProperSatisfies}\ \frac{\varphi_1 \sqsubseteq \varphi_2 \qquad \mathcal{E}, h_1, h_2, v \vDash \varphi_1}{\mathcal{E}, h_1, h_2, v \vDash \varphi_2} \qquad \textsc{ProperSeq}\ \frac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1; \varphi_3 \sqsubseteq \varphi_2; \varphi_4}$$

$$\textsc{ProperReq}\ \frac{\varphi_1 \sqsubseteq \varphi_2}{\mathbf{req}\ H\ \varphi_1 \sqsubseteq \mathbf{req}\ H\ \varphi_2} \qquad \textsc{RReset}\ \frac{\varphi_1 \sqsubseteq \varphi_2}{\langle \varphi_1 \rangle \sqsubseteq \langle \varphi_2 \rangle}$$

Fig. 6.4. Select rewriting lemmas

entailment lemmas, and rely on rewriting to justify them.

**Theorem 3.3.1** (Soundness of Reduction Rules)**.**

$$\varphi_1 \rightsquigarrow \varphi_2 \textit{ iff } \mathcal{E}, h_1, h_2, R \vDash \varphi_1 \Rightarrow \mathcal{E}, h_1, h_2, R \vDash \varphi_2.$$

*Proof.* The soundness of reduction follows directly from the soundness of entailment. □

### 6.1.1 Limitations of the model

While the model formalised so far in this chapter is sufficient to prove properties of many programs discussed in this dissertation, it does not work for all of them. The main issue is that the model contains *syntax* – specification environments $\mathcal{E}$ are maps containing values of type `flow` – which must be subject to a further operational interpretation. This prevents some statements from being proved. For example, consider Lemma 6.1.

**Lemma 6.1** (Associativity of seq. – untrue!)**.** $(\varphi_1; \varphi_2); \varphi_3 \Leftrightarrow \varphi_1; (\varphi_2; \varphi_3)$.

To see intuitively why this lemma cannot be proved, suppose that $\varphi_1$ contained a shift. According to the semantics, we would get shift outcomes on both sides, with continuations $(\square; \varphi_2); \varphi_3$ and $\square; (\varphi_2; \varphi_3)$. The logical next step would be reassociate the continuations, perhaps inductively. Unfortunately, the definition of entailment (Definition 2.1) requires the outcomes to be *syntactically equal*, and does not allow for this.

Intuitively, however, Lemma 6.1 should certainly be true. We demonstrate this formally by strengthening the definition of entailment to allow for this inductive reassociation.

**Definition 6.1** (Generalised entailment)**.**

$$\textsc{GEntailsBase}\ \frac{\forall \mathcal{E}_1\ \mathcal{E}_2\ h_1\ h_2\ v.\quad \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, v\,\varphi_1 \Rightarrow \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, v\,\varphi_2}{\varphi_1 \subseteq_0 \varphi_2} \qquad \textsc{GEntailsShift}\ \frac{\begin{array}{l}\forall \mathcal{E}_1\ \mathcal{E}_2\ h_1\ h_2\ \varphi_b\ \varphi_k.\\ \quad \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, \varphi_1 \Rightarrow\\ \quad \exists \varphi_b'\ \varphi_k'.\ \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, \varphi_2 \wedge\\ \quad (\varphi_b \subseteq_n \varphi_b') \wedge (\forall v.\ \varphi_k[r/v] \subseteq_n \varphi_k'[r/v])\end{array}}{\varphi_1 \subseteq_{(n+1)} \varphi_2}$$

A statement of generalised entailment $\varphi_1 \subseteq_n \varphi_2$ is indexed by the height of its derivations. At index 0, there are no shifts, and this is simply standard entailment for values. At index $n+1$, there is at least one more shift, and we inductively require the resulting continuations to be related by generalised entailment at index $n$. Intuitively, an index of $n$ allows us to talk about the behaviour of continuations *after n future shifts*. With this, associativity can be proved as we would expect, with $\varphi_1 \equiv_n \varphi_2 \triangleq \varphi_1 \subseteq_n \varphi_2 \wedge \varphi_2 \subseteq_n \varphi_1$.

**Lemma 6.2** (Generalised associativity)**.** $(\varphi_1; \varphi_2); \varphi_3 \equiv_n \varphi_1; (\varphi_2; \varphi_3)$

*Proof.* By induction on $n$. □

Unfortunately, when used as an entailment, the index exposed in generalised entailment is too constraining.

Another way to sidestep the issue is to ignore the cases where $\varphi_1$ and $\varphi_2$ have shifts.

**Lemma 6.3** (Associativity of seq. – shift-free)**.**

$$\textit{If } \varphi_1^{\#} \textit{ and } \varphi_2^{\#}\textit{, then } (\varphi_1; \varphi_2); \varphi_3 \Longleftrightarrow \varphi_1; (\varphi_2; \varphi_3).$$

This can also be proved, but it is less than ideal, as we cannot reason completely about programs containing shifts.

Despite the shortcomings of the current model, it does validate the soundness of the framework, at least sufficiently to construct an automated tool.

A more expressive model which overcomes these problems is described in Section 6.2.

### 6.1.2 Evaluation

Table 6.1
Case studies verified in Rocq

| **Program** | **Features** | **LoC** |
|---|---|---|
| *foldr* (Section 2.1) | HO, ind., imp. | 512 |
| Landin's knot (Section 5.1) | HO, ind., imp. | 436 |
| Total | | 998 |

The mechanisation of the metatheory of staged logic, as well as basic automation, spans a development of 6392 LoC in Rocq. It is built on top of the mechanisation of separation logic from the *Separation Logic Foundations* book [47].

To evaluate the feasibility of the theory, I attempted a number of case studies (Table 6.1) to get a sense of the effort and issues involved. Each example is shown with a list of its interesting features (use of higher-order functions, induction, imperative state, or multi-shot continuations), followed by the LoC (comprising all of program, specification, and lemma definitions, and proof script).

Included with the mechanisation is a library of tactics which automate applying the rules of Section 2.3, but with fewer heuristics, optimising instead for predictability in interactive use. This significantly reduces the length of proof scripts.

## 6.2 Towards a model using contextual refinement

In Section 6.1, I described a mechanisation with a simple form of behavioural refinement as the notion of entailment. It is complete enough to be able to handle a number of case studies interactively, with tactics for working at the level of the logic.

However, as shown in Section 6.1.1, the underlying model is not sufficiently expressive to be able to give a full treatment of the logic, leading to odd limitations, like the lack of associativity, and being unable to rewrite on the right side of a shift. It is also an *encoding* of the logic and does not model every feature in it faithfully, e.g., using function pointers instead of general lambda expressions.

In this section, I sketch an alternative interpretation of the logic, outlining the path towards a complete mechanisation. The theory presented in this section been mechanised in Rocq.[3]

### 6.2.1 A more expressive notion of refinement

We first restate the problem from Section 6.1.1. It arose because of the use of shift outcomes, which contain syntax representing a shift body or continuation. Because, according to the semantics (Fig. 6.3), shift outcomes are "inert" and cannot be subject to further interpretation outside a reset, we ended up with semantically-equal objects that were syntactically distinct, and so could not be accepted by our insufficiently-expressive refinement relation.

It turns out that a version of this problem also occurs in the plain lambda calculus with lambda abstraction values – for example, $\lambda x.\, x \sqsubseteq \lambda x.\, (\lambda y.\, y)\, x$ also does not hold. The missing piece is the more fine-grained notion of *contextual refinement*, which abstracts away syntactic detail by enclosing terms in an arbitrary context, which, intuitively, captures all possible ways we can *con-*

[3] https://github.com/dariusf/staged/tree/ctx-equiv/ctx-equiv-ixfree

*tinue interpretation* on shift outcomes/lambda values. Any such suspended values which cannot be distinguished by arbitrary further use will be contained in the relation.

Contextual refinement is difficult to prove directly [16]. An alternative is to use a binary *logical relation*, which provides a definition that is sound and complete with respect to contextual refinement, yet is easier to work with. Logical relations are typically defined inductively over the structure of types, and are often *step-indexed* to cope with circularities that cannot be resolved by the structure of types alone. I use a step-indexed, but *untyped* treatment [100, 16], in the interest of both generality (types were not hitherto mentioned in this work) and simplicity (a typed treatment would have to handle things like answer-type modification). The technique of *biorthogonality* [164], which is to define logical relations for contexts as well as expressions/values in a mutually recursive manner, is also used.

$$
\begin{aligned}
\mathcal{L} &\triangleq \{\ (e_1, e_2)\ \mid (\forall\, v.\ e_1 = v \Rightarrow e_2 \Downarrow)\ \wedge\ (\forall\, e.\ e_1 \longrightarrow e \Rightarrow (e, e_2) \in \triangleright\mathcal{L})\} \\
\mathcal{O} &\triangleq \{\ (e_1, e_2)\ \mid (e_1, e_2) \in \mathcal{L} \wedge (e_2, e_1) \in \mathcal{L}\} \\
\mathcal{V} &\triangleq \{\ (v_1, v_2)\ \mid \forall\, v_1'\ v_2' \in \triangleright\mathcal{V}.\ (v_1 v_1', v_2 v_2') \in \mathcal{R}\} \\
\mathcal{K} &\triangleq \{\ (K_1, K_2)\ \mid \forall\, v_1\ v_2 \in \mathcal{V}, M_1\ M_2 \in \mathcal{M}.\ (M_1[K_1[v_1]], M_2[K_2[v_2]]) \in \mathcal{O}\} \\
\mathcal{E} &\triangleq \{\ (e_1, e_2)\ \mid \forall\, K_1\ K_2 \in \mathcal{K}, M_1\ M_2 \in \mathcal{M}.\ (M_1[K_1[e_1]], M_2[K_2[e_2]]) \in \mathcal{O}\} \\
\mathcal{R} &\triangleq \{\ (e_1, e_2)\ \mid \forall\, K_1\ K_2 \in \triangleright\mathcal{K}, M_1\ M_2 \in \triangleright\mathcal{M}.\ (M_1[K_1[e_1]], M_2[K_2[e_2]]) \in \mathcal{O}\} \\
\mathcal{M} &\triangleq \{\ (M_1, M_2)\ \mid \forall\, v_1\ v_2 \in \mathcal{V}.\ (M_1[v_1], M_2[v_2]) \in \mathcal{O}\}
\end{aligned}
$$

Fig. 6.5. Logical relation for a lambda calculus with shift/reset

The resulting binary logical relation is shown in Fig. 6.5. It is defined using contextual equivalence (contextual refinement in both directions), and for a lambda calculus with shift/reset.[4][5] It assumes the underlying logic provides a notion of *logical step-indexing* [168, 74], using the later modality to make definitions well-founded.

The different relations are named mnemonically. The relation $\mathcal{L}$ is one

[4]The missing parts of staged logic are addressed in Section 6.2.3.

[5]As staged logic is now directly encoded as a lambda calculus, there is no longer a need for specification environments or function pointers.

side of the observational equivalence relation $\mathcal{O}$ for closed programs. $\longrightarrow$ is a *meta-contextual step*: a small-step operational semantics for shift/reset with context and meta-context [61].[6] $\mathcal{V}$ is for values; since we are dealing with a lambda calculus, the only values are lambda abstractions, which are in the relation if applying them to values in the relation after one step produces results in the relation. $\mathcal{K}$ is for contexts, which are related if plugged with and into related values and meta-contexts, respectively, if the result is observationally equivalent. $\mathcal{E}$ is for expressions, which are related if plugged into related contexts and meta-contexts and observationally equivalent. $\mathcal{R}$ is the same as $\mathcal{E}$ with the exception of the later modalities; it is only used in $\mathcal{V}$ and is there for technical reasons [16]. An increment over Balik, Biernacki, and Polesiuk [16], where continuations are undelimited, is the addition of meta-contexts to the definitions and to the $\mathcal{M}$ relation.

### 6.2.2 Encoding in Rocq

The addition of contexts and contextual semantics necessitates a deep embedding of staged logic, so the syntax is defined as an inductive datatype. As staged logic is a lambda calculus at heart, this is in keeping with observations made in the literature that it is more appropriate to deeply-embed "dynamic" objects (such as programs, involving state transitions) and shallowly-embed "static" objects (such as assertions, which describe sets of states) [208].

Given the deep embedding, binding is handled via the so-called "functorial approach" [169], using well-scoped de Bruijn indices.

The next salient feature of the encoding is the use of IxFree [168] for a step-indexed base logic with a later modality [74], which hides the explicit index arithmetic. Iris [108] could equally be used as a base logic; I used it for an early prototype, but it was generally easier to proceed with IxFree due to its simpler core and focus.

[6] Notably, the bubble-up semantics and shift outcomes are no longer needed.

Finally, contextual equivalence is a congruence (and contextual refinement a *precongruence*), so the congruence lemmas required are easy to define using the completeness theorem.

### 6.2.3 Remaining work

Extending this model to cope with all of staged logic remains work in progress. Many of the ideas from Section 6.1 can be reused, but must be adapted to the logical relations setting. The different fragments of staged logic require different techniques whose combination is nontrivial.

- (Angelic) nondeterminism must be handled with may-observational equivalence [34].
- Heaps must be stratified, like the CMRAs [108] of Iris or the memories in *indirection theory* [98], as they are able to contain functions.
- Since both the languages of programs and specifications are deeply embedded, triples (Section 2.2.2) can be defined as cross-language logical relations [160].
- Effect handlers also admit a logical relations account [214].

## 6.3 Summary

Table 6.2 gives a summary of both mechanisations. The main differences between them are 1. the definition of entailment, and 2. how the syntax of staged logic is represented.

The simple one uses a shallow embedding for the underlying (separation) logic, which is conventional [208] and allows the reuse of existing proof automation [47]. Its syntax is deeply embedded, but with anything involving values shallowly embedded (using HOAS), as necessitated by the shallow

Table 6.2
Summary of both mechanisations

| | | **Simple** | **Contextual refinement** |
|---|---|---|---|
| Syntax | Separation logic | Shallow | Deep |
| | Staged logic | Deep + shallow | |
| Semantics | Programs | Big-step | Contextual small-step |
| | Delimited continuations | Bubble-up | List of contexts |
| Entailment | | Refinement | Contextual refinement |
| Sound | | ✓ | ✓ |
| Complete | | ✗ | ✓ |
| Higher-order functions | | ✓ | ✓ |
| **req/ens** | | ✓ | ✗ |
| Heaps | | ✓ | ✗ |
| shift/reset | | ✓ | ✓ |
| Tactics | | ✓ | ✗ |
| LoC | | 6392 | 5931+ |
| Case studies | | 5 | - |

embedding of the underlying logic. Limitations of shallow embedding, such as the inability to perform induction over syntax or have function values due to negative occurrences, are worked around or suffered.

The one based on contextual refinement uses a deep embedding for everything. This is more general but significantly more complex, necessitating things like an intrinsically-typed de Bruijn representation, as well as definitions for evaluation and general program contexts.

The simple mechanisation is a relatively practical one, with a small number of case studies foundationally verified, using a number of specialised tactics. It also supports the whole language of staged logic. However, there are entailments which cannot be proved in it – this is what is meant by lack of completeness.

To overcome this problem, the second one is based on contextual refinement, which is defined using a step-indexed, biorthogonal logical relation. It is unfinished at the time of writing and does not cover the entire language of staged logic; there are also no tactics or case studies using it yet. However, its size is already comparable to the simple one, reflecting the significant

increase in complexity.

# RELATED WORK

## 7.1 Program logics

Deductive verification is typically carried out with the help of *program logics*, formal systems which give programs an axiomatic semantics. In this section, we give a brief history of the development of program logics, leading up to the modern developments that we build on.

### 7.1.1 Hoare logics

The field of program logics could be said to have begun with *Hoare logic* [97], which introduced the *Hoare triple* $\{P\}\, e\, \{Q\}$ as a way of characterising the behaviours of programs. Dijkstra [72] developed the idea of automating the verification of Hoare triples using *weakest preconditions*, which half a century later forms the basis of many mature tools, such as Dafny [134] and Why3 [79]. As these tools matured, they came to be structured as *verification platforms*, supporting *intermediate verification languages* [17, 149]: shared infrastructure to which the verification of programs in different languages could be reduced. Cameleer [161, 187, 185] is one such extension of the Why3 platform, enabling automated verification of OCaml programs. It has a backend-agnostic specification language, GOSPEL [188], which can also be interpreted via Ortac [101] for more lightweight runtime checking.

While tools based on standard Hoare logic are highly mature, they have limited support for more dynamic language features, such as heap manipulation, higher-order programs, and effects; there is not yet consensus on how best to support them. Staged logic is one answer, bridging Hoare logic with its sister methodology of refinement. Seen as an extension of Hoare logic, staged logic uses triples in much the same way, and can be seen as enriching

them with an underlying logic that is *behavioural*, satisfied by traces rather than single states.

**Separation logic** Reynolds [177] and O'Hearn [155] proposed *separation logic*, a substructural logic of resources, as a way to solve the frame problem of Hoare logic and enable reasoning about heap-manipulating programs. It became the basis of a new generation of automated verification tools, such as VeriFast [107, 106], HIP/SLEEK [49], and Viper [149], which automate program proofs using *symbolic execution* [26]. Separation logic is also the foundation of the Infer tool, which pioneered the use of *biabduction* [42] for *begin-anywhere* static analysis.

Staged logic adopts separation logic as a means of describing program states, while handling the behavioural aspects. It also relies on biabduction to simplify terms, in a manner similar to symbolic execution.

**Mechanisation** While there are many automated program verifiers in continued and successful use, a major alternative direction has been to encode program/separation logics in proof assistants, to inherit both their interactive facilities as well as their foundational soundness guarantees.

A key design choice when mechanising a logic is whether to embed each aspect of it *shallowly* or *deeply* [208]. A shallow embedding allows for maximal reuse of the proof assistant's features, but limits metatheoretical reasoning, as well as what can be encoded, if there is no precisely-matching feature in the metalogic – an example is in the use of higher-order abstract syntax (HOAS), where binding is represented by metalogic functions, and negative occurrences limit what can be expressed and require nontrivial workarounds. Dually, a deep embedding results in no semantic mismatch and full control over every aspect of the object logic, with the tradeoff that every feature of the logic has to be reimplemented – to continue the example of binding, a first-order encoding requires necessitates representations such as de Bruijn

indices, as well as theorems to say that terms are closed.

Modern separation logics mostly shallowly embed the language of propositions and differ on whether to deeply embed the program. These two correspond roughly to whether the logic is seen as something "extrinsic" [123] to the program, and related to it only by a separate relation like a Hoare triple, or "intrinsically" part of it, e.g., encoded in its type.

CFML [46, 45] is based on the notion of *characteristic formulae*, relations between pre- and postconditions that can be computed from the (deeply-embedded) syntax of programs, such that the syntax no longer features in verification conditions. It is simple and elegant, and is used for teaching in the *Separation Logic Foundations* [47] textbook. It is common in Iris [108] to deeply embed programs and their contexts [87, 202]. The logic of Iris is shallowly embedded, but atop a sophisticated step-indexed model of higher-order logic, in order to support impredicative quantification and higher-order state. The Verified Software Toolchain (VST) [7] makes similar choices with respect to programs, but implements a step-indexed model using *indirection theory* [98], a general method of implementing stratified models. These logics mechanise the process of symbolic execution using magic wands, weakest preconditions, and the ramified frame rule [99].

Hoare Type Theory (HTT) [151] pioneered the approach of using dependent types to embed specifications into the types of (shallowly-embedded) monadic programs. This approach was followed by Swamy et al. [197], who proposed *Dijkstra monads* and the use of weakest preconditions instead to automate the computation of specifications; this was later generalised to work for any *specification monad* [140, 210] and more combinations of effects. Pulse [86, 196, 77] is a separation logic embedded in F⋆, based on these ideas, which also marries them with indirection theory for reasoning about concurrency.

In the first mechanisation (Section 6.1), staged logic takes a variation of

the former position: staged logic terms are deeply embedded but use HOAS and a shallow embedding of the underlying (separation) logic. The result is an incomplete but more practical mechanisation, suitable for interactive use. In the second (Section 6.2), because of the dynamic nature of the logic and its ability to manipulate higher-order values, it is completely deeply embedded, as a program would be. This mechanisation is more faithful but is significantly more complex.

### 7.1.2 Refinement

**Temporal logics and Hoare logics** Refinement is a foundational concept in program verification, serving as a means of relating descriptions of systems. Perhaps the most well-known and widely-used temporal logic, TLA$^+$ [125], uses it to relate state machines to protocols or temporal properties, which are all expressed in the same logic. Refinement is also used in Hoare logics, typically to justify that an effectful program implements some pure function, by including the latter directly in the postcondition; we rely on this sort of use in our work.

**Contextual refinement** A related property is *contextual refinement*, a relation between *open* programs which quantifies over their contexts:

$$\forall\, C.\ C[e] \sqsubseteq C[e']$$

This intuitively means that an arbitrary client $C$ cannot distinguish $e$ from $e'$.

Proving contextual refinement directly by induction over contexts is known to be difficult [16]. It is instead usually proved by via the technique of logical relations [16]. The idea is to define relations which are sound and complete with respect to contextual refinement, but for which it is easier to prove membership.

Logical relations were classically defined inductively over the structure

of types. However, semantically-cyclic language features such as recursive types [8] and general references [3] necessitated induction over some other index; this led to the technique of *step-indexed* logical relations. Step-indexing can be treated logically [74], hidden behind a *later* modality. Modern logics such as IxFree [168] and Iris [108] take this approach, and provide facilities for defining predicates that are well-founded based on the index. These can then be used to construct logics for reasoning about contextual refinement.

One approach is to reason about contextual refinements directly. Earlier approaches, such as Timany and Birkedal [202], worked with the logical relation directly, which was defined in terms of weakest preconditions; they introduced the idea of context-local triples to simplify reasoning when contexts were not manipulated, but ultimately unfold the logical relation and work with its definition. ReLoC Reloaded [87] is a newer approach which uses a *typed* relation of the form $e_1 \precsim e_2 : \tau$, and structural rules to manipulate this judgment, remaining at the level of refinements.

An alternative is to use a relational program logic to reason compositionally about it. Simuliris [88] takes this approach, with a simulation relation (similar to the judgment of Relational Hoare Logic [25]) $\{P\}\, e_1 \preceq e_2 \,\{r_1 r_2.\, Q\}$.

The approach I take in the latest mechanisation (Section 6.2) is similar to the one used in Timany and Birkedal [202], working with the logical relation directly – a simple, baseline approach – with a view towards eventually adapting the approach used in ReLoC Reloaded (to the untyped setting, and to programs with control operators), to realise the goal of reasoning at the level of the entailment relation.

**Refinement calculi** Refinement calculi [14, 147, 13, 43] are another family of program logics, originating in the methodology of *stepwise refinement*, a deductive program synthesis method for deriving programs from specifica-

tions. The distinguishing features of refinement calculi are that they embed *specification statements* in programs, allowing both to be expressed in the same abstract programming language, and a binary *refinement* relation which orders abstract programs, serving as a means of giving them specifications. This allows working and reasoning entirely at the level of programs.

Staged logic bears superficial resemblance to refinement calculi, given that it mixes specifications and program elements in a single language. It is, however, a logic with constructs for *representing* programs, rather than a programming language, and is designed for verification rather than synthesis.

Modern routes to mechanisation for refinement calculi commonly involve shallow, monadic embeddings of effectful programs. For example, Boulmé [36] uses a state monad equipped with a least fixed point operator to verify higher-order imperative programs. Another line of work [4, 12, 198] from Swierstra and Baanen [198] uses free monads to uniformly implement a variety of effects. None of these tackle delimited continuations, which is not an algebraic effect [103, 166] and cannot be expressed as a free monad. It is unclear how to shallowly embed effect handlers or delimited continuations alongside e.g., a weakest precondition semantics.

### 7.1.3 Monadic encodings of effects

Monads are a useful semantic device for shallowly embedding effects, and another line of work using them for reasoning about programs is *monadic equational reasoning* [90, 2]. Staged logic similarly advocates for the idea of treating effects equationally, a simple and effective approach. However, staged logic also deals with specifications, and, consequently, inequations (refinement). The semantics of specifications can be shallowly embedded using a weakest precondition semantics, as in the encodings of refinement calculi mentioned in Refinement calculi. However, it is unclear how to extend this to non-algebraic effects. Staged logic thus uses a deep embedding for the

semantics of specifications.

Another program verification approach leveraging monads is the ordered *specification monads* of the Dijkstra monad line of work [140, 210]. This approach differs from staged logic in that specifications are given intrinsically, rather than extrinsically – the monad is only used to the encode the effectful program, and the specifications are given in a standard weakest precondition style.

### 7.1.4 Relational verification

Much of the program verification literature is dedicated to *trace properties*, predicates on program executions of programs. *Relational properties* or *hyperproperties* generalise this to relations between executions.

In the relational verification literature, *refinement* refers to a relational $\forall\exists$ property which copes with a nondeterminism reference program [24, Definition 2]. This is not the sense in which it is used in this dissertation, which uses the classic sense of behavioural inclusion.

Relational verification is mainly carried out in two ways [19].

The first is to use a *relational program logic* [25], a bespoke formal system for proving specific kinds of relational properties. Some examples are RHLE [71] ($\forall\exists$ properties), CHL [191] (*k-safety* or $\forall^*$ properties), and Hyper Hoare Logic [63], which allows arbitrary quantifier alternations.

The second way is to encode relational verification problems into standard Hoare logic, to allow reuse of existing tools. This requires different encodings depending on the type of property. *Self-composition*, or more generally, *product programs* [21] for $\forall\forall$ properties. *Asymmetric product programs* [20] extend this to $\forall\exists$ properties. This can also be done at the level of logics [212].

Regardless of the verification method, the issue of *alignment* is essential. Intuitively, it is the act of bringing parts of the programs being related into

correspondence, to make verification easier. For example, relational program logics often contain "lockstep" rules which only apply to two structurally-similar programs. Alignment appears in different forms depending on the verification approach being considered. In the self-composition approach, it may require duplicating loop bodies, unrolling loops, and so on, while in relational program logics, relational invariants allow loop states to, and there may be rules for executing one program, given that it does not change the state relevant to the relation. For some formulations, it is possible to automatically search for alignments [204, 70, 5].

This dissertation focuses mostly on trace properties. A case study on encoding program equivalence (a $\forall\forall$ property) using staged logic is given in Section 5.3.1. Other kinds of $\forall\forall$ properties, as well as other relational properties, are left to the future work.

## 7.2 Higher-order imperative programs

### 7.2.1 Specifications

Before we can verify higher-order programs, we must first specify what it means for them to be correct. There are two broad families of approaches for doing so, with contemporary approaches fusing them to combine their benefits.

**Nested Hoare triples and invariants** The use of a Hoare triple as a proposition to give a function parameter a specification is a natural and "obvious" [178] way to treat higher-order programs. Such a Hoare triple is typically coupled with an *invariant*, a predicate of the metalogic, to describe the effect of a function; an introduction to this pattern was given in Section 2.1. For this reason it requires higher-order quantification [129, Section 2.1], and so works well in program logics designed for interactive verification, such as

iHTT [195], Iris [108, 33], and CFML [46].

It can also be adapted to automated verification, either directly [48] or by first-order axiomatisation: using axioms to associate specifications and invariants with otherwise uninterpreted higher-order functions. The latter approach was pioneered by Régis-Gianas and Pottier [175], extended to the imperative setting by Kanig and Filliâtre [112], and remains commonly used in other verifiers [185] in the Why3 [79] lineage, as well as newer tools like Verus [127]. *Refinement types*, such as in LiquidHaskell [206], can also be seen as an intrinsic encoding of nested triples.

**Greybox specifications** Büchi and Weck [40] proposed the idea of *greybox specifications*, observing that refinement was a good way to specify higher-order programs with side effects, due to the difficulty of abstracting them in general; Boulmé [36] also makes this observation in the context of OCaml programs. Shaner, Leavens, and Naumann [181] adapted the greybox approach to JML in the form of *model programs*, which extends [129] to aspect-oriented and context-oriented language features.

Kanig and Filliâtre [112] noted that "one is often confronted with the problem of specifying the behaviour of a program where the most natural description of the behaviour is the program itself!" In Why3 [79] and Dafny [134], pure functions require no further specification, as they are already mathematical objects, and so can be given directly to SMT solvers, even in higher-order contexts. This technique can be seen as an instance of (and, retrospectively, motivation for) the greybox approach as well.

Prusti [211] primarily uses a nested triple approach (*specification entailments*), but also proposes *call descriptions*, which specify the *uses* of closures within higher-order functions, something which would be are otherwise opaque to clients, in order to inform clients about their effects. Penninckx, Timany, and Jacobs [159] uses *abstract* nested triples to specify I/O

in a similar way.

Coma [158] is a recent addition to the Why3 platform, with facilities for enforcing abstraction barriers, allowing the user to vary how much of a program is specified, and which parts are subject to the greybox approach.

The approach I describe in this dissertation also fuses both paradigms, combining the symbolic execution of separation logic triples with the flexibility of refinement as the means of specification.

### 7.2.2 Syntactic methods

Given the right specification, (syntactic) verification is often natural. With a nested triple, one must instantiate any higher-order quantifiers (which typically requires human input) before using the standard function application rule. Model programs are equally simple to verify and instantiate for use, though symbolic execution is not considered there, something we remedy.

In this section we survey additional syntactic techniques useful for verification.

**Defunctionalization** Defunctionalization [176] is a classic technique for compiling higher-order programs to first-order ones, and it can similarly be used to reduce the verification of a higher-order program to a first-order setting. It was first proposed by Soares [186] for Cameleer, using axiomatised specifications parametrised over states to handle effectful programs, as Régis-Gianas and Pottier [174] do. Creusot [69] models closures as stateful objects, following the *closure conversion* Rust uses, a closely related concept to defunctionalization.

**Type system guarantees** An expressive type system can significantly simplify how higher-order state is specified and managed. For example, in Rust, closures have exclusive ownership over mutable locations that they capture;

Prusti [211] exploits the fact that they can maintain *history invariants* to provide stronger guarantees about captured state. Creusot [69] uses *prophecies*, similar in spirit to ghost variables allowing one to talk about the values of mutable locations at the end of borrows, to specify mutable state; it then treats closures as if they have constant prophecies relating to their captured state.

### 7.2.3 Semantic methods

In the earlier section on Mechanisation, we briefly surveyed the techniques generally used to implement higher-order program logics in interactive proof assistants. Here we focus more specifically on ways in which they support the verification of higher-order imperative programs and the issues encountered therein.

As today's mature proof assistants are based on type theories or higher-order logic, they do not natively support closures or imperative updates in their metalanguages. Embedding a program logic thus begins with an embedding of programs, which may be *deep* or *shallow*. Deep embeddings are generally handled in an *extrinsic* way, with (nested) Hoare triple propositions associating specifications with programs, while shallow embeddings are handled in an *intrinsic* manner, with their syntax represented as a value of a monadic type indexed with a specification.

**Nested triples** Hoare triples are the usual way of specifying higher-order programs in proof assistants. Due to higher-order quantification and the impredicativity of, e.g., Rocq's `Prop`, they are naturally usable in a nested manner, e.g., in CFML [46, 47].

**Dijkstra monads** Dijkstra monads [197, 140] were originally introduced to automate the computation of verification conditions, but their use case was

to verify higher-order, stateful programs. The intrinsic style of verification with dependently-typed monads lends itself very naturally to higher-order programs, with state naturally handled by computing weakest preconditions (which transform predicates over states), and the effectful monadic layer taking care of the effects.

**Impredicativity** A problem that plagues both extrinsic and intrinsic approaches alike is that of impredicativity. It shows up in higher-order programs in semantically-cyclic features such as *higher-order store*. This necessitates constructing models stratified by some notion of step, typically a small step in the operational semantics, and using a *later modality* to guard recursive occurrences of predicates, ensuring they take place at a smaller step to keep definitions are well-founded. Iris [108] is perhaps the most influential and well-known example of this approach. *Indirection theory* [98] is a general approach for building stratified models and viewing them through the lens of approximation; VST [7] and PulseCore [77] are recent examples building on it.

*Syntactic indirection* [6, 50, Chapter 35] is a means of working around the issue of impredicativity without a later modality, typically using a syntactic approach at the cost of some expressiveness. VeriFast [106] is a recent example of program logic with this approach. XCAP [153] was an early example, doing this to verify low-level programs with function pointers. It works around the problem using both syntactic assertions, and a modular approach where programs are verified under local assumptions and deferring the check for whether assumptions are all globally consistent. This can also seen as deeply-embedding part of the assertion language in order to defer what would have otherwise been a cyclic interpretation, breaking the cycle. Chapter 6 illustrates how we use a similar approach to work around the issues with impredicativity.

## 7.3 Effect handlers

### 7.3.1 In practice

Effect handlers originated in experimental programming languages such as Eff [23, 22], Effekt [38, 37], and Koka [130, 131], but have been steadily finding their way into mainstream languages; natively in OCaml 5 [183] and WebAssembly [162], and in the form of libraries in Scala 3 [203] and Haskell [171].

Effect handlers have many applications [157], including concurrency [183], control inversion [200], automatic differentiation [67], reactivity [85], model-checking [75], and probabilistic programming [152, 78].

### 7.3.2 Reasoning

**Protocols** The dominant paradigm for reasoning about effects today is the protocol-based approach, pioneered by Hazel [66, 64, 67] and Maze [64], and implemented in Cameleer [187, 185]. An extensive introduction to it was given in Section 3.1.1. In summary, the user provides two pieces of specification: a *protocol*, a contract for an effect that a client can rely on and that a handler must implement; and a *handler invariant*, a means of reasoning about the recursive nature of deep handlers. This allows modular verification in Hoare logic, but at the cost of requiring a global specification that commits to an interpretation of effects. Hazel and Maze define a protocol as a predicate transformer in Iris' logic, while Cameleer builds it in primitively, encoding an effect as a WhyML exception and using defunctionalization and axiomatisation to represent continuations and their specifications. Only Maze supports multishot continuations, with restrictions that resources cannot be framed around them.

**Refinement** An emerging approach for reasoning about effect handlers is to internalise them and talk directly about them using refinement. This is the basis of the approach introduced in Section 3.1.2 in this dissertation. The idea was pioneered by Song, Foo, and Chin [189], focusing on temporal verification for pure functional programs. It was extended with heap-manipulation by Foo, Song, and Chin [83] and to effects by Song, Foo, and Chin [190]. The key difference compared to the protocol approach are that the interpretations of effects are deferred until handlers are known and yet admit a modular, *greybox* specification; consequently, no novel specification mechanisms are required. I believe it is possible to marry both approaches, as the interpreted and uninterpreted approaches are complimentary and are each useful for different applications.

**Type-and-effect systems** Traditionally, effects have been specified via type-and-effect systems, which refine type systems with annotations, for example in the form of a set of effect labels [132]. Recent approaches are based on refinement types [113] and modal types [215, 199]. These approaches typically aim to verify less expressive properties, such as the absence of unhandled effects. Functional correctness is supported to an extent, with limitations on language features; heap-manipulation is often not handled, and advanced language features such as modules continue to be open [183].

**Monads** While fixing the composition issues with monads and transformers was a key motivation for effect handlers, monads continue to used to encode effects in proof assistants for reasoning. There are two distinct approaches: directly working with a monadic program and proving properties via equational reasoning [90, 2] or refinement [12, 198, 4, 36]; or having a computation monad (which specifies the effects) indexed by a specification monad [197, 139], to build a program logic for the former. In both cases, it is not common to internalise handlers in the object language, unlike what

we do; effect handlers are defined as folds over monadic types at the meta level instead.

## 7.4 Delimited continuations

### 7.4.1 In practice

Delimited continuation operators were classically implemented in research languages, most prominently Racket, which supports the widest variety [55], but like effect handlers, they have recently begun to enter the mainstream, most recently in Scala 3 [146] (via a program transformation), as (untyped) primitives in GHC [114], where they have replaced monads as the basis for modern effect systems [171], and as libraries in OCaml [116, 184].

It is common to implement either $\mathbf{control}_0$ (as in GHC), or a set of four lower-level operations *newPrompt*, *pushPrompt*, *takeSubCont*, and *pushSubCont* for manipulating the control stack [184, 116, 115, 76, 93] with named prompts for generality.

Many of the use cases of effects apply equally to delimited continuations. Additional ones include partial evaluation [128, 59], web interactions [91], operating systems [120], automatic differentiation [207], probabilistic programming [51], and linguistics [180].

### 7.4.2 Reasoning

**Equational reasoning and refinement** Reasoning about delimited continuations has classically been carried out either by direct appeal to their semantics in CPS [28, Section 3], or by the use of equations [109]. Encodings of this approach in proof assistants are possible using the *generalised monad* [11], which also takes an input parameter. It has been used as a means of implementing a continuation monad admitting ATM [117, 193] and can support

proofs by equational reasoning [96], at the cost of having to annotate the types of subterms, and find encodings for other kinds of effects, such as state. The approach we take with refinement sits in this category, generalising the equational approach to allow the use of specifications, and using a separation logic for stateful behaviours.

**Program logics** $\mathrm{HTT}_{cc}$ [68] is a mechanised program logic for (algebraic) callcc and throw. They use nested triples and focus on reconciling state changes with continuations, using so *large-footprint assertions*, which do not admit a general frame rule and must quantify over heaps to indicate framing.

Timany and Birkedal [202] develop a relational program logic which proposes the idea of *context-local* and *non-context-local* (Section 7.4) triples: the former are used for programs free of control operators and allow modular reasoning, while the latter are for programs with control changes and allow global reasoning reflecting the operational semantics.

$$\frac{\{\,P\,\}\,C[v]\,\{\,Q\,\}}{\{\,P\,\}\,C[\mathbf{shift}\ k.\ (k\ v)]\,\{\,Q\,\}}$$

They use contextual refinement to show that a cooperatively-concurrent program implemented using delimited control refines a specification program with primitive concurrency.

Effect handlers and (the 0 variants of) delimited continuation operators are known to be interderivable [84, 163]. The correspondence is, roughly:

$$\mathbf{shift}_0\ k.\ e_b \approx \mathbf{perform}\ \mathit{Shift}$$

$$\langle e \rangle \approx \mathbf{try}\ e\ \mathbf{with}\ \mathit{Shift}, k \to e_b$$

de Vilhena [64] exploits this to adapt their protocol-based program logic for effect handlers to callcc and throw; more on this in Protocols. The converse has been done by Cong and Asai [54], but for a type system.

**Type systems** Danvy and Filinski [60] gave a type system supporting the ATM of shift and reset soon after they were proposed. It turns the typical typing judgment $\Gamma \vdash e : \tau$ into a 5-place version $\Gamma, \alpha \vdash e : \tau, \beta$, where $\alpha$ and $\beta$ track the (input and output) *answer types* of an expression $e$ (i.e., the type of an enclosing reset), which may be modified by the execution of the $e$. Asai and Kameyama [9] generalise it with *answer type polymorphism*, and implement it in OchaCaml [141], a variant of OCaml which tracks the answer types of functions using an arrow of the form $\tau_1/\alpha \to \tau_2/\beta$. Cong and Asai [53] extend this to a dependently typed system.

Sekiyama and Unno [179] propose *answer effect modification*, which generalises answer types with refinements, allowing their use for reasoning about delimited control operators. Kawamata et al. [113] adapts this idea to effect handlers. This approach is currently limited to pure programs.

**Protocols** The protocol approach can be adapted to handle (un)delimited control operators. Rules for *callcc* and *throw* can be found in de Vilhena [64, Section 6.3.2]. They are reproduced here to illustrate the key ideas. First, recall their reduction rules.

$$K[\mathit{callcc}\ k\ e] \rightsquigarrow K[e[\tilde{K}/k]] \qquad K[\mathit{throw}\ \tilde{K}\ v] \rightsquigarrow \tilde{K}[v]$$

With these in mind,

$$\textsc{callcc}\ \frac{\forall\, k.\ \mathit{isCont}\ k\ \Phi \mathrel{-\!\!*} \mathit{ewp}\ e\ \langle CT\rangle\{\Phi\}}{\mathit{ewp}\ (\mathit{callcc}\ k\ e)\ \langle CT\rangle\{\Phi\}} \qquad \textsc{throw}\ \frac{\mathit{isCont}\ k\ \Phi \qquad \Phi\ w}{\mathit{ewp}\ (\mathit{throw}\ k\ w)\ \langle CT\rangle\{\mathit{false}\}}$$

to prove that a use of *callcc* satisfies a postcondition $\Phi$, we must prove that the body $e$ satisfies $\Phi$ under the assumption $\mathit{isCont}\ k\ \Phi$: that $k$ is a continuation whose *input* satisfies $\Phi$. This is because the result of $\mathit{callcc}\ k\ e$ is the result of $e$ with $k$ bound (so both must satisfy $\Phi$), and this result will be plugged into the continuation of $\mathit{callcc}\ k\ e$, $k$, therefore its input must also satisfy $\Phi$. *CT* is

a protocol allowing the use of the *callcc* and *throw* effects, which underlies this encoding.

The rule for *throw* $k$ $w$ simply requires that $k$ is a continuation whose input $w$ satisfies $\Phi$. Its postcondition is *false* because *throw* does not produce a result, can be used in any context, and *escapes* this context by resuming $k$.

The extension to shift/reset [170, pages 87-88] builds on these ideas. The protocol *CT* is abstracted away, and *ewp* is replaced with a predicate $wp\ e\ \langle Some\ \Phi\rangle\{\Phi'\}$, a weakest precondition predicate with an "answer postcondition" $\Phi$.

$$\text{SHIFT}\quad \frac{\square\,(\forall\, w.\ \Phi'(w) \mathrel{-\!\!*} wp\ (k\ w)\ \langle\_\rangle\{\Phi\}) \mathrel{-\!\!*} wp\ e\ \langle Some\ \Phi\rangle\{\Phi'\}}{wp\ (shift\ k\ e)\ \langle Some\ \Phi\rangle\{\Phi'\}} \qquad \text{RESET}\quad \frac{wp\ e\ \langle Some\ \Phi\rangle\{\Phi\}}{wp\ (reset\ e)\ \langle\_\rangle\{\Phi\}}$$

$\Phi$ describes 1. the postcondition of the enclosing reset, and also the input to the metacontinuation, and 2. the postcondition of $k$. $\Phi'$ is the postcondition of shift and the shift body, which also constrains the *input* to $k$. "Answer postcondition modification" is hence possible, as $\Phi$ and $\Phi'$ can be different. $k$ has no effects (as it is enclosed in a reset, per the semantics of shift) and can be used many times in $e$.

The rule for *reset* handles all "effects" and supplies the "answer postcondition" $\Phi$.

Comparing the approach once again with staged logic, the main and fundamental difficulty with using protocols is that one has to come up with $\Phi$ and $\Phi'$ upfront, similar to how one has to state invariants upfront (Section 3.1).

Secondly, the extension to `control` not obvious. It is not just a matter of adding an operational semantics for it, as in staged logic; we would also need a protocol, which seems nontrivial to write. I am not aware of any program logics for `control`, so it is unclear what the protocol should be parametrised over.

CHAPTER 8

# CONCLUSION

In this dissertation, I have described *staged logic*, a comprehensive and novel approach to using refinement to reason about effectful higher-order programs. Staged logic underlies the implementation of a new automated verifier for OCaml programs, HEIFER, and is mechanised in Rocq. I have shown that it is feasible to automate, by verifying a number of challenging programs with it, as well as that it is sound, via the mechanisation.

**Future work** There are several avenues for future work.

- Section 6.2 details the current state of the mechanisation, and Section 6.2.3 gives the next steps for completing it, with the goal of enabling interactive use in Rocq.
- A certification back-end for HEIFER, to allow it to generate certificates for the Rocq mechanisation, would usefully integrate the two tools, and also confirm the soundness of the former.
- Exploring the use of staged logic in new domains, such as relational verification and concurrency. This combination is particularly interesting due to the use of effect handlers to model concurrency, and the difficulty of verifying such libraries built in this way today.

# BIBLIOGRAPHY


[1] *Add effect syntax*. https://github.com/ocaml/ocaml/pull/12309 (Cited on page 51).

[2] Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. "A practical formalization of monadic equational reasoning in dependent-type theory". In: *J. Funct. Program.* 35 (2025). DOI: 10.1017/S0956796824000157. URL: https://doi.org/10.1017/s0956796824000157 (Cited on pages 134, 142).

[3] Amal Jamil Ahmed. *Semantics of types for mutable state*. Princeton University, 2004 (Cited on pages 88, 133).

[4] João Alpuim and Wouter Swierstra. "Embedding the refinement calculus in Coq". In: *Sci. Comput. Program.* 164 (2018), pp. 37–48. DOI: 10.1016/J.SCICO.2017.04.003. URL: https://doi.org/10.1016/j.scico.2017.04.003 (Cited on pages 134, 142).

[5] Timos Antonopoulos et al. "An Algebra of Alignment for Relational Verification". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 573–603. DOI: 10.1145/3571213. URL: https://doi.org/10.1145/3571213 (Cited on page 136).

[6] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN: 978-1-10-704801-0. URL: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB (Cited on page 140).

[7] Andrew W. Appel. "Verified Software Toolchain". In: *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Ed. by Alwyn Goodloe and Suzette Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012,

p. 2. DOI: 10.1007/978-3-642-28891-3_2. URL: https://doi.org/10.1007/978-3-642-28891-3_2 (Cited on pages 131, 140).

[8] Andrew W. Appel and David A. McAllester. “An indexed model of recursive types for foundational proof-carrying code”. In: *ACM Trans. Program. Lang. Syst.* 23.5 (2001), pp. 657–683. DOI: 10.1145/504709.504712. URL: https://doi.org/10.1145/504709.504712 (Cited on page 133).

[9] Kenichi Asai and Yukiyoshi Kameyama. “Polymorphic Delimited Continuations”. In: *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*. Ed. by Zhong Shao. Vol. 4807. Lecture Notes in Computer Science. Springer, 2007, pp. 239–254. DOI: 10.1007/978-3-540-76637-7_16. URL: https://doi.org/10.1007/978-3-540-76637-7_16 (Cited on page 145).

[10] Kenichi Asai and Oleg Kiselyov. “Introduction to Programming with Shift and Reset”. In: 2011. URL: https://api.semanticscholar.org/CorpusID:62358634 (Cited on page 86).

[11] Robert Atkey. “Parameterised notions of computation”. In: *J. Funct. Program.* 19.3-4 (2009), pp. 335–376. DOI: 10.1017/S095679680900728X. URL: https://doi.org/10.1017/S095679680900728X (Cited on page 143).

[12] Tim Baanen. “Algebraic effects, specification and refinement”. MA thesis. 2019 (Cited on pages 134, 142).

[13] Ralph-Johan Back. “A Calculus of Refinements for Program Derivations”. In: *Acta Informatica* 25.6 (1988), pp. 593–624. DOI: 10.1007/BF00291051. URL: https://doi.org/10.1007/BF00291051 (Cited on page 133).

[14] Ralph-Johan Back. *On the correctness of refinement steps in program development*. Department of Computer Science, University of Helsinki Helsinki, Finland, 1978 (Cited on page 133).

[15] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. ISBN: 978-0-387-98417-9. DOI: 10.1007/978-1-4612-1674-2. URL: https://doi.org/10.1007/978-1-4612-1674-2 (Cited on page 119).

[16] Patrycja Balik, Dariusz Biernacki, and Piotr Polesiuk. "Untyped Logical Relations at Work: Control Operators, Contextual Equivalence and Full Abstraction". In: *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday*. OLIVIERFEST '25. Singapore, Singapore: Association for Computing Machinery, 2025, pp. 128–141. ISBN: 9798400721502. DOI: 10.1145/3759427.3760374. URL: https://doi.org/10.1145/3759427.3760374 (Cited on pages 124, 125, 132).

[17] Michael Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17. URL: https://doi.org/10.1007/11804192_17 (Cited on page 129).

[18] Clark W. Barrett et al. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1_14 (Cited on page 48).

[19] Gilles Barthe. *An introduction to relational program verification*. 2020 (Cited on page 135).

[20] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification". In: *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*. Ed. by Sergei N. Artëmov and Anil Nerode. Vol. 7734. Lecture Notes in Computer Science. Springer, 2013, pp. 29–43. DOI: 10.1007/978-3-642-35722-0_3. URL: https://doi.org/10.1007/978-3-642-35722-0_3 (Cited on page 135).

[21] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational Verification Using Product Programs". In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Ed. by Michael J. Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17. URL: https://doi.org/10.1007/978-3-642-21437-0_17 (Cited on page 135).

[22] Andrej Bauer and Matija Pretnar. *Eff*. http://www.eff-lang.org/. 2020 (Cited on page 141).

[23] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *J. Log. Algebraic Methods Program.* 84.1 (2015), pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001. URL: https://doi.org/10.1016/j.jlamp.2014.02.001 (Cited on page 141).

[24] Bernhard Beckert and Mattias Ulbrich. "Trends in Relational Program Verification". In: *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*. Ed. by Peter Müller and Ina Schaefer. Springer, 2018, pp. 41–58. DOI: 10.

1007/978-3-319-98047-8_3. URL: https://doi.org/10.1007/978-3-319-98047-8_3 (Cited on page 135).

[25] Nick Benton. "Simple relational correctness proofs for static analyses and program transformations". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 14–25. DOI: 10.1145/964001.964003. URL: https://doi.org/10.1145/964001.964003 (Cited on pages 104, 133, 135).

[26] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 115–137. DOI: 10.1007/11804192_6. URL: https://doi.org/10.1007/11804192_6 (Cited on page 130).

[27] Karthikeyan Bhargavan et al. "Everest: Towards a Verified, Drop-in Replacement of HTTPS". In: *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 1:1–1:12. DOI: 10.4230/LIPICS.SNAPL.2017.1. URL: https://doi.org/10.4230/LIPIcs.SNAPL.2017.1 (Cited on page 2).

[28] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. "An Operational Foundation for Delimited Continuations in the CPS Hierarchy". In: *Logical Methods in Computer Science* 1.2:5 (Nov. 2005), pp. 1–39 (Cited on pages 76, 143).

[29] Dariusz Biernacki and Olivier Danvy. “Theoretical Pearl: A simple proof of a folklore theorem about delimited control”. In: *J. Funct. Program.* 16.3 (2006), pp. 269–280. DOI: 10.1017/S0956796805005782. URL: https://doi.org/10.1017/S0956796805005782 (Cited on page 76).

[30] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. “A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations”. In: *ACM Trans. Program. Lang. Syst.* 38.1 (2015), 2:1–2:25. DOI: 10.1145/2794078. URL: https://doi.org/10.1145/2794078 (Cited on pages 76, 82).

[31] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. “On the Static and Dynamic Extents of Delimited Continuations”. In: 60.3 (2006), pp. 274–297 (Cited on pages 76, 82).

[32] Richard S. Bird and Philip Wadler. *Introduction to functional programming*. Prentice Hall International series in computer science. Prentice Hall, 1988. ISBN: 978-0-13-484197-7 (Cited on page 112).

[33] Lars Birkedal and Aleš Bizjak. “Lecture Notes on Iris: Higher-order Concurrent Separation Logic”. In: *Lectures notes, September* (2023). URL: https://iris-project.org/tutorial-material.html (Cited on pages 3, 13–15, 88, 90, 91, 137).

[34] Lars Birkedal, Ales Bizjak, and Jan Schwinghammer. “Step-Indexed Relational Reasoning for Countable Nondeterminism”. In: *Log. Methods Comput. Sci.* 9.4 (2013). DOI: 10.2168/LMCS-9(4:4)2013. URL: https://doi.org/10.2168/LMCS-9(4:4)2013 (Cited on page 126).

[35] Olivier Bouissou et al. “Space software validation using abstract interpretation”. In: *The International Space System Engineering Conference: Data Systems in Aerospace-DASIA 2009*. Vol. 1. European Space Agency. 2009, pp. 1–7 (Cited on page 1).

[36] Sylvain Boulmé. “Intuitionistic Refinement Calculus”. In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 54–69. DOI: 10.1007/978-3-540-73228-0_6. URL: https://doi.org/10.1007/978-3-540-73228-0_6 (Cited on pages 134, 137, 142).

[37] Jonathan Immanuel Brachthäuser and Philipp Schuster. “Effekt: extensible algebraic effects in Scala (short paper)”. In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*. Ed. by Heather Miller, Philipp Haller, and Ondrej Lhoták. ACM, 2017, pp. 67–72. DOI: 10.1145/3136000.3136007. URL: https://doi.org/10.1145/3136000.3136007 (Cited on page 141).

[38] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. “Effects as capabilities: effect handlers and lightweight effect polymorphism”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 126:1–126:30. DOI: 10.1145/3428194. URL: https://doi.org/10.1145/3428194 (Cited on page 141).

[39] Martin Brain and Elizabeth Polgreen. “A Pyramid Of (Formal) Software Verification”. In: *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*. Ed. by André Platzer et al. Vol. 14934. Lecture Notes in Computer Science. Springer, 2024, pp. 393–419. DOI: 10.1007/978-3-031-71177-0_24. URL: https://doi.org/10.1007/978-3-031-71177-0_24 (Cited on page 1).

[40] Martin Büchi and Wolfgang Weck. *The Greybox Approach: When Blackbox Specification Hide too much*. Turku Centre for Computer Science, 1999 (Cited on page 137).

[41] Cristiano Calcagno et al. “Compositional Shape Analysis by Means of Bi-Abduction”. In: *J. ACM* 58.6 (2011), 26:1–26:66. DOI: 10.1145/2049697.2049700. URL: https://doi.org/10.1145/2049697.2049700 (Cited on page 43).

[42] Cristiano Calcagno et al. “Compositional Shape Analysis by means of Bi-abduction”. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 289–300. DOI: 10.1145/1480881.1480917. URL: https://doi.org/10.1145/1480881.1480917 (Cited on pages 19, 42, 130).

[43] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. “Refinement: An overview”. In: *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004, Revised Lectures*. Ed. by Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Vol. 3167. Lecture Notes in Computer Science. Springer, 2004, pp. 1–17. DOI: 10.1007/11889229_1. URL: https://doi.org/10.1007/11889229_1 (Cited on page 133).

[44] Aleks Chakarov et al. “Formally verified cloud-scale authorization”. In: (2025). URL: https://www.amazon.science/publications/formally-verified-cloud-scale-authorization (Cited on page 2).

[45] Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs. (Un nouveau regard sur la Logique de Séparation pour les programmes séquentiels)*. 2023. URL: https://tel.archives-ouvertes.fr/tel-04076725 (Cited on page 131).

[46] Arthur Charguéraud. “Characteristic formulae for the verification of imperative programs”. In: *Proceeding of the 16th ACM SIGPLAN in-*

*ternational conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 418–430. DOI: 10.1145/2034773.2034828. URL: https://doi.org/10.1145/2034773.2034828 (Cited on pages 131, 137, 139).

[47] Arthur Charguéraud. *Separation Logic Foundations*. 2024 (Cited on pages 116, 122, 126, 131, 139).

[48] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. "Crowfoot: A Verifier for Higher-Order Store Programs". In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Springer, 2012, pp. 136–151. DOI: 10.1007/978-3-642-27940-9_10. URL: https://doi.org/10.1007/978-3-642-27940-9_10 (Cited on page 137).

[49] Wei-Ngan Chin et al. "Automated verification of shape, size and bag properties via user-defined predicates in separation logic". In: *Sci. Comput. Program.* 77.9 (2012), pp. 1006–1036. DOI: 10.1016/J.SCICO.2010.07.004. URL: https://doi.org/10.1016/j.scico.2010.07.004 (Cited on page 130).

[50] Adam Chlipala. "A verified compiler for an impure functional language". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 93–106. DOI: 10.1145/1706299.1706312. URL: https://doi.org/10.1145/1706299.1706312 (Cited on page 140).

[51] Alexander Collins and Vinod Grover. “Probabilistic Programming with CuPPL”. In: *CoRR* abs/2010.08454 (2020). arXiv: 2010.08454. URL: https://arxiv.org/abs/2010.08454 (Cited on page 143).

[52] Commenters. *Multi-shot continuations gone forever?* https://discuss.ocaml.org/t/multi-shot-continuations-gone-forever/9072. 2022 (Cited on page 53).

[53] Youyou Cong and Kenichi Asai. “Handling delimited continuations with dependent types”. In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 69:1–69:31. DOI: 10.1145/3236764. URL: https://doi.org/10.1145/3236764 (Cited on page 145).

[54] Youyou Cong and Kenichi Asai. “Understanding Algebraic Effect Handlers via Delimited Control Operators”. In: *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*. Ed. by Wouter Swierstra and Nicolas Wu. Vol. 13401. Lecture Notes in Computer Science. Springer, 2022, pp. 59–79. DOI: 10.1007/978-3-031-21314-4_4. URL: https://doi.org/10.1007/978-3-031-21314-4_4 (Cited on page 144).

[55] *Continuations*. https://docs.racket-lang.org/reference/cont.html (Cited on pages 76, 143).

[56] Emanuele D’Osualdo, Azadeh Farzan, and Derek Dreyer. “Proving hypersafety compositionally”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 289–314. DOI: 10.1145/3563298. URL: https://doi.org/10.1145/3563298 (Cited on pages 112, 114).

[57] Olivier Danvy. “Folding left and right matters: Direct style, accumulators, and continuations”. In: *J. Funct. Program.* 33 (2023), e2. DOI: 10.1017/S0956796822000156. URL: https://doi.org/10.1017/S0956796822000156 (Cited on page 112).

[58] Olivier Danvy. “Functional Unparsing”. In: *Journal of Functional Programming* 8.6 (1998), pp. 621–625 (Cited on page 71).

[59] Olivier Danvy. “Type-Directed Partial Evaluation”. In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 242–257. DOI: 10.1145/237721.237784. URL: https://doi.org/10.1145/237721.237784 (Cited on page 143).

[60] Olivier Danvy and Andrzej Filinski. *A Functional Abstraction of Typed Contexts*. DIKU Rapport 89/12. Copenhagen, Denmark: DIKU, Computer Science Department, University of Copenhagen, July 1989 (Cited on pages 71, 72, 76, 145).

[61] Olivier Danvy and Andrzej Filinski. “Abstracting Control”. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. Ed. by Gilles Kahn. ACM, 1990, pp. 151–160. DOI: 10.1145/91556.91622. URL: https://doi.org/10.1145/91556.91622 (Cited on pages 70, 76, 86, 95, 125).

[62] Olivier Danvy and Lasse R. Nielsen. “Defunctionalization at Work”. In: *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, September 5-7, 2001, Florence, Italy*. ACM, 2001, pp. 162–174. DOI: 10.1145/773184.773202. URL: https://doi.org/10.1145/773184.773202 (Cited on pages 93, 96).

[63] Thibault Dardinier and Peter Müller. “Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties”. In: *Proc. ACM Program. Lang.* 8.PLDI (2024),

pp. 1485–1509. DOI: 10.1145/3656437. URL: https://doi.org/10.1145/3656437 (Cited on page 135).

[64] Paulo de Vilhena. “Proof of Programs with Effect Handlers. (Preuve de Programmes avec Effect Handlers)”. PhD thesis. Paris Cité University, France, 2022. URL: https://tel.archives-ouvertes.fr/tel-03891381 (Cited on pages 51–54, 141, 144, 145).

[65] Paulo de Vilhena. “The Theory and Practice of First-Class Prompts”. In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Ed. by Jeanne Ferrante and Peter Mager. ACM Press, 1988, pp. 180–190. DOI: 10.1145/73560.73576. URL: https://doi.org/10.1145/73560.73576 (Cited on page 76).

[66] Paulo Emílio de Vilhena and François Pottier. “A separation logic for effect handlers”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28. DOI: 10.1145/3434314. URL: https://doi.org/10.1145/3434314 (Cited on pages 3, 49, 52, 54, 68, 141).

[67] Paulo Emílio de Vilhena and François Pottier. “Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library”. In: *Log. Methods Comput. Sci.* 19.4 (2023). DOI: 10.46298/LMCS-19(4:5)2023. URL: https://doi.org/10.46298/lmcs-19(4:5)2023 (Cited on page 141).

[68] Germán Andrés Delbianco and Aleksandar Nanevski. “Hoare-style reasoning with (algebraic) continuations”. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 363–376. DOI: 10.1145/2500365.2500593. URL: https://doi.org/10.1145/2500365.2500593 (Cited on pages 54, 86, 144).

[69] Xavier Denis and Jacques-Henri Jourdan. “Specifying and Verifying Higher-order Rust Iterators”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 93–110. DOI: 10.1007/978-3-031-30820-8_9. URL: https://doi.org/10.1007/978-3-031-30820-8_9 (Cited on pages 3, 47, 138, 139).

[70] Robert Dickerson, Prasita Mukherjee, and Benjamin Delaware. “KestRel: Relational Verification using E-Graphs for Program Alignment”. In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (2025), pp. 1073–1100. DOI: 10.1145/3720474. URL: https://doi.org/10.1145/3720474 (Cited on pages 104, 136).

[71] Robert Dickerson et al. “RHLE: Modular Deductive Verification of Relational ∀ ∃ Properties”. In: *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13658. Lecture Notes in Computer Science. Springer, 2022, pp. 67–87. DOI: 10.1007/978-3-031-21037-2_4. URL: https://doi.org/10.1007/978-3-031-21037-2%5C_4 (Cited on page 135).

[72] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (1975), pp. 453–457. DOI: 10.1145/360933.360975. URL: https://doi.org/10.1145/360933.360975 (Cited on page 129).

[73] *Documentation of Closures*. https://github.com/viperproject/prusti-dev/issues/1431. 2024 (Cited on page 47).

[74] Derek Dreyer, Amal Ahmed, and Lars Birkedal. “Logical Step-Indexed Logical Relations”. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, 2009, pp. 71–80. DOI: 10.1109/LICS.2009.34. URL: https://doi.org/10.1109/LICS.2009.34 (Cited on pages 124, 125, 133).

[75] *DSCheck - tool for testing concurrent OCaml programs*. https://github.com/ocaml-multicore/dscheck (Cited on page 141).

[76] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. “A monadic framework for delimited continuations”. In: *J. Funct. Program.* 17.6 (2007), pp. 687–730. DOI: 10.1017/S0956796807006259. URL: https://doi.org/10.1017/S0956796807006259 (Cited on pages 75, 76, 143).

[77] Gabriel Ebner et al. “PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs”. In: *2025 Programming Language Design and Implementation*. Accepted for publication, to appear. ACM. 2025. URL: https://www.microsoft.com/en-us/research/publication/pulsecore-an-impredicative-concurrent-separation-logic-for-dependently-typed-programs/ (Cited on pages 3, 131, 140).

[78] *EffPPL*. https://github.com/Arnhav-Datar/EffPPL (Cited on page 141).

[79] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 - Where Programs Meet Provers”. In: *European Symposium on Programming*. 2013. URL: https://api.semanticscholar.org/CorpusID:14572425 (Cited on pages 48, 129, 137).

[80] Robert Bruce Findler and Matthias Felleisen. “Contracts for Higher-order Functions”. In: *Proceedings of the Seventh ACM SIGPLAN In-*

*ternational Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 48–59. DOI: 10.1145/581478.581484. URL: https://doi.org/10.1145/581478.581484 (Cited on page 46).

[81] Darius Foo and Wei-Ngan Chin. "Tracing OCaml Programs". In: *CoRR* abs/2304.04937 (2023). DOI: 10.48550/ARXIV.2304.04937. arXiv: 2304.04937. URL: https://doi.org/10.48550/arXiv.2304.04937 (Cited on page 7).

[82] Darius Foo, Andreea Costea, and Wei-Ngan Chin. "Protocol Conformance with Choreographic PlusCal". In: *Theoretical Aspects of Software Engineering - 17th International Symposium, TASE 2023, Bristol, UK, July 4-6, 2023, Proceedings*. Ed. by Cristina David and Meng Sun. Vol. 13931. Lecture Notes in Computer Science. Springer, 2023, pp. 126–145. DOI: 10.1007/978-3-031-35257-7_8. URL: https://doi.org/10.1007/978-3-031-35257-7_8 (Cited on page 7).

[83] Darius Foo, Yahui Song, and Wei-Ngan Chin. "Staged Specification Logic for Verifying Higher-Order Imperative Programs". In: *FM 2024: Formal Methods - 17th International Symposium on Formal Methods, Milan, Italy, Sept 9-13, 2024. Proceedings*. Lecture Notes in Computer Science. Springer, 2024 (Cited on pages 7, 22, 23, 37, 41, 44, 142).

[84] Yannick Forster et al. "On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control". In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 13:1–13:29. DOI: 10.1145/3110257. URL: https://doi.org/10.1145/3110257 (Cited on page 144).

[85] *Friendship ended with Monads: Testing out Algebraic effects in OCaml for Animations*. https://gopiandcode.uk/logs/log-bye-bye-monads-algebraic-effects.html (Cited on page 141).

[86] Aymeric Fromherz et al. "Steel: proof-oriented programming in a dependently typed concurrent separation logic". In: *Proceedings of the ACM on Programming Languages* 5 (2021), pp. 1–30. URL: https://api.semanticscholar.org/CorpusID:233474595 (Cited on page 131).

[87] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity". In: *Log. Methods Comput. Sci.* 17.3 (2021). DOI: 10.46298/LMCS-17(3:9)2021. URL: https://doi.org/10.46298/lmcs-17(3:9)2021 (Cited on pages 131, 133).

[88] Lennard Gäher et al. "Simuliris: a separation logic framework for verifying concurrent program optimizations". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498689. URL: https://doi.org/10.1145/3498689 (Cited on page 133).

[89] Cristian Gherghina et al. "Expressive program verification via structured specifications". In: *Int. J. Softw. Tools Technol. Transf.* 16.4 (2014), pp. 363–380. DOI: 10.1007/S10009-014-0306-5. URL: https://doi.org/10.1007/s10009-014-0306-5 (Cited on page 4).

[90] Jeremy Gibbons and Ralf Hinze. "Just do it: simple monadic equational reasoning". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 2–14. DOI: 10.1145/2034773.2034777. URL: https://doi.org/10.1145/2034773.2034777 (Cited on pages 134, 142).

[91] Paul T. Graunke et al. “Modeling Web Interactions”. In: *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Pierpaolo Degano. Vol. 2618. Lecture Notes in Computer Science. Springer, 2003, pp. 238–252. DOI: 10.1007/3-540-36575-3_17. URL: https://doi.org/10.1007/3-540-36575-3_17 (Cited on page 143).

[92] Ronghui Gu et al. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels”. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 653–669. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu (Cited on page 2).

[93] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. “A Generalization of Exceptions and Control in ML-like Languages”. In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. Ed. by John Williams. ACM, 1995, pp. 12–23. DOI: 10.1145/224164.224173. URL: https://doi.org/10.1145/224164.224173 (Cited on pages 76, 143).

[94] Chris Hawblitzel et al. “IronFleet: proving safety and liveness of practical distributed systems”. In: *Commun. ACM* 60.7 (2017), pp. 83–92. DOI: 10.1145/3068608. URL: https://doi.org/10.1145/3068608 (Cited on page 2).

[95] Daniel Hillerström and Sam Lindley. “Shallow Effect Handlers”. In: *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*. Ed.

by Sukyoung Ryu. Vol. 11275. Lecture Notes in Computer Science. Springer, 2018, pp. 415–435. DOI: 10.1007/978-3-030-02768-1_22. URL: https://doi.org/10.1007/978-3-030-02768-1_22 (Cited on page 51).

[96] Noriko Hirota and Kenichi Asai. "Correctness of functions with shift and reset". In: *Proceedings of the 2011 ACM SIGPLAN Continuation Workshop*. 2011, pp. 23–25. URL: https://www.cs.tsukuba.ac.jp/~kam/cw2011/book.pdf (Cited on pages 78, 144).

[97] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (Cited on page 129).

[98] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. "A theory of indirection via approximation". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 171–184. DOI: 10.1145/1706299.1706322. URL: https://doi.org/10.1145/1706299.1706322 (Cited on pages 126, 131, 140).

[99] Aquinas Hobor and Jules Villard. "The ramifications of sharing in data structures". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 523–536. DOI: 10.1145/2429069.2429131. URL: https://doi.org/10.1145/2429069.2429131 (Cited on page 131).

[100] Dániel Horpácsi, Péter Bereczky, and Simon J. Thompson. "Program equivalence in an untyped, call-by-value functional language with uncurried functions". In: *J. Log. Algebraic Methods Program.* 132 (2023),

p. 100857. DOI: 10.1016/J.JLAMP.2023.100857. URL: https://doi.org/10.1016/j.jlamp.2023.100857 (Cited on page 124).

[101] Nikolaus Huber et al. “Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III*. Ed. by Arie Gurfinkel and Marijn Heule. Vol. 15698. Lecture Notes in Computer Science. Springer, 2025, pp. 3–22. DOI: 10.1007/978-3-031-90660-2_1. URL: https://doi.org/10.1007/978-3-031-90660-2_1 (Cited on page 129).

[102] R. John M. Hughes. “A Novel Representation of Lists and its Application to the Function "reverse"”. In: *Inf. Process. Lett.* 22.3 (1986), pp. 141–144. DOI: 10.1016/0020-0190(86)90059-1. URL: https://doi.org/10.1016/0020-0190(86)90059-1 (Cited on page 72).

[103] Martin Hyland et al. “Combining algebraic effects with continuations”. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 20–40. DOI: 10.1016/J.TCS.2006.12.026. URL: https://doi.org/10.1016/j.tcs.2006.12.026 (Cited on page 134).

[104] *Implementing simple generators with effect handlers*. https://discuss.ocaml.org/t/implementing-simple-generators-with-effect-handlers/16020/7 (Cited on page 51).

[105] Chiaki Ishio and Kenichi Asai. “Type System for Four Delimited Control Operators”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*. Ed. by Bernhard Scholz and Yukiyoshi Kameyama. ACM, 2022, pp. 45–

58. DOI: 10.1145/3564719.3568691. URL: https://doi.org/10.1145/3564719.3568691 (Cited on page 75).

[106] Bart Jacobs. "VeriFast's separation logic: a higher-order(ish) logic without laters for modular verification of fine-grained concurrent programs". In: *CoRR* abs/2505.04500 (2025). DOI: 10.48550/ARXIV.2505.04500. arXiv: 2505.04500. URL: https://doi.org/10.48550/arXiv.2505.04500 (Cited on pages 3, 130, 140).

[107] Bart Jacobs et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4. URL: https://doi.org/10.1007/978-3-642-20398-5_4 (Cited on page 130).

[108] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151. URL: https://doi.org/10.1017/S0956796818000151 (Cited on pages 125, 126, 131, 133, 137, 140).

[109] Yukiyoshi Kameyama. "Axioms for Delimited Continuations in the CPS Hierarchy". In: *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*. Ed. by Jerzy Marcinkowski and Andrzej Tarlecki. Vol. 3210. Lecture Notes in Computer Science. Springer, 2004, pp. 442–457. DOI: 10.1007/978-3-540-30124-0_34. URL: https://doi.org/10.1007/978-3-540-30124-0_34 (Cited on page 143).

[110] Yukiyoshi Kameyama and Masahito Hasegawa. “A sound and complete axiomatization of delimited continuations”. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. Ed. by Colin Runciman and Olin Shivers. ACM, 2003, pp. 177–188. DOI: 10.1145/944705.944722. URL: https://doi.org/10.1145/944705.944722 (Cited on page 99).

[111] Yukiyoshi Kameyama and Takuo Yonezawa. “Typed Dynamic Control Operators for Delimited Continuations”. In: *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*. Ed. by Jacques Garrigue and Manuel V. Hermenegildo. Vol. 4989. Lecture Notes in Computer Science. Springer, 2008, pp. 239–254. DOI: 10.1007/978-3-540-78969-7_18. URL: https://doi.org/10.1007/978-3-540-78969-7_18 (Cited on page 81).

[112] Johannes Kanig and Jean-Christophe Filliâtre. “Who: a verifier for effectful higher-order programs”. In: *Proceedings of the 2009 ACM SIGPLAN workshop on ML*. 2009, pp. 39–48 (Cited on page 137).

[113] Fuga Kawamata et al. “Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers”. In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 115–147. DOI: 10.1145/3633280. URL: https://doi.org/10.1145/3633280 (Cited on pages 142, 145).

[114] Alexis King. *Delimited continuation primops*. https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7942. 2022 (Cited on page 143).

[115] Oleg Kiselyov. “Delimited control in OCaml, abstractly and concretely”. In: *Theor. Comput. Sci.* 435 (2012), pp. 56–76. DOI: 10.1016/J.TCS.

2012.02.025. URL: https://doi.org/10.1016/j.tcs.2012.02.025 (Cited on page 143).

[116] Oleg Kiselyov. "Delimited Control in OCaml, Abstractly and Concretely: System Description". In: *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Vol. 6009. Lecture Notes in Computer Science. Springer, 2010, pp. 304–320. DOI: 10.1007/978-3-642-12251-4_22. URL: https://doi.org/10.1007/978-3-642-12251-4_22 (Cited on page 143).

[117] Oleg Kiselyov. *Genuine shift/reset in Haskell98*. https://okmij.org/ftp/continuations/implementations.html. 2007 (Cited on page 143).

[118] Oleg Kiselyov. *Three new monad transformers for multi-prompt delimited control*. https://okmij.org/ftp/continuations/implementations.html#CC-monads. 2010 (Cited on pages 62, 83).

[119] Oleg Kiselyov. *Undelimited continuations do not occur in practice*. https://okmij.org/ftp/continuations/against-callcc.html#illusion. 2012 (Cited on page 84).

[120] Oleg Kiselyov and Chung-chieh Shan. "Delimited Continuations in Operating Systems". In: *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007, Roskilde, Denmark, August 20-24, 2007, Proceedings*. Ed. by Boicho N. Kokinov et al. Vol. 4635. Lecture Notes in Computer Science. Springer, 2007, pp. 291–302. DOI: 10.1007/978-3-540-74255-5_22. URL: https://doi.org/10.1007/978-3-540-74255-5_22 (Cited on page 143).

[121] Oleg Kiselyov and Chung-chieh Shan. "Embedded Probabilistic Programming". In: *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009, Oxford, UK, July 15-17, 2009, Proceedings*. Ed. by

Walid Mohamed Taha. Vol. 5658. Lecture Notes in Computer Science. Springer, 2009, pp. 360–384. DOI: 10.1007/978-3-642-03034-5_17. URL: https://doi.org/10.1007/978-3-642-03034-5_17 (Cited on page 50).

[122] Gerwin Klein et al. "seL4: formal verification of an OS kernel." In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596 (Cited on page 2).

[123] Joomy Korkut. *Intrinsic vs. extrinsic verification*. https://joomy.korkutblech.com/posts/2024-10-30-intrinsic-vs-extrinsic-verification.html. 2024 (Cited on page 131).

[124] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217. DOI: 10.1145/3009837.3009855. URL: https://doi.org/10.1145/3009837.3009855 (Cited on page 41).

[125] Leslie Lamport. *Specifying systems*. Vol. 388. Addison-Wesley Boston, 2002 (Cited on page 132).

[126] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *Comput. J.* 6.4 (1964), pp. 308–320. DOI: 10.1093/COMJNL/6.4.308. URL: https://doi.org/10.1093/comjnl/6.4.308 (Cited on page 88).

[127] Andrea Lattuada et al. "Verus: Verifying Rust Programs using Linear Ghost Types". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023),

pp. 286–315. DOI: 10.1145/3586037. URL: https://doi.org/10.1145/3586037 (Cited on page 137).

[128] Julia L. Lawall and Olivier Danvy. “Continuation-Based Partial Evaluation”. In: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*. Ed. by Robert R. Kessler. ACM, 1994, pp. 227–238. DOI: 10.1145/182409.182483. URL: https://doi.org/10.1145/182409.182483 (Cited on page 143).

[129] Gary T. Leavens et al. “Specifying and Verifying Advanced Control Features”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9953. Lecture Notes in Computer Science. 2016, pp. 80–96. DOI: 10.1007/978-3-319-47169-3_7. URL: https://doi.org/10.1007/978-3-319-47169-3_7 (Cited on pages 136, 137).

[130] Daan Leijen. *Algebraic effects for functional programming*. Tech. rep. Microsoft, 2016 (Cited on page 141).

[131] Daan Leijen. *Koka*. https://www.microsoft.com/en-us/research/project/koka/. 2012 (Cited on page 141).

[132] Daan Leijen. “Koka: Programming with Row Polymorphic Effect Types”. In: *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*. Ed. by Paul Blain Levy and Neel Krishnaswami. Vol. 153. EPTCS. 2014, pp. 100–126. DOI: 10.4204/EPTCS.153.8. URL: https://doi.org/10.4204/EPTCS.153.8 (Cited on page 142).

[133] Daan Leijen. “Structured Asynchrony with Algebraic Effects”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-*

*Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*. Ed. by Sam Lindley and Brent A. Yorgey. ACM, 2017, pp. 16–29. DOI: 10.1145/3122975.3122977. URL: https://doi.org/10.1145/3122975.3122977 (Cited on page 50).

[134] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: https://doi.org/10.1007/978-3-642-17511-4_20 (Cited on pages 3, 129, 137).

[135] Xavier Leroy et al. "CompCert – A formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016 (Cited on page 2).

[136] Stéphane Lescuyer. "Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)". PhD thesis. University of Paris-Sud, Orsay, France, 2011. URL: https://tel.archives-ouvertes.fr/tel-00713668 (Cited on page 48).

[137] Benjamin Livshits et al. "In defense of soundiness: a manifesto". In: *Commun. ACM* 58.2 (2015), pp. 44–46. DOI: 10.1145/2644805. URL: https://doi.org/10.1145/2644805 (Cited on page 1).

[138] Andreas Lööw et al. "Compositional Symbolic Execution for Correctness and Incorrectness Reasoning". In: *38th European Conference on Object-Oriented Programming, ECOOP 2024, Vienna, Austria, September 16-20, 2024*. Ed. by Jonathan Aldrich and Guido Salvaneschi.

LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 25:1–25:28. DOI: 10.4230/LIPICS.ECOOP.2024.25. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2024.25 (Cited on page 21).

[139] Kenji Maillard. "Principles of Program Verification for Arbitrary Monadic Effects. (Principes de la Vérification de Programmes à Effets Monadiques Arbitraires)". In: 2019 (Cited on page 142).

[140] Kenji Maillard et al. "Dijkstra Monads for All". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 104:1–104:29. DOI: 10.1145/3341708. URL: https://doi.org/10.1145/3341708 (Cited on pages 3, 131, 135, 139).

[141] Moe Masuko and Kenichi Asai. "Caml Light + shift/reset = Caml Shift". In: *Theory and Practice of Delimited Continuations (TPDC 2011)* (2011), pp. 33–46 (Cited on page 145).

[142] Marek Materzok and Dariusz Biernacki. "Subtyping delimited continuations". In: *Proceeding of the 16th ACM SIGPLAN intrernational conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 81–93. DOI: 10.1145/2034773.2034786. URL: https://doi.org/10.1145/2034773.2034786 (Cited on pages 76, 83, 86).

[143] Philip McGrath. *Racket comes with a continuation-based web server framework that makes extensive use of resuming continuations multiple times*. https://github.com/WebAssembly/stack-switching/issues/110#issuecomment-2927824629. 2025 (Cited on page 50).

[144] *Modular Specification and Verification of Closures in Rust (artefact)*. https://zenodo.org/records/5482557. 2021 (Cited on pages 46, 47).

[145] Eugenio Moggi. “Notions of Computation and Monads”. In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (Cited on page 6).

[146] Guillem Bartrina I Moreno. “Lexical Delimited Continuations for Scala 3”. PhD thesis. École Polytechnique Fédérale de Lausanne, 2025 (Cited on page 143).

[147] Carroll Morgan. “The Refinement Calculus”. In: *NATO ASI PDC*. 1994. URL: https://api.semanticscholar.org/CorpusID:10923264 (Cited on page 133).

[148] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3_24 (Cited on page 48).

[149] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Dependable Software Systems Engineering*. Ed. by Alexander Pretschner, Doron Peled, and Thomas Hutzelmann. Vol. 50. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2017, pp. 104–125. DOI: 10.3233/978-1-61499-810-5-104. URL: https://doi.org/10.3233/978-1-61499-810-5-104 (Cited on pages 17, 21, 129, 130).

[150] *Multicont: Continuations with multi-shot semantics in OCaml*. https://github.com/dhil/ocaml-multicont (Cited on page 53).

[151] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. "Hoare type theory, polymorphism and separation". In: *J. Funct. Program.* 18.5-6 (2008), pp. 865–911. DOI: 10.1017/S0956796808006953. URL: https://doi.org/10.1017/S0956796808006953 (Cited on page 131).

[152] Minh Nguyen et al. "Modular probabilistic models via algebraic effects". In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 381–410. DOI: 10.1145/3547635. URL: https://doi.org/10.1145/3547635 (Cited on page 141).

[153] Zhaozhong Ni and Zhong Shao. "Certified assembly programming with embedded code pointers". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 320–333. DOI: 10.1145/1111037.1111066. URL: https://doi.org/10.1145/1111037.1111066 (Cited on page 140).

[154] Liam O'Connor. *Lightweight Formal Methods for Programming*. https://liamoc.net/work_with_me.html. 2024 (Cited on page 1).

[155] Peter W. O'Hearn. "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 286–318. DOI: 10.3233/978-1-61499-028-4-286. URL: https://doi.org/10.3233/978-1-61499-028-4-286 (Cited on page 130).

[156] Peter W. O'Hearn. "Incorrectness Logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 10:1–10:32. DOI: 10.1145/3371078. URL: https://doi.org/10.1145/3371078 (Cited on page 1).

[157] *OCaml multicore examples*. https://github.com/ocaml-multicore/effects-examples (Cited on page 141).

[158] Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre. "Coma, an Intermediate Verification Language with Explicit Abstraction Barriers". In: *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*. Ed. by Viktor Vafeiadis. Vol. 15695. Lecture Notes in Computer Science. Springer, 2025, pp. 175–201. DOI: 10.1007/978-3-031-91121-7_8. URL: https://doi.org/10.1007/978-3-031-91121-7_8 (Cited on page 138).

[159] Willem Penninckx, Amin Timany, and Bart Jacobs. "Specifying I/O using abstract nested Hoare triples in separation logic". In: *Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2019, London, United Kingdom, July 15, 2019*. Ed. by Toby Murray and Gidon Ernst. ACM, 2019, 5:1–5:7. DOI: 10.1145/3340672.3341118. URL: https://doi.org/10.1145/3340672.3341118 (Cited on page 137).

[160] James T. Perconti and Amal Ahmed. "Verifying an Open Compiler Using Multi-language Semantics". In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 128–148. DOI: 10.1007/978-3-642-

54833-8_8. URL: https://doi.org/10.1007/978-3-642-54833-8%5C_8 (Cited on page 126).

[161] Mário Pereira and António Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31. URL: https://doi.org/10.1007/978-3-030-81688-9_31 (Cited on pages 46, 129, 187).

[162] Donald Pinckney, Arjun Guha, and Yuriy Brun. "Wasm/k: delimited continuations for WebAssembly". In: *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020*. Ed. by Matthew Flat. ACM, 2020, pp. 16–28. DOI: 10.1145/3426422.3426978. URL: https://doi.org/10.1145/3426422.3426978 (Cited on page 141).

[163] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. "Typed Equivalence of Effect Handlers and Delimited Control". In: *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, Dortmund, Germany, June 24-30, 2019*. Ed. by Herman Geuvers. Vol. 131. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 30:1–30:16. DOI: 10.4230/LIPICS.FSCD.2019.30. URL: https://doi.org/10.4230/LIPIcs.FSCD.2019.30 (Cited on page 144).

[164] Andrew Pitts and Ian Stark. "Operational Reasoning for Functions with Local State". In: *Higher Order Operational Techniques in Semantics*. Ed. by Andrew Gordon and Andrew Pitts. Publications of the Newton Institute, Cambridge University Press, 1998, pp. 227–273.

URL: http://www.inf.ed.ac.uk/~stark/operfl.html (Cited on page 124).

[165] Gordon D. Plotkin and John Power. "Algebraic Operations and Generic Effects". In: *Appl. Categorical Struct.* 11.1 (2003), pp. 69–94. DOI: 10.1023/A:1023064908962. URL: https://doi.org/10.1023/A:1023064908962 (Cited on page 49).

[166] Gordon D. Plotkin and John Power. "Notions of Computation Determine Monads". In: *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Mogens Nielsen and Uffe Engberg. Vol. 2303. Lecture Notes in Computer Science. Springer, 2002, pp. 342–356. DOI: 10.1007/3-540-45931-6_24. URL: https://doi.org/10.1007/3-540-45931-6%5C_24 (Cited on page 134).

[167] Gordon D. Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 80–94. DOI: 10.1007/978-3-642-00590-9_7. URL: https://doi.org/10.1007/978-3-642-00590-9_7 (Cited on page 49).

[168] Piotr Polesiuk. "IxFree: Step-indexed logical relations in Coq". In: *3 rd International Workshop on Coq for Programming Languages (CoqPL)*. 2017 (Cited on pages 124, 125, 133).

[169] Piotr Polesiuk and Filip Sieczkowski. *Functorial Syntax for All*. 2024 (Cited on page 125).

[170] *Proof of Programs with Effect Handlers*. 2022. URL: https://devilhena-paulo.github.io/files/phd-defense-slides.pdf (Cited on page 146).

[171] prophet. *Why are there so many libraries for algebraic effects? (comment)*. https://discourse.haskell.org/t/why-are-there-so-many-libraries-for-algebraic-effects/11844/29. 2025 (Cited on pages 141, 143).

[172] Olivia Proust and Frédéric Loulergue. "Verified Scalable Parallel Computing with Why3". In: *Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings*. Ed. by Carla Ferreira and Tim A. C. Willemse. Vol. 14323. Lecture Notes in Computer Science. Springer, 2023, pp. 246–262. DOI: 10.1007/978-3-031-47115-5_14. URL: https://doi.org/10.1007/978-3-031-47115-5%5C_14 (Cited on page 13).

[173] Brian Randell. "The 1968/69 NATO Software Engineering reports". In: *History of Software Engineering* 37 (1996) (Cited on page 1).

[174] Yann Régis-Gianas and François Pottier. "A Hoare Logic for Call-by-Value Functional Programs". In: *International Conference on Mathematics of Program Construction*. 2008. URL: https://api.semanticscholar.org/CorpusID:1095142 (Cited on page 138).

[175] Yann Régis-Gianas and François Pottier. "A Hoare Logic for Call-by-Value Functional Programs". In: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lecture Notes in Computer Science. Springer, 2008, pp. 305–335. DOI: 10.1007/978-3-540-70594-9_17. URL:

https://doi.org/10.1007/978-3-540-70594-9_17 (Cited on page 137).

[176] John C. Reynolds. "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*. Ed. by John J. Donovan and Rosemary Shields. ACM, 1972, pp. 717–740. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852 (Cited on page 138).

[177] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817. URL: https://doi.org/10.1109/LICS.2002.1029817 (Cited on page 130).

[178] Jan Schwinghammer et al. "Nested Hoare Triples and Frame Rules for Higher-Order Store". In: *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*. Ed. by Erich Grädel and Reinhard Kahle. Vol. 5771. Lecture Notes in Computer Science. Springer, 2009, pp. 440–454. DOI: 10.1007/978-3-642-04027-6_32. URL: https://doi.org/10.1007/978-3-642-04027-6_32 (Cited on page 136).

[179] Taro Sekiyama and Hiroshi Unno. "Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 2079–2110. DOI: 10.1145/3571264. URL: https://doi.org/10.1145/3571264 (Cited on page 145).

[180] Chung-chieh Shan. “Delimited continuations in natural language: quantification and polarity sensitivity”. In: *CoRR* cs.CL/0404006 (2004). URL: http://arxiv.org/abs/cs/0404006 (Cited on page 143).

[181] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. “Modular verification of higher-order methods with mandatory calls specified by model programs”. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications* (2007) (Cited on page 137).

[182] Rahul Sharma et al. “Verification as Learning Geometric Concepts”. In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 388–411. DOI: 10.1007/978-3-642-38856-9_21. URL: https://doi.org/10.1007/978-3-642-38856-9%5C_21 (Cited on page 103).

[183] K. C. Sivaramakrishnan et al. “Retrofitting effect handlers onto OCaml”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 206–221. DOI: 10.1145/3453483.3454039. URL: https://doi.org/10.1145/3453483.3454039 (Cited on pages 49, 141, 142).

[184] KC Sivaramakrishnan. *delimcc_of_fxhandler - Delimcc primitives from OCaml 5 effect handlers*. https://github.com/kayceesrk/delimcc_of_fxhandler. 2022 (Cited on page 143).

[185] Tiago Soares and Mário Pereira. “A Framework for the Automated Verification of Algebraic Effects and Handlers (extended version)”. In: *CoRR* abs/2302.01265 (2023). DOI: 10.48550/ARXIV.2302.

01265. arXiv: 2302.01265. URL: https://doi.org/10.48550/arXiv.2302.01265 (Cited on pages 49, 52, 129, 137, 141).

[186] Tiago Lopes Soares. "A Deductive Verification Framework For Higher Order Programs". In: *CoRR* abs/2011.14044 (2020). arXiv: 2011.14044. URL: https://arxiv.org/abs/2011.14044 (Cited on pages 3, 138).

[187] Tiago Lopes Soares. "Handle with Care and Confidence - Extending Cameleer with Algebraic Effects and Effect Handlers". In: (2022) (Cited on pages 129, 141).

[188] Tiago Lopes Soares, Ion Chirica, and Mário Pereira. "Static and Dynamic Verification of OCaml Programs: The Gospel Ecosystem". In: *Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings, Part III*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 15221. Lecture Notes in Computer Science. Springer, 2024, pp. 247–265. DOI: 10.1007/978-3-031-75380-0_14. URL: https://doi.org/10.1007/978-3-031-75380-0_14 (Cited on page 129).

[189] Yahui Song, Darius Foo, and Wei-Ngan Chin. "Automated Temporal Verification for Algebraic Effects". In: *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13658. Lecture Notes in Computer Science. Springer, 2022, pp. 88–109. DOI: 10.1007/978-3-031-21037-2_5. URL: https://doi.org/10.1007/978-3-031-21037-2_5 (Cited on pages 7, 142).

[190] Yahui Song, Darius Foo, and Wei-Ngan Chin. "Specification and Verification for Unrestricted Algebraic Effects and Handling". In: *Proc. ACM Program. Lang.* ICFP (2024) (Cited on pages 7, 22, 23, 86, 142).

[191] Marcelo Sousa and Isil Dillig. “Cartesian hoare logic for verifying k-safety properties”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 57–69. DOI: 10.1145/2908080.2908092. URL: https://doi.org/10.1145/2908080.2908092 (Cited on page 135).

[192] Matthieu Sozeau. “A New Look at Generalized Rewriting in Type Theory”. In: *J. Formaliz. Reason.* 2.1 (2009), pp. 41–62. DOI: 10.6092/ISSN.1972-5787/1574. URL: https://doi.org/10.6092/issn.1972-5787/1574 (Cited on page 119).

[193] Matthieu Sozeau. *The Proved Program of The Month - Type-safe printf via delimited continuations*. https://web.archive.org/web/20080513223632/http://www.lri.fr/perso/~sozeau/repos/coq/misc/shiftreset/GenuineShiftReset.html. 2008 (Cited on page 143).

[194] Kasper Svendsen. *Modular Specification and Verification for Higher-order Languages with State*. IT-Universitetet i København, 2013 (Cited on page 46).

[195] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. “Verifying Generics and Delegates”. In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Ed. by Theo D’Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 175–199. DOI: 10.1007/978-3-642-14107-2_9. URL: https://doi.org/10.1007/978-3-642-14107-2_9 (Cited on page 137).

[196] Nikhil Swamy et al. “SteelCore: an extensible concurrent separation logic for effectful dependently typed programs”. In: *Proc. ACM Pro-*

*gram. Lang.* 4.ICFP (2020), 121:1–121:30. DOI: 10.1145/3409003. URL: https://doi.org/10.1145/3409003 (Cited on page 131).

[197] Nikhil Swamy et al. "Verifying higher-order programs with the Dijkstra monad". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 387–398. DOI: 10.1145/2491956.2491978. URL: https://doi.org/10.1145/2491956.2491978 (Cited on pages 3, 131, 139, 142).

[198] Wouter Swierstra and Tim Baanen. "A predicate transformer semantics for effects (functional pearl)". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 103:1–103:26. DOI: 10.1145/3341707. URL: https://doi.org/10.1145/3341707 (Cited on pages 134, 142).

[199] Wenhao Tang et al. "Modal Effect Types". In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (2025), pp. 1130–1157. DOI: 10.1145/3720476. URL: https://doi.org/10.1145/3720476 (Cited on page 142).

[200] *The OCaml manual - Language extensions - Effect handlers*. https://ocaml.org/manual/5.3/effects.html (Cited on pages 51, 141).

[201] Joe Tidy. *CrowdStrike IT outage affected 8.5 million Windows devices, Microsoft says*. https://www.bbc.com/news/articles/cpe3zgznwjno. 2024 (Cited on page 1).

[202] Amin Timany and Lars Birkedal. "Mechanized relational verification of concurrent programs with continuations". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 105:1–105:28. DOI: 10.1145/3341709. URL: https://doi.org/10.1145/3341709 (Cited on pages 3, 54, 70, 71, 131, 133, 144).

[203] Turbolift. *Turbolift*. https://marcinzh.github.io/turbolift/. 2024 (Cited on page 141).

[204] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. “Constraint-Based Relational Verification”. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 742–766. DOI: 10.1007/978-3-030-81685-8_35. URL: https://doi.org/10.1007/978-3-030-81685-8%5C_35 (Cited on page 136).

[205] Viktor Vafeiadis and Matthew J. Parkinson. “A Marriage of Rely/Guarantee and Separation Logic”. In: *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*. Ed. by Luís Caires and Vasco Thudichum Vasconcelos. Vol. 4703. Lecture Notes in Computer Science. Springer, 2007, pp. 256–271. DOI: 10.1007/978-3-540-74407-8_18. URL: https://doi.org/10.1007/978-3-540-74407-8%5C_18 (Cited on page 27).

[206] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. “LiquidHaskell: experience with refinement types in the real world”. In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 39–51. DOI: 10.1145/2633357.2633366. URL: https://doi.org/10.1145/2633357.2633366 (Cited on page 137).

[207] Fei Wang et al. “Demystifying differentiable programming: shift/reset the penultimate backpropagator”. In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 96:1–96:31. DOI: 10.1145/3341700. URL: https://doi.org/10.1145/3341700 (Cited on page 143).

[208] Zhongye Wang, Qinxiang Cao, and Yichen Tao. “Verifying Programs with Logic and Extended Proof Rules: Deep Embedding v.s. Shallow Embedding”. In: *CoRR* abs/2310.17616 (2023). DOI: 10.48550/

ARXIV.2310.17616. arXiv: 2310.17616. URL: https://doi.org/10.48550/arXiv.2310.17616 (Cited on pages 125, 126, 130).

[209] Patrick Wardle. *Initial details about why CrowdStrike's CSAgent.sys crashed*. https://news.ycombinator.com/item?id=41021366. 2024 (Cited on page 1).

[210] Théo Winterhalter et al. "Partial Dijkstra monads for all". In: *Proceedings of the International Conference on Types for Proofs and Programs (TYPES)*. 2022 (Cited on pages 131, 135).

[211] Fabian Wolff et al. "Modular specification and verification of closures in Rust". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–29. DOI: 10.1145/3485522. URL: https://doi.org/10.1145/3485522 (Cited on pages 3, 46, 137, 139).

[212] Shushu WU, Xiwei WU, and Qinxiang Cao. "Encode the ∀∃ Relational Hoare Logic into Standard Hoare Logic". In: *CoRR* abs/2504.17444 (2025). DOI: 10.48550/ARXIV.2504.17444. arXiv: 2504.17444. URL: https://doi.org/10.48550/arXiv.2504.17444 (Cited on page 135).

[213] Hongseok Yang. "Relational separation logic". In: *Theoretical Computer Science* 375.1-3 (2007), pp. 308–334 (Cited on page 107).

[214] Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. "Handling bidirectional control flow". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 139:1–139:30. DOI: 10.1145/3428207. URL: https://doi.org/10.1145/3428207 (Cited on page 126).

[215] Nikita Zyuzin and Aleksandar Nanevski. "Contextual modal types for algebraic effects and handlers". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473580. URL: https://doi.org/10.1145/3473580 (Cited on page 142).

# HIGHER-ORDER FUNCTIONS SPECIFIED IN CAMELEER

```
let rec map (f:'a -> 'b) (xs:'a list) =
  match xs with
  | [] -> []
  | x :: xs1 -> f x :: map f xs1
(*@ ys = map f xs
      variant xs
      ensures length ys = length xs
      ensures forall i. 0 <= i < length ys ->
                ys[i] = f (xs[i]) *)

(*@ lemma index_shift: forall x:'a, xs:'a list, i:int.
   1 <= i /\ i < length (Cons x xs) ->
     (Cons x xs)[i] = xs[i-1] *)

let rec foldr ((inv : 'b -> 'a seq -> bool) [@ghost])
  (f : 'a -> 'b -> 'b) (xs : 'a list) (acc : 'b)
= match xs with
  | [] -> acc
  | x :: t -> f x (foldr inv f t acc)
(*@ r = foldr inv f xs acc
      requires inv acc []
      requires forall acc x ys.
                 inv acc ys -> inv (f x acc) (cons x
   ys)
      variant  xs
      ensures  inv r xs *)
```

Fig. A.1. `foldr` and `map` in Cameleer [161]

Cameleer specifications for `map` and `foldr` are shown in Fig. A.1. These may be compared to the specifications written for HEIFER in Section 2.1.

Because of the need to summarize the effects of `f`, `map`'s postcondition uses a quantifier (over sequence indices) to talk about the elements of the input and output lists. This then necessitates lemmas such as `index_shift` for relating indices to list destructuring, finally requiring more lines of specification per line of code.

The `foldr` specification is similar to the one for Iris (Section 2.1), but does not use a higher-order triple, instead requiring a ghost argument for an invariant that must be preserved between calls to `f`. This approach is representative of many verifiers, including Dafny, WhyML, and vanilla F⋆. As mentioned before, this parametrisation of the specification with a summary of `f` is nontrivial, in that it cannot be mechanically done for every higher-order function, as this pair of examples shows.