Continuation Passing as an Abstract Syntax for Deductive Verification

Andrei Paskevich

Continuation Passing as an Abstract Syntax for Deductive Verification

Andrei Paskevich

Abstract

Continuation-passing style allows us to devise an extremely economical abstract syntax for a generic algorithmic language. This syntax is flexible enough to naturally express conditionals, loops, (higher-order) function calls, and exception handling. It is type-agnostic and state-agnostic, which means that we can combine it with a wide range of type and effect systems.

We argue that this syntax is also well suited for the purposes of deductive verification. Indeed, we show how it can be augmented in a natural way with specification annotations, ghost code, and side-effect discipline. We define the rules of verification condition generation for this syntax, and we show that the resulting formulas are nearly identical to what traditional approaches, like the weakest precondition calculus, produce for the equivalent algorithmic constructions.

To sum up, we propose a minimalistic yet versatile abstract syntax for annotated programs for which we can compute verification conditions without sacrificing their size, legibility, and amenability to automated proof, compared to more traditional methods. We believe that it makes it an excellent candidate for internal code representation in program verification tools.

1 Introduction

Suppose we want to develop a deductive verification tool for a simple imperative language. We start with the basic WHILE core — assignment, sequence, conditionals, and `while` loops — and, for the purposes of specification — assertions and loop invariants, and also loop variants, to prove termination. Then we add functions and procedures with their preconditions and postconditions, and auxiliary parameters, to catch the previous states of the modified variables. Auxiliary variables are nice but still quite limited, and so we add ghost variables, ghost function parameters, and ghost function results. Then we get adventurous and implement exception handling, which we already had in a rudimentary state in the form of `break`, `continue`, and `return` statements. Before long, our language has the abstract syntax tree with two dozen cases, some of which, like function calls or pattern matching, have a lot of components and are quite complex to handle.

We are fully aware that some (most?) of this complexity is justified and inevitable. And yet, we cannot help looking at lambda calculus, a universal computation formalism with mere three syntax rules and a single evaluation rule, and feeling like we have lost some truth, or at least some beauty, in the maelstrom of technical details.

The present work attempts to recover some of this loss, and it comes of one simple intuition: *In a structural language, a block of code has a single entry but multiple exits*. The "multiple exits" part means not just that there may be multiple exit points inside a block, but, more importantly, that there may be multiple ways in which a code block may end, leading to different execution paths afterwards, and with different postconditions to verify on exit.

This does not seem to be a universal perception: after all, the foundational tool of deductive verification, the Hoare triple, comes with a single postcondition. The notion that a program expression can only end in a single way and has to satisfy this postcondition on exit is a mental habit that we have to overcome when we start dealing with `break`, `continue`, `return`, and exceptions, both in code and in specification. We recognize the same habit in the type systems of ML-like languages, where exception-raising instructions are given type $\forall\alpha.\alpha$, which is a formal way to say "whatever". The type system simply cannot express that the expression does not produce any value (nor that it raises

an exception), and the best we can do is to say that the produced value can have any type at all. The usual way to explain this inadequacy is to say that exceptions are side effects, and the type system, with its noble lambda pedigree, cares not for such shenanigans. However, we firmly believe that it is much more useful to see exceptions not as a side effect but as a pure control structure, a sequence step towards a different code point. In fact, from the standpoint of specification and verification (as opposed to compilation and actual execution), there is not much reason to distinguish a "normal" and an "exceptional" way to leave a block of code — they are just different kinds of `return`.

From the same standpoint, and perhaps more radically, there is not much reason to distinguish function calls and function returns. Indeed, we describe them in a very similar manner: both are parametrized by data (received and returned), and both carry conditions (pre- and post-) to verify when the execution reaches the corresponding point.

Of course, an approach that treats exits as entries — and function returns as function calls — exists since the early days of computer science: it is called *continuation-passing style*. In this setting, leaving a block of code is done by calling another one (its *continuation*), and the multiplicity of possible outcomes comes naturally, as we are free to call any particular continuation at our disposal. It is an extremely powerful notion, which has seen no end of theoretical and practical applications, to which this work wants to add another one: being a great abstract syntax for deductive verification of imperative structural programs.

Let us consider, as an example, a function that computes the factorial of a natural number, in the syntax which we are going to use throughout this paper:

```
1 factorial (n: int) { 0 ⩽ n }
2  (return: (m: int) { m = n! })
3 = fact 1 n
4   / fact (r: int) (k: int)
5       { 0 ⩽ k ⩽ n ∧ r·k! = n! }
6     = if (k > 0) tt ff
7       / tt → fact (r * k) (k - 1)
8       / ff → return r
```

Lines 1 and 2 introduce and specify a *handler*, which is our name for a subroutine. The handler is called `factorial` and it expects a single data parameter, an integer named `n`, with a precondition that requires `n` to be non-negative. It also expects a single continuation parameter, named `return`, which takes an integer argument `m`, and the precondition for `return` is that `m` is the factorial of `n`.

Lines 3-8 contain the implementation of `factorial`. It consists in defining a recursive handler `fact`, expecting two data parameters, `r` and `k`, in lines 4-8, and calling it with arguments `1` and `n`, respectively, in line 3. The slash symbol separates the call from the definition. The precondition of `fact` is given on line 5. The implementation of `fact` tests whether `k` is positive, using a globally defined handler `if`, which has the following specification:

```
if (c: bool) (then: { c }) (else: { ¬c })
```

The `if` handler expects a single data parameter `c` along with two nullary continuation parameters, named `then` and `else`. The precondition of `then` is that `c` is true, and the precondition of `else` is that `c` is false. It is worth emphasizing that `if` is not part of the language, but just a handler, like `factorial` or `fact`, and its specification above is sufficient for verification.

Going back to `fact`, the actual handler arguments supplied to `if` are defined in the last two lines. Unlike the data argument `(k > 0)` which is evaluated eagerly, the handler arguments `tt` and `ff` are continuations, to be called by `if` depending on the value of the data argument. Whenever `k` is strictly positive, `if` executes `tt`, which makes a recursive call of `fact`. Otherwise, if `k` is zero or negative, `if` calls `ff`, which passes `r` to the `return` handler, leaving `factorial`. There is a small peculiarity in the way we define `tt` and `ff`: the right arrow instead of the equality sign means that these handlers are not given a complete specification, and their code must be verified separately at each invocation.

Note that `fact` never returns to the caller: to do that, it would need to receive a continuation parameter and call it, like `if` and `factorial` do. Instead, `fact` escapes by calling `return` at the end

of computation. In this respect, `fact` behaves rather like a loop than a recursive function. Indeed, its continuation is determined statically, by its lexical context, rather than dynamically by its caller. Consequently, there is no distinct postcondition associated to `fact`: in our language, postconditions are preconditions of continuation parameters, and `fact` has none thereof.

To verify a handler definition, we can compute the weakest precondition for the handler's body and check that it holds for all values of handler's parameters that satisfy the handler's precondition. Let us see how this works on our example (here, we ignore termination and we write $\mathcal{V}_{N\text{-}M}$ to denote the weakest precondition for the expression written in lines N to M):

$$
\begin{aligned}
\mathcal{V}_{\texttt{factorial}} &= \forall \mathsf{n}.\, 0 \leqslant \mathsf{n} \to \mathcal{V}_{3\text{-}8} \\
\mathcal{V}_{3\text{-}8} &= \mathcal{V}_3 \ \wedge\ \mathcal{V}_{\texttt{fact}} \\
\mathcal{V}_3 &= (0 \leqslant \mathsf{k} \leqslant \mathsf{n} \ \wedge\ \mathsf{r} \cdot \mathsf{k}! = \mathsf{n}!)\,[\mathsf{r} \mapsto 1, \mathsf{k} \mapsto \mathsf{n}] \\
&= 0 \leqslant \mathsf{n} \leqslant \mathsf{n} \ \wedge\ 1 \cdot \mathsf{n}! = \mathsf{n}! \\
\mathcal{V}_{\texttt{fact}} &= \forall \mathsf{r}\, \mathsf{k}.\, (0 \leqslant \mathsf{k} \leqslant \mathsf{n} \ \wedge\ \mathsf{r} \cdot \mathsf{k}! = \mathsf{n}!) \ \to\ \mathcal{V}_{6\text{-}8} \\
\mathcal{V}_{6\text{-}8} &= (\mathsf{k} > 0 \to \mathcal{V}_7) \ \wedge\ (\neg(\mathsf{k} > 0) \to \mathcal{V}_8) \\
\mathcal{V}_7 &= (0 \leqslant \mathsf{k} \leqslant \mathsf{n} \ \wedge\ \mathsf{r} \cdot \mathsf{k}! = \mathsf{n}!)\,[\mathsf{r} \mapsto \mathsf{r} \cdot \mathsf{k}, \mathsf{k} \mapsto \mathsf{k} - 1] \\
&= 0 \leqslant \mathsf{k} - 1 \leqslant \mathsf{n} \ \wedge\ \mathsf{r} \cdot \mathsf{k} \cdot (\mathsf{k} - 1)! = \mathsf{n}! \\
\mathcal{V}_8 &= (\mathsf{m} = \mathsf{n}!)\,[\mathsf{m} \mapsto \mathsf{r}] \\
&= \mathsf{r} = \mathsf{n}!
\end{aligned}
$$

The weakest precondition for an expression consists of the weakest precondition for the top-most handler call and the verification conditions for the attached handler definitions ($\mathcal{V}_{\texttt{fact}}$). Calls without continuation parameters ($\mathcal{V}_3$, $\mathcal{V}_7$, $\mathcal{V}_8$) require us to merely verify the instantiated precondition of the callee. Calls with continuation parameters ($\mathcal{V}_{6\text{-}8}$) also require checking that the actual handler arguments satisfy the specification of the formal continuation parameters. In our example, this means that the preconditions of the two continuation parameters in the contract of `if` must imply, respectively, the weakest preconditions of the actual continuations defined in lines 7 and 8.

Outline. Below we present our language: its syntax, semantics, type system, and procedures for effect analysis, deductive verification, and ghost code handling. The features of the language are introduced incrementally. We start with the pure fragment without annotations in Section 2. Then we introduce mutable state in Section 3, in a way that ensures the alias safety of well-typed programs. In Section 4, we add *pre-write annotations* that describe the effect that a given computation has on the program state. These annotations allow us to translate programs with mutable state into equivalent pure programs using a fine-grained monadic encoding.

In Section 5, we equip our handlers with functional specifications and show how to compute verification conditions for partial and total correctness. In particular, we demonstrate that classical Dijkstra-style weakest preconditions and efficient weakest preconditions as described by Flanagan and Saxe [**?**]. can both be derived as equivalent forms of the same verification condition.

Finally, in Section 6, we introduce the notion of ghost data: program variables that do not affect the main computation and are added solely to facilitate specification and proof. We devise a procedure that eliminates ghost variables and the code that depends on them, while checking that this elimination does not change the program behaviour.

Even before the formal introduction of various specification annotations, we systematically use them in our examples in order to better describe the behaviour of annotated handlers. In all such cases we provide an informal explanation of the meaning and purpose of these annotations.

2 Pure fragment

The building blocks of our language are handler definitions and handler calls. Contrary to the usual practice of functional programming languages, we separate code and data on the syntactic level: data terms (which we call simply *terms*) occur inside code terms (which we call *expressions*) as arguments in handler calls, but an expression cannot reduce (evaluate) to a term.

Here is the formal grammar of our base fragment:

$$
\begin{array}{rcl}
\mathit{expression} & ::= & \mathit{handler}\ \ \mathit{argument}* \\
& \mid & \mathit{expression}\ \ /\ \ \mathit{abbreviation} \\
& \mid & \mathit{expression}\ \ /\ \ \mathit{definition}+ \\
\mathit{argument} & ::= & \mathit{term} \mid \mathit{handler} \\
\mathit{abbreviation} & ::= & \mathit{handler}\ \ \mathit{contract}\ \ \twoheadrightarrow\ \ \mathit{expression} \\
\mathit{definition} & ::= & \mathit{handler}\ \ \mathit{contract}\ \ =\ \ \mathit{expression} \\
\mathit{contract} & ::= & \mathit{annotation}* \\
\mathit{annotation} & ::= & \mathit{variable}\ \ :\ \ \mathit{termType} \\
& \mid & \mathit{handler}\ \ :\ \ \mathit{contract}
\end{array}
$$

Just as we distinguish expressions and terms, we distinguish the names we assign to them, and call them *handlers* and *variables*, respectively. Thus, in handler contracts (which at this point are just lists of formal parameters), variables serve as data parameters and handlers as continuation parameters. From now on, we will call the former *term parameters* and the latter, *handler parameters*.

We introduce two kinds of handler definitions which behave in the same way during execution but are quite different from the verification standpoint. The first kind, called *abbreviations*, are verified independently at each invocation site, as if the handler definition were inlined in the code. The second kind, called *definitions* proper, are verified just once, when introduced, and any subsequent invocation of the handler is merely required to satisfy the preconditions in the contract. In other words, the contract in a handler definition must provide a complete specification of the handler's behaviour, sufficient to abstract over its implementation. This allows for recursive handler definitions, without the risk of introducing circular dependencies in verification conditions. Abbreviations, on the other hand, are not allowed to be recursive.

We use letters e and d for expressions, s and t for terms, x, y, z for variables, f, g, h for handlers, a and b for arguments (which can be terms or handlers), γ and δ for definitions and abbreviations, ϱ and π for contracts, τ and θ for term types. We use the underscore symbol (_) to denote a fresh "dummy" handler name, which is not supposed to be bound or used anywhere else in the program. In a contract, no variable or handler parameter can appear twice, just like no handler can be defined twice in a group of handler definitions.

In formal rules and definitions, we use two separator symbols, $\succ$ and $\prec$, to represent, respectively, *abstraction* and *application*. A pair $\varrho \succ e$ is a nameless handler definition, an implementation hidden behind a contract. A pair $\varrho \prec \bar{a}$ or a triple $(\varrho \succ e) \prec \bar{a}$ stand for an application of some unnamed handler, described by its contract and, possibly, its implementation, to a specific list of arguments. We denote empty sequences of annotations, arguments or definitions with the box symbol ($\Box$). Since empty definition groups are not allowed in our syntax, $e\ /\ \Box$ is identified with e.

Before we proceed to further presentation and analysis of our language, we shall briefly discuss its data side. For our purposes, we do not need precise definitions of the syntax and semantics of terms and term types. Instead, we assume a typing relation $\Gamma \vdash s : \tau$ that associates type τ with term s in typing context Γ, which provides types for the free variables in s. We also assume a type-preserving normalization operator $[\![s]\!]_\Sigma$ that reduces s to a ground normal form in evaluation context Σ, which provides values for the free variables in s. Handlers cannot occur inside terms. We require that terms can be directly used inside logical formulas (preconditions, verification conditions) with the same

meaning as in program code. As a consequence, terms must be total and pure, i.e., normalization can neither fail nor produce side effects; all partial or effectful computations are to be done by handlers. It is possible to restrict terms to variables and constants only, so that normalization is limited to variable instantiation, but in a practical implementation, it is convenient to admit some operations, too.

For concrete examples in our paper, we will use unbounded integers, Booleans, and polymorphic lists, together with a handful of basic operations on them:

$$
\begin{array}{rcl}
\mathit{term} & ::= & \mathit{variable} \\
& | & \texttt{true} \mid \texttt{false} \mid \ldots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
& | & \mathit{term}\ \texttt{+}\ \mathit{term} \mid \mathit{term}\ \texttt{-}\ \mathit{term} \mid \mathit{term}\ \texttt{*}\ \mathit{term} \\
& | & \mathit{term}\ \texttt{=}\ \mathit{term} \mid \mathit{term}\ \texttt{<}\ \mathit{term} \mid \mathit{term}\ \texttt{>}\ \mathit{term} \\
& | & \texttt{cons}\ \mathit{term}\ \mathit{term} \mid \texttt{nil} \mid \texttt{isCons}\ \mathit{term} \mid \ldots \\
\mathit{termType} & ::= & \mathit{typeVariable} \\
& | & \texttt{bool} \mid \texttt{int} \mid \texttt{list}\ \mathit{termType} \mid \ldots
\end{array}
$$

Typing and normalization rules for this fragment are as usual, and we do not write them here.

We compute the free variables and free handlers in expressions and abstractions as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{FV}(h\ \square) & \triangleq & \varnothing & \mathrm{FV}(e\ /\ h\ \varrho \to d) & \triangleq & \mathrm{FV}(e) \cup \mathrm{FV}(\varrho \succ d) \\
\mathrm{FV}(h\ \bar{a}\ s) & \triangleq & \mathrm{FV}(h\ \bar{a}) \cup \mathrm{FV}(s) & \mathrm{FV}(e\ /\ \bar{\gamma}\ (h\ \varrho = d)) & \triangleq & \mathrm{FV}(e\ /\ \bar{\gamma}) \cup \mathrm{FV}(\varrho \succ d) \\
\mathrm{FV}(h\ \bar{a}\ g) & \triangleq & \mathrm{FV}(h\ \bar{a}) & \mathrm{FV}((x:\tau)\ \varrho \succ e) & \triangleq & \mathrm{FV}(\varrho \succ e) \setminus \{x\} \\
\mathrm{FV}(\square \succ e) & \triangleq & \mathrm{FV}(e) & \mathrm{FV}((g:\pi)\ \varrho \succ e) & \triangleq & \mathrm{FV}(\varrho \succ e) \cup \mathrm{FV}(\pi \succ \_\ \square)
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{FH}(h\ \square) & \triangleq & \{h\} & \mathrm{FH}(e\ /\ h\ \varrho \to d) & \triangleq & (\mathrm{FH}(e) \setminus \{h\}) \cup \mathrm{FH}(\varrho \succ d) \\
\mathrm{FH}(h\ \bar{a}\ s) & \triangleq & \mathrm{FH}(h\ \bar{a}) & \mathrm{FH}(e\ /\ \bar{\gamma}\ (h\ \varrho = d)) & \triangleq & \mathrm{FH}(e\ /\ (\_\ \varrho \to d)\ /\ \bar{\gamma}) \setminus \{h\} \\
\mathrm{FH}(h\ \bar{a}\ g) & \triangleq & \mathrm{FH}(h\ \bar{a}) \cup \{g\} & \mathrm{FH}((x:\tau)\ \varrho \succ e) & \triangleq & \mathrm{FH}(\varrho \succ e) \\
\mathrm{FH}(\square \succ e) & \triangleq & \mathrm{FH}(e) & \mathrm{FH}((g:\pi)\ \varrho \succ e) & \triangleq & \mathrm{FH}(\varrho \succ e) \setminus \{g\}
\end{array}
$$

We call an expression *closed* if it contains no free variables. The abstraction $\pi \succ \_\ \square$ used in the last rule for FV allows us to compute the free variables in contract π by creating an abstraction over a dummy handler applied to an empty list of arguments. The actual name of the handler does not matter as it does not affect the result. Though at this point there are no free occurrences of variables in contracts, this will change in the following sections when we add effect annotations and preconditions. We do not apply this trick for FH, as handlers do not, and will not, freely occur in contracts.

Also noteworthy is the rule to compute the free handlers in an expression with a group of handler definitions. Since handler definitions can be recursive, the definition body, d, may contain handlers defined in $\bar{\gamma}$. To account for this, we transform the definition $h\ \varrho = d$ into an abbreviation $\_\ \varrho \to d$ and move it under $\bar{\gamma}$. Again, the name of the dummy handler does not matter here, as it is fresh and thus cannot occur in e. After computing the free handlers in the resulting expression, we remove h from the set. This rule also applies when $\bar{\gamma}$ is empty: as we said before, $e\ /\ \square$ is just e.

Operational semantics. We define a small-step operational semantics of our language as follows. An *evaluation step* is presented as transition relation $\Lambda \vdash e \bullet \Sigma \longrightarrow e' \bullet \Sigma'$. Here, Λ is a *handler evaluation context* that maps handlers to abstractions, and Σ, Σ' are *variable evaluation contexts* that map variables to normalized ground terms. Handler context Λ keeps track of handler definitions and abbreviations. Variable contexts Σ and Σ' store values of mutable variables (Section 3), in the state before and after the evaluation step, respectively.

Here are the evaluation rules for the current syntax:

$$\frac{\Lambda, h \mapsto \varrho \succ d \vdash e \bullet \Sigma \longrightarrow e' \bullet \Sigma'}{\Lambda \vdash e \,/\, h\ \varrho \to d \bullet \Sigma \longrightarrow e' \,/\, h\ \varrho \to d \bullet \Sigma'} \quad \text{(E-Abbr)}$$

$$\frac{\Lambda, h \mapsto \varrho \succ d \vdash e \,/\, \bar{\gamma} \bullet \Sigma \longrightarrow e' \,/\, \bar{\gamma} \bullet \Sigma'}{\Lambda \vdash e \,/\, \bar{\gamma}\ (h\ \varrho = d) \bullet \Sigma \longrightarrow e' \,/\, \bar{\gamma}\ (h\ \varrho = d) \bullet \Sigma'} \quad \text{(E-Defn)}$$

$$\frac{h \mapsto \varrho \succ d \in \Lambda \qquad \Sigma \vdash (\varrho \succ d) \prec \bar{a} \longrightarrow^* (\square \succ e) \prec \square}{\Lambda \vdash h\ \bar{a} \bullet \Sigma \longrightarrow e \bullet \Sigma} \quad \text{(E-Call)}$$

$$\Sigma \vdash ((x : \tau)\ \varrho \succ e) \prec s\ \bar{a} \longrightarrow (\varrho \succ e)[x \mapsto [\![s]\!]_\Sigma] \prec \bar{a} \quad \text{(E-AppT)}$$

$$\Sigma \vdash ((g : \pi)\ \varrho \succ e) \prec f\ \bar{a} \longrightarrow (\varrho \succ e)[g \mapsto f] \prec \bar{a} \quad \text{(E-AppH)}$$

Rules E-Abbr and E-Defn evaluate expressions under handler abbreviations and definitions, by converting those into abstractions, which are added to the handler context Λ. Rule E-Call applies when the called handler is introduced by an abstraction or a definition, and therefore is found in Λ. An auxiliary transition relation $\Sigma \vdash (\varrho \succ e) \prec \bar{a} \longrightarrow (\varrho' \succ e') \prec \bar{a}'$, which we use here, is essentially β-reduction: it replaces formal term and handler parameters with the actual arguments; the term arguments are normalized under Σ. We assume that substitution into abstractions and expressions, as performed by rules E-AppT and E-AppH, renames bound variables and handlers as needed to avoid illegal capture. A single E-Call step requires full parameter application: all binders in the contract must be lifted and all arguments must be substituted into the handler body.

Of course, these evaluation rules are not enough. Handler calls and definitions are mere control structures, and all actual work has to be done by predefined *library handlers*, such as `if` which we used in the example in the introduction.

The semantics of library handlers is axiomatized by evaluation rules like these:

$$\frac{[\![s]\!]_\Sigma = \texttt{true}}{\Lambda \vdash \texttt{if}\ s\ f\ g \bullet \Sigma \longrightarrow f \bullet \Sigma} \qquad \frac{[\![s]\!]_\Sigma = \texttt{false}}{\Lambda \vdash \texttt{if}\ s\ f\ g \bullet \Sigma \longrightarrow g \bullet \Sigma}$$

$$\frac{[\![s]\!]_\Sigma = \texttt{cons}\ t_1\ t_2}{\Lambda \vdash \texttt{unList}\ s\ f\ g \bullet \Sigma \longrightarrow f\ t_1\ t_2 \bullet \Sigma} \qquad \frac{[\![s]\!]_\Sigma = \texttt{nil}}{\Lambda \vdash \texttt{unList}\ s\ f\ g \bullet \Sigma \longrightarrow g \bullet \Sigma}$$

$$\frac{[\![t]\!]_\Sigma \neq 0 \quad [\![s]\!]_\Sigma \operatorname{div} [\![t]\!]_\Sigma = n}{\Lambda \vdash \texttt{div}\ s\ t\ f \bullet \Sigma \longrightarrow f\ n \bullet \Sigma} \qquad \frac{[\![t]\!]_\Sigma \neq 0 \quad [\![s]\!]_\Sigma \bmod [\![t]\!]_\Sigma = n}{\Lambda \vdash \texttt{mod}\ s\ t\ f \bullet \Sigma \longrightarrow f\ n \bullet \Sigma}$$

Note that just like the conditional, pattern matching does not need to be part of the core language syntax and can be delegated to a handler. Of course, if we were implementing a compiler, we would still have to implement the actual pattern matching functionality. But since we are interested in static analysis and verification, we only need the specification of the library handlers:

```
if (c: bool) (then: { c }) (else: { ¬c })

unList (l: list α)
  (onCons: (h: α) (t: list α) { l = cons h t })
  (onNil: { l = nil })

div (a: int) (b: int) { b ≠ 0 }
  (return: (q: int) { ∃r. 0 ⩽ r < |b| ∧ a = b · q + r })

mod (a: int) (b: int) { b ≠ 0 }
  (return: (r: int) { 0 ⩽ r < |b| ∧ ∃q. a = b · q + r })
```

which can be readily expressed in our language and used for verification condition generation in the same way as for user-defined handlers.

Typing rules. While our language is compatible with various type systems, we endow it here with ML-style polymorphic rank-1 types, to benefit from effective Hindley-Milner type inference. We will however restrict polymorphism to data types; the arity of handlers is invariant. Here is the grammar of expression and handler types:

$$\begin{array}{rcl} \textit{expressionType} & ::= & \bot \\ & | & \textit{termType} \;\to\; \textit{expressionType} \\ & | & \textit{expressionType} \;\to\; \textit{expressionType} \\ \textit{handlerType} & ::= & \langle\ \textit{typeVariable}{*}\ \rangle\ \textit{expressionType} \end{array}$$

Expression types are also assigned to contracts and abstractions. Type $\bot$ is that of expressions that expect no additional arguments and can be readily executed. This type is also given to nullary contracts and abstractions. Handler calls with an incomplete argument list have arrow types; so do the contracts and abstractions that have formal parameters. Handler types can be generalized in some of their type variables and carry the list of these type variables inside angled brackets before their base type. When a handler type is monomorphic, i.e., not generalized in any of its type variables, we omit the prefix. Library handlers are always fully generalized. For example, the type of `unList` is $\langle\alpha\rangle$ `list` $\alpha \to (\alpha \to$ `list` $\alpha \to \bot) \to \bot \to \bot$. We denote expression types with letters ξ and ζ.

Typing judgements $\Gamma \vdash X : \xi$, where X is an expression, a contract or an abstraction, are made under typing context Γ, which binds variables to term types and handlers to handler types. The types of library handlers are always present in Γ. Here are the typing rules:

$$\frac{h : \langle\bar\alpha\rangle\xi \in \Gamma}{\Gamma \vdash h\ \square\ :\ \xi[\bar\alpha \mapsto \bar\tau]}\ \text{(T-Call)} \qquad \frac{\Gamma \vdash h\ \bar a : \tau \to \xi \qquad \Gamma \vdash s : \tau}{\Gamma \vdash h\ \bar a\ s : \xi}\ \text{(T-AppT)}$$

$$\frac{\Gamma \vdash \varrho \succ d : \xi \qquad \Gamma, h : \xi \vdash e : \bot}{\Gamma \vdash e\ /\ h\ \varrho \twoheadrightarrow d : \bot}\ \text{(T-Abbr)} \qquad \frac{\Gamma \vdash h\ \bar a : \zeta \to \xi \qquad \Gamma \vdash g\ \square : \zeta}{\Gamma \vdash h\ \bar a\ g : \xi}\ \text{(T-AppH)}$$

$$\frac{\Gamma \vdash \varrho : \xi \qquad \Gamma, h : \langle\bar\alpha\rangle\xi \vdash e\ /\ (\_\ \varrho \twoheadrightarrow d)\ /\ \bar\gamma : \bot \qquad \text{no type variable in } \bar\alpha \text{ is free in } \Gamma}{\Gamma \vdash e\ /\ \bar\gamma\ (h\ \varrho = d) : \bot}\ \text{(T-Defn)}$$

$$\frac{\Gamma, \_ : \bot \vdash \pi \succ \_\ \square : \zeta}{\Gamma \vdash \pi : \zeta}\ \text{(T-Contract)} \qquad \frac{\Gamma, x : \tau \vdash \varrho \succ e : \xi}{\Gamma \vdash (x : \tau)\ \varrho \succ e : \tau \to \xi}\ \text{(T-ParT)}$$

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash \square \succ e : \bot}\ \text{(T-Nullary)} \qquad \frac{\Gamma \vdash \pi : \zeta \qquad \Gamma, g : \zeta \vdash \varrho \succ e : \xi}{\Gamma \vdash (g : \pi)\ \varrho \succ e : \zeta \to \xi}\ \text{(T-ParH)}$$

Rule T-Call specializes handler types by instantiating their generalized type variables with arbitrary data types. When we pass a handler as an argument (rule T-AppH), we check that it can be specialized into the expected type ζ by typechecking its call with an empty argument list.

Rule T-Defn handles mutually recursive definitions using the same technique as the corresponding rule for FH: each definition is converted into an abbreviation under a fresh dummy handler name and pushed inside the expression. Handlers introduced by a definition are generalized in type variables that are not fixed by the typing context.

Handlers introduced by an abbreviation (rule T-Abbr) are not generalized. While there is no fundamental reason why they should not be, this restriction simplifies some of our procedures in later sections. Moreover, abbreviations are expected to represent proper continuations, i.e., code that retrieves and processes the results of a given computation. In ML-like languages, such code is not generalized: when we write `let x =` e_1 `in` e_2, the type of `x` is fixed in e_2.

Rule T-Contract derives the type of a contract from the type of an abstraction over a call of a dummy handler with an empty argument list, similarly to how we defined FV for handler parameters.

Finally, the types of handler parameters (rule T-ParH) are not generalized, in agreement with the rules of prenex polymorphism.

```
let rec find_greater (n: int) (l: int list) : int
= match l with
  | [] → raise Not_found
  | h :: t → if h > n then h else find_greater n t

let check_greater (n: int) (l: int list) : bool
= try let _ = find_greater n l in true
  with Not_found → false
```

Figure 1: Simple OCaml program.

```
find_greater (n: int) (l: list int)
  (return: (r: int) { r ∈ l ∧ r > n })
  (not_found: { ∀x ∈ l. x ⩽ n })
= unList l on_cons not_found
  / on_cons (h: int) (t: list int) →
      if (h > n) tt ff
      / tt → return h
      / ff → find_greater n t return not_found

check_greater (n: int) (l: list int)
  (return: (r: bool) { r ↔ ∃x ∈ l. x > n })
= find_greater n l found not_found
  / found (x: int) → return true
  / not_found → return false
```

Figure 2: The program from Fig. 1, translated.

Control structures. We conclude with another example showing how usual programming constructs — recursion, pattern matching, and exception handling — are expressed in our language. In Figure 1, we define two OCaml functions. Recursive function `find_greater` returns the first element in a singly-linked list that exceeds a given integer or raises the `Not_found` exception if none such is found. Function `check_greater` uses `find_greater` to test whether a list contains an element greater than a given integer, and catches the `Not_found` exception to return a negative result.

In Figure 2, we translate these functions into our language and add functional specifications. We use the previously shown library handler `unList` to deconstruct a list value. Notice that we do not put any precondition in the contract of `on_cons`: the body of this abbreviation will be verified in the context of its usage. Exceptions are translated using additional handler parameters: `find_greater` receives one handler parameter named `return` for normal termination, and another, named `not_found`, for exceptional termination. The caller, `check_greater`, provides a continuation for both outcomes. If the OCaml version of `check_greater` did not catch and handle the exceptional outcome of `find_greater`, then its specification in our language would receive an additional handler parameter to reflect that `check_greater` can raise an exception, and this parameter would be passed verbatim to `find_greater`.

discuss automated translation?

This example underlines an important feature of a continuation-based representation: it naturally admits multiple outcomes for a unit of computation and makes no distinction between normal and exceptional outcomes. Each possible outcome is shown in the handler's contract and characterized by its own specification, modeled as a precondition of the corresponding handler parameter.

Furthermore, by using the same notion of a handler (i.e., continuation) to represent entries and exits, units of computation and their outcomes, we achieve a rather elegant economy of concepts: results are modeled as parameters, postconditions as preconditions, etc.

3 Mutable State

Our next extension consists in adding mutable variables. We shall restrict our language to alias-free programs, for which simple and efficient techniques of program verification, like weakest precondition calculus, can be effectively applied. We do not introduce a pointer type, but rather treat certain variables as *references* to mutable memory cells. For this, we extend the syntax of expressions and handler contracts as follows:

$$\begin{aligned} \textit{expression} &::= \ \ldots \mid \textit{expression}\ \texttt{/ \&}\ \textit{variable}\ \texttt{=}\ \textit{term} \\ \textit{argument} &::= \ \ldots \mid \texttt{\&}\ \textit{variable} \\ \textit{annotation} &::= \ \ldots \mid \texttt{\&}\ \textit{variable}\ \texttt{:}\ \textit{termType} \end{aligned}$$

On the lexical level, references are represented with ordinary variables. In particular, they can be used inside terms and preconditions. On introduction and usage, we distinguish references with the ampersand symbol &. We denote references with letters p, q, r.

Let us extend the definitions of FV and FH for the new constructs:

$$\begin{aligned} \mathrm{FV}(e\ \texttt{/ \&}r\ \texttt{=}\ s) &\triangleq (\mathrm{FV}(e) \setminus \{r\}) \cup \mathrm{FV}(s) & \mathrm{FH}(e\ \texttt{/ \&}r\ \texttt{=}\ s) &\triangleq \mathrm{FH}(e) \\ \mathrm{FV}(h\ \bar{a}\ \texttt{\&}r) &\triangleq \mathrm{FV}(h\ \bar{a}) \cup \{r\} & \mathrm{FH}(h\ \bar{a}\ \texttt{\&}r) &\triangleq \mathrm{FH}(h\ \bar{a}) \\ \mathrm{FV}((\texttt{\&}r : \tau)\ \varrho \succ e) &\triangleq \mathrm{FV}(\varrho \succ e) \setminus \{r\} & \mathrm{FH}((\texttt{\&}r : \tau)\ \varrho \succ e) &\triangleq \mathrm{FH}(\varrho \succ e) \end{aligned}$$

In the syntax rules above, the added productions correspond, respectively, to reference allocation, passing reference as an argument, and declaring a formal reference parameter. We do not need special syntax for dereferencing, as references that occur inside terms as variables are dereferenced implicitly. As for assignment, this functionality is, once again, delegated to a library handler:

```
assign (&r: α) (v: α) (return: [r] { r = v })
```

Library handler `assign` takes a reference parameter `r` and a term parameter `v` of the same generic type α. The handler parameter `return` takes no parameters (that is, `assign` returns no result), and the precondition of `return` states that the value stored in `r` when `assign` returns is equal to `v`. This can only be ensured if `assign` changes the content of `r`, and, indeed, the *pre-write annotation* `[r]` attached to `return` means that reference `r` is potentially modified during the execution of `assign` before `return` is called. Pre-write annotations are part of the functional specification and will be formally introduced in Section 4.

To give a better intuition of the new constructs, let us see how dereferencing and allocation can be implemented as handlers:

```
access (&r: α) (return: (v: α) { v = r }) = return r

alloc (v: α) (return: (&r: α) { r = v }) = return &q / &q = v
```

Handler `access` takes a reference parameter `r` and a handler parameter `return`. The latter expects a single term parameter whose value must be equal to the value of `r` at the moment when `return` is called. Application `return r` treats `r` as a term argument (note the absence of & in front of `r`), and therefore passes to `return` the current value stored in `r`, which satisfies the precondition of `return`.

Handler `alloc` takes a term parameter `v` and returns (via its handler parameter) a reference containing the value of `v`. The no-alias restriction ensures that the reference returned by `alloc` is fresh, and we do not need to specify this explicitly in the contract. The implementation of `alloc` allocates a fresh reference `q` initialized to `v` and returns it. This does not create a "dangling pointer", since there is no stack shrinking on return: rule E-Call replaces a call to `alloc` with its appropriately instantiated body, and however the actual return handler is evaluated, it always stays in the scope of the new reference.

Operational semantics. Here are the evaluation rules for expressions with references:

$$\Sigma \vdash ((\&q : \tau)\ \varrho \succ e) \prec \&r\ \bar{a} \longrightarrow (\varrho \succ e)[q \mapsto r] \prec \bar{a} \qquad \text{(E-AppR)}$$

$$\frac{\Lambda \vdash e \bullet \Sigma, r \mapsto [\![s]\!]_\Sigma \longrightarrow e' \bullet \Sigma', r \mapsto t}{\Lambda \vdash e \mathbin{/} \&r = s \bullet \Sigma \longrightarrow e' \mathbin{/} \&r = t \bullet \Sigma'} \qquad \text{(E-Alloc)}$$

Rule E-AppR (which participates together with E-AppT and E-AppH in evaluating handler calls) instantiates a formal reference parameter q with an actual reference argument r. Note that the latter is not normlalized under Σ unlike the term argument in E-AppT: normalization replaces references with their values in the variable context.

Rule E-Alloc evaluates the underlying expression e in the extended variable context that binds the new reference to the normalized initialization term. The value of this reference in the post-state is stored inside the allocation clause. In other words, allocations play the role of the mutable store inside expressions. In this way, we can preserve the lexical scope relationship between references and handlers during evaluation, after allocation takes place.

The operational semantics of `assign` is defined as follows:

$$\frac{\Sigma = \Sigma_1, r \mapsto t, \Sigma_2 \qquad \Sigma' = \Sigma_1, r \mapsto [\![s]\!]_\Sigma, \Sigma_2}{\Lambda \vdash \texttt{assign}\ \&r\ s\ f \bullet \Sigma \longrightarrow f \bullet \Sigma'}$$

For instance, (`assign &q (q*q+q) k / ... / &q = 6`) evaluates to (`k / ... / &q = 42`).

Typing rules. Reference parameters add a new kind of type for expressions:

$$\textit{expressionType} \ ::= \ \ldots \mid \& \ \textit{termType} \ \rightarrow \ \textit{expressionType}$$

The typing rules for the new constructs are as follows:

$$\frac{\Gamma \vdash s : \tau \qquad \Gamma, \&r : \tau \vdash e : \bot}{\Gamma \vdash e \mathbin{/} \&r = s : \bot}\ \text{(T-Alloc)} \qquad \frac{\Gamma, \&q : \tau \vdash \varrho \succ e : \xi}{\Gamma \vdash (\&q : \tau)\ \varrho \succ e : \&\tau \rightarrow \xi}\ \text{(T-ParR)}$$

$$\frac{\Gamma, \Delta' \vdash h\ \bar{a} : \&\tau \rightarrow \xi \qquad \Delta' \text{ is } \Delta \text{ with all handler bindings removed}}{\Gamma, \&r : \tau, \Delta \vdash h\ \bar{a}\ \&r : \xi}\ \text{(T-AppR)}$$

In typing contexts, we distinguish references from ordinary variables using the & symbol. The rules for allocation and abstraction are straightforward; notice that just like handler definitions, reference allocations can only be put over $\bot$-typed, that is, executable, expressions.

Rule T-AppR, which handles passing of reference arguments, requires the applicand to be type-checked in a restricted context, from which the reference argument and all handlers that come after it have been evicted. This is done to ensure that well-typed programs are alias-free. In our language, the only way to create an alias — a possibility to access the same memory location under two different names — is to give another name to a reference by passing it as a reference argument to an expression. Rule T-Call verifies that this expression does access this reference in any other way, neither directly, nor through a handler. To see how this restriction works, consider two typical cases of aliasing:

```
f &r &r / &r = 0                          g &r / g (&p: int) = ...
        / f (&p: int) (&q: int) = ...           / &r = 0
```

In the expression on the left, call (`f &r &r`) evaluates to the body of `f`, where both `p` and `q` are instantiated with `r`, creating an alias. In the expression on the right, call (`g &r`) evaluates to the body of `g`, where `p` is instantiated with `r`. This also creates an alias, since `g` is already in the scope of reference `r` and may access it directly.

Rule T-Call excludes both of these programs. Indeed, (`f &r &r`) cannot be type-checked, because the typing context for (`f &r`) does not contain a binding for `&r`. Call (`g &r`) cannot be type-checked either, because the typing context for `g` does not contain a binding for `g` itself.

As a small exercise, consider the following handler definitions:

```
alias1 (&x: int) (return: (&y: int)) = return &x
alias2 (return: (&y: int)) (&x: int) = return &x
```

Are these definitions accepted by our type system? Is application of these handlers alias-safe?

4 Effect Computation

The next element of specification we add to our language are the *pre-write annotations* that specify, for any given handler h, which references, visible to h, are potentially modified after h is introduced and before it is executed. We consider that a handler is introduced when its parent handler is executed; this holds both for defined handlers and handler parameters. We extend the syntax of contracts:

$$\textit{contract} \;::=\; \textit{pre-write}* \;\; \textit{annotation}*$$

$$\textit{pre-write} \;::=\; \texttt{[}\; \textit{variable} \;\texttt{]}$$

We do not put the & symbol before the reference name, since non-mutable variables cannot appear in a pre-write annotation and thus there is no possible ambiguity. We assume that all references in the pre-write annotations in any given contract are distinct. The references listed as pre-writes in the contract of handler h are said to be the *formal pre-writes* of h.

References in pre-write annotations are considered to occur freely in the contract:

$$\mathrm{FV}(\texttt{[}p\texttt{]}\ \varrho \succ e) \;\triangleq\; \mathrm{FV}(\varrho \succ e) \cup \{p\} \qquad\qquad \mathrm{FH}(\texttt{[}p\texttt{]}\ \varrho \succ e) \;\triangleq\; \mathrm{FH}(\varrho \succ e)$$

Being part of specification, pre-writes do not affect evaluation:

$$\Sigma \vdash (\texttt{[}p\texttt{]}\ \varrho \succ e) \prec \bar{a} \longrightarrow (\varrho \succ e) \prec \bar{a} \tag{E-Eff}$$

Typing rules are extended in a straightforward way:

$$\frac{\&p : \tau \in \Gamma \qquad \Gamma \vdash \varrho \succ e : \xi}{\Gamma \vdash \texttt{[}p\texttt{]}\ \varrho \succ e : \xi} \tag{T-Eff}$$

Effect checking. The pre-write annotations in a program must be exhaustive in the sense that every actual write effect that may happen during evaluation is accounted for. We can effectively check that user-written annotations are exhaustive; for this, we compute an overapproximation of *actual pre-writes* for each handler bound in our code, and we check that each reference in this overapproximation is listed as a formal pre-write for that handler. In the rest of this section, we describe this procedure and discuss the restrictions that it imposes on the programs we can write.

An *effect set* or simply an *effect* is a set of pairs (r, h) such that reference r is a potential pre-write for handler h. We can compute the effect set of any given expression, abstraction or application. For example, if we consider a handler call `assign &p 42 g`, which writes `42` into reference `p` and then passes control to handler `g`, then `p` is a pre-write for `g`, and the effect of the call is the singleton set $\{(\mathsf{p}, \mathsf{g})\}$. We denote effect sets with capital letter E.

In an abstraction $\texttt{[}p\texttt{]}\ \varrho \succ e$, reference p is considered as a potential pre-write for every handler occurring in e and not bound by ϱ. The corresponding effect set is denoted $p \otimes \varrho \succ e$ and is formally defined as the cartesian product $\{p\} \times \mathrm{FH}(\varrho \succ e)$.

When a reference is bound in an expression or an abstraction, it is evicted from the underlying effect set. We denote this operation $E \ominus r$ and define it as $\{\, (q, g) \in E \mid q \neq r \,\}$.

Similarly, when a handler is bound in an expression or an abstraction, we forget the pre-writes computed for this handler, on condition that they are all duly listed in the handler's contract:

$$E \ominus h \;\triangleq\; \begin{cases} \{\, (q, g) \in E \mid g \neq h \,\} & \text{if for each } (p, h) \text{ in } E,\ p \text{ is a formal pre-write of } h, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For the sake of simplicity, we assume the contract of handler h to be known from the context.

Operator $\mathcal{E}$ computes the effect set for a given expression, abstraction or application:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{E}(h\ \bar{a}) & \triangleq & \mathcal{E}(\varrho_h \prec \bar{a}) & \mathcal{E}(e\ /\ \bar{\gamma}\ (h\ \varrho = d)) & \triangleq & \mathcal{E}(e\ /\ (_\ \varrho \twoheadrightarrow d)\ /\ \bar{\gamma}) \ominus h \\
\mathcal{E}(e\ /\ \&r = s) & \triangleq & \mathcal{E}(e) \ominus r & \mathcal{E}(e\ /\ h\ \varrho \twoheadrightarrow d) & \triangleq & (\mathcal{E}(e) \ominus h) \cup \mathcal{E}(\varrho \succ d) \\
\mathcal{E}([p]\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) \cup (p \otimes \varrho \succ e) & \mathcal{E}([p]\ \varrho \prec \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\
\mathcal{E}((x:\tau)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) & \mathcal{E}((x:\tau)\ \varrho \prec s\ \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\
\mathcal{E}((\&q:\tau)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) \ominus q & \mathcal{E}((\&q:\tau)\ \varrho \prec \&r\ \bar{a}) & \triangleq & \mathcal{E}(\varrho[q \mapsto r] \prec \bar{a}) \\
\mathcal{E}((f:\pi)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) \ominus f & \mathcal{E}((f:\pi)\ \varrho \prec h\ \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \cup \mathcal{E}(\pi \succ h\ \overline{xqg}_\pi) \\
\mathcal{E}(\square \succ e) & \triangleq & \mathcal{E}(e) & \mathcal{E}(\square \prec \square) & \triangleq & \varnothing
\end{array}$$

In the first rule, ϱ_h stands for a specialized (that is, type-instantiated and with all binders refreshed) contract of h. The fourth rule for applications, $\mathcal{E}((f:\pi)\ \varrho \prec h\ \bar{a})$, deals with handler parameters. The code to which we pass handler h may perform write operations before calling h. These writes are listed as formal pre-writes in contract π and they constitute actual potential pre-writes for h. In order to compute the corresponding effect set, we form an abstraction $\pi \succ h\ \overline{xqg}_\pi$, where $\overline{xqg}_\pi$ denotes the list of all formal parameters of π, in the order of their appearance in the contract (we use this three-letter notation to remind of the three kinds of parameters a handler can have, namely, term, reference, and handler parameters). In other words, we create a *virtual definition*, where the actual handler argument h is used to implement the formal handler parameter f. The same rule will be applied to the handler parameters in $\overline{xqg}_\pi$, which will effectively allow us to verify that the contracts of the handler parameters of h are compatible with what is supplied by π.

Let us return to the call `assign &p 42 g`. The contract of `assign`, which we saw in Section 3, is `(&r: α) (v: α) (return: [r] { r = v })`. Ignoring the precondition of `return`, which is irrelevant here, the effect of the application is computed as follows:

$$\begin{array}{l}
\mathcal{E}((\&\mathtt{r}\colon \alpha)\ (\mathtt{v}\colon \alpha)\ (\mathtt{return}\colon [\mathtt{r}]) \prec \&\mathtt{p}\ 42\ \mathtt{g}) = \\
\qquad \mathcal{E}((\mathtt{v}\colon \alpha)\ (\mathtt{return}\colon [\mathtt{p}]) \prec 42\ \mathtt{g}) = \mathcal{E}((\mathtt{return}\colon [\mathtt{p}]) \prec \mathtt{g}) = \\
\qquad\quad \mathcal{E}(\square \prec \square) \cup \mathcal{E}([\mathtt{p}] \succ \mathtt{g}) = \varnothing \cup \mathcal{E}(\square \succ \mathtt{g}) \cup (\mathtt{p} \otimes \square \succ \mathtt{g}) = \\
\qquad\qquad \mathcal{E}(\mathtt{g}) \cup \{(\mathtt{p}, \mathtt{g})\} = \mathcal{E}(\varrho_{\mathtt{g}} \prec \square) \cup \{(\mathtt{p}, \mathtt{g})\} = \{(\mathtt{p}, \mathtt{g})\}
\end{array}$$

We do not specify the contract of `g`, but since this handler takes no parameters (we assume the initial expression to be well-typed), the effect set $\mathcal{E}(\varrho_{\mathtt{g}} \prec \square)$ cannot be anything but empty.

The effect $\mathcal{E}(X)$ is undefined when an actual pre-write computed for a handler introduced in X (this includes the handler parameters in virtual definitions) is not listed as a formal pre-write in the handler's contract. A program whose effect set is undefined is considered ill-specified and is rejected. In what follows, we assume that all programs we consider have a well-defined effect.

Effect inference. The effect-checking rules above can be used to infer pre-write annotations in user-written code that does not come with ones. Indeed, we can treat missing formal pre-writes as meta-variables designing sets of references, and the conditions of the $E \ominus h$ would supply constraints on those sets that must be satisfied for the program to be acceptable. While multiple solutions are generally possible (it is legal to declare a reference as a formal pre-write even if it is never modified), it is reasonable to look for minimal sufficient annotations by default.

It is important to notice that the solution space is finite. Indeed, the typing rules require any reference listed as a formal pre-write of a handler to be present in the typing context of that handler; thus, there is only a finite number of references that can appear in the pre-write annotation of any handler in our code. The effect sets computed by the $\mathcal{E}$ operator are similarly bounded. Indeed, it is easy to see that for any expression or abstraction X, any handler that occurs in $\mathcal{E}(X)$ must freely occur in X. The same is not necessarily true for references, since a pre-write may be introduced via the contract of a called handler, without occurring explicitly in the call. However, since this handler must be in the scope of the concerned reference, and X must be in the scope of the handler, we can conclude that any reference in the effect set of X must be visible in X.

```
let rec iter (l: int list)
         (fn: int → unit) : unit
= match l with
  | [] → ()
  | h :: t → fn h; iter t fn

let sum (l: int list) : int
= let r = ref 0 in
  let add (i: int) : unit =
    r := i + !r in
  iter l add;
  !r
```

```
iter (l: list int)
  (fn: (i: int) (ret_fn:)) (return:)
= unList l on_cons return
  / on_cons h t → fn h tail
    / tail → iter t fn return

sum (l: list int) (return: (s: int))
= iter l add out
  / add [r] (i: int) (ret_add: [r])
    = assign &r (i + r) ret_add
  / out [r] → return r
  / &r = 0
```

Figure 3: Illicit hidden side effects.

No hidden side effects. Our effect system is sufficiently precise to allow effective computation of verification conditions, as we will see in the coming sections. However, this precision comes at a price, as it further restricts the expressiveness of our language by forbidding programs to have hidden side effects.

Consider the code in Figure 3, OCaml version on the left, translation in our syntax on the right. In this example, we omit all functional specification. Function `iter` is an idiomatic second-order list iterator, and function `sum` uses it to compute the sum of a list of integers in a local reference `r`, whose value is then returned as the result.

Handler `add` on the right changes the value of `r` before returning control to the caller via `ret_add`. The effect of the body of `add` is, according to the contract of `assign`, the singleton $\{(\texttt{r}, \texttt{ret\_add})\}$. This effect does not escape from the definition of `add`, since `ret_add` is bound in the definition. Instead, it is compared to the list of formal pre-writes of `ret_add` in the contract of `add`; and `r` is correctly listed as one. This pre-write is the write effect of `add` when it returns via `ret_add`. As for the pre-writes of handlers `add` and `out`, they should reflect the writes that may be performed during execution of `sum` before `add` and `out` are called. The correct annotation in both cases is `[r]`, since `iter` may call `add` several times (and thus modify `r` before the second call) before calling `out`.

Unfortunately, our effect system cannot accommodate this code — let us see why. Computing $\mathcal{E}(\texttt{iter l add out})$ requires us, for the first handler parameter `add`, to compute the effect of the virtual definition $\mathcal{E}(\texttt{(i: int) (ret\_fn:)} \succ \texttt{add i ret\_fn})$. The effect of the body is $\{(\texttt{r}, \texttt{ret\_fn})\}$, according to the pre-write annotation of `ret_add` in the contract of `add`. However, `r` is not listed as a formal pre-write of `ret_fn` in the contract of `fn`, and thus the overall effect is undefined. Informally, we cannot pass `add` as an argument to `iter`, because `add` modifies the program state (as reflected in the non-empty pre-write list of `ret_add`), whereas `iter` expects a handler without side effects (as reflected in the empty pre-write list of `ret_fn`).

To make this code accepted, the potential side effects of handler parameter `fn` in the contract of `iter` must be listed as formal pre-writes of `ret_fn`. This requires the concerned references to be visible to `iter`. One way of doing it is to move the definition of `iter` inside the scope of `r`. This approach is similar to implicit inlining of effectful higher-order functions proposed by Filliâtre et al. [**?**]. Another possibility is to pass `r` to `iter` as an argument. Here is what the specification of `iter` would look like in this case:

```
iter (&q: int) (l: list int) (fn: [q] (i: int) (ret_fn: [q])) (return: [q])
```

Despite the fact that `iter` never touches the reference parameter `q` in its own code, its handler parameter `fn` might do so, as reflected in the pre-write of `ret_fn`. From that, by analysing the body of `iter`, we can correctly infer that `q` is potentially modified during the execution of `iter` before `fn` or `return` are called, and so `q` must appear in their pre-writes, too.

State Elimination. Pre-write annotations allow us to translate an alias-free program with side effects into an equivalent pure program. This is essentially a monadic transformation where the state is passed as an additional parameter from one handler to another. However, effect computation allows us to refine the encoding by only passing the relevant parts of the state. Indeed, the pre-write annotations of a handler h indicate which references accessible to h may have changed their value between the start of the parent of h and the start of h itself. It is (the current values of) these references that must be passed to h as extra parameters. As for other accessible references, their last values can be accessed directly from the lexical context that was established at the start of the parent of h, since those references have not been modified since.

Operator $[\![\cdot]\!]$, defined below, applies to expressions, contracts, and applications. It translates expressions into pure expressions and contracts into contracts without pre-writes or reference parameters. Applications are transformed into sequences of pure arguments followed by abbreviations; these abbreviations are appended to the translated handler call.

$$
\begin{array}{rcl@{\qquad}rcl}
[\![h\ \bar{a}]\!] & \triangleq & h\ [\![\varrho_h \prec \bar{a}]\!] & [\![e\ /\ \bar{\gamma}\ (h\ \varrho = d)]\!] & \triangleq & [\![e\ /\ \bar{\gamma}]\!]\ (h\ [\![\varrho]\!] = [\![d]\!]) \\
[\![e\ /\ \&r = s]\!] & \triangleq & [\![e]\!][r \mapsto s] & [\![e\ /\ h\ \varrho \twoheadrightarrow d]\!] & \triangleq & [\![e]\!]\ /\ h\ [\![\varrho]\!] \twoheadrightarrow [\![d]\!] \\
[\![[p]\ \varrho]\!] & \triangleq & (p : \tau_p)\ [\![\varrho]\!] & [\![[p]\ \varrho \prec \bar{a}]\!] & \triangleq & p\ [\![\varrho \prec \bar{a}]\!] \\
[\![(x : \tau)\ \varrho]\!] & \triangleq & (x : \tau)\ [\![\varrho]\!] & [\![(x : \tau)\ \varrho \prec s\ \bar{a}]\!] & \triangleq & s\ [\![\varrho \prec \bar{a}]\!] \\
[\![(\&q : \tau)\ \varrho]\!] & \triangleq & (q : \tau)\ [\![\varrho]\!] & [\![(\&q : \tau)\ \varrho \prec \&r\ \bar{a}]\!] & \triangleq & r\ [\![\varrho[q \mapsto r] \prec \bar{a}]\!] \\
[\![(f : \pi)\ \varrho]\!] & \triangleq & (f : [\![\pi]\!])\ [\![\varrho]\!] & [\![(f : \pi)\ \varrho \prec h\ \bar{a}]\!] & \triangleq & f\ [\![\varrho \prec \bar{a}]\!]\ /\ f\ [\![\pi]\!] \twoheadrightarrow [\![h\ \overline{xqg}_\pi]\!] \\
[\![\square]\!] & \triangleq & \square & [\![\square \prec \square]\!] & \triangleq & \square
\end{array}
$$

Once again, ϱ_h is a specialized contract of h and $\overline{xqg}_\pi$ denotes the list of all formal parameters of π. We write τ_p to denote the type of reference p in the original program.

Our transformation converts reference parameters and pre-written references into term parameters. To preserve the well-typedness of the transformed code, each actual handler argument is encapsulated in a local abbreviation whose contract is supplied by the corresponding formal handler parameter. Without this precaution, the actual handler argument might have a different arity from what is expected by the callee, due to different formal pre-writes.

Notice that converting pre-writes into term parameters inevitably creates collisions, as each pre-write annotation must be in the scope of the corresponding reference. This is exactly our intention: by the rules of lexical scoping, each occurrence of a given reference is now captured by the closest (innermost) binder, which allows it to be instantiated with a correct value during execution.

Of course, we must also update the specification and semantics of library handlers that manipulate the state. For example, here is the converted contract of `assign`:

```
assign (r: α) (v: α) {} (return (r: α) { r = v })
```

The operational semantics of this pure version is as follows: $\Lambda \vdash \texttt{assign}\ t\ s\ f \bullet \varnothing \longrightarrow f\ [\![s]\!]_\varnothing \bullet \varnothing$. This rule can be obtained from the original one by applying $[\![\cdot]\!]$ to both sides of the rule and then by replacing each former reference argument with its value in the corresponding variable context. The latter is replaced by an empty set (as state elimination removes all references from the program).

In Figure 4, we return to the example from Figure 3, with an added reference parameter for `iter` (as suggested at the end of Section 4), to ensure that the effects are specified correctly. On the left side of Figure 4, we show the stateful definitions of `iter` and `sum`, and on the right side, their stateless encoding. For simplicity, we omit the preconditions and inline the trivial abbreviations generated for handler arguments whose contract has exactly the same pre-writes as the corresponding formal handler parameter. Then the only added abbreviation is `onNil` in the translated definition of `iter`. Indeed, library handler `unList` does not modify the program state, and therefore its handler parameters `onCons` and `onNil` have no pre-writes. Handler `return`, on the other hand, does have a pre-write which becomes a term parameter in translation. Abbreviation `onNil → return q` allows us to pass `return` to `unList` with this term parameter properly instantiated with `q`.

```
iter (&q: int) (l: list int)
  (fn: [q] (i: int)
    (ret_fn: [q]))
  (return: [q])
= unList l on_cons return
  / on_cons h t → fn h tail
    / tail [q] → iter q t fn return
```

```
iter (q: int) (l: list int)
  (fn: (q: int) (i: int)
    (ret_fn: (q: int)))
  (return: (q: int))
= unList l on_cons onNil
  / onNil → return q
  / on_cons h t → fn q h tail
    / tail q → iter q t fn return
```

```
sum (l: list int) (return: (s: int))
= iter r l add out
  / add [r] (i: int) (ret_add: [r])
    = assign &r (i + r) ret_add
  / out [r] → return r
  / &r = 0
```

```
sum (l: list int) (return: (s: int))
= iter 0 l add out
  / add (r: int) (i: int)
      (ret_add: (r: int))
    = assign r (i + r) ret_add
  / out (r: int) → return r
```

Figure 4: Program with side effects (left) and its stateless encoding (right)

5 Functional Correctness

We equip handler contracts with functional specifications, called *preconditions*, that describe the expected program state at the beginning of the handler execution. As functional specifications often need to relate the current program state to previous ones, we also add *auxiliary variables* (or simply *auxiliaries*) that capture reference values at the start of a handler:

$$\begin{array}{rcl} \textit{annotation} & ::= & \ldots \\ & \mid & \{\ \textit{formula}\ \} \\ & \mid & \textit{variable}\ =\ \textit{term} \end{array}$$

The non-terminal *formula* represents statements in a suitable logical language; we require that data terms can be used inside formulas, and handlers can not. We denote formulas with letters φ and ψ.

Here are the additional rules to compute the free variables and handlers in abstractions:

$$\begin{array}{rcl@{\qquad}rcl} \mathrm{FV}(\{\varphi\}\ \varrho \succ e) & \triangleq & \mathrm{FV}(\varrho \succ e) \cup \mathrm{FV}(\varphi) & \mathrm{FH}(\{\varphi\}\ \varrho \succ e) & \triangleq & \mathrm{FH}(\varrho \succ e) \\ \mathrm{FV}((z = s)\ \varrho \succ e) & \triangleq & (\mathrm{FV}(\varrho \succ e) \setminus \{z\}) \cup \mathrm{FV}(s) & \mathrm{FH}((z = s)\ \varrho \succ e) & \triangleq & \mathrm{FH}(\varrho \succ e) \end{array}$$

Evaluation rules are extended as follows:

$$\Sigma \vdash (\{\varphi\}\ \varrho \succ e) \prec \bar{a} \longrightarrow (\varrho \succ e) \prec \bar{a} \qquad \text{(E-Pre)}$$

$$\Sigma \vdash ((z = s)\ \varrho \succ e) \prec \bar{a} \longrightarrow (\varrho \succ e)[z \mapsto [\![s]\!]_\Sigma] \prec \bar{a} \qquad \text{(E-Aux)}$$

Assuming a predefined typing relation $\Gamma \vdash \varphi : \mathtt{bool}$ for formulas, here are the new typing rules:

$$\frac{\Gamma \vdash \varphi : \mathtt{bool} \qquad \Gamma \vdash \varrho \succ e : \xi}{\Gamma \vdash \{\varphi\}\ \varrho \succ e : \xi}\ \text{(T-Pre)} \qquad \frac{\Gamma \vdash s : \tau \qquad \Gamma, z : \tau \vdash \varrho \succ e : \xi}{\Gamma \vdash (z = s)\ \varrho \succ e : \xi}\ \text{(T-Aux)}$$

Effect computation and state elimination ignore preconditions and auxiliaries:

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{E}((z = s)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) & \mathcal{E}((z = s)\ \varrho \prec \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\ \mathcal{E}(\{\varphi\}\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) & \mathcal{E}(\{\varphi\}\ \varrho \prec \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\ [\![(z = s)\ \varrho]\!] & \triangleq & (z = s)\ [\![\varrho]\!] & [\![(z = s)\ \varrho \prec \bar{a}]\!] & \triangleq & [\![\varrho \prec \bar{a}]\!] \\ [\![\{\varphi\}\ \varrho]\!] & \triangleq & \{\varphi\}\ [\![\varrho]\!] & [\![\{\varphi\}\ \varrho \prec \bar{a}]\!] & \triangleq & [\![\varrho \prec \bar{a}]\!] \end{array}$$

Verification conditions are produced in a restricted fragment of the first-order language with local predicate definitions. This fragment can be described by the following grammar:

$$
\begin{array}{rcl}
\mathit{vc} & ::= & \top \quad | \quad \mathit{vc} \wedge \mathit{vc} \\
& | & \mathit{formula} \quad | \quad \mathit{formula} \rightarrow \mathit{vc} \\
& | & \forall\ \mathit{variable} : \mathit{termType}\ .\ \mathit{vc} \\
& | & \mathit{vc}\ /\ \mathit{handler}\ (\mathit{variable} : \mathit{termType})* \equiv \mathit{vc} \\
& | & \mathit{handler}\ \mathit{term}*
\end{array}
$$

In this syntax, handlers act as second-order variables that denote locally defined predicates: verification conditions, parametrized by a number of first-order values. Predicate definitions are not recursive and are not type-generalized. Note that *vc* can not occur in the antecedent of an implication. Moreover, since handlers can not occur in a *formula*, all occurrences of locally defined predicates and, therefore, of *vc* are necessarily positive. We denote verification conditions with Φ and Ψ.

Operator $\mathcal{V}$ computes the verification condition for a given expression, abstraction or application:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{V}(h\ \bar{a}) & \triangleq & \mathcal{V}(\varrho_h^\star \prec \bar{a}) & \mathcal{V}(e\ /\ \bar{\gamma}\ (h\ \varrho = d)) & \triangleq & \mathcal{V}(e\ /\ \bar{\gamma}) \wedge \mathcal{V}(\varrho \succ d) \\
\mathcal{V}(e\ /\ \&r = s) & \triangleq & \mathcal{V}(e)[r \mapsto s] & \mathcal{V}(e\ /\ h\ \varrho \twoheadrightarrow d) & \triangleq & \mathcal{V}(e)\ /\ h\ \overline{pxqz}_\varrho \equiv \mathcal{V}(d) \\
\mathcal{V}([p]\ \varrho \succ e) & \triangleq & \forall p : \tau_p\ .\ \mathcal{V}(\varrho \succ e) & \mathcal{V}([p]\ \varrho \prec \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a}) \\
\mathcal{V}((x:\tau)\ \varrho \succ e) & \triangleq & \forall x : \tau\ .\ \mathcal{V}(\varrho \succ e) & \mathcal{V}((x:\tau)\ \varrho \prec s\ \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a})[x \mapsto s] \\
\mathcal{V}((\&q:\tau)\ \varrho \succ e) & \triangleq & \forall q : \tau\ .\ \mathcal{V}(\varrho \succ e) & \mathcal{V}((\&q:\tau)\ \varrho \prec \&r\ \bar{a}) & \triangleq & \mathcal{V}(\varrho[q \mapsto r] \prec \bar{a}) \\
\mathcal{V}((f:\pi)\ \varrho \succ e) & \triangleq & \mathcal{V}(\varrho \succ e) & \mathcal{V}((f:\pi)\ \varrho \prec h\ \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a}) \wedge \mathcal{V}(\pi \succ h\ \overline{xqg}_\pi) \\
\mathcal{V}((z = s)\ \varrho \succ e) & \triangleq & \mathcal{V}(\varrho \succ e)[z \mapsto s] & \mathcal{V}((z = s)\ \varrho \prec \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a})[z \mapsto s] \\
\mathcal{V}(\{\varphi\}\ \varrho \succ e) & \triangleq & \varphi \rightarrow \mathcal{V}(\varrho \succ e) & \mathcal{V}(\{\varphi\}\ \varrho \prec \bar{a}) & \triangleq & \varphi \wedge \mathcal{V}(\varrho \prec \bar{a}) \\
\mathcal{V}(\square \succ e) & \triangleq & \mathcal{V}(e) & \mathcal{V}(\square \prec \square) & \triangleq & \top
\end{array}
$$

$$
\varrho_h^\star \triangleq \begin{cases} \varrho_h\ \{h\ \overline{pxqz}_{\varrho_h}\} & \text{if } h \text{ is an abbreviation,} \\ \varrho_h & \text{otherwise.} \end{cases}
$$

As before, ϱ_h is a specialized contract of handler h, τ_p is the type of reference p, and $\overline{xqg}_\pi$ are the formal parameters of contract π. Local predicate definitions hold verification conditions of handler abbreviations; they are parametrized by the pre-writes, term and reference parameters, and auxiliary variables of the handler's contract ϱ, collectively denoted $\overline{pxqz}_\varrho$ (the missing type annotations for pre-writes and auxiliaries can be pulled from the typing context). Predicates defined in such a way are added, as an extra precondition, to the contract of the corresponding handler whenever it is executed. We thus prove the instantiated verification condition of the abbreviation body separately at each call site, whereas handler definitions are verified just once, for all possible entry states and parameters. In what follows, we always simplify the occurrences of $\top$ in verification conditions.

It is important to notice that in the rules for application, we substitute reference arguments early, directly in the contract, but we instantiate term parameters and auxiliary variables late, in the computed verification condition. The reason for the former is that reference parameters of a handler can appear as formal pre-writes of its subsequent handler parameters: for instance, the reference parameter of `assign` is a pre-write of its handler parameter. These pre-writes will get bound when we verify the virtual definitions for those handler parameters; see the rules for $\mathcal{V}((f:\pi)\ \varrho \prec h\ \bar{a})$ and $\mathcal{V}([p]\ \varrho \succ e)$. We must therefore instantiate reference parameters early, in order to bind correct variables in $h\ \overline{xqg}_\pi$. On the other hand, term parameters and auxiliaries must not be instantiated early. Indeed, the terms we substitute into them may contain references, but these occurrences represent dereferenced values at the start of the handler call. If we substitute such a term into the contract, it may pass under a pre-write annotation which will turn into a universal quantifier over the reference variable, making it refer to some later value. To make this more clear, let us consider some examples.

Example: Increment. The handler defined below increments a given reference by 1.

```
incr (&p: int) (o = p) (out: [p] { p = 1 + o })
= assign &p (p + 1) out
```

Let us compute the verification condition of the corresponding abstraction.

$$
\begin{aligned}
&\mathcal{V}((\&p{:}\mathsf{int})\ (o = p)\ (\mathsf{out}{:}\ [p]\ \{\, p = 1 + o \,\}) \succ \mathsf{assign}\ \&p\ (p+1)\ \mathsf{out}) = \\
&\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}((o = p)\ (\mathsf{out}{:}\ [p]\ \{\, p = 1 + o \,\}) \succ \mathsf{assign}\ \&p\ (p+1)\ \mathsf{out}) = \\
&\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}((\mathsf{out}{:}\ [p]\ \{\, p = 1 + o \,\}) \succ \mathsf{assign}\ \&p\ (p+1)\ \mathsf{out})[o \mapsto p] = \\
&\quad\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}(\square \succ \mathsf{assign}\ \&p\ (p+1)\ \mathsf{out})[o \mapsto p] = \\
&\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}(\mathsf{assign}\ \&p\ (p+1)\ \mathsf{out})[o \mapsto p] = \\
&\forall\, p{:}\mathsf{int}\,.\, \mathcal{V}((\&r{:}\mathsf{int})\ (v{:}\mathsf{int})\ (\mathsf{return}{:}\ [r]\ \{\, r = v \,\}) \prec \&p\ (p+1)\ \mathsf{out})[o \mapsto p] = \\
&\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}(((v{:}\mathsf{int})\ (\mathsf{return}{:}\ [r]\ \{\, r = v \,\}))[r \mapsto p] \prec (p+1)\ \mathsf{out})[o \mapsto p] = \\
&\quad\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}((v{:}\mathsf{int})\ (\mathsf{return}{:}\ [p]\ \{\, p = v \,\}) \prec (p+1)\ \mathsf{out})[o \mapsto p] = \\
&\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}((\mathsf{return}{:}\ [p]\ \{\, p = v \,\}) \prec \mathsf{out})[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\mathcal{V}(\square \prec \square) \wedge \mathcal{V}([p]\ \{\, p = v \,\} \succ \mathsf{out}))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\top \wedge \forall\, p{:}\mathsf{int}\,.\, \mathcal{V}(\{\, p = v \,\} \succ \mathsf{out}))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\forall\, p{:}\mathsf{int}\,.\, p = v \rightarrow \mathcal{V}(\square \succ \mathsf{out}))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\forall\, p{:}\mathsf{int}\,.\, p = v \rightarrow \mathcal{V}(\mathsf{out}))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\forall\, p{:}\mathsf{int}\,.\, p = v \rightarrow \mathcal{V}([p]\ \{\, p = 1 + o \,\} \prec \square))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\forall\, p{:}\mathsf{int}\,.\, p = v \rightarrow \mathcal{V}(\{\, p = 1 + o \,\} \prec \square))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\forall\, p{:}\mathsf{int}\,.\, p = v \rightarrow p = 1 + o \wedge \mathcal{V}(\square \prec \square))[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, (\forall\, q{:}\mathsf{int}\,.\, q = v \rightarrow q = 1 + o \wedge \top)[v \mapsto p+1][o \mapsto p] = \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \forall\, p{:}\mathsf{int}\,.\, \forall\, q{:}\mathsf{int}\,.\, q = p+1 \rightarrow q = 1 + p
\end{aligned}
$$

The topmost universal quantifier binds the reference parameter of `incr`. The underlying formula is the verification condition of the definition body, that is, of the `assign` call. Since `assign` does not have preconditions, we only need to verify that the postcondition of `assign` (i.e., the precondition of its formal handler parameter `return`) implies the precondition of `out`. Since the handler parameter of `assign` has a formal pre-write — the modified reference `p` which replaced the formal reference parameter `r` — the implication must be universally quantified over `p`. In other words, the postcondition of `incr` must hold for every value of `p` that satisfies the postcondition of `assign`.

Example: Russian Peasant Multiplication. The code in Figures 5 and 6, written first in OCaml and then in our language, implements the classical Russian peasant multiplication algorithm. For simplicity, we define the algorithm on natural numbers and use Euclidean division and remainder operators in terms. This code produces the following verification condition:

$$
\begin{aligned}
&\forall\, a : \mathsf{nat}\,.\ \forall\, b : \mathsf{nat}\,.\ a \cdot b + 0 = a \cdot b \ \wedge \\
&\quad \forall\, p : \mathsf{nat}\,.\ \forall\, q : \mathsf{nat}\,.\ \forall\, r : \mathsf{nat}\,. \\
&\qquad p \cdot q + r = a \cdot b \rightarrow (q > 0 \rightarrow \mathsf{next}) \wedge (q \not> 0 \rightarrow \mathsf{last}) \\
&\qquad\quad / \ \mathsf{last} \equiv r = a \cdot b \\
&\qquad\quad / \ \mathsf{next} \equiv (q \bmod 2 = 1 \rightarrow \mathsf{write\_r}) \wedge (q \bmod 2 \neq 1 \rightarrow \mathsf{write\_p}\ r) \\
&\qquad\qquad / \ \mathsf{write\_r} \equiv \forall\, v : \mathsf{nat}\,.\ v = r + p \rightarrow \mathsf{write\_p}\ v \\
&\qquad\qquad / \ \mathsf{write\_p}\ (r : \mathsf{nat}) \equiv \forall\, v : \mathsf{nat}\,.\ v = p + p \rightarrow \mathsf{write\_q}\ v\ r \\
&\qquad\qquad / \ \mathsf{write\_q}\ (p : \mathsf{nat})\ (r : \mathsf{nat}) \equiv \forall\, v : \mathsf{nat}\,.\ v = q \operatorname{div} 2 \rightarrow p \cdot v + r = a \cdot b
\end{aligned}
$$

Notice that the first conjunct $a \cdot b + 0 = a \cdot b$ generated by the initial call to `loop` corresponds exactly to the initialisation of a loop invariant. Similarly, the assertion $p \cdot v + r = a \cdot b$ coming from the second,

```
let product (a: nat) (b: nat) : nat
= let p,q,r = ref a, ref b, ref 0 in
  while !q > 0 do
    if !q mod 2 = 1 then r := !r + !p;
    p := !p + !p;
    q := !q div 2
  done;
  !r
```

Figure 5: Russian Peasant Multiplication in OCaml.

```
product (a: nat) (b: nat) (return: (c: nat) { c = a·b })
= loop
  / loop [p] [q] [r] { p·q+r = a·b }
    = if (q > 0) next last
      / last → return r
      / next → if (q mod 2 = 1) write_r write_p
        / write_r → assign &r (r + p) write_p
        / write_p [r] → assign &p (p + p) write_q
        / write_q [p] [r] → assign &q (q div 2) loop
  / &p = a / &q = b / &r = 0
```

Figure 6: Russian Peasant Multiplication, translated.

recursive, call to `loop`, corresponds to the preservation of the invariant after a single iteration. In a traditional functional language, encoding loops as tail-recursive functions is slightly more difficult, since one also has to provide (and then prove) a postcondition. In our translation, when the loop condition becomes false, we simply call the after-the-loop continuation `return r`.

Termination proof. Assuming that library handlers always terminate (meaning that they either halt or return to the caller by executing one of their handler parameters), the only potential source of divergence in our language are recursive handler definitions. Indeed, without those, any program can be translated into a stateless one, as shown in Section 4, and then into a typed lambda-term, which are known to be strongly normalizing [**?**].

We can prove that a given recursive definition cannot loop on itself forever by associating a certain measure to the handler entry state and by showing that each recursive call makes this measure decrease with respect to some well-founded ordering.

Let us select some type θ and a well-founded relation $\prec$ on it. If we want to prove a recursive definition group $\bar{\gamma}$ to be well-founded, we place into the contract of each handler h in $\bar{\gamma}$ a distinctive *variant auxiliary* (or simply *variant*) ($\hat{u}_h = s_h$) of type θ. We only recognize variant auxiliaries in the contracts of handlers introduced through definitions (but not in abbreviations or handler parameters), and we allow at most one variant in a contract.

Now we extend the $\mathcal{V}$ rule for handler calls as follows:

$$\varrho_h^\star \triangleq \begin{cases} \varrho_h\,\{\,h\ \overline{pxqz}_{\varrho_h}\,\} & \text{if } h \text{ is an abbreviation,} \\ \varrho_h\,\{\,\hat{u} \prec \hat{v}\,\} & \text{if } \hat{u} \text{ is a variant auxiliary in } \varrho_h, \\ & \quad \hat{v} \text{ appears in the current typing context,} \\ & \quad \hat{v} \text{ is a variant auxiliary of handler } g, \text{ and} \\ & \quad g \text{ is } h \text{ or defined in the same group as } h, \\ \varrho_h & \text{otherwise.} \end{cases}$$

The condition "$\hat{v}$ appears in the current typing context" verifies that handler h is used — i.e., called directly or passed as a handler argument — inside the definition of handler g, which can be either h itself or its sibling from the same definition group. In other words, we only add the extra precondition for recursive calls (passing a handler argument is treated as a call in a virtual definition), when both handlers have a variant auxiliary. Apart from this added precondition, variants are treated in exactly the same way as ordinary auxiliary variables with respect to typing, evaluation, etc. It is possible to use different types and orderings for variants in different definition groups, but inside a single group, all variants must be of the same type and must be compared via the same well-founded relation.

To prove termination of `product`, we add a variant `(û = q)` to the contract of `loop`. Notice that `product` itself is not recursively defined and therefore does not need a variant. This results in the following verification condition:

```
∀ a : nat . ∀ b : nat . a · b + 0 = a · b ∧
  ∀ p : nat . ∀ q : nat . ∀ r : nat .
    p · q + r = a · b → (q > 0 → next) ∧ (q ≯ 0 → last)
      / last ≡ r = a · b
      / next ≡ (q mod 2 = 1 → write_r) ∧ (q mod 2 ≠ 1 → write_p r)
        / write_r ≡ ∀ v : nat . v = r + p → write_p v
        / write_p (r : nat) ≡ ∀ v : nat . v = p + p → write_q v r
        / write_q (p : nat) (r : nat) ≡ ∀ v : nat . v = q div 2 → p · v + r = a · b ∧ v < q
```

Notice that the extra precondition only appears when `loop` is used in its own definition, but not when it is called from `product` at the beginning. The standard ordering $<$ is well-founded on $\mathbb{N}$; if we used integers, we would have to add a precondition to `loop` requiring `q` to be non-negative — and similarly for `product` and `b` — and use the ordering relation $u < v \wedge v \geqslant 0$ which is well-founded on $\mathbb{Z}$. The relation $0 \leqslant u < v$ would work in this example, too, but is much less useful in the general case, where we often need to admit a negative number of the left-hand side.

Elimination of predicate definitions. If we want to use automated theorem provers to discharge verification conditions, we have to translate them into a format understood by those provers: usually, some variation of the first-order language. Assuming that user-written preconditions in our code are already first-order[1], local predicate definitions are the only feature of verification conditions that may require translation.

One method to eliminate predicate definitions is simply to inline them (remember that they cannot be recursive). By doing this, we obtain a logical formula that is very close to what is produced by the classical Dijkstra-style weakest precondition calculus. For example, below is the verification condition for `product` (without variant), obtained in this fashion. In addition to predicate inlining, we also simplify occurrences of $(\forall x.\, x = s \to \Phi)$ to $\Phi[x \mapsto s]$, when term s does not contain x:

```
∀ a : nat . ∀ b : nat . a · b + 0 = a · b ∧
  ∀ p : nat . ∀ q : nat . ∀ r : nat . p · q + r = a · b →
    (q > 0 → (q mod 2 = 1 → (p + p) · (q div 2) + (r + p) = a · b) ∧
              (q mod 2 ≠ 1 → (p + p) · (q div 2) + r = a · b)) ∧
    (q ≯ 0 → r = a · b)
```

A well-known inconvenience of this method is that whenever some code fragment is reachable via two different code paths — which in our language is expressed by an abbreviation that is called from two different points in the underlying expression — its weakest precondition will appear twice in the verification condition. An example of this in the code of `product` is the `write_p` handler, called once from `next` and once from `write_r`. By chaining such constructions, we can easily come up with examples that demonstrate exponential growth of verification conditions compared to the size

[1] In our examples, we do not distinguish between `bool`-typed terms and formulas, but this is rather easy to handle.

of the original program. Indeed, a simple sequence of conditional statements suffices to produce an oversized verification condition.

A different method of verification condition generation was proposed by Flanagan and Saxe [**?**]. It consists in computing a single formula that describes all code paths leading to a given sequence point, and avoids the combinatorial explosion in a large number of cases, including all WHILE programs that do not use exceptions. The trade-off here is the increased complexity of the resulting formula. Leino demonstrated that verification conditions produced by this method can be obtained by interweaving application of classical procedures for the strict and liberal weakest precondition computation [**?**].

Below we show that these "compact" verification conditions can be also obtained directly from our verification conditions by a chain of truth-preserving transformations. Moreover, at any point of this process we can resort to inlining, meaning that we can freely mix the classical and the compact approaches in the same verification condition, balancing its complexity against its size.

Consider a verification condition of the form $\Phi \mathbin{/} h\ (x_1 : \tau_1) \ldots (x_n : \tau_n) \equiv \Phi_h$. We assume that Φ contains no predicate definitions; if it does, we eliminate them first. Then the grammar of verification conditions ensures that each application of h in Φ occurs at a positive position, under a number of conjunctions, implications, and universal quantifiers. We want to rearrange Φ in such a way that multiple applications of h in it are merged into one. Once we obtain an equivalent verification condition $\Phi' \mathbin{/} h\ (x_1 : \tau_1) \ldots (x_n : \tau_n) \equiv \Phi_h$, where h is only applied once inside Φ', we can inline the predicate definition without increasing the size of Φ'.

Our first step is to abstract out the different arguments passed to h in Φ. We pick n fresh variables $z_1, \ldots, z_n$ and consider the following verification condition:

$$\forall z_1 : \tau_n. \ldots \forall z_n : \tau_n. (\Phi_0 \mathbin{/} h\ (x_1 : \tau_n) \ldots (x_n : \tau_n) \equiv \Phi_h),$$

where formula Φ_0 is obtained from Φ by replacing each occurrence of $h\ s_1 \ldots s_n$ with

$$(z_1 = s_1 \wedge \cdots \wedge z_n = s_n) \rightarrow h\ z_1 \ldots z_n.$$

It is obvious that the new verification condition is well-typed (as predicate definitions are not type-generalized) and equivalent to the initial one. Indeed, the universal quantifiers over $z_1, \ldots, z_n$ can be pushed down in Φ_0, and $\forall \bar{z}. (\bar{z} = \bar{s} \rightarrow h\ \bar{z})$ is equivalent to $h\ \bar{s}$ — since variables $\bar{z}$ are fresh, they cannot occur in terms $\bar{s}$.

Now we can move the occurrences of subformulas $(\varphi \rightarrow h\ \bar{z})$ up in Φ_0, and merge them when two or more appear in the same conjunction. To this end, we apply to Φ_0 the following rewriting rules, starting from the lowermost subformulas:

$$\begin{aligned}
(\varphi_1 \rightarrow h\ \bar{z}) \wedge (\varphi_2 \rightarrow h\ \bar{z}) &\implies (\varphi_1 \vee \varphi_2) \rightarrow h\ \bar{z} \\
\varphi_1 \rightarrow (\varphi_2 \rightarrow h\ \bar{z}) &\implies (\varphi_1 \wedge \varphi_2) \rightarrow h\ \bar{z} \\
\varphi_1 \rightarrow ((\varphi_2 \rightarrow h\ \bar{z}) \wedge \Psi) &\implies ((\varphi_1 \wedge \varphi_2) \rightarrow h\ \bar{z}) \wedge (\varphi_1 \rightarrow \Psi) \\
\forall \bar{y}. (\varphi \rightarrow h\ \bar{z}) &\implies (\exists \bar{y}. \varphi) \rightarrow h\ \bar{z} \\
\forall \bar{y}. ((\varphi \rightarrow h\ \bar{z}) \wedge \Psi) &\implies ((\exists \bar{y}. \varphi) \rightarrow h\ \bar{z}) \wedge \forall \bar{y}. \Psi
\end{aligned}$$

Here we assume that Ψ contains no occurrences of $h\ \bar{z}$ and we treat conjunction modulo associativity and commutativity. In the last two rules, we use the fact that variables $\bar{z}$ are bound above the definition of h, and thus variables $\bar{y}$ cannot occur in formula $h\ \bar{z}$. Again, it is easy to see that each rewriting step produces an equivalent formula.

We stop when the formula under the definition of h contains only one application of h; at that point we can simply inline the definition. It is also perfectly possible to inline some of the applications of h at the very beginning, directly in Φ, instead of merging them with the other applications. This selective handling allows us to consider some code paths separately from the others, e.g., because they require some special proof techniques.

Notice that this approach still does not eliminate the possibility of exponential growth: the third rewriting rule duplicates formula φ_1.

more elaborate analysis of sources of explosion
+ correspondence to Flanagan and Saxe VCs.

Here is the first-order verification condition obtained by merging two applications of `write_p` in the original verification condition. All other predicate variables are applied only once, and thus their definitions can be inlined directly. As before, we simplify the trivial variable definitions.

$$
\begin{aligned}
&\forall\, a : \mathsf{nat}\,.\ \forall\, b : \mathsf{nat}\,.\ a \cdot b + 0 = a \cdot b\ \wedge \\
&\quad \forall\, p : \mathsf{nat}\,.\ \forall\, q : \mathsf{nat}\,.\ \forall\, r : \mathsf{nat}\,.\ p \cdot q + r = a \cdot b\ \longrightarrow \\
&\qquad (q > 0\ \longrightarrow\ \forall\, z : \mathsf{nat}\,.\ ((q \bmod 2 = 1 \wedge z = r + p)\ \vee \\
&\qquad\qquad\qquad (q \bmod 2 \neq 1 \wedge z = r))\ \longrightarrow\ (p + p) \cdot (q \operatorname{div} 2) + z = a \cdot b))\ \wedge \\
&\qquad (q \not> 0\ \longrightarrow\ r = a \cdot b)
\end{aligned}
$$

Show how this also allows to infer postconditions: must lift all the way up, not just until the single occurrence

6 Ghost Code

In the practice of deductive verification, the term "ghost code" refers to the parts of our program that do not participate in the computation of the final result but help writing adequate specification and verifying the program correctness. In the last section, we saw two examples of ghost code: auxiliary variable `o` in the contract of `incr` and variant auxiliary $\hat{u}$ for the loop in `product`. Without the former, we are unable to write the postcondition of `incr`, and we need the latter to prove termination of `loop` — but none of these variables affects the actual computation. Indeed, we can erase them from our code, together with the preconditions that may depend on them, without changing the behaviour of the program in any meaningful way.

Ghost code is a versatile and powerful instrument. Local ghost variables and ghost function parameters allow us to better describe the progression of an algorithm; for example, when looking for a maximal element in an array, we may use a ghost variable to store the index of the current best candidate, eliminating the need in an existential quantifier in the loop invariant. Ghost fields in data structures can provide a mathematical model of the data, independent of the implementation details, so that the operations on the data structure can be specified in terms of the abstract model rather than concrete implementation. One can even write an entire ghost reimplementation of a program, to be executed in lockstep with the original code, so that the correctness proof of one version could be transferred to the other [**?**].

From the point of view of type-checking, evaluation, effect analysis or VC generation, ghost code and ghost data are treated in exactly the same way as non-ghost, or *material*, parts of our program. Moreover, we want to use the same data types and operations over material and ghost data, without introducing a parallel standard library for ghost integers, ghost references, ghost arrays, etc. We must however ensure that ghost code in our program does not affect the material computation, that it can be erased from the program without changing its flow or outcome. One can see ghost data as a sort of classified information that should never leak to the lower-security material code. One way to verify this *non-interference* property is to recognize ghost data at the type level and use the typing rules to track dependencies between values, so that the material computation is never contaminated by ghost data. Filliâtre et al. used this approach in their formal framework for ghost code in a functional language with polymorphic types and mutable state [**?**].

In this work, we explore a slightly different method. Assuming that every variable in our code is explicitly marked as material or ghost, we determine which parts of our program can be safely skipped, as they produce no material data for subsequent computation nor make a choice between different continuations. Then we "deghostify" our program: we erase all ghost parameters and ghost values, and all skippable expressions. If the resulting program is well-formed, we know that ghost data is never leaked in the material computation, and the original program is admissible. If, on the

other hand, we are left with code where a handler expecting a material parameter is given a ghost argument, and this handler call can not be skipped — then we have found an interference and must reject the program.

This analysis crucially depends on the functional correctness of our program — in particular, on its termination proof. Indeed, if some part of code may diverge, it obviously can not be erased, as that would alter the set of possible behaviours (unless the divergence is the only possible behaviour). Consequently, in order to see what expressions are skippable, we must annotate our well-founded recursive definitions with variants and prove the termination.

Let us proceed to formal definitions. We identify each variable name as either *material* or *ghost*. This denomination is extended to data terms: a term is considered *ghost* if it contains occurrences of ghost variables, otherwise it is *material*. In our examples, we will distinguish ghost variables by starting them with a capital letter. In our formal rules, to indicate the status of a variable or term X, we write $\circ X$ and $\bullet X$ to say "ghost X" and "material X", respectively. In absence of such an indicator, X can be either material or ghost, unless its status is established elsewhere. Handlers and expressions are not considered ghost or material.

An expression in a program is called *unrestrained* if it contains a recursive definition without a variant or is itself the right-hand side of such a definition; otherwise it is *restrained*. Assuming that no handler in our code is introduced twice, we say that handler h is *escaping* when it satisfies one of the three conditions:

- h is a library handler whose evaluation may diverge or halt;
- h is a handler parameter;
- h is introduced via a definition $h\ \varrho = d$ or an abbreviation $h\ \varrho \twoheadrightarrow d$, where d is unrestrained or $\mathrm{FH}(\varrho \succ d)$ contains an escaping handler.

This definition is circular for recursively defined handlers, as they freely occur in their own bodies. We resolve the ambiguity by considering the minimal set of escaping handlers. Then the status of each handler in the program can be effectively determined by proceeding top-down (i.e., from right to left) and by computing the least fixed point when treating recursive definitions with variants.

Identifying escaping handlers is important for code flow analysis, as they do not always return to the caller via one of their handler parameters. Instead, they may escape either by halting the program (imagine a library handler `exit`), or by entering a non-terminating computation, or by passing control to some other escaping handler. We consider that handler parameters, whose actual implementation is *a priori* unknown, may always escape. Alternatively, we could introduce escaping as a new effect annotation in contracts and distinguish between escaping and non-escaping handler parameters.

An expression e is *skippable* up to handler h whenever the following conditions are satisfied:

1. e is inside the scope of h;
2. h does not have handler parameters and all term and reference parameters of h are ghost;
3. all formal pre-writes of h that appear in $\mathcal{E}(e)$ are ghost;
4. e is restrained;
5. every escaping handler in $\mathrm{FH}(e) \setminus \{h\}$ is introduced via a definition or an abbreviation whose right-hand side is skippable up to h.

Any expression that is skippable to handler h can be replaced with a call to h when we erase ghost data from our program. Indeed, h is accessible from that position and would not require arguments. The skipped expression does not modify material references accessible to h and does not diverge, meaning that no observable effects are lost when we skip it. The last condition ensures that the target handler cannot be bypassed: every escaping handler in the skipped expression is either h itself or can

be replaced with h. Here, we only need to consider escaping handlers, as the non-escaping ones must necessarily return to the caller via one of the handler parameters.

An expression can be skippable up to multiple handlers. Indeed, if expression e is skippable up to handler h, and the body of h is itself skippable up to handler g, then e is also skippable up to g. In a practical implementation, we would obviously want to skip as far as we can.

Condition 5 above is circular on recursive definitions. If there are other escaping handlers in the definition body that are not introduced in the same definition group (this includes the formal handler parameters), the ambiguity is lifted. A recursive definition group that does not include any escaping handler from the outside of the group is obviously divergent, as the computation has no way to leave the confines of that group. Then it either has an unrestrained right-hand side, breaking the circularity by condition 4, or its "termination" is proved due to a contradiction in premises, meaning that these definitions are unreachable code. In the latter case, we can indeed replace the right-hand sides with any suitable handler, provided that conditions 1–3 are respected.

Operator $[\![\cdot]\!]$ applies to expressions, contracts, and applications. It removes skippable subexpressions, ghost parameters, and preconditions from expressions and contracts. It converts applications into sequences of material arguments followed by abbreviations; these abbreviations are appended to the translated handler call. This operator is partial: it is undefined whenever ghost data is passed to material code. In this case, the original program is considered invalid and is rejected.

$$[\![e]\!] \triangleq \begin{cases} h\ \square & \text{if } e \text{ is skippable up to handler } h, \\ \lfloor e \rfloor & \text{otherwise.} \end{cases}$$

$$\begin{aligned}
\lfloor h\ \bar{a} \rfloor &\triangleq h\ [\![\varrho_h \prec \bar{a}]\!] & \lfloor e\ /\ \bar{\gamma}\ (h\ \varrho = d) \rfloor &\triangleq [\![e\ /\ \bar{\gamma}]\!]\ (h\ [\![\varrho]\!] = [\![d]\!]) \\
\lfloor e\ /\ \&\bullet r = \bullet s \rfloor &\triangleq [\![e]\!]\ /\ \&r = s & \lfloor e\ /\ h\ \varrho \twoheadrightarrow d \rfloor &\triangleq [\![e]\!]\ /\ h\ [\![\varrho]\!] \twoheadrightarrow [\![d]\!] \\
\lfloor e\ /\ \&\bullet r = \circ s \rfloor &\text{ is undefined} & \lfloor e\ /\ \&\circ r = s \rfloor &\triangleq [\![e]\!]
\end{aligned}$$

$$\begin{aligned}
[\![[\bullet p]\ \varrho]\!] &\triangleq [p]\ [\![\varrho]\!] & [\![[\bullet p]\ \varrho \prec \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!] \\
[\![(\bullet x : \tau)\ \varrho]\!] &\triangleq (x : \tau)\ [\![\varrho]\!] & [\![(\bullet x : \tau)\ \varrho \prec \bullet s\ \bar{a}]\!] &\triangleq s\ [\![\varrho \prec \bar{a}]\!] \\
[\![(\&\bullet q : \tau)\ \varrho]\!] &\triangleq (\&q : \tau)\ [\![\varrho]\!] & [\![(\&\bullet q : \tau)\ \varrho \prec \&\bullet r\ \bar{a}]\!] &\triangleq \&r\ [\![\varrho[q \mapsto r] \prec \bar{a}]\!] \\
[\![(f : \pi)\ \varrho]\!] &\triangleq (f : [\![\pi]\!])\ [\![\varrho]\!] & [\![(f : \pi)\ \varrho \prec h\ \bar{a}]\!] &\triangleq f\ [\![\varrho \prec \bar{a}]\!]\ /\ f\ [\![\pi]\!] \twoheadrightarrow [\![h\ \overline{xqg}_\pi]\!] \\
[\![(\bullet z = \bullet s)\ \varrho]\!] &\triangleq (z = s)\ [\![\varrho]\!] & [\![(\bullet z = \bullet s)\ \varrho \prec \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!] \\
[\![\square]\!] &\triangleq \square & [\![\square \prec \square]\!] &\triangleq \square
\end{aligned}$$

$$\begin{aligned}
[\![(\bullet z = \circ s)\ \varrho]\!] &\text{ is undefined} & [\![(\bullet x : \tau)\ \varrho \prec \circ s\ \bar{a}]\!] &\text{ is undefined} