*Thesis Proposal*

# Automated Verification of Effectful Higher-order Programs

**Darius Foo**

11 September 2025

Department of Computer Science
School of Computing
National University of Singapore

# Contents

# ABSTRACT

This dissertation describes *staged logic*, a new approach to the automated verification of effectful higher-order programs. Such programs are completely natural to write in most programming languages, but are challenging to handle in verifiers, due to the need to reason simultaneously about the ordering and semantics of effects and higher-order control. Today, support for effectful higher-order programs is largely limited to interactive verifiers, each employing disparate, often semantic specification and verification approaches; many automated verifiers either handle them partially or not at all.

Staged logic offers a unified approach to effectful reasoning that is designed to bridge this gap. To support automation, it is syntactic in nature, based on refinement and rewriting. It is simple and general, and scales to complex language features, including effect handlers and delimited continuation operators. It serves as the basis for an automated tool, HEIFER, that verifies annotated OCaml programs via SMT, and is mechanised in Rocq. HEIFER has been used to verify a number of complex and challenging programs, demonstrating that staged logic can be effectively automated. This automation is achieved via a novel proof search procedure and a method of lemma synthesis for reasoning about control operators.

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 [169]. The root cause was a memory error due to a parser mishandling invalid data [176]. This is just another example in a *software crisis* that has been in effect since the advent of software engineering [141], 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 they are free of bugs. Two families of approaches [41] 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" [132], and abstract interpretation is relied on in safety-critical domains [37]. The tradeoff is that static analyses suffer from false positives when overapproximations prove too coarse-grained [134], and false negatives when approximations are not possible [120]. 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 kernel seL4 [107], the C compiler CompCert [118], the TLS stack in Project Everest [32], and, most recently, the Amazon Web Services (AWS) authorization engine [46]. The tradeoff, however, is that it scales mainly with human effort and expertise, which is expensive [83]; 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. This will be done by increasing the capacity for automation.

## 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 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 for the past 12 years, is dedicated to this problem.

While there has been much progress on it, 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 [117, 93]; build it into their tools primitively [178], or encode partial support for it in their underlying logics [66, 154].

Correspondingly little attention has been paid to *specifications*. Amongst interactive tools, which support varying subsets of effects, there is also not yet consensus on a uniform means of specification, which range from invariants [36] (Section 2.1) to protocols [63] (Section 3.1), non-context-local triples [170] (Section 4.1), and monadic types [122, 165, 69] (Section 6.1.2).

To solve these problems, I propose to rethink the means of verification, which I argue fundamentally makes these problems awkward to solve. I describe a simple logic of refinement, *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 is a logic for precisely specifying programs. It is similar in spirit to an *intermediate verification language* [128], in that it contains specifications alongside (representations of) program elements and logical connectives. A high-level overview of its syntax is given in Fig. 1.1.

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

Fig. 1.1. Staged logic

Instead of the Hoare triple, the *refinement relation* $\varphi \sqsubseteq \varphi'$ is taken as the starting point, with $\varphi$ typically representing the program and $\varphi'$ the specification.

$$
\dfrac{\dfrac{e' = \mathit{skip} \qquad P' \vdash Q}{\{P'\}\, e'\, \{Q\}} \qquad P, e \rightsquigarrow^* P', e'}{\{P\}\, e\, \{Q\}}
\qquad\qquad
\dfrac{\dfrac{\varphi_2 = \mathbf{ens}\ H_2 \qquad H_1 \vdash H_2}{\mathbf{ens}\ H_1 \sqsubseteq \varphi_2} \qquad \varphi_1 \rightsquigarrow^* \mathbf{ens}\ H_1}{\varphi_1 \sqsubseteq \varphi_2}
$$

(a) Separation logic (b) Staged logic

Fig. 1.2. Program proofs by symbolic execution

Proofs are carried out in a manner analogous to the (forward) symbolic execution performed by separation logic verifiers. Fig. 1.2 contains a high-level illustration of the proof states encountered, comparing both logics.

On the left, one begins with needing to prove a Hoare triple $\{P\}\, e\, \{Q\}$, where $P$ and $Q$ are heap assertions. Progress is made by repeatedly executing $e$ against the symbolic state $P$, transforming them both in a syntax-directed manner, interleaving applications of the frame rule, etc. At some point, when the program is reduced to a *skip*, a final entailment between the "current state" $P'$ and the postcondition $Q$ is proved.

On the right, we begin with a proof obligation $\varphi_1 \sqsubseteq \varphi_2$, where $\varphi_1$ is a precise representation of $e$. $\varphi_1$ is iteratively rewritten until we reach a state where we can cancel some part of it with something that appears in the specification; here, for example, we find an **ens** $H_1$ on the left, and if there were an **ens** $H_2$ on the right, we would be able to eliminate both by proving $H_1 \vdash H_2$, simplifying the proof obligation. This is repeated until the refinement becomes trivial.

In a sense, staged logic generalises Hoare logic: the Hoare triple may be recovered in two ways, where the assertions on the right relate programs and staged logic formulae.

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

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

The main benefit of staged logic, and of giving verification conditions using refinement, is that reasoning *remains precise*, using formulae which represent programs directly. For programs, this means that it is possible to talk about their effects in an embarrassingly simple way: using syntax. For specifications, given that they are also staged logic formulae, this means that they are able to retain descriptions of effectful behaviour in cases where it is difficult to invent an abstraction for it, allowing proofs to be deferred to a more appropriate point, where an abstraction may no longer be necessary. With this way of doing things, automation does not suffer, and is actually *improved* by the aforementioned ability to defer proofs. Moreover, all the techniques developed for Hoare logic continue to apply; staged logic can be seen as a bridge between the worlds of Hoare logic and refinement, inheriting the benefits of both approaches.

## 1.3 Thesis statement & contributions

The thesis I defend is captured by the following statement.

**Staged logic is a effective, sound, and novel approach to the**

**automated verification of effectful, higher-order programs.**

By "effectful", I mean using *computational effects* [126]. This includes primitive mutable state (thought of as an effect), algebraic effects, as well as delimited control operators.

By "effective", I mean that staged logic enables automated verification of the aforementioned class of programs. This is supported by the proof search algorithm described in Section 2.3, its implementation (Section 2.3.4), and experimental evaluation (Sections 2.4, 3.5 and 4.4).

By "sound", I mean that staged logic is compatible with an encoding in a foundational proof assistant. The soundness of the metatheory is thus checked.

The novelty of my work is justified by Chapter 6, which provides an extensive survey of the landscape of verifiers for such programs and situates stated logic in it.

**Contributions** The main contributions of this thesis may be summarised as follows.

I propose *staged logic*, a refinement-based approach to modularly specifying and verifying higher-order effectful programs. It improves the state of the art with simpler specifications, not requiring intermediate abstractions, such as protocols, while being equally capable at verification.

I demonstrate its effectiveness in automated verification by formulating a proof search algorithm and implementing it in a new SMT-based verifier, HEIFER[1].

I justify its soundness by a developing an encoding for it in Rocq and implementing a set of tactics for proving staged logic entailments interactively.

Finally, I investigate its expressiveness by demonstrating how it uniformly handles three distinct flavours of effectful higher-order program, hitherto

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

treated in very disparate ways.

## 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. *Proc. ACM Program. Lang.*, (ICFP), 2024

- Darius Foo, Yahui Song, and Wei-Ngan Chin, "A Staged Hoare Logic for Imperative Shift/Reset Programs" (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) [73]

- Yahui Song, Darius Foo, and Wei-Ngan Chin, "Automated Temporal Verification for Algebraic Effects" (APLAS 2022) [157]

- Darius Foo, Andreea Costea, and Wei-Ngan Chin, “Protocol Conformance with Choreographic PlusCal” (TASE 2023) [74]
- Yahui Song, Darius Foo, and Wei-Ngan Chin, “Specifying and Verifying Future Conditions” (SAS 2025; to appear)

## 1.5 How to read this dissertation

For an intuitive look at the key ideas and a first pass through examples, see: Sections 2.1 and 2.2 (an overview of staged logic), Section 3.1 (effect handlers), Section 4.1 (delimited continuations).

Chapter 6 provides an extensive and mostly self-contained literature review of this corner of the field of deductive verification, and can be read standalone.

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 combined use of higher-order functions and mutable state is routine in everyday programming. Consider the following program, which computes the sum of the elements of a list `xs` using the higher-order function `foldr`.

```
let x = ref [] in
foldr (fun c t -> x := c :: !x; c + t) xs 0
```

`foldr` is passed a function which captures some mutable reference `x` from its environment, and mutates it as the list is traversed.

To verify this program, one might write the following specification for the call to `foldr`, given as a Hoare triple (Fig. 2.1).

$\{x \mapsto a\}$
```
foldr (fun c t -> x := c :: !x; c + t) xs 0
```
$\{res.\ x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]\}$

Fig. 2.1. A modular specification for `foldr`

The specification is *modular*, and so begins with a *precondition*, a separation logic *heap formula* stating the requirements of the program on its environment. Here, all that is required initially is for there to be some (mutable) location x with arbitrary value `a`.

A *postcondition* follows, stating the property that we would like to verify of the program: that it computes the sum of `xs`, and also mutates `x` by prepending all of `xs` to it. Intuitively, this is true because `foldr` traverses the list from right to left, and the elements of `xs` are individually prepended in *reverse* order, resulting in the whole list being prepended in order. The square brackets denote lifting a pure formula into a heap formula.

Now, how might we prove such a Hoare triple automatically? Perhaps surprisingly, there isn't an easy answer today – in short, many automated verifiers do not allow effects in higher-order functions, and the ones that do tend to rely on interactive proof. A thorough survey of the landscape of these tools is given in Section 6.2.

Let us first review a standard approach to proving such a triple, to see why it is difficult to automate.

The first obstacle is that we need a specification of *foldr*, to modularly say what a call to it does. The following specification (Fig. 2.2) is taken from the Iris lecture notes [36].

$$\forall P, \mathit{Inv}, f, \mathit{xs}, l.\ \left\{ \begin{array}{l} (\forall x, a', \mathit{ys}.\ \{P\ x * \mathit{Inv}\ \mathit{ys}\ a'\}\ f(x, a')\ \{r.\ \mathit{Inv}\ (x{::}\mathit{ys})\ r\}) \\ * \mathit{isList}\ l\ \mathit{xs} * \mathit{all}\ P\ \mathit{xs} * \mathit{Inv}\ []\ a \end{array} \right\}$$
$$\mathit{foldr}\ f\ \ a\ l$$
$$\{r.\ \mathit{isList}\ l\ \mathit{xs} * \mathit{Inv}\ \mathit{xs}\ r\}$$

Fig. 2.2. A specification for *foldr*

Its most salient feature is that it is parameterised over an *invariant Inv*, a formula in separation logic that provides a means of talking about the effects of the function argument $f$. *Inv* relates the *suffix* of the input list traversed to the result of *foldr*, and is preserved by *foldr*: assuming the invariant holds initially, when the list has not been traversed at all and the result is $a$, it will hold after the evaluation of *foldr*, when the entire list *xs* has been traversed and the eventual result is $r$.

Next, to verify that the body of *foldr* preserves the invariant, we have to know that the function argument $f$ – which will be called at some point by *foldr*, and which we know nothing else about – preserves it too. This knowledge is expressed in the form of a nested triple in the precondition, giving requirements on $f$. This is an elegant solution because the invariant also abstractly describes the effects of $f$ – it takes us from a state over the suffix traversed *ys* and the accumulator $a'$ to one where the suffix has been extended by the current element $x$, and the accumulator has been updated to the result of $f$, $r$.

The rest of the triple illustrates other orthogonal but useful qualifies of *foldr* that one might wish to include in its specification. Suppose a client wanted to assume something about the content of the list being traversed in $f$, here a unary property $P$; this must be "pushed out" to *foldr* using a pure higher-order function such as *all*. Also, the structure of the list $l$ may be described using a *shape predicate isList*, which relates $l$ to its content *xs*. Its inclusion in both pre- and postcondition expresses that *foldr* does not change the list.

Unfortunately, despite the elegance of this approach, it is fundamentally quite difficult to automate.

The first problem is that at a call site such as in Fig. 2.1, one must come up with an invariant to instantiate *Inv* with, in order to explain the effect of the function argument in terms of an arbitrary call to $f$. This is an arbitrary semantic property and can be quite difficult to infer.

The second problem that one must come up with *the specification itself* (Fig. 2.2), an abstraction of the behaviour of *foldr*. As we can see, it took some insight. Moreover, there is no single most general specification one might give *foldr*. This problem is noted by its authors [36, Section 5]:

> The specification of foldr is higher-order in the sense that it involves nested Hoare triples (here in the precondition). The reason being that foldr takes a function f as argument, hence we can't specify foldr without having some knowledge or specification for the function f. *Different clients may instantiate foldr 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.*

The end result is that we may require one abstraction of *foldr* per client. For example, a client may require a $P$ of arity higher than one, necessitating a different specification.

### 2.1.1 A solution using refinement

Where did our woes begin? Arguably, they started when we came up with an abstraction for the effect of the function argument $f$ as a metalogic predicate. As we saw in Fig. 2.2, this takes *insight*, as it is ultimately a semantic account of the behaviour of $f$. Moreover, it takes *expressiveness*; this is also the reason why automated verifiers such as Dafny and Why are unable to handle effectful higher-order functions, as they embed function arguments as terms in first-order, SMT-friendly metalogics.

Seeing as the problem was the difficulty of an adequate(ly general) semantic account of higher-order unknowns, one might wonder if a syntactic solution could solve the problem. Indeed, it can, and this is the path we take.

We start with the language of *staged logic*[1] from Fig. 1.1, which allows us to both *model* and *specify* programs. In the former view, it is a (higher-order) verification intermediate language [128], where one may mix specifications and programs. In the latter, it is a logic, in which one may use quantifiers and other standard logical connectives, assert and assume arbitrary propositions, and talk about programs using fragments of syntax.

First, we model the *foldr* program (Fig. 2.3). We use disjunction to model the branches of the `match`, **ens** for the assumptions in each branch and existentials for the destructured variables, and **ens** again to talk about the result.

[1]The name *staged logic* was chosen in homage to Gherghina et al. [80], 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 broadly similar way, though here it also functions as a means of specification and has semantic implications. Finally, the partial evaluation around unknowns in staged logic (Section 2.3.2) also bears some similarity to the same mechanism in multi-stage programming, though this was not the original intent behind the choice of name.

Calls to recursive and unknown functions are represented directly, in their sequential order.

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

(a) foldr

$$\begin{aligned} &\mathit{foldr}(f,\mathit{init},\mathit{xs},\mathit{res}) = \\ &\quad \mathbf{ens}\ \mathit{res}.\mathit{xs}{=}[\,]\wedge \mathit{res}{=}\mathit{init} \\ &\quad \vee\ \exists h,t.\ \mathbf{ens}\ \mathit{xs}{=}h{::}t; \\ &\qquad\quad \exists r.\ \mathit{foldr}(f,\mathit{init},t,r); \\ &\qquad\qquad f\,(h,r,\mathit{res}) \end{aligned}$$

(b) *foldr*

Fig. 2.3. *foldr* program and formula

Next, we state a specification as a refinement. This is the same statement as in Fig. 2.1, but expressed in the logic.

$$\mathit{foldr}(g,\mathit{init},\mathit{xs},\mathit{res}) \sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ \mathit{res}.\ x \mapsto (\mathit{xs} +\!\!+\, a) * [\mathit{res} = \mathit{sum}\ \mathit{xs}]$$

$g$ is the staged logic counterpart of the closure. We denote this using a *specification assertion* (**fun** c t -> x := c :: !x; c + t) ::: $g$.

What is perhaps surprising is that the proof from this point can be completely automated. Readers who are unsurprised, or who would prefer to skip over the details at this point, can safely advance to Section 2.1.2.

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

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

Rewriting the call to *foldr* using the induction hypothesis and unfolding $f$, we get a lengthy sequence of **req** and **ens**.

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

It is not as complex as it looks: it is merely a summary 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* [43].

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

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 is the following.

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

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.

$$\frac{H_2 \vdash H_1}{\mathbf{req}\ H_1 \sqsubseteq \mathbf{req}\ H_2} \qquad \frac{Q_1 \vdash Q_2}{\mathbf{ens}\ Q_1 \sqsubseteq \mathbf{ens}\ Q_2}$$

The two entailments follow.

$$\forall a_1.\ x \mapsto a_1 \vdash x \mapsto a \qquad \begin{aligned}
&x \mapsto (h :: (t +\!\!+ a)) * \\
&\quad [res = h + r \wedge r = sum\ t \wedge xs = h :: t] \\
&\vdash x \mapsto (xs +\!\!+ a) * [res = sum\ xs]
\end{aligned}$$

The one on the right can can be reduced to the following first-order obligation, which is then easily dispatched by an SMT solver, given standard lemmas about *append* and *cons*.

$$res = h + (sum\ t) \wedge xs = h :: t \Rightarrow h :: (t \mathbin{++} a) = xs \mathbin{++} a \wedge res = sum\ xs \qquad \square$$

### 2.1.2 Deferring abstraction

What exactly enabled the automation? The crucial difference from the invariant approach is that we did not have to come up with an abstraction for the behaviour of *foldr* immediately. Instead, we left most of it as syntax until we knew enough (i.e. saw the definition of the closure argument) that going directly to the final specification was easy. Only the essential parts – the ordering of effects – was left, while things like the concrete state changes which occurred were abstracted away by **req** and **ens**.

The use of partial implementation detail in this manner has been referred to as the *greybox* approach [42, 150, 112] to verification. Some of the research questions this dissertation answers around it are: Given refinement as the starting point, what sorts of verification problems would be easier, and what sorts of abstractions could we dispense with? How could automation be improved? Could we eventually marry refinement with the Hoare triples and separation logic to get a happy middle ground?

In the rest of this chapter, we answer these questions for effectful higher-order functions. We start by fixing a syntax and semantics for staged logic, then describe 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 an 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

$$
\begin{array}{rll}
\varphi ::= & \mathbf{req}\ \sigma \mid \mathbf{ens}\ r.\,\sigma & \text{Specifications} \\
\mid & f(v,r) \mid \varphi_1; \varphi_2 & \text{Programs} \\
\mid & \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \exists\, x.\ \varphi \mid \forall\, x.\ \varphi & \text{Logical connectives}
\end{array}
$$

Fig. 2.4. Syntax of staged logic

**Syntax** The syntax of staged logic is given in Fig. 2.4. Its formulae consist firstly of **req** and **ens** constructs, which are best thought of as similar to the corresponding annotations in standard Hoare logic verifiers. Something unusual here is that they are internalised in the logic, i.e. used to construct propositions, and not merely delimiters for different portions of a Hoare triple. $\sigma$ is a state formula whose models are program states. Henceforth we take it to be a separation logic heap formula $H$, i.e. $\sigma ::= H$.

Informally, **req** $\sigma\ \varphi$ may be understood as an obligation to prove that some program state satisfies $\sigma$, and **ens** $r.\,\sigma$ as an assumption that we have a program state satisfying $\sigma$ (with result $r$), though like their annotation counterparts, their roles flip depending on which side of an call site (or entailment) they are on. We will make this precise shortly.

The next two constructs are what allow effectful behaviour to be represented natively in the logic: we can talk directly about a call to an unknown, possibly-effectful function $f$, with argument $v$ and result $r$, and also retain the order of such unknown calls in specifications.

The remaining ones are standard logical connectives lifted to specifications and programs.

The following abbreviations are also useful:

$$
\mathbf{req}\ H; \varphi \triangleq \mathbf{req}\ H\ \varphi \qquad \mathbf{ret}\ v \triangleq \mathbf{ens}\ r.\,[r{=}v]
$$

$$
\mathbf{ens}\ H \triangleq \mathbf{ens}\ r.\,[r{=}()] * H \qquad \mathbf{assert}\ H \triangleq \mathbf{req}\ H; \mathbf{ens}\ H
$$

$$
\mathbf{empty} \triangleq \mathbf{ens}\ \mathit{true} \qquad \bot \triangleq \mathbf{ens}\ \mathit{false} \qquad \top \triangleq \mathbf{req}\ \mathit{false}; \mathbf{empty}
$$

**Semantics** Staged logic admits 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, which is a finite partial map from locations $\ell$ to values $v$. We represent traces intensionally as an initial heap, a final heap, and a result value, and a staged formula $\varphi$ relating them. Finally, 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 formulae functions $\lambda v\, r.\, \varphi$, to give them interpretations.

The semantics of a staged formula can thus be given via the *satisfies* relation $\mathcal{E}, h_1, h_2, v \models \varphi$, defined in Fig. 2.5. It is similar to a nondeterministic big-step semantics containing logical connectives.

SReq shows that, as mentioned earlier, a **req** represents the obligation to prove that the "current" program state $h_1$ satisfies $H$. In separation logic terms, this means the initial heap $h_1$ can be partitioned into two heaplets $h_p$ and $h_r$, such that $h_p$ satisfies $H$ (and subsequently vanishes), and the remaining piece $h_r$ may serve as the initial heap for satisfying $\varphi$. Dually, SEns shows that we may assume a piece of heap $h_3$ satisfying $Q$ into existence and add it to the initial heap $h_1$. These internalise the operational meaning of heap entailment in symbolic execution-style separation logic verification, broadly similar to the *exhale* and *inhale* operations from Viper [128][2].

Next are functions and sequencing. The semantics of an unknown function call is simply that of its interpretation in $\mathcal{E}$ in SFn. Sequencing in SSeq is similar to relational composition of *satisfies* assertions on heaps, discarding the intermediate result $v_1$.

The remaining rules lift the standard connectives to staged logic.

[2]They are only similar insofar as they produce and consume pieces of heap. **req** and **ens** flip in meaning depending on which side of an entailment they are on, whereas *exhale* and *inhale* are ghost operations with a fixed meaning – in other words, **req** is translated to *either inhale or exhale* depending on the side of an entailment it occurs on.

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

$$\text{SReq}\quad \frac{\forall h_p\, h_r.\, \left( \begin{array}{c} h_1 = h_p \circ h_r \wedge h_p \vDash H \Rightarrow \\ \mathcal{E}, h_r, h_2, v \vDash \varphi \end{array} \right)}{\mathcal{E}, h_1, h_2, v \vDash \textbf{req}\, H\, \varphi} \qquad \text{SEns}\quad \frac{\begin{array}{c} h_3, v \vDash Q \\ h_2 = h_1 \circ h_3 \end{array}}{\mathcal{E}, h_1, h_2, v \vDash \textbf{ens}\, Q}$$

$$\text{SFn}\quad \frac{\mathcal{E}(f) = g \quad \mathcal{E}, h_1, h_2, r \vDash g(v, r)}{\mathcal{E}, h_1, h_2, r \vDash f(v, r)} \qquad \text{SSeq}\quad \frac{\begin{array}{c} \mathcal{E}, h_1, h_3, v_1 \vDash \varphi_1 \\ \mathcal{E}, h_3, h_2, v \vDash \varphi_2 \end{array}}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1; \varphi_2}$$

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

$$\text{SDisjL}\quad \frac{\mathcal{E}, h_1, h_2, v \vDash \varphi_1}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \vee \varphi_2} \qquad \text{SDisjR}\quad \frac{\mathcal{E}, h_1, h_2, v \vDash \varphi_2}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \vee \varphi_2} \qquad \text{SConj}\quad \frac{\begin{array}{c} \mathcal{E}, h_1, h_2, v \vDash \varphi_1 \\ \mathcal{E}, h_1, h_2, v \vDash \varphi_2 \end{array}}{\mathcal{E}, h_1, h_2, v \vDash \varphi_1 \wedge \varphi_2}$$

Fig. 2.5. Semantics of staged logic

### 2.2.1 Entailment

To support reasoning at the level of staged formulae, 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 formula as a program. We define it intensionally, as a direct lifting of implication to staged logic.

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

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

This semantic definition is useful to justify the soundness of the entailment rules in this section in this section and in Section 2.3.2. In Section 2.3.4, we give an alternative, syntactic definition suitable for automated proof.

**Properties of entailment** Entailment is reflexive and transitive, and equivalence is an equivalence relation. Using both, we can now state properties of the formulae of staged logic.

Many standard properties apply to lifted connectives: for example, disjunction is associative, commutative, idempotent, and has $\bot$ as an identity element and $\top$ as annihilator. We omit discussion of these and focus only on properties of the new constructs, in Fig. 2.6.

$\top$ and $\bot$ function as one would expect as the staged analogues of *true* and *false*. NSeqBotL shows that when combined with sequencing, $\bot$ has its usual interpretation as "dead code" or "unreachability", allowing anything after it to be ignored.

Important properties of **req** and **ens** are that they are contravariant and covariant, respectively, results familiar from the theory of behavioural subtyping [111]. NReqPureL and NEnsPureR further illustrate the duality of **req** and **ens**, showing that to introduce **req** (resp. **ens**) on the left (resp. right), one must be able to prove the condition $P$; with **req** (resp. **ens**) on the *right* (resp. left), one *assumes* $P$. **empty** is the left identity of sequencing, and its right as well only if $\varphi$ has the unit value as its result. Sequencing is associative, and disjunction distributes over it. Quantifiers distribute over sequencing and vice versa; note that these are not equivalences because $\varphi_1$ may depend on $x$. Finally, ReqReassoc shows that **req** may always be rewritten

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

$$\textsc{NBot}\ \frac{}{\bot \sqsubseteq \varphi} \qquad \textsc{NTop}\ \frac{}{\varphi \sqsubseteq \top} \qquad \textsc{NSeqBotL}\ \frac{}{\bot; \varphi \equiv \bot} \qquad \textsc{ReqContra}\ \frac{H_2 \vdash H_1 \quad \varphi_1 \sqsubseteq \varphi_2}{\mathbf{req}\ H_1\ \varphi_1 \sqsubseteq \mathbf{req}\ H_2\ \varphi_2}$$

$$\textsc{EnsCov}\ \frac{H_1 \vdash H_2}{\mathbf{ens}\ r.H_1 \sqsubseteq \mathbf{ens}\ r.H_2} \qquad \textsc{NReqPureL}\ \frac{P}{\mathbf{req}\ [P]\ \varphi \equiv \varphi} \qquad \textsc{NEnsPureR}\ \frac{P}{\mathbf{empty} \equiv \mathbf{ens}\ [P]}$$

$$\textsc{NSeqEmpL}\ \frac{}{\mathbf{empty}; \varphi \equiv \varphi} \qquad \textsc{NSeqEmpR}\ \frac{\varphi[()]}{\varphi; \mathbf{empty} \equiv \varphi} \qquad \textsc{NSeqAssoc}\ \frac{}{(\varphi_1; \varphi_2); \varphi_3 \equiv \varphi_1; (\varphi_2; \varphi_3)}$$

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

$$\textsc{NForAllDistrL}\ \frac{}{(\forall x.\ \varphi_2); \varphi_1 \sqsubseteq \forall x.\ \varphi_2; \varphi_1} \qquad \textsc{ReqReassoc}\ \frac{}{(\mathbf{req}\ H\ \varphi_1); \varphi_2 \sqsubseteq \mathbf{req}\ H\ (\varphi_1; \varphi_2)}$$

Fig. 2.6. Select properties of entailment

into right-associative form, i.e. where anything sequenced after it may be "moved under" the **req**, a key property which ensures that the nesting of **req** and sequencing does not matter[3]. Note that the converse of this last property is not true, as $\varphi_2$ may be dependent on an assumption in $H$.

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

*Proof.* See Section 5.2. □

### 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.

[3]This ensures that the abbreviation for **req** defined at the start of Section 2.2 does not cause any loss of expressiveness.

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

Fig. 2.7. Core language

**Core language** We begin with the definition of an ML-like core language (Fig. 2.7), 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 *nil* 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. 2.8) 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, environments 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.

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

$$\textsc{BigVal}\ \frac{}{E, h, v \leadsto h, v} \qquad \textsc{BigFun}\ \frac{}{E, h, (\lambda x.\, e) \leadsto h, (\lambda x.\, e)} \qquad \textsc{BigFix}\ \frac{}{E, h, (\lambda f\, x.\, e) \leadsto h, (\lambda f\, x.\, e)}$$

$$\textsc{BigAssert}\ \frac{}{E, h, (\mathit{assert\ true}) \leadsto h, ()} \qquad \textsc{BigAppFun}\ \frac{E, h_1, e[v/x] \leadsto h_2, R}{E, h_1, (\lambda x.\, e)\ v \leadsto h_2, R}$$

$$\textsc{BigAppUnk}\ \frac{E, h_1, (E(f)\ v) \leadsto h_2, R}{E, h_1, (f\ \ v) \leadsto h_2, R} \qquad \textsc{BigAppFix}\ \frac{E, h_1, e[v/x][(\lambda f\, x.\, e)/f] \leadsto h_2, R}{E, h_1, (\lambda f\, x.\, e)\ v \leadsto h_2, R}$$

$$\textsc{BigLet}\ \frac{\begin{array}{c} E, h_1, e_1 \leadsto h_3, v \\ E, h_3, e_2[v/x] \leadsto h_2, R \end{array}}{E, h_1, (\mathit{let}\ x{=}e_1\ \mathit{in}\ e_2) \leadsto h_2, R} \qquad \textsc{BigRef}\ \frac{\begin{array}{c} \ell \notin \mathit{dom}(h_1) \\ h_2 = h_1[\ell \mapsto v] \end{array}}{E, h_1, (\mathit{ref}\ v) \leadsto h_2, \ell}$$

$$\textsc{BigIfT}\ \frac{E, h_1, e_1 \leadsto h_2, R}{E, h_1, (\mathit{if\ true\ then}\ e_1\ \mathit{else}\ e_2) \leadsto h_2, R} \qquad \textsc{BigDeref}\ \frac{\ell \in \mathit{dom}(h)}{E, h, (!\ell) \leadsto h, h(\ell)}$$

$$\textsc{BigIfF}\ \frac{E, h_1, e_2 \leadsto h_2, R}{E, h_1, (\mathit{if\ false\ then}\ e_1\ \mathit{else}\ e_2) \leadsto h_2, R} \qquad \textsc{BigAssign}\ \frac{\ell \in \mathit{dom}(h)}{E, h, (\ell := v) \leadsto h[\ell \mapsto v], R}$$

Fig. 2.8. Big-step semantics

**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 many staged formulae which cannot directly be written as Hoare triples, at least without further parameterization at the meta-logical level; a simple example is the *open* program $e; f(v, r) ::: \mathbf{req}\, P; \mathbf{ens}\, Q; f(v, r)$, 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, staged logic can be seen as a means of importing some syntactic reasoning into the language of specifications.

Fig. 2.9 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 $\varphi_1 \sqsubseteq \varphi_2$.

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

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

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

$$\textsc{FIf}\ \frac{e_1 ::: \varphi_1 \quad e_2 ::: \varphi_2}{(\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2) ::: (\mathbf{ens}\ [b=\mathit{true}]; \varphi_1) \lor (\mathbf{ens}\ [b=\mathit{false}]; \varphi_2)} \qquad \textsc{FDeref}\ \frac{}{!x ::: \forall v.\ \mathbf{req}\ x \mapsto v\ \ \mathbf{ens}\ r.[r=v] * x \mapsto v}$$

$$\textsc{FAssign}\ \frac{}{(\ell := v) ::: \mathbf{req}\ x \mapsto v\ \ \mathbf{ens}\ x \mapsto \ell} \qquad \textsc{FRef}\ \frac{}{(\mathit{ref}\ v) ::: \exists \ell.\ \ \mathbf{ens}\ r.[r=\ell] * \ell \mapsto v}$$

$$\textsc{FAssert}\ \frac{}{\mathit{assert}\ v ::: \mathbf{req}\ [v=\mathit{true}]}$$

Fig. 2.9. Specification assertions

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 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 assumptions on the value of the conditional in both disjuncts. The FDeref, FAssign, FRef rules specify heap-manipulating primitives. The FAssert rule gives assertions the interpretation of requiring properties. Crucially, it cannot change the heap due to the use of a pure predicate. Properties of heap data can still be stated by first reading them into variables.

**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 2.2** (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 $\mathcal{E}, h_1, h_2, v$.*

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

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 2.3** (Environment compatibility)**.**

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

Finally, we close the definition by quantifying over environments.

**Definition 2.4** (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. 2.9, then it is valid.*

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

**History triples** Staged logic [75] originally used a *history triple*, a ternary relation between two staged formulae 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.

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

$$\textsc{HHistFrame}\ \frac{\{\,\varphi_h\,\}\,e\,\{\,\varphi\,\}}{\{\,\varphi_f;\varphi_h\,\}\,e\,\{\,\varphi_f;\varphi\,\}} \qquad \textsc{HConseq}\ \frac{\varphi_1 \sqsubseteq \varphi_3 \quad \varphi_4 \sqsubseteq \varphi_2 \quad \{\,\varphi_3\,\}\,e\,\{\,\varphi_4\,\}}{\{\,\varphi_1\,\}\,e\,\{\,\varphi_2\,\}}$$

$$\textsc{HVal}\ \frac{}{\{\,\varphi_h\,\}\,v\,\{\,\varphi_h;\mathbf{ens}\ r.[r=n]\,\}} \qquad \textsc{HAssert}\ \frac{}{\{\,\varphi_h\,\}\,v\,\{\,\varphi_h;\mathbf{req}\ [v=\mathit{true}]\,\}}$$

$$\textsc{HDeref}\ \frac{}{\left\{\,\varphi_h\,\right\}\,!x\,\left\{\,\varphi_h;\forall\,v.\ \mathbf{req}\ x\mapsto v \atop \mathbf{ens}\ r.[r=v]*x\mapsto v\,\right\}} \qquad \textsc{HAssign}\ \frac{}{\left\{\,\varphi_h\,\right\}\,x:=v\,\left\{\,\varphi_h;\mathbf{req}\ x\mapsto v_0 \atop \mathbf{ens}\ r.[r=v]*x\mapsto v\,\right\}}$$

$$\textsc{HRef}\ \frac{}{\{\,\varphi_h\,\}\,\mathit{ref}\ v\,\{\,\varphi_h;\exists\,\ell.\ \mathbf{ens}\ r.[r=\ell]*\ell\mapsto v\,\}} \qquad \textsc{HIf}\ \frac{\{\,\varphi_h;\mathbf{req}\ [b=\mathit{true}]\,\}\,e_1\,\{\,\varphi_1\,\} \atop \{\,\varphi_h;\mathbf{req}\ [b=\mathit{false}]\,\}\,e_2\,\{\,\varphi_2\,\}}{\{\,\varphi_h\,\}\,v\,\{\,\varphi_1\vee\varphi_2\,\}}$$

$$\textsc{HLet}\ \frac{\{\,\varphi_h\,\}\,e_1\,\{\,\varphi_1\,\} \quad \varphi_1[v] \atop \{\,\varphi_1\,\}\,e_2[v/x]\,\{\,\varphi_2\,\}}{\{\,\varphi_h\,\}\,\mathit{let}\ x{=}e_1\ \mathit{in}\ e_2\,\{\,\varphi_2\,\}} \qquad \textsc{HAppFun}\ \frac{\{\,\varphi_h\,\}\,e[v/x]\,\{\,\varphi\,\}}{\{\,\varphi_h\,\}\,(\lambda x.e)\ v\,\{\,\varphi\,\}}$$

$$\textsc{HAppFix}\ \frac{\{\,\varphi_h\,\}\,e[v/x][(\lambda f\,x.e)/f]\,\{\,\varphi\,\}}{\{\,\varphi_h\,\}\,(\lambda f\,x.e)\ v\,\{\,\varphi\,\}} \qquad \textsc{HAppUnk}\ \frac{}{\{\,\varphi_h\,\}\,f\ \ v\,\{\,\varphi_h;\exists\,r.\ f(v,r)\,\}}$$

Fig. 2.10. History triples

The rules are given in Fig. 2.10. 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 express the constraints on the condition as a preconditions in the history. This is functionally the same thanks to NREQPUREL and NENSPURER, 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). $e ::: \varphi \iff \{\,\textbf{empty}\,\}\, e\, \{\,\varphi\,\}$

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.

$$\textsc{HRes}\quad \frac{\{\,\varphi_h; \textbf{empty}\,\}\, e\, \{\,\varphi\,\}}{\{\,\varphi_h\,\}\, e\, \{\,\varphi\,\}} \qquad\qquad \textsc{HAppend}\quad \frac{e ::: \varphi}{\{\,\varphi_h\,\}\, e\, \{\,\varphi_h; \varphi\,\}}$$

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 2.5** (Validity of history triples)**.**

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

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 2.2.

**Theorem 2.2** (Soundness of history triples)**.** *Every history triple derivable using the rules of Fig. 2.10 is valid.* □

## 2.3 Entailment checking

We return in this section to the subject of 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 automated proof search.

### 2.3.1 Rewriting and sequent rules

The first idea required is that there are really two flavours of entailment rules. We saw some properties of entailment earlier, in Fig. 2.6, and most are *rewriting rules*, which have $\sqsubseteq$ only "below the line". For example, many of the earlier steps in Section 2.1.1 went by rewriting, using rules such as Biab.

The other kind of rule is a *sequent rule* – it has exactly one $\sqsubseteq$ premise, and can be seen as a *transformation* of the entire sequent, usually to lift things into the metalogic. An example is ReqContra, which simplifies the proof state by reducing a matching **req** prefix to a separation logic entailment. More examples are shown in Fig. 2.11.

$$\frac{\forall x.\, (\varphi_1 \sqsubseteq \varphi_2)}{\varphi_1 \sqsubseteq \forall x.\, \varphi_2}\;\textsc{EForAllR} \qquad \frac{\exists x.\, (\varphi_1 \sqsubseteq \varphi_2)}{\forall x.\, \varphi_1 \sqsubseteq \varphi_2}\;\textsc{EForAllL} \qquad \frac{\mathbf{ens}\; H; \varphi_1 \sqsubseteq \varphi_2}{\varphi_1 \sqsubseteq \mathbf{req}\; H\; \varphi_2}\;\textsc{EReqR}$$

Fig. 2.11. Examples of sequent rules

While in theory some sequent rules can also be used for rewriting, in practice it is useful to consider them separately. Rewriting rules must be applicable regardless of context – we discuss how to guarantee this later, in Section 5.4 – while sequent rules apply to goals of a fixed form.

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

## 2.3.2 Simplification to a normal form

Many rewriting rules can always be fruitfully applied, and iterated until they no longer change the 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.

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

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

Fig. 2.12. Select simplification rules

The important rules are given in Fig. 2.12. Generally, they aim to widen the scope of quantifiers, associate sequences and **req** to the right, distribute all connectives over disjunction, 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.

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

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 NReqPureL; in this sense it is broadly similar in role to the *spatial context* [108] in Iris. The rules are designed to manipulate and maintain this context. For example, EReqR normalises **req** assumptions into this form.

The original paper on staged logic [75] identified the following stronger normal form and used it to define an entailment checking procedure.

$$\bigvee \left(\textbf{req}\ H\ \textbf{ens}\ Q; f(v,r);\right)^{*} \textbf{req}\ H\ \textbf{ens}\ Q \sqsubseteq \varphi$$

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

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

Fig. 2.13. Biabduction

A key rewriting rule is NTranspose (Fig. 2.13), presented earlier informally as Biab. 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* [43] 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)**.**

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

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

NTRANSPOSE is not an equivalence, as solutions to (bi)abduction problems are not unique [44]. 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.

The approach in this section extends and generalises the one we proposed originally [75], 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 sequent for propagating frames.

Algorithm 1 is structured as a pair of mutually-recursive functions in a

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

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

$$\textsc{EPureEns}\ \frac{\pi \Rightarrow (\varphi_1 \sqsubseteq \varphi_2)}{\textbf{ens}\ \pi; \varphi_1 \sqsubseteq \varphi_2} \qquad \textsc{EPureReq}\ \frac{\pi \Rightarrow (\varphi_1 \sqsubseteq \varphi_2)}{\varphi_1 \sqsubseteq \textbf{req}\ \pi; \varphi_2} \qquad \textsc{ECancelFn}\ \frac{x_1{=}x_2 \quad (r_1{=}r_2 \Rightarrow \varphi_1 \sqsubseteq \varphi_2)}{f(x_1, r_1); \varphi_1 \sqsubseteq f(x_2, r_2); \varphi_2}$$

$$\textsc{EWFInd}\ \frac{f(x,r) \sqsubseteq \varphi_f \quad \begin{array}{c}(\forall\, x_0\, r_0.\ x_0 \leq x \Rightarrow f(x_0, r_0) \sqsubseteq \\ \varphi[x_0/x][r_0/r]) \Rightarrow \varphi_f \sqsubseteq \varphi\end{array}}{f(x,r) \sqsubseteq \varphi} \qquad \textsc{EForAllL}\ \frac{\exists x.\ (\varphi_1 \sqsubseteq \varphi_2)}{(\forall\, x.\ \varphi_1) \sqsubseteq \varphi_2}$$

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

Fig. 2.14. Essential sequent rules

**Algorithm 1:** Entailment proof search

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

tactic monad *'a tactic*, a function of type *pstate* → (*'a* × *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. 2.14) 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[4], 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.

## 2.4 Evaluation

To evaluate the effectiveness of Heifer's approach, we 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 bench-

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

marks 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
Comparison with Cameleer and Prusti

| | Heifer | | | | Cameleer [136] | | | Prusti [178] | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 [163] | 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 [72] | 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 | |

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 "✗". 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 [136]. 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 [178]. 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 [10], 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 timeouts and crashes). Support for closures is also not yet in mainline Prusti [11]. 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 formulae (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 [66, 10]; 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 [61] and Why3 [71] (with Z3, Alt-Ergo [119], and CVC4 [24] as backends) 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[5].

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

CHAPTER 3

# VERIFYING PROGRAMS WITH ALGEBRAIC EFFECTS & HANDLERS

## 3.1 Introduction

*Algebraic effects* were introduced by Plotkin and Power [138] as a means of reasoning about computational effects *algebraically*, using equations relating different operations. *Effect handlers* [139] 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 [151] as the main means of structuring concurrency.

Our story picks up at this point. First, we give a gentle introduction to effect handlers and related concepts. We then survey briefly how verification tools treat them today before covering our perspective on the issue.

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

let client () =
  a ();
  let x : int =
    perform (E "?")
  in b ()
```

(a) Client

```
let main () =
  match client () with
  | effect (E y), k ->
    c ();
    let z =
      continue k 42
    in e ()
  | v -> d (); v
```

(b) Handler

Fig. 3.1. Effect handlers

Fig. 3.1 illustrates how an effect handler works in OCaml. To help explicate the control flow of the example, functions are called in the order `a`, `b`, `c`, `d`, `e`, and variables are bound in the order `x`, `y`, `z`.

We start on the left, with the declaration of an effect `E`. Like an exception type, it extends an extensible variant `Effect.t`. It has an optional input type to the left of the arrow, and an output type to the right.

Below it, we have a program which *performs* effects, using the primitive **`perform`**. We refer to this program as the *client*. At this point, `E` is *uninterpreted*: we don't know what it does, only that it accepts a string from the client and returns to it an integer.

On the right is the entry point of the program, `main`, which invokes the client under a **`match`**, the syntax for an effect handler. Operationally, an effect handler may be seen as a generalisation of an exception handler which also gets access to a *delimited continuation* `k`. The **`perform`** thus causes a jump to the *effect case* of the handler, where `y` in the handler is bound to `"?"`, and `k` is bound to (a reification of) `let x : int = □ in b ()`. It should be clear now why `k` is called a delimited continuation: it consists of the rest of the computation of `client` following the **`perform`**, delimited by the extent of the **`match`**, and it can be *resumed* by plugging some value into the □.

This particular handler resumes the continuation using **`continue`**, calling functions `c` and `e` before and after. Notably, the continuation is only resumed once, and is thus called *one-shot*. Control goes back to the client, where `x` is bound to `42`. `b` is then invoked, and at some point after, the client returns its result.

Another notable point is that the handler remains installed over resumptions of continuations, handling subsequent effects that arise from their invocation; it is a *deep* handler. Because of this, the value the client returns is then bound to `v` in the second, *value case* of the **`match`**.

Finally, when the value case returns, its value is bound to `z` in the first case of the match, and `e` is executed. The intuition for this is that **`continue`** may be thought of as similar to a function call in the way it uses the call stack. A resumption of a continuation thus later returns to the point of resumption, and continues with whatever comes after.

**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 denotation: in Fig. 3.1, the meaning of `E` only became known given a handler, and one could imagine it being invoked under different handlers for e.g. testing, instrumentation, or simply begin given an alternative semantics.

**Deep and shallow handlers** In Fig. 3.1, we used a *deep* handler, which continues to apply to resumptions of continuations. It is also possible to 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 [6] and generators [9]. Semantically, a deep handler is a fold over a computation tree of trees, while a shallow handler is a case split [85].

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 our examples, we 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 [85], 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 [62].

**Zero-shot, one-shot, & multishot continuations** In Fig. 3.1, the handler used `k` in a manner that made it a *one-shot* continuation. 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 [106], a Unix-style fork [116], or continuation-based web servers [125] – as well as new challenges, which we go into shortly.

### 3.1.1 Reasoning about effect handlers

To illustrate the current paradigm for specifying and verifying effect-handling programs, consider the following example, which has been discussed in multiple papers [153, 63].

```
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. 3.2. XCHG effect

The XCHG effect 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. The implementation of the handler (Fig. 3.2, right) expresses this directly. The **try** syntax abbreviates a **match** with an identity value case.

A client which exchanges twice (Fig. 3.2, left) 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 for this program, but how can we specify and verify this in general?

Consider what we will require to verify the client modularly: eventually we will need a specification for the **perform** of XCHG. The same goes for the handler and its use of **continue** on the continuation k. Furthermore, we will need a way to treat the handler in a way that expresses its deep behaviour.

de Vilhena and Pottier [63]'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 both client and handler adhere to. It "describes functionality on which the client can rely, and which the handler must implement" [62]. Soares and Pereira [153] 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. 3.3. Cameleer specifications for XCHG

Fig. 3.3 shows the specifications needed in GOSPEL, which Soares and Pereira use in their automated verification setting. First is the protocol, which pertains to a particular effect. Here it consists of a relational postcondition that allows the client to know what **perform**ing XCHG will do. Verification of exchange is immediate given the protocol, and verification of client is easy, by composing the postcondition twice.

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

[1]de Vilhena and Pottier 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 postcondition of the protocol is really the *precondition* of the continuation: 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. Correspondingly, the precondition of the protocol can be assumed to hold at the start of the hander body.

Moving on, 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.

**Multishot continuations and context-locality** The protocol-based solution is elegant, and adequate for stock OCaml. However, OCaml may be extended [5] with multishot continuations. It has been noted that they enable some applications and make others notably simpler, and that more research is required to understand how best to support them [53].

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 [62], in Fig. 3.4.

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

Fig. 3.4. 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 [170] noted this problem, which de Vilhena and Pottier [63] 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 [63]. This problem has shown up many times in the literature, with different solutions. Delbianco and Nanevski [65] 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 [170] 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 [62] 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 A solution using refinement

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. 3.4, then the one in Fig. 3.2.

First, 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 $\textbf{assert}\ D \triangleq \textbf{req}\ D; \textbf{ens}\ r. D \wedge r{=}()$.

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

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.

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

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, *deferring abstraction* 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.

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

Notably, we have added a *bind* construct $\varphi_1; r.\ \varphi_2$ to the logic, which is essential for working with continuations (Section 3.2).

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.

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

The reasoning goes very operationally, but at a high level of abstraction, using lemmas corresponding to intuitive simplifying transformations we would make on pen and paper.

Next, we cover the extensions to staged logic required to realise the reasoning above. 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

$$
\begin{aligned}
\varphi &::= \textbf{req}\ \sigma\ \varphi \mid \textbf{ens}\ r.\,\sigma \\
&\quad\mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\ \varphi \mid \forall x.\ \varphi \\
&\quad\mid \varphi_1; r.\ \varphi_2 \mid f(v) \mid \epsilon(v) \mid \textbf{match}^{\delta}\ \varphi\ \textbf{with}\,\mathcal{H} \\
\delta &::= s? \\
\mathcal{H} &::= x \to \varphi \mid \epsilon(v), k \to \varphi \mid \mathcal{H}
\end{aligned}
$$

Fig. 3.5. Staged logic for effect handlers

We first extend the syntax of staged logic to be able to model effect handlers. The first two rows of Fig. 3.5, for specifications and logical connectives, are the same as in Fig. 2.4; we only extend the syntax with forms to model the new program elements.

**Existential problems** The first change is that we have generalised sequencing to *binding*. The notation $\varphi_1; r.\ \varphi_2$ alludes to this, denoting that the result of $\varphi_1$ is named $r$ in $\varphi_2$. The reason for this is that existentials are no longer adequate for modelling the results of programs using multishot continuations, which can return multiple times and thereby have *multiple results*; in contrast, an existential can be instantiated to exactly one result value. We resolve this by modelling the continuations of programs explicitly, allowing them to be taken multiple times with different values.

With bind in the language, we no longer have to write functions in relational form to be able to talk about their results, and so use the standard applicative form, bringing staged logic closer to the language of programs. This is more general – the relational form we used up to this point can be reinterpreted as $f(v,r); \varphi \triangleq f(v); r.\ \varphi$.

**Effect handlers** Next are effects $\epsilon(v)$ and handlers, which 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 (and their context) in specifications, sans abstraction, in keeping with our proposed methodology.

Handlers may be shallow (indicated by the superscript $s$) or deep (no superscript, and the default). **try** can be interpreted as the following abbreviation:

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

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

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

$$\textsc{SEff}\ \frac{R = \epsilon(v, \lambda r.\, \textbf{ret}\ r)}{\mathcal{E}, h, h, R \vDash \epsilon(v)} \qquad \cancel{\textsc{SSeq}\ \frac{\mathcal{E}, h_1, h_3, R \vDash \varphi_1 \quad \mathcal{E}, h_3, h_2, R \vDash \varphi_2}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; \varphi_2}} \qquad \textsc{SBindVal}\ \frac{\mathcal{E}, h_1, h_3, v \vDash \varphi_1 \quad \mathcal{E}, h_3, h_2, R \vDash \varphi_2[v/r]}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\ \varphi_2}$$

$$\textsc{SBindEff}\ \frac{\mathcal{E}, h_1, h_2, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi_1 \quad R = \epsilon(v, \lambda x.\, \varphi_k; r.\ \varphi_2)}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1; r.\ \varphi_2} \qquad \textsc{SUnhandled}\ \frac{\mathcal{E}, h_1, h_2, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi \quad \epsilon \notin \mathcal{H} \quad R = \epsilon(v, \lambda x.\, \textbf{match}\ \varphi_k\ \textbf{with}\ \mathcal{H})}{\mathcal{E}, h_1, h_2, R \vDash \textbf{match}\ \varphi\ \textbf{with}\ \mathcal{H}}$$

$$\textsc{SHandle}\ \frac{\mathcal{E}, h_1, h_3, \epsilon(v, \lambda x.\, \varphi_k) \vDash \varphi \quad \epsilon(x), k \to \varphi_b \in \mathcal{H} \quad \varphi_k' = \textbf{match}\ \varphi_k\ \textbf{with}\ \mathcal{H} \quad \mathcal{E}[k := \lambda x.\, \varphi_k'], h_3, h_2, R \vDash \varphi_b[v/x]}{\mathcal{E}, h_1, h_2, R \vDash \textbf{match}\ \varphi\ \textbf{with}\ \mathcal{H}} \qquad \textsc{SHandleVal}\ \frac{\mathcal{E}, h_1, h_3, v \vDash \varphi \quad x \to \varphi_b \in \mathcal{H} \quad \mathcal{E}, h_3, h_2, R \vDash \varphi_b[v/x]}{\mathcal{E}, h_1, h_2, R \vDash \textbf{match}\ \varphi\ \textbf{with}\ \mathcal{H}}$$

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

Fig. 3.6. Extended semantics of staged logic, on top of Fig. 2.5

**Semantics** The extensions to the semantics are shown in Fig. 3.6. First, we add the notion of an *outcome* $R$, inspired by the operational view of effects as a generalisation of exceptions – 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. 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, and has an intermediate identity continuation. The SSEQ rule from before is superseded by two new rules, SBINDVAL and SBINDEFF, which distinguish the outcome of $\varphi_1$. The former is simply a generalisation of the original rule to binding. The latter picks up after SEFF: given an unhandled effect, it 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. A strength of the operational approach we take is that extensions to new control operators are very direct.

## 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. This can be seen as a form of simplification, but driven by the semantics of program constructs, rather the laws relating logical connectives in Section 2.3.2. Biabduction (Section 2.3.3) also falls into this category, if we see it as the symbolic execution of state-changing operations. We refer to this specific kind of simplification as *reduction*.

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

Fig. 3.7. New simplification rules

First, some prerequisites. Most of the rules of Fig. 2.12 can be generalised directly to bind. We add a few more in Fig. 3.7, as well as a rule relating handlers and disjunction.

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

Fig. 3.8. **match#**

Next, we introduce a handler variant called **match#** (Fig. 3.8). 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, which gives us a way to keep track of the progress being made in reduction.

We are now ready to tackle the reduction rules of Fig. 3.9. 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

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

$$\textsc{RNorm}\quad \frac{x \to \varphi_b \in \mathcal{H} \qquad \varphi \setminus \mathcal{H}}{\textbf{match}\ \varphi\ \textbf{with}\ \mathcal{H} \rightsquigarrow \varphi; x.\ \varphi_b} \qquad \textsc{RSkip}\quad \frac{\varphi_1 \setminus \mathcal{H}}{\begin{array}{l}\textbf{match}\ \varphi_1; r.\ \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \\ \varphi_1; r.\ \textbf{match}\ \varphi_2\ \textbf{with}\ \mathcal{H}\end{array}}$$

$$\textsc{RDeep}\quad \frac{\textbf{match}\ \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \varphi_c}{\begin{array}{l}\textbf{match}\ \varphi_1; r.\ \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \\ \textbf{match}\ \varphi_1 \mathbin{\#} r.\ \varphi_c\ \textbf{with}\ \mathcal{H}\end{array}} \qquad \textsc{RShallow}\quad \frac{}{\begin{array}{l}\textbf{match}^s\ \varphi_1; r.\ \varphi_2\ \textbf{with}\ \mathcal{H} \rightsquigarrow \\ \textbf{match}\ \varphi_1 \mathbin{\#} r.\ \varphi_2\ \textbf{with}\ \mathcal{H}\end{array}}$$

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

Fig. 3.9. Reduction rules for effect handlers

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 reaches a turning point 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 5.5. □

## 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 relate programs and staged logic formulae using their semantics. This was justifiable because there was some semantic distance between the two languages, with state changes modelled by separation logic **req** and **ens**, higher-order 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. Moreover, reasoning about continuations necessitated a bind operator, making staged logic much closer to a programming language.

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 verification intermediate language. Entailment then becomes the workhorse for everything, even specification assertions.

**Definition 3.1** (Concrete staged formula)**.** *A concrete staged formula $\Phi^e$ 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.

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

*Proof.* By construction. □

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

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

$$e ::: \varphi \triangleq translate(e) \sqsubseteq \varphi$$

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 [63] and extended with additional stateful operations. Programs 3 and 7 are from the benchmarks of the *multicont*[2] library, which provides practical examples for multishot continuations. Program 5 revises

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

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 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, we generalise our framework to handle them, as a case study in how to extend it to new language features – a key advantage of the operational, refinement-based approach that we take.

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.

```
4 + reset (1 + shift k (k 2 + k 3))
```

Fig. 4.1. A simple example using shift/reset

Fig. 4.1 demonstrates the use of shift and reset with a small program in an ML-like syntax. The `shift` operator captures its *continuation* – the remaining computation that its result will be used in – and *reifies* it as a function value, which will then be bound to `k` in its body `(k 2 + k 3)`, which is also called the *meta-context*. Here, the continuation of `shift` is `1 + □`. The outer expression `4 + □` is not included due to the presence of `reset`, which *delimits* the extent of the continuation which is captured. `reset` has no other operational behaviour, and its result is simply that of its body once the latter is *shift-free* – if it reduces to a value with no further shifts.

How does this program execute? The reified continuation $(\lambda x.\, 1 + x)$ is bound to `k` in `(k 2 + k 3)`, which is executed next; it uses the continuation twice. The result of the program will thus be $4+((\underline{1+2})+(\underline{1+3}))=7$, where uses of the continuation are underlined. Note that in general, the continuation could contain an arbitrary (and dynamic) amount of computation.

Specifying and reasoning about programs using control operators compositionally is challenging, due to their fundamentally *non-context-local* [170] nature: an arbitrary expression $e$ may contain a control operator, causing reasoning about $e$ to depend on its continuation. This necessitates an approach such as *non-context-local* triples $\{P\}C[e]\{Q\}$ [170]. We defer a survey of approaches to Section 6.4.2.

**Answer type modification** The ability to manipulate the meta-context of an expression gives rise to some subtle issues, one of which is known as *answer type modification* [58] (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

Consider the program in Fig. 4.2. It is an encoding of the classic *printf* function, with the format string represented *shallowly*, as a program which uses `shift`, and 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 `reset` has no operational behaviour besides delimiting a `shift` and simply acts as the identity when its body is a value.

The explanation for this phenomenon is that the meta-context is allowed to *modify the answer type* of the `reset`. Here, the `shift` binds `k` to (a reification of) □ `^ 2` in the meta-context, a function. This function is then the result of the **`reset`** (changing its answer type), and it accepts an integer argument before converting it into a string and plugging it into `k`. The result is then a string. As this example demonstrates, ATM is useful: being able to repeatedly change the answer type to functions allowed us to accept a variable number of arguments, depending on the expressions appearing in the **`reset`**.

ATM complicates reasoning about programs, due to the need to know about possible answer type changes in **`reset`** bodies in order to type/verify them. Again, more approaches are covered in Section 6.4.2; 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_delim` [56]

`append_delim` in Fig. 4.3 uses ATM to implement a *difference list* [91]: 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 (in the *cons* branch) to a function, making the meta-context *the continuation itself*. 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 perhaps easy to see that it implements the standard list append function. How can we prove this formally?

First, we write the specification as a refinement.

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

Like before, we assume a staged logic with constructs **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 $[\,]$.

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

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

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

| | | |
|---|---|---|
| | $\langle \mathit{append\_sh}([\,]); r.\ \mathbf{ret}\ zs +\!\!+\, r\rangle; f.\ f\,(ys)$ | |
| $\sqsubseteq$ | $\langle(\mathbf{shift}\ k.\ k); r.\ \mathbf{ret}\ zs +\!\!+\, r\rangle; f.\ f\,(ys)$ | Unfold |
| $\sqsubseteq$ | $\langle \mathbf{ret}\ (\lambda r.\ \langle \mathbf{ret}\ zs +\!\!+\, r\rangle)\rangle; f.\ f\,(ys)$ | Reduction[1] |
| $\sqsubseteq$ | $\mathbf{ret}\ (\lambda r.\, \mathbf{ret}\ zs +\!\!+\, r); f.\ f\,(ys)$ | Pure |
| $\sqsubseteq$ | $\mathbf{ret}\ zs +\!\!+\, ys$ | Simplify |
| $\sqsubseteq$ | $\mathbf{ret}\ (zs +\!\!+\, [\,]) +\!\!+\, ys \quad \square$ | Lem. ++ |

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

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

Reduction[1] (used in the base case) eliminates **shift**s from formulae via symbolic execution, similar to Handle in Section 3.1.2. Here, the result of the meta-continuation is $k$, which is simply what follows the **shift** expression, delimited by the reset.

The proofs presented here harken back to the classical use of (monadic) equational reasoning [86, 12], with two key differences. The first is that with refinement, we have the added freedom to use specifications or abstractions of the pieces involved. The second is that this approach is itself a refinement of the one from Chapter 2, so stateful and higher order reasoning, as well as automation, are supported naturally.

### 4.1.2 Beyond shift and reset

Next, we seek to generalise our framework, so that it may be applied to other continuation operators, in the same spirit as Ishio and Asai [92]. Dybvig et al. [68] present a taxonomy of such operators. We compare their semantics in Fig. 4.4. On the left are the classifications given by Dybvig et al., while the name of the operator we use appears in the reduction semantics.

$$
\begin{array}{lrcll}
 & \langle v\rangle & \Longrightarrow & v & \\
+\mathcal{F}+ & \langle E[\mathbf{shift}\ k.\ e]\rangle & \Longrightarrow & \langle(\lambda k.\, e)\ (\lambda v.\, \langle E[v]\rangle)\rangle & [59] \\
-\mathcal{F}+ & \langle E[\mathbf{shift}_0\ k.\ e]\rangle & \Longrightarrow & (\lambda k.\, e)\ (\lambda v.\, \langle E[v]\rangle) & [84, 124] \\
+\mathcal{F}- & \langle E[\mathbf{control}\ k.\ e]\rangle & \Longrightarrow & \langle(\lambda k.\, e)\ (\lambda v.\, E[v])\rangle & [70] \\
-\mathcal{F}- & \langle E[\mathbf{control}_0\ k.\ e]\rangle & \Longrightarrow & (\lambda k.\, e)\ (\lambda v.\, E[v]) & [82]
\end{array}
$$

Fig. 4.4. Reduction semantics of various control operators [68, 8]

The operators differ essentially in whether they leave a delimiter around the continuation (indicated by + on the right) and the meta-continuation (+ on the left). Intuitively, these differences allow for more "dynamic" [35] behaviours, along distinct axes: **control** is able to capture parts of the *meta-continuation*, while **shift**$_0$ is able to "escape" to enclosing delimiters [124].

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 [34].

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

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

Fig. 4.5. `traverse` [34, 33]

The function `traverse` (Fig. 4.5) returns a copy of the given list. Interestingly, if the **control** operator is instead used in `visit`, `traverse` *reverses* the list. To prove this, we start with specifications (Fig. 4.6): pure functions *copy* and *reverse*. The latter is implemented using an accumulator.

$$\begin{aligned}
&\mathit{copy}(xs) = \\
&\qquad \mathbf{ens}\ xs{=}[\,];\mathbf{ret}\ [\,] \\
&\quad \lor \exists\, x\, xs_1.\ \mathbf{ens}\ xs{=}x :: xs_1; \mathit{copy}(xs_1); r_1.\ \mathbf{ret}\ x :: r_1 \\
&\\
&\mathit{rev}(xs, acc) = \\
&\qquad \mathbf{ens}\ xs{=}[\,];\mathbf{ret}\ acc \\
&\quad \lor \exists\, x\, xs_1.\ \mathbf{ens}\ xs{=}x :: xs_1; \mathit{rev}(xs_1, x :: acc)
\end{aligned}$$

Fig. 4.6. Specifications for *copy* and *rev*

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

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

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

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

| | | |
|---|---|---|
| | $\langle \mathit{visit}(x :: xs_1) \rangle$ | |
| $\sqsubseteq$ | $\langle (\textbf{shift}\ k.\ k(xs_1); r.\ \textbf{ret}\ x :: r); r_1.\ \mathit{visit}(r_1) \rangle$ | Unfold |
| $\sqsubseteq$ | $\textbf{ens}\ k{=}(\lambda r_1.\ \langle \mathit{visit}(r_1) \rangle); \langle k(xs_1); r.\ \textbf{ret}\ x :: r \rangle$ | Reduction$^2$ |
| $\sqsubseteq$ | $\langle \langle \mathit{visit}(xs_1) \rangle; r.\ \textbf{ret}\ x :: r \rangle$ | Simplify |
| $\sqsubseteq$ | $\langle \mathit{copy}(xs_1); r.\ \textbf{ret}\ x :: r \rangle$ | Ind. hyp. |
| $\sqsubseteq$ | $\mathit{copy}(xs_1); r.\ \textbf{ret}\ x :: r \quad \square$ | Pure |

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

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

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

$$\langle \mathit{visit}'(xs_1); r.\ \textbf{ret}\ x :: r \rangle$$

The key idea is that we must generalise the entailment over the *continuation* of *visit′*. In general, nontrivial generalisation may be required [86].

Suppose we did that, giving us the following statement. Now, what is the appropriate generalisation of *rev* over *k*?

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

A naive idea would be to apply the CPS transformation to *rev*, resulting in a definition with signature $\mathit{rev}'(xs, acc, k)$. 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 $\boxed{\text{do not know}}$ its definition, nor the two proofs demonstrating that it is indeed an intermediate step. This situation is shown as the following incomplete proof tree.

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

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 taking the current sequent(s) *as the definition* for $\varphi'$, 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** $\langle visit'([\,]);r.\ k(r)\rangle \sqsubseteq rev_1([\,],k)$

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

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

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

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. 4.4).

The final unproved goals together give us an inductive definition for $rev_1$ (Fig. 4.7), by assembling together the two cases under the assumptions on $xs$ in each case.

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

Fig. 4.7. Synthesised definition for $rev_1$

Interestingly, due to the use of reduction, both **control** and delimiter have been eliminated, resulting in a continuation-passing 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.

$$rev_1(xs, \lambda y.\, acc \mathbin{++} y) \sqsubseteq rev(xs, acc)$$

This statement can now be proved automatically.

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

| | | |
|---|---|---|
| | $rev([\,], \lambda y.\, \mathbf{ret}\; acc \mathbin{++} y)$ | Unfold |
| $\sqsubseteq$ | $\mathbf{ens}\; xs{=}[\,]; \mathbf{ret}\; acc$ | Simplify |
| $\sqsubseteq$ | $rev(xs, acc) \quad \square$ | Fold |

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

| | | |
|---|---|---|
| | $rev(x :: xs_1, \lambda y.\, \mathbf{ret}\ acc +\!\!+\ y)$ | |
| $\sqsubseteq$ | $rev(xs_1, \lambda y.\, \mathbf{ret}\ acc +\!\!+\ y; r_1.\ \mathbf{ret}\ x :: r_1)$ | Unfold |
| $\sqsubseteq$ | $rev(xs_1, \lambda y.\, \mathbf{ret}\ x :: (acc +\!\!+\ y))$ | Simplify |
| $\sqsubseteq$ | $rev(xs_1, \lambda y.\, \mathbf{ret}\ (x :: acc) +\!\!+\ y)$ | Lem. cons, ++ |
| $\sqsubseteq$ | $rev(x :: xs_1, acc) \quad \square$ | Ind. hyp. |

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 – an surprisingly systematic approach given its heuristic nature. Interestingly, it did not go through the standard CPS transformation, which for **control** is more involved [97].

Finally, given the proofs for both *copy* and *rev*, *why* does changing the control operator cause these functions to differ in behaviour, and how does the proof shed light on this? Consider the steps *after* Reduction$^2$ and Reduction$^3$. The **ret** $x :: r$ portion is really part of the meta-continuation, which means that these steps differ in whether subsequent control operators in *visit*/*visit'* are allowed to capture that. In the case of Reduction$^2$, they cannot, 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 captures*. Moreover, this became part of the inductive argument, pointing to how *visit' repeatedly* captures that part of the meta-continuation and nests it one level deeper, resulting in the reversal.

In the next section, we cover the extensions to staged logic required to realise the reasoning above.

## 4.2 Extensions to staged logic

We follow the technical development of Chapter 3, extending staged logic with delimited continuation operators. Because the latter are more general, we define their semantics directly instead of encoding them as effects.

$$
\begin{aligned}
\varphi &::= \mathbf{req}\ \sigma\ \varphi \mid \mathbf{ens}\ r.\sigma \\
&\quad \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\ \varphi \mid \forall x.\ \varphi \\
&\quad \mid \varphi_1; r.\ \varphi_2 \mid f(v) \\
&\quad \mid \mathbf{shift}^c_n\ k.\ \varphi\ (\lambda x.\ \varphi_k) \mid \mathbf{control}^c_n\ k.\ \varphi\ (\lambda x.\ \varphi_k) \mid \langle \varphi \rangle \\
n &::= 0? \\
\mathbf{id} &\triangleq \lambda x.\ \mathbf{ret}\ x \\
\mathbf{shift}_n\ k.\ \varphi &\triangleq \mathbf{shift}^c_n\ k.\ \varphi\ \mathbf{id} \\
\mathbf{control}_n\ k.\ \varphi &\triangleq \mathbf{control}^c_n\ k.\ \varphi\ \mathbf{id}
\end{aligned}
$$

Fig. 4.8. Staged logic with delimited control operators

**Syntax** We add both **shift** and **control**, with both zero and nonzero variants, and a single delimiter (Fig. 4.8). 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 [124], and in that case we would take the same approach to reduction as in the previous chapter.

**Semantics** Inspired by the exception-like semantics of effects given in Section 3.2, we use a similar semantics for control operators. This semantics is novel to the best of my knowledge, though it similar to an abstract machine-style semantics. Again, the construction of the delimited continuation is op-

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

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

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
$$\frac{\begin{array}{c}\mathcal{E}, h_1, h_3, v \vDash \varphi_1 \\ \mathcal{E}, h_3, h_2, R \vDash \varphi_2[v/r]\end{array}}{\mathcal{E}, h_1, h_2, R \vDash \varphi_1;\, r.\ \varphi_2}$$

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

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

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

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

SDelimShift0
$$\frac{\begin{array}{c}\mathcal{E}, h_1, h_3, \mathbf{shift}_0(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \vDash \varphi \\ \mathcal{E}[k := \lambda x.\, \langle \varphi_k \rangle], h_3, h_2, r \vDash \varphi_b\end{array}}{\mathcal{E}, h_1, h_2, R \vDash \langle \varphi \rangle}$$

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

SDelimControl0
$$\frac{\begin{array}{c}R_0 = \mathbf{control}_0(\lambda k.\, \varphi_b, \lambda x.\, \varphi_k) \\ \mathcal{E}, h_1, h_3, R_0 \vDash \varphi \\ \mathcal{E}[k := \lambda x.\, \varphi_k], h_3, h_2, r \vDash \varphi_b\end{array}}{\mathcal{E}, h_1, h_2, R \vDash \langle \varphi \rangle}$$

Fig. 4.9. Semantics of control operators in staged logic, extending Fig. 3.6

erationalised in it.

The semantics (Fig. 4.9) starts with a definition of outcomes: values, or "unhandled" control operators; the latter directly correspond to $\mathbf{shift}^c$ and $\mathbf{control}^c$. The rules for them, SShift and SControl, are hence quite trivial and 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 [104].

SBindVal is the same as in Fig. 3.6, generalising sequencing to binding. 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 meta-context body, with the the continuation appropriately bound, and delimiters around both. The remaining rules are similar, but directly implement the semantics of each operator by varying where the delimiters are placed, as in Fig. 4.4.

## 4.3 Reduction

Control operators can be reasoned about via the same semantics-directed *reduction* first introduced 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.*

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

Shift-freedom can be proved compositionally (Fig. 4.10).

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

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

Fig. 4.10. 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 meta-context is always placed under another reset. Crucially, this last rule is not true for the zero operators.

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

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

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

Fig. 4.11. Shift/reset reduction

We may now define the reduction rules (Fig. 4.11). Again, the goal is to minimise the scopes of resets, trying to move things out 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 **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 **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 the meta-context, 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 [17] | HO | 21 | 2 | 4.84 | 0.02 |
| Alice-Cat [124] | Multi | 3 | 1 | 0.33 | 0.02 |
| Printf [17] | ATM | 30 | 6 | 2.20 | 0.05 |
| Flip [59] | Multi | 9 | 1 | 3.89 | 0.04 |
| Toss [158] | Imp | 8 | 1 | 6.50 | 0.08 |
| HTTcc-inc3 [65] | Imp | 5 | 1 | 5.05 | 0.05 |
| Times [17] | 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, and so mainly serve to validate the claims that staged logic is a novel and effective means of verifying programs containing delimited continuations, and is feasible to automate. The lines of specifications required were 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. This suggests potential for being usable for automated verification method of larger, more realistic programs.

CHAPTER 5

# MECHANISING STAGED LOGIC

Chapters 2 to 4 gave an introduction to the use of staged logic, using an intuitive and idealised model. While this was convenient for exposition (and sufficient for the implementation of an automated verifier), it is not the full story. This chapter gives more precise definitions for implementation in a proof assistant. To that end, I formalise the theory presented in Chapter 4 (which builds on Chapter 2) in Rocq and prove its soundness.

## 5.1 Staged formulae

```
Inductive flow : Type :=
  | req : hprop → flow → flow
  | ens : (val → hprop) → flow
  | bind : flow → (val → flow) → flow
  | fex : forall (A:Type), (A → flow) → flow
  | fall : forall (A:Type), (A → flow) → flow
  | unk : var → val → flow
  | intersect : flow → flow → flow
  | disj : flow → flow → flow
  | shc : var → flow → (val → flow) → flow
  | rs : flow → flow
  | defun : var → (val → flow) → flow
  | discard : var → flow.
```

Fig. 5.1. Deep embedding of the syntax of staged logic

**Syntax** The syntax of staged formulae (Fig. 2.4) is represented using an inductive datatype (Fig. 5.1). A deep embedding is used both to sidestep both impredicativity and the positivity restriction (to represent uninterpreted functions; see Section 5.3), and because of the operational character of the semantics of the control operators, which makes it difficult to give them a modular denotation.

Separation logic propositions appear within **req** and **ens** specifications as the type `hprop` (defined to be the Rocq type `heap` → `Prop`), and so are *shallowly* embedded, to reuse standard approaches to automation [49].

To highlight a number of subtleties and differences:

- `unk` ("unknown") represents uninterpreted functions, which take a type of names `var`.
- `val` is a type of (dynamically-typed) values which also includes the constructor `vfptr` : `var` → `val` for *function pointers*.
- `shc` (representing **shift**$^c$, from Section 4.2) and `rs` represent shift and reset. The shift body is deeply embedded, but the continuation is shallowly embedded; the reason that the shift body allows its argument (a continuation) to "escape" via a function pointer value, as in **shift** $k$. **ret** $k$, while the continuation is only built up during reduction and never inspects its argument.
- `defun k (fun x ⇒ f)` represents (**ens** $k{=}(\lambda x.\,\varphi)$), which cannot be directly implemented, as lambdas are not in `flow` due to Rocq's positivity restriction on definitions of inductive data types. Instead, `defun` associates a definition with the name `k` in the *specification environment* (Section 2.2). `discard` undoes the effect of a `defun` and is required for technical reasons, as the changes to the model made by `defun` must be undone in refinement proofs where the specification does not make a corresponding change. The model is also extended to have *two* specification environments, in order to capture the effects of `defun`. More on this in Section 5.3.

**Semantics** The semantics of staged formulae is defined inductively via a separate interpretation relation. This follows the presentation in Fig. 2.5

and 4.9 closely. The new cases for `defun` are shown in Fig. 5.2 to give a flavour for how things are encoded.

```
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. 5.2. Semantics of staged logic (select cases)

`defun` adds a new name to the specification environment, while `discard` removes it. The name must be new so `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 a "nonterminating" derivation.

## 5.2 Entailment

We work entirely with an extensional account of entailment, 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 encoding are twofold: (1) it ensures that all the various entailment rules (Fig. 2.12, 2.14, 3.9 and 4.11) 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 **req** also come for free.

Entailment does double duty in interactive proofs, serving as a means for stating rewriting rules (Fig. 2.12) as well as a *sequent* (Fig. 2.14), appearing "below the line" in the Rocq proof state display as a representation of both the staged logic context and goal. This in turn supports the view of staged logic proofs as performing a sequence of entailment-preserving rewrites, leading to a trivial goal that can be solved using an automated tactic.

**Theorem 2.2.1** (Soundness of Entailment Rules)**.** *Given an entailment $\varphi_a \sqsubseteq \varphi_c$, the consequent overapproximates 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. □

## 5.3 Avoiding impredicativity

As staged logic is designed for reasoning about higher-order programs, it admits many higher-order constructions:

- Lambda values which may be stored in the heap, yet may be annotated with specifications
- Lambda expressions at the specification level
- First-class continuations, used in constructions such as **shift** $k.\ k$, yet whose definition is only known in the course of reduction

These are convenient to have on pen and paper, as well as in the implementation of an automated verifier, but they are difficult to encode in Rocq, as they create a cycle between the definitions of values and the interpretation of staged formulae.

We use a combination of syntactic techniques to support these use cases.

- To give specifications to functions, the user can prove a specification for an identical expression as a separate Rocq `Lemma`, then instantiate it as needed.

- The specification environment $\mathcal{E}$ links such specifications to their occurrences.

- To allow functions in the heap, we reify the names of function stages as *function pointer values* [131] which refer to definitions in the specification environment.

- To support dynamic definition of continuations, we allow the specification environment to be updated by two new staged formula primitives, **defun** and **discard**. We extend the model of Section 5.1 with a second specification environment, resulting in the interpretation relation $\mathcal{E}_1, \mathcal{E}_2, h_1, h_2, R \vDash \varphi$. **defun** $f\ (\lambda x.\varphi)$ is morally equivalent to the pen-and-paper **ens** $f=(\lambda x.\varphi)$, and extends $\mathcal{E}$ with a binding $[f \leftarrow (\lambda x.\varphi)]$, while **discard** simply removes a name that is already present – its purpose is mainly technical (Fig. 5.2).

The end result is that we can justify our pen-and-paper and automated proofs, while using a sound (but lower-level) representation when building proofs within Rocq.

## 5.4 Rewriting

Staged logic proofs primarily consist of rewrites. We make heavy use of Rocq's *generalized rewriting* [162] 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 each constructor of staged logic is *monotonic* [22] or *proper* [162] 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.

$$\textsc{ProperEntails}\quad \frac{\varphi_3 \sqsupseteq \varphi_1 \qquad \varphi_4 \sqsubseteq \varphi_2 \qquad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1 \sqsubseteq \varphi_2}$$

```
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. 5.3.

$$\textsc{ProperSatisfies}\quad \frac{\varphi_1 \sqsubseteq \varphi_2 \qquad \mathcal{E}, h_1, h_2, v \vDash \varphi_1}{\mathcal{E}, h_1, h_2, v \vDash \varphi_2} \qquad\qquad \textsc{ProperSeq}\quad \frac{\varphi_1 \sqsubseteq \varphi_2 \qquad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1; \varphi_3 \sqsubseteq \varphi_2; \varphi_4}$$

$$\textsc{ProperReq}\quad \frac{\varphi_1 \sqsubseteq \varphi_2}{\mathbf{req}\ H\ \varphi_1 \sqsubseteq \mathbf{req}\ H\ \varphi_2} \qquad\qquad \textsc{RReset}\quad \frac{\varphi_1 \sqsubseteq \varphi_2}{\langle \varphi_1 \rangle \sqsubseteq \langle \varphi_2 \rangle}$$

Fig. 5.3. Select rewriting lemmas

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.

## 5.5 Simplification and reduction

With the ability to rewrite inside arbitrary contexts (Section 5.4), we are able to state the reduction rules of Section 4.3 simply as 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. □

## 5.6 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 using vanilla entailment (Section 2.3). For example, consider Lemma 5.1.

**Lemma 5.1** (Associativity of seq. – untrue!)**.** $(\varphi_1; \varphi_2); \varphi_3 \Longleftrightarrow \varphi_1; (\varphi_2; \varphi_3)$.

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 5.1 should certainly be true. We demonstrate this formally by strengthening the definition of entailment to allow for this inductive reassociation.

**Definition 5.1** (Generalised entailment).

$$\frac{\begin{array}{l}\forall \mathcal{E}_1\ \mathcal{E}_2\ h_1\ h_2\ v.\\ \quad \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, v\varphi_1 \Rightarrow\\ \quad\quad \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, v\varphi_2\end{array}}{\varphi_1 \subseteq_0 \varphi_2}\ \textsc{GEntailsBase}$$

$$\frac{\begin{array}{l}\forall \mathcal{E}_1\ \mathcal{E}_2\ h_1\ h_2\ \varphi_b\ \varphi_k.\\ \quad \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, \varphi_1 \Rightarrow\\ \quad\quad \exists \varphi_b'\ \varphi_k'.\ \mathcal{E}_1, h_1 \vDash \mathcal{E}_2, h_2, \varphi_2 \land\\ \quad\quad\quad (\varphi_b \subseteq_n \varphi_b') \land (\forall v.\ \varphi_k[r/v] \subseteq_n \varphi_k'[r/v])\end{array}}{\varphi_1 \subseteq_{(n+1)} \varphi_2}\ \textsc{GEntailsShift}$$

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 $\varphi_1 \equiv_n \varphi_2 \triangleq \varphi_1 \subseteq_n \varphi_2 \land \varphi_2 \subseteq_n \varphi_1$.

**Lemma 5.2** (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 a sequent, 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 5.3** (Associativity of seq. – shift-free).

*If $\varphi_1^{\#}$ and $\varphi_2^{\#}$, 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. I intend to search for a more expressive model, such as a denotational or step-indexed one in which proofs might be conducted foundationally, as future work.

### 5.6.1 Evaluation

Table 5.1
Case studies verified in Rocq

| **Program** | **Desc.** | **#Lemma** | **LoP** |
|---|---|---|---|
| Multi [158] | Multi | 0 | 50 |
| Times [17] | Ind | 5 | 201 |
| Toss [158] | Ind, Imp | 1 | 342 |
| Total | | 6 | 593 |

The formalisation of the metatheory of staged logic, as well as basic automation, spans a development of 5907 LoC in Rocq. It is built on top of a textbook mechanisation of separation logic [49].

As mentioned in Section 5.6, the model proposed is not sufficiently expressive to be able to verify case studies without additional axioms – which are verified with respect to a different, incompatible notion of entailment (Definition 5.1). Nevertheless, to evaluate the feasibility of the theory, under the assumption that a more expressive model will eventually be found, I attempted a number of case studies (Table 5.1, where LoP stands for lines of proof) to get a sense of the effort and issues involved. Each example is shown with a list of its interesting features, followed by the LoC (comprising all of program, specification, and lemma definitions, and proof script).

Included with the formalisation is a library of tactics which automate applying the rules of Section 2.3, but with fewer heuristics, optimizing instead for predictability in interactive use. This significantly reduces the length of proof scripts.

The formalisation can be used standalone, but is intended to be complementary to HEIFER, to be used to prove lemmas that cannot be handled by the current heuristics.

CHAPTER 6

# RELATED WORK

## 6.1 Program logics

Deductive verification is typically carried out with the help of *program logics*, formal systems which ascribe to 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.

### 6.1.1 Hoare logics

The field of program logics could be said to have begun with *Hoare logic* [87], which introduced the *Hoare triple* $\{P\}\, e \,\{Q\}$ as a way of characterising the behaviours of programs. Dijkstra [67] 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 [117] and Why3 [71]. As these tools matured, they came to be structured as *verification platforms*, supporting *verification intermediate languages* [23, 128]: shared infrastructure to which the verification of programs in different languages could be reduced. Cameleer [136, 155, 153] is one such extension of the Why3 platform, enabling automated verification of OCaml programs. It has a backend-agnostic specification language, GOSPEL [156], which can also be interpreted via Ortac [90] 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 in the different 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 [145] and O'Hearn [133] 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 quickly became the basis of a new generation of automated verification tools, such as VeriFast [94, 93], HIP/SLEEK [51], and Viper [128], which automate program proofs using *symbolic execution* [31]. Separation logic is also the foundation of the Infer tool, which pioneered the use of *biabduction* [43] for *begin-anywhere* static analysis.

Staged logic is heavily inspired by separation logic, adopting it as a means of describing program states, while handling the behavioural aspects. It also relies on biabduction to simplify formulae, 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* [175]. A shallow embedding maps an object logic feature directly to a corresponding feature in the metalogic; this 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. 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, which can be an untenable amount of work.

Modern separation logics essentially all shallowly embed the language of propositions and differ on whether to deeply embed the program.

CFML [47, 48] 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* [49] textbook. It is common in Iris [95] to deeply embed programs and their contexts [78, 170]. The logic of Iris is technically shallowly embedded, but atop a sophisticated step-indexed model of higher-order logic, in order to support impredicative quantification and other higher-order features. The Verified Software Toolchain (VST) [14] makes similar choices with respect to programs, but implements a step-indexed model using *indirection theory* [89], 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 [88].

Hoare Type Theory (HTT) [129] 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. [165], 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* [122, 177] and more combinations of effects. Pulse [77, 166, 69] is a separation logic embedded in F$\star$, based on these ideas, which also marries them with indirection theory for reasoning about concurrency.

### 6.1.2 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$^+$ [109], 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 popular means of specifying and relating *open* programs, by quantifying over contexts: $\forall C.\ C[e] \sqsubseteq C[e']$ intuitively means that an arbitrary client $C$ cannot distinguish $e$ from $e'$. Rather than prove it by induction over contexts, which can be complex, much contemporary work on proving contextual refinement builds on Iris [95], discussing it in a typed setting using step-indexed logical relations. Examples of this approach include ReLoC Reloaded [78], which uses a *typed* refinement relation of the form $e_1 \precsim e_2 : \tau$, and Simuliris [79], which uses a relational verification approach $\{P\}\, e_1 \preceq e_2 \,\{r_1 r_2.\, Q\}$. Timany and Birkedal [170] use contextual refinement to show that a program using continuation-based for concurrency implements a primitively-concurrent version faithfully. They also introduce the idea of *context-local triples* that we revisit in Section 6.4.

Contextual refinement is fundamentally based on comparing *programs*; working at the level of specifications, as we do, allows the context to be described in other ways, such as via separation logic and framing. In that light, our work can be seen as similar to Song et al. [160], in the sense that it marries refinement and separation logic.

**Refinement calculi** Refinement calculi [20, 127, 21, 45] are another family of program logics, classically used in the methodology of *stepwise refinement*, a deductive program synthesis method for deriving programs from specifications. 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 [38] uses a state monad equipped with a least fixed point operator to verify higher-order imperative programs. There is some resemblance to the ordered *specification monads* of the Dijkstra monad approach [122, 177], if these were enriched with effects; with Dijkstra monads it is more typical that *computations* have effects, while specifications are left pure. Another line of work [13, 19, 167] from Swierstra and Baanen [167] uses free monads to uniformly implement a variety of effects. None of these tackle delimited continuations.

**Relational verification** Refinement is expressible as a relational verification [25] problem. Due to the more abstract program typically being nondeterministic, it is not a *relational safety property* [30, Definition 2], i.e. cannot be expressed using only universal quantification; instead it is a ∀∃ property.

*Self-composition*, or more generally, *product programs* [26], is the most common way to carry out relational verification today for ∀∀ properties. *Asymmetric product programs* [27] extend this to ∀∃ properties. While it copes well with nondeterminism, it is more suitable when the programs to be related are concrete, and it is easy to construct an *alignment* (a synchronous product) upfront, e.g. when the programs are very similar. Both of these are not the case for our setting.

Due to its ability to express arbitrary quantifier alternations, a promising solution could likely also be found in Hyper Hoare Logic [60].

Dynamic logic is another relational verification approach that copes well with refinement [172]. It is an approach dual to the refinement calculus in the sense that programs are instead objects in the logic, with modal operators allowing quantification over their behaviours. Refinement is naturally expressed there as $e_1 \sqsubseteq e_2 \triangleq \forall P.\ [e_1]P \Rightarrow [e_2]P$, where $[e]P$ is a modal operator meaning "after every terminating execution of $e$, proposition $P$ is true".

## 6.2 Higher-order imperative programs

### 6.2.1 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" [147] 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 [112, Section 2.1], and so works well in program logics designed for interactive verification, such as iHTT [164], Iris [95, 36], and CFML [47].

It can also be adapted to automated verification, either directly [50] 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 [142], extended to the imperative setting by Kanig and Filliâtre [98], and remains commonly used in other verifiers [153] in the Why3 [71] lineage. *Refinement types*, such as in LiquidHaskell [173], can also be seen as an intrinsic encoding of nested triples.

**Greybox specifications** Büchi and Weck [42] 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 [38] also makes this observation in the context of OCaml programs. Shaner et al. [150] adapted the greybox approach to JML in the form of *model programs*, which extends [112] to aspect-oriented and context-oriented language features.

Kanig and Filliâtre [98] 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 [71] and Dafny [117], 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.

**Contemporary approaches** Prusti [178] 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.

Coma [135] 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.

### 6.2.2 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 [144] 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 [154] for Cameleer, using axiomatised specifications parameterised over states to handle effectful programs, as Régis-Gianas and Pottier [143] do. Creusot [66] 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 [178] exploits the fact that they can maintain *history invariants* to provide stronger guarantees about captured state. Creusot [66] 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.

### 6.2.3 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 [47, 49].

**Dijkstra monads** Dijkstra monads [165, 122] 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 [95] is perhaps the most influential and well-known example of this approach. *Indirection theory* [89] is a general approach for building stratified models and viewing them through the lens of approximation; VST [14] and PulseCore [69] are recent examples building on it.

*Syntactic indirection* [15, 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 [93] is a recent example of program logic with this approach. XCAP [131] 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 5 illustrates how we use a similar approach to work around the issues with impredicativity.

## 6.3 Algebraic effects & handlers

### 6.3.1 In practice

Effect handlers originated in experimental programming languages such as Eff [28, 29], Effekt [40, 39], and Koka [115, 113], but have been steadily finding their way into mainstream languages; natively in OCaml 5 [151] and WebAssembly [137], and in the form of libraries in Scala 3 [171] and Haskell [140].

Effect handlers have many applications [7], including concurrency [151], control inversion [6], automatic differentiation [64], reactivity [4], model-checking [2], and probabilistic programming [130, 3].

### 6.3.2 Reasoning

**Protocols** The dominant paradigm for reasoning about effects today is the protocol-based approach, pioneered by Hazel [63, 62, 64] and Maze [62], and implemented in Cameleer [155, 153]. An extensive introduction to it was given in Section 3.1. 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 paradigm for reasoning about effect handlers is to internalise them and talk directly about them using refinement. This is the basis of our approach in this dissertation and was also thoroughly introduced in Section 3.1 The idea was pioneered by Song et al. [157], focusing on temporal verification for pure functional programs. It was extended with heap-manipulation by Foo et al. [75] and to effects by Song et al. [159]. 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. We believe it is possible to marry both approaches and get the best of both worlds, 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 [114]. Recent approaches are based on refinement types [99] and modal types [179, 168]. 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 [151].

**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 [12] or refinement [19, 167, 13, 38]; or having a computation monad (which specifies the effects) indexed by a specification monad [165, 121], 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.

## 6.4 Delimited continuations

### 6.4.1 In practice

Delimited continuation operators are implemented in many research languages, most prominently Racket, which supports the widest variety [8]. Like effect handlers, they have recently begun to enter the mainstream, most famously as (untyped) primitives in GHC [100], where they are now used to implement effect systems [140] (replacing stacks of monads), and less recently as libraries in Scala [146] and OCaml [102].

It is common to implement either **control**$_0$ (as in GHC), or a set of four lower-level operations *newPrompt*, *pushPrompt*, *takeSubCont*, and *pushSubCont* for manipulating the control stack [152, 102, 103, 68, 82] with named prompts for maximum generality.

Many of the use cases of effects apply equally to delimited continuations. Additional ones include partial evaluation [110, 57], web interactions [81], operating systems [105], automatic differentiation [174], probabilistic programming [52], and linguistics [149].

### 6.4.2 Reasoning

**Equational reasoning and refinement** Reasoning about delimited continuations has classically been carried out either by direct appeal to their semantics in CPS [33, Section 3], or by the use of equations [96]. Encodings of this approach in proof assistants are possible using the *generalised monad* [18], which also takes an input parameter. It has been used as a means of implementing a continuation monad admitting ATM [101, 161] and can support proofs by equational reasoning [86], 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** HTT$_{cc}$ [65] is an early 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 [170] develop a relational program logic which proposes the idea of *context-local* and *non-context-local* 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. They use contextual refinement to show that a cooperatively-concurrent program implemented using delimited control refines a specification program with primitive concurrency.

Forster et al. [76] proved that effect handlers and delimited continuation operators are interderivable (modulo typeability). de Vilhena [62] exploits this to adapt their protocol-based program logic for effect handlers to callcc and throw. The converse has also been done by Cong and Asai [55].

Coma [135] is a recent verification intermediate language with explicit support for continuations.

**Type systems** Danvy and Filinski [58] 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 enclosing reset), which may be modified by the execution of the $e$. Asai and Kameyama [16] generalise it with *answer type polymorphism*, and illustrate it lucidly in OchaCaml [123], a variant of OCaml which tracks the answer types of functions using an arrow of the form $\tau_1/\alpha \rightarrow \tau_2/\beta$. Cong and Asai [54] extend this to a dependently typed system.

A recent line of work using refinement types had advanced the idea of *answer effect modification* [148], which can prove functional correctness of programs with $\textbf{shift}_0$, through ATM, and go further to the temporal verification setting. Kawamata et al. [99] adapts this idea to effect handlers.

CHAPTER 7

# 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.

## 7.1 Remaining work

In the time until the final version of the dissertation is submitted, I intend to complete the following work:

- The paper on which Chapter 4 is based is currently under submission. I intend to continue to strengthen the evaluation results for it by adding more case studies to Tables 4.1 and 5.1.
- I intend to complete the implementation of HEIFER supporting Chapter 4, to make it a more robust prototype that others can build on.

## 7.2 Future work

- While the current semantic model (described in Chapter 5) is sufficient for proving the soundness of the framework, it is limited in the kinds of programs that can be expressed, particularly in cases involving dynamic continuations. This limits the use of staged logic in an interactive setting.

  I intend to investigate alternative encodings, based on specification monads [12, 19, 122] as well as indirection theory [89], to more conclusively answer the question of how best to encode the logic in a proof assistant.

- 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 automated one. I hope to develop it in tandem with the refinements to the mechanisation that would first allow interactive proofs.

# BIBLIOGRAPHY

[1] Add effect syntax. https://github.com/ocaml/ocaml/pull/12309.

[2] Dscheck - tool for testing concurrent OCaml programs. https://github.com/ocaml-multicore/dscheck.

[3] Effppl. https://github.com/Arnhav-Datar/EffPPL.

[4] Friendship ended with monads: Testing out algebraic effects in OCaml for animations. https://gopiandcode.uk/logs/log-bye-bye-monads-algebraic-effects.html.

[5] Multicont: Continuations with multi-shot semantics in OCaml. https://github.com/dhil/ocaml-multicont.

[6] The OCaml manual - language extensions - effect handlers. https://ocaml.org/manual/5.3/effects.html, .

[7] OCaml multicore examples. https://github.com/ocaml-multicore/effects-examples, .

[8] Continuations. https://docs.racket-lang.org/reference/cont.html.

[9] Implementing simple generators with effect handlers. https://discuss.ocaml.org/t/implementing-simple-generators-with-effect-handlers/16020/7.

[10] Modular specification and verification of closures in Rust (artefact). https://zenodo.org/records/5482557, 2021.

[11] Documentation of closures. https://github.com/viperproject/prusti-dev/issues/1431, 2024.

[12] Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. A practical formalization of monadic equational reasoning in dependent-type theory. *J. Funct. Program.*, 35, 2025. doi: 10.1017/S0956796824000157. URL https://doi.org/10.1017/s0956796824000157.

[13] João Alpuim and Wouter Swierstra. Embedding the refinement calculus in coq. *Sci. Comput. Program.*, 164:37–48, 2018. doi: 10.1016/J.SCICO.2017.04.003. URL https://doi.org/10.1016/j.scico.2017.04.003.

[14] Andrew W. Appel. Verified software toolchain. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, page 2. Springer, 2012. doi: 10.1007/978-3-642-28891-3_2. URL https://doi.org/10.1007/978-3-642-28891-3_2.

[15] 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.

[16] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2007. doi: 10.1007/978-3-540-76637-7_16. URL https://doi.org/10.1007/978-3-540-76637-7_16.

[17] Kenichi Asai and Oleg Kiselyov. Introduction to programming with shift and reset. 2011. URL https://api.semanticscholar.org/CorpusID:62358634.

[18] Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi: 10.1017/S095679680900728X. URL https://doi.org/10.1017/S095679680900728X.

[19] Tim Baanen. Algebraic effects, specification and refinement. Master's thesis, 2019.

[20] Ralph-Johan Back. *On the correctness of refinement steps in program development*. Department of Computer Science, University of Helsinki Helsinki, Finland, 1978.

[21] Ralph-Johan Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988. doi: 10.1007/BF00291051. URL https://doi.org/10.1007/BF00291051.

[22] 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.

[23] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. doi: 10.1007/11804192_17. URL https://doi.org/10.1007/11804192_17.

[24] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14.

[25] Gilles Barthe. An introduction to relational program verification, 2020.

[26] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011. doi: 10.1007/978-3-642-21437-0_17. URL https://doi.org/10.1007/978-3-642-21437-0_17.

[27] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013. doi: 10.1007/978-3-642-35722-0_3. URL https://doi.org/10.1007/978-3-642-35722-0_3.

[28] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123,

2015. doi: 10.1016/j.jlamp.2014.02.001. URL https://doi.org/10.1016/j.jlamp.2014.02.001.

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

[30] Bernhard Beckert and Mattias Ulbrich. Trends in relational program verification. In Peter Müller and Ina Schaefer, editors, *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 41–58. Springer, 2018. doi: 10.1007/978-3-319-98047-8_3. URL https://doi.org/10.1007/978-3-319-98047-8_3.

[31] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. doi: 10.1007/11804192_6. URL https://doi.org/10.1007/11804192_6.

[32] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 1:1–

1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPICS.SNAPL.2017.1. URL https://doi.org/10.4230/LIPIcs.SNAPL.2017.1.

[33] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005.

[34] Dariusz Biernacki and Olivier Danvy. Theoretical pearl: A simple proof of a folklore theorem about delimited control. *J. Funct. Program.*, 16 (3):269–280, 2006. doi: 10.1017/S0956796805005782. URL https://doi.org/10.1017/S0956796805005782.

[35] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Sci. Comput. Program.*, 60(3):274–297, 2006. doi: 10.1016/J.SCICO.2006.01.002. URL https://doi.org/10.1016/j.scico.2006.01.002.

[36] Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic. *Lectures notes, September*, 2023. URL https://iris-project.org/tutorial-material.html.

[37] Olivier Bouissou, Eric Conquet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Khalil Ghorbal, Eric Goubault, David Lesens, Laurent Mauborgne, Antoine Miné, et al. Space software validation using abstract interpretation. In *The International Space System Engineering Conference: Data Systems in Aerospace-DASIA 2009*, volume 1, pages 1–7. European Space Agency, 2009.

[38] Sylvain Boulmé. Intuitionistic refinement calculus. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages

54–69. Springer, 2007. doi: 10.1007/978-3-540-73228-0_6. URL https://doi.org/10.1007/978-3-540-73228-0_6.

[39] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: extensible algebraic effects in scala (short paper). In Heather Miller, Philipp Haller, and Ondrej Lhoták, editors, *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, pages 67–72. ACM, 2017. doi: 10.1145/3136000.3136007. URL https://doi.org/10.1145/3136000.3136007.

[40] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30, 2020. doi: 10.1145/3428194. URL https://doi.org/10.1145/3428194.

[41] Martin Brain and Elizabeth Polgreen. A pyramid of (formal) software verification. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*, volume 14934 of *Lecture Notes in Computer Science*, pages 393–419. Springer, 2024. doi: 10.1007/978-3-031-71177-0_24. URL https://doi.org/10.1007/978-3-031-71177-0_24.

[42] Martin Büchi and Wolfgang Weck. *The Greybox Approach: When Blackbox Specification Hide too much*. Turku Centre for Computer Science, 1999.

[43] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th*

*ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009. doi: 10.1145/1480881.1480917. URL https://doi.org/10.1145/1480881.1480917.

[44] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011. doi: 10.1145/2049697.2049700. URL https://doi.org/10.1145/2049697.2049700.

[45] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement: An overview. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004, Revised Lectures*, volume 3167 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2004. doi: 10.1007/11889229_1. URL https://doi.org/10.1007/11889229_1.

[46] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Mike Hicks, Sam Huang, Georges Axel Jaloyan, Anjali Joshi, Rustan Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clément Pit Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, John Tristan, Lucas Wagner, Mike Whalen, Remy Willems, Jenny Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Wang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. Formally verified cloud-scale authorization. 2025. URL https://www.amazon.science/publications/formally-verified-cloud-scale-authorization.

[47] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011. doi: 10.1145/2034773.2034828. URL https://doi.org/10.1145/2034773.2034828.

[48] 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.

[49] Arthur Charguéraud. *Separation Logic Foundations*. 2024.

[50] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Crowfoot: A verifier for higher-order store programs. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2012. doi: 10.1007/978-3-642-27940-9_10. URL https://doi.org/10.1007/978-3-642-27940-9_10.

[51] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012. doi: 10.1016/J.SCICO.2010.07.004. URL https://doi.org/10.1016/j.scico.2010.07.004.

[52] Alexander Collins and Vinod Grover. Probabilistic programming with cuppl. *CoRR*, abs/2010.08454, 2020. URL https://arxiv.org/abs/2010.08454.

[53] Commenters. Multi-shot continuations gone forever? https://discuss.ocaml.org/t/multi-shot-continuations-gone-forever/9072, 2022.

[54] Youyou Cong and Kenichi Asai. Handling delimited continuations with dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):69:1–69:31, 2018. doi: 10.1145/3236764. URL https://doi.org/10.1145/3236764.

[55] Youyou Cong and Kenichi Asai. Understanding algebraic effect handlers via delimited control operators. In Wouter Swierstra and Nicolas Wu, editors, *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*, volume 13401 of *Lecture Notes in Computer Science*, pages 59–79. Springer, 2022. doi: 10.1007/978-3-031-21314-4_4. URL https://doi.org/10.1007/978-3-031-21314-4_4.

[56] Olivier Danvy. Programming with tighter control. *Special issue of the BIGRE journal Putting the Scheme Language to Work, Brest, France (July 1989)*, pages 10–29, 1989.

[57] Olivier Danvy. Type-directed partial evaluation. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *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*, pages 242–257. ACM Press, 1996. doi: 10.1145/237721.237784. URL https://doi.org/10.1145/237721.237784.

[58] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[59] Olivier Danvy and Andrzej Filinski. Abstracting control. In Gilles Kahn, editor, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 151–160. ACM, 1990. doi: 10.1145/91556.91622. URL https://doi.org/10.1145/91556.91622.

[60] Thibault Dardinier and Peter Müller. Hyper Hoare logic: (dis-)proving program hyperproperties. *Proc. ACM Program. Lang.*, 8(PLDI):1485–1509, 2024. doi: 10.1145/3656437. URL https://doi.org/10.1145/3656437.

[61] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[62] 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.

[63] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi: 10.1145/3434314. URL https://doi.org/10.1145/3434314.

[64] Paulo Emílio de Vilhena and François Pottier. Verifying an effect-handler-based define-by-run reverse-mode AD library. *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.

[65] Germán Andrés Delbianco and Aleksandar Nanevski. Hoare-style reasoning with (algebraic) continuations. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 363–376. ACM, 2013. doi: 10.1145/2500365.2500593. URL https://doi.org/10.1145/2500365.2500593.

[66] Xavier Denis and Jacques-Henri Jourdan. Specifying and verifying higher-order Rust iterators. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *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*, volume 13994 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2023. doi: 10.1007/978-3-031-30820-8_9. URL https://doi.org/10.1007/978-3-031-30820-8_9.

[67] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975. URL https://doi.org/10.1145/360933.360975.

[68] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6): 687–730, 2007. doi: 10.1017/S0956796807006259. URL https://doi.org/10.1017/S0956796807006259.

[69] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil

Swamy. Pulsecore: An impredicative concurrent separation logic for dependently typed programs. In *2025 Programming Language Design and Implementation*. ACM, June 2025. URL https://www.microsoft.com/en-us/research/publication/pulsecore-an-impredicative-concurrent-separation-logic-for-dependently-type Accepted for publication, to appear.

[70] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52:205–237, 1987. doi: 10.1016/0304-3975(87)90109-5. URL https://doi.org/10.1016/0304-3975(87)90109-5.

[71] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *European Symposium on Programming*, 2013. URL https://api.semanticscholar.org/CorpusID:14572425.

[72] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 48–59. ACM, 2002. doi: 10.1145/581478.581484. URL https://doi.org/10.1145/581478.581484.

[73] Darius Foo and Wei-Ngan Chin. Tracing OCaml programs. *CoRR*, abs/2304.04937, 2023. doi: 10.48550/ARXIV.2304.04937. URL https://doi.org/10.48550/arXiv.2304.04937.

[74] Darius Foo, Andreea Costea, and Wei-Ngan Chin. Protocol conformance with Choreographic PlusCal. In Cristina David and Meng Sun, editors, *Theoretical Aspects of Software Engineering - 17th International Symposium, TASE 2023, Bristol, UK, July 4-6, 2023, Proceedings*, volume 13931 of *Lecture Notes in Computer Science*, pages

126–145. Springer, 2023. doi: 10.1007/978-3-031-35257-7_8. URL https://doi.org/10.1007/978-3-031-35257-7_8.

[75] 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.

[76] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP): 13:1–13:29, 2017. doi: 10.1145/3110257. URL https://doi.org/10.1145/3110257.

[77] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proceedings of the ACM on Programming Languages*, 5: 1 – 30, 2021. URL https://api.semanticscholar.org/CorpusID:233474595.

[78] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity. *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.

[79] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022. doi: 10.1145/3498689. URL https://doi.org/10.1145/3498689.

[80] Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Expressive program verification via structured specifications. *Int. J. Softw. Tools Technol. Transf.*, 16(4):363–380, 2014. doi: 10.1007/S10009-014-0306-5. URL https://doi.org/10.1007/s10009-014-0306-5.

[81] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In Pierpaolo Degano, editor, *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*, volume 2618 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2003. doi: 10.1007/3-540-36575-3_17. URL https://doi.org/10.1007/3-540-36575-3_17.

[82] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 12–23. ACM, 1995. doi: 10.1145/224164.224173. URL https://doi.org/10.1145/224164.224173.

[83] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, 2017. doi: 10.1145/3068608. URL https://doi.org/10.1145/3068608.

[84] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In David A. Padua, editor, *Proceedings of the Second ACM SIGPLAN*

*Symposium on Princiles & Practice of Parallel Programming (PPOPP), Seattle, Washington, USA, March 14-16, 1990*, pages 128–136. ACM, 1990. doi: 10.1145/99163.99178. URL https://doi.org/10.1145/99163.99178.

[85] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi: 10.1007/978-3-030-02768-1_22. URL https://doi.org/10.1007/978-3-030-02768-1_22.

[86] Noriko Hirota and Kenichi Asai. Correctness of functions with shift and reset. In *Proceedings of the 2011 ACM SIGPLAN Continuation Workshop*, pages 23–25, 2011. URL https://www.cs.tsukuba.ac.jp/~kam/cw2011/book.pdf.

[87] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

[88] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 523–536. ACM, 2013. doi: 10.1145/2429069.2429131. URL https://doi.org/10.1145/2429069.2429131.

[89] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid,*

*Spain, January 17-23, 2010*, pages 171–184. ACM, 2010. doi: 10.1145/1706299.1706322. URL https://doi.org/10.1145/1706299.1706322.

[90] Nikolaus Huber, Naomi Spargo, Nicolas Osborne, Samuel Hym, and Jan Midtgaard. Dynamic verification of OCaml software with gospel and ortac/qcheck-stm. In Arie Gurfinkel and Marijn Heule, editors, *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*, volume 15698 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2025. doi: 10.1007/978-3-031-90660-2_1. URL https://doi.org/10.1007/978-3-031-90660-2_1.

[91] R. John M. Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986. doi: 10.1016/0020-0190(86)90059-1. URL https://doi.org/10.1016/0020-0190(86)90059-1.

[92] Chiaki Ishio and Kenichi Asai. Type system for four delimited control operators. In Bernhard Scholz and Yukiyoshi Kameyama, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, pages 45–58. ACM, 2022. doi: 10.1145/3564719.3568691. URL https://doi.org/10.1145/3564719.3568691.

[93] Bart Jacobs. Verifast's separation logic: a higher-order(ish) logic without laters for modular verification of fine-grained concurrent programs. *CoRR*, abs/2505.04500, 2025. doi: 10.48550/ARXIV.2505.04500. URL https://doi.org/10.48550/arXiv.2505.04500.

[94] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi: 10.1007/978-3-642-20398-5_4. URL https://doi.org/10.1007/978-3-642-20398-5_4.

[95] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi: 10.1017/S0956796818000151. URL https://doi.org/10.1017/S0956796818000151.

[96] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457. Springer, 2004. doi: 10.1007/978-3-540-30124-0_34. URL https://doi.org/10.1007/978-3-540-30124-0_34.

[97] Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2008. doi: 10.1007/978-3-540-78969-7_18. URL https://doi.org/10.1007/978-3-540-78969-7_18.

[98] Johannes Kanig and Jean-Christophe Filliâtre. Who: a verifier for effectful higher-order programs. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 39–48, 2009.

[99] Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. Answer refinement modification: Refinement type system for algebraic effects and handlers. *Proc. ACM Program. Lang.*, 8(POPL):115–147, 2024. doi: 10.1145/3633280. URL https://doi.org/10.1145/3633280.

[100] Alexis King. Delimited continuation primops. https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7942, 2021.

[101] Oleg Kiselyov. Genuine shift/reset in haskell98. https://okmij.org/ftp/continuations/implementations.html, 2007.

[102] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 304–320. Springer, 2010. doi: 10.1007/978-3-642-12251-4_22. URL https://doi.org/10.1007/978-3-642-12251-4_22.

[103] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, 2012. doi: 10.1016/J.TCS.2012.02.025. URL https://doi.org/10.1016/j.tcs.2012.02.025.

[104] Oleg Kiselyov. Undelimited continuations do not occur in practice. https://okmij.org/ftp/continuations/against-callcc.html#illusion, 2012.

[105] Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. In Boicho N. Kokinov, Daniel C. Richardson, Thomas

Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007, Roskilde, Denmark, August 20-24, 2007, Proceedings*, volume 4635 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2007. doi: 10.1007/978-3-540-74255-5_22. URL https://doi.org/10.1007/978-3-540-74255-5_22.

[106] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In Walid Mohamed Taha, editor, *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009, Oxford, UK, July 15-17, 2009, Proceedings*, volume 5658 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2009. doi: 10.1007/978-3-642-03034-5_17. URL https://doi.org/10.1007/978-3-642-03034-5_17.

[107] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[108] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi: 10.

1145/3009837.3009855. URL https://doi.org/10.1145/3009837.3009855.

[109] Leslie Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002.

[110] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Robert R. Kessler, editor, *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, pages 227–238. ACM, 1994. doi: 10.1145/182409.182483. URL https://doi.org/10.1145/182409.182483.

[111] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.*, 37(4):13:1–13:88, 2015. doi: 10.1145/2766446. URL https://doi.org/10.1145/2766446.

[112] Gary T. Leavens, David A. Naumann, Hridesh Rajan, and Tomoyuki Aotani. Specifying and verifying advanced control features. In Tiziana Margaria and Bernhard Steffen, editors, *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*, volume 9953 of *Lecture Notes in Computer Science*, pages 80–96, 2016. doi: 10.1007/978-3-319-47169-3_7. URL https://doi.org/10.1007/978-3-319-47169-3_7.

[113] Daan Leijen. Koka. https://www.microsoft.com/en-us/research/project/koka/, 2012.

[114] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Program-*

*ming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014. doi: 10.4204/EPTCS.153.8. URL https://doi.org/10.4204/EPTCS.153.8.

[115] Daan Leijen. Algebraic effects for functional programming. Technical report, Microsoft, 2016.

[116] Daan Leijen. Structured asynchrony with algebraic effects. In Sam Lindley and Brent A. Yorgey, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, pages 16–29. ACM, 2017. doi: 10.1145/3122975.3122977. URL https://doi.org/10.1145/3122975.3122977.

[117] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20.

[118] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016.

[119] 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.

[120] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015. doi: 10.1145/2644805. URL https://doi.org/10.1145/2644805.

[121] Kenji Maillard. Principles of program verification for arbitrary monadic effects. (principes de la vérification de programmes à effets monadiques arbitraires). 2019.

[122] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29, 2019. doi: 10.1145/3341708. URL https://doi.org/10.1145/3341708.

[123] Moe Masuko and Kenichi Asai. Caml light+ shift/reset= caml shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pages 33–46, 2011.

[124] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN intrernational conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 81–93. ACM, 2011. doi: 10.1145/2034773.2034786. URL https://doi.org/10.1145/2034773.2034786.

[125] 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.

[126] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93

(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4. URL https://doi.org/10.1016/0890-5401(91)90052-4.

[127] Carroll Morgan. The refinement calculus. In *NATO ASI PDC*, 1994. URL https://api.semanticscholar.org/CorpusID:10923264.

[128] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017. doi: 10.3233/978-1-61499-810-5-104. URL https://doi.org/10.3233/978-1-61499-810-5-104.

[129] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi: 10.1017/S0956796808006953. URL https://doi.org/10.1017/S0956796808006953.

[130] Minh Nguyen, Roly Perera, Meng Wang, and Nicolas Wu. Modular probabilistic models via algebraic effects. *Proc. ACM Program. Lang.*, 6(ICFP):381–410, 2022. doi: 10.1145/3547635. URL https://doi.org/10.1145/3547635.

[131] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 320–333. ACM, 2006. doi: 10.1145/1111037.1111066. URL https://doi.org/10.1145/1111037.1111066.

[132] Liam O'Connor. Lightweight formal methods for programming. https://liamoc.net/work_with_me.html, 2024.

[133] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 286–318. IOS Press, 2012. doi: 10.3233/978-1-61499-028-4-286. URL https://doi.org/10.3233/978-1-61499-028-4-286.

[134] Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020. doi: 10.1145/3371078. URL https://doi.org/10.1145/3371078.

[135] Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre. Coma, an intermediate verification language with explicit abstraction barriers. In Viktor Vafeiadis, editor, *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*, volume 15695 of *Lecture Notes in Computer Science*, pages 175–201. Springer, 2025. doi: 10.1007/978-3-031-91121-7_8. URL https://doi.org/10.1007/978-3-031-91121-7_8.

[136] Mário Pereira and António Ravara. Cameleer: A deductive verification tool for OCaml. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 677–689. Springer, 2021. doi: 10.1007/978-3-030-81688-9_31. URL https://doi.org/10.1007/978-3-030-81688-9_31.

[137] Donald Pinckney, Arjun Guha, and Yuriy Brun. Wasm/k: delimited continuations for webassembly. In Matthew Flat, editor, *DLS 2020: Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, Virtual Event, USA, November 17, 2020*, pages 16–28. ACM, 2020. doi: 10.1145/3426422.3426978. URL https://doi.org/10.1145/3426422.3426978.

[138] Gordon D. Plotkin and A. John Power. Computational effects and operations: An overview. In Martín Escardó and Achim Jung, editors, *Proceedings of the Workshop on Domains VI 2002, Birmingham, UK, September 16-19, 2002*, volume 73 of *Electronic Notes in Theoretical Computer Science*, pages 149–163. Elsevier, 2002. doi: 10.1016/J.ENTCS.2004.08.008. URL https://doi.org/10.1016/j.entcs.2004.08.008.

[139] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *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*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi: 10.1007/978-3-642-00590-9_7. URL https://doi.org/10.1007/978-3-642-00590-9_7.

[140] 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.

[141] Brian Randell. The 1968/69 NATO software engineering reports. *History of Software Engineering*, 37, 1996.

[142] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, 2008. doi: 10.1007/978-3-540-70594-9_17. URL https://doi.org/10.1007/978-3-540-70594-9_17.

[143] 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.

[144] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972. doi: 10.1145/800194.805852. URL https://doi.org/10.1145/800194.805852.

[145] 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*, pages 55–74. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029817. URL https://doi.org/10.1109/LICS.2002.1029817.

[146] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 317–328. ACM, 2009. doi: 10.

1145/1596550.1596596. URL https://doi.org/10.1145/1596550.1596596.

[147] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 2009. doi: 10.1007/978-3-642-04027-6_32. URL https://doi.org/10.1007/978-3-642-04027-6_32.

[148] Taro Sekiyama and Hiroshi Unno. Temporal verification with answer-effect modification: Dependent temporal type-and-effect system with delimited continuations. *Proc. ACM Program. Lang.*, 7(POPL):2079–2110, 2023. doi: 10.1145/3571264. URL https://doi.org/10.1145/3571264.

[149] Chung-chieh Shan. Delimited continuations in natural language: quantification and polarity sensitivity. *CoRR*, cs.CL/0404006, 2004. URL http://arxiv.org/abs/cs/0404006.

[150] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, 2007.

[151] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*,

pages 206–221. ACM, 2021. doi: 10.1145/3453483.3454039. URL https://doi.org/10.1145/3453483.3454039.

[152] KC Sivaramakrishnan. delimcc_of_fxhandler - delimcc primitives from OCaml 5 effect handlers. https://github.com/kayceesrk/delimcc_of_fxhandler, 2022.

[153] Tiago Soares and Mário Pereira. A framework for the automated verification of algebraic effects and handlers (extended version). *CoRR*, abs/2302.01265, 2023. doi: 10.48550/ARXIV.2302.01265. URL https://doi.org/10.48550/arXiv.2302.01265.

[154] Tiago Lopes Soares. A deductive verification framework for higher order programs. *CoRR*, abs/2011.14044, 2020. URL https://arxiv.org/abs/2011.14044.

[155] Tiago Lopes Soares. Handle with care and confidence - extending Cameleer with algebraic effects and effect handlers. 2022.

[156] Tiago Lopes Soares, Ion Chirica, and Mário Pereira. Static and dynamic verification of OCaml programs: The gospel ecosystem. In Tiziana Margaria and Bernhard Steffen, editors, *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*, volume 15221 of *Lecture Notes in Computer Science*, pages 247–265. Springer, 2024. doi: 10.1007/978-3-031-75380-0_14. URL https://doi.org/10.1007/978-3-031-75380-0_14.

[157] Yahui Song, Darius Foo, and Wei-Ngan Chin. Automated temporal verification for algebraic effects. In Ilya Sergey, editor, *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, volume 13658 of

*Lecture Notes in Computer Science*, pages 88–109. Springer, 2022. doi: 10.1007/978-3-031-21037-2_5. URL https://doi.org/10.1007/978-3-031-21037-2_5.

[158] Yahui Song, Darius Foo, and Wei-Ngan Chin. Specification and verification for unrestricted algebraic effects and handling. *Proc. ACM Program. Lang.*, (ICFP), 2024.

[159] Yahui Song, Darius Foo, and Wei-Ngan Chin. Specification and verification for unrestricted algebraic effects and handling. *Proc. ACM Program. Lang.*, (ICFP), 2024.

[160] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. Conditional contextual refinement. *Proc. ACM Program. Lang.*, 7(POPL):1121–1151, 2023. doi: 10.1145/3571232. URL https://doi.org/10.1145/3571232.

[161] 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.

[162] Matthieu Sozeau. A new look at generalized rewriting in type theory. *J. Formaliz. Reason.*, 2(1):41–62, 2009. doi: 10.6092/ISSN.1972-5787/1574. URL https://doi.org/10.6092/issn.1972-5787/1574.

[163] Kasper Svendsen. *Modular specification and verification for higher-order languages with state*. IT-Universitetet i København, 2013.

[164] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Verifying generics and delegates. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture*

*Notes in Computer Science*, pages 175–199. Springer, 2010. doi: 10.1007/978-3-642-14107-2_9. URL https://doi.org/10.1007/978-3-642-14107-2_9.

[165] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013. doi: 10.1145/2491956.2491978. URL https://doi.org/10.1145/2491956.2491978.

[166] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP):121:1–121:30, 2020. doi: 10.1145/3409003. URL https://doi.org/10.1145/3409003.

[167] Wouter Swierstra and Tim Baanen. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP): 103:1–103:26, 2019. doi: 10.1145/3341707. URL https://doi.org/10.1145/3341707.

[168] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. Modal effect types. *Proc. ACM Program. Lang.*, 9(OOPSLA1):1130–1157, 2025. doi: 10.1145/3720476. URL https://doi.org/10.1145/3720476.

[169] Joe Tidy. Crowdstrike it outage affected 8.5 million windows devices, microsoft says. https://www.bbc.com/news/articles/cpe3zgznwjno, 2024.

[170] Amin Timany and Lars Birkedal. Mechanized relational verification of

concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP):105:1–105:28, 2019. doi: 10.1145/3341709. URL https://doi.org/10.1145/3341709.

[171] Turbolift. Turbolift. https://marcinzh.github.io/turbolift/, 2024.

[172] Mattias Ulbrich. *Dynamic Logic for an Intermediate Language: Verification, Interaction and Refinement*. PhD thesis, Karlsruhe Institute of Technology, 2013. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000041169.

[173] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: experience with refinement types in the real world. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51. ACM, 2014. doi: 10.1145/2633357.2633366. URL https://doi.org/10.1145/2633357.2633366.

[174] Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019. doi: 10.1145/3341700. URL https://doi.org/10.1145/3341700.

[175] Zhongye Wang, Qinxiang Cao, and Yichen Tao. Verifying programs with logic and extended proof rules: Deep embedding v.s. shallow embedding. *CoRR*, abs/2310.17616, 2023. doi: 10.48550/ARXIV.2310.17616. URL https://doi.org/10.48550/arXiv.2310.17616.

[176] Patrick Wardle. Initial details about why crowdstrike's csagent.sys crashed. https://news.ycombinator.com/item?id=41021366, 2024.

[177] Théo Winterhalter, Cezar-Constantin Andrici, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, and Exequiel Rivas. Partial dijkstra monads for all. In *Proceedings of the International Conference on Types for Proofs and Programs (TYPES)*, 2022.

[178] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi: 10.1145/3485522. URL https://doi.org/10.1145/3485522.

[179] Nikita Zyuzin and Aleksandar Nanevski. Contextual modal types for algebraic effects and handlers. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi: 10.1145/3473580. URL https://doi.org/10.1145/3473580.

# 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 *)
% \end{minted}
%
```

Fig. A.1. `foldr` and `map` in Cameleer [136]

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 parameterization 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.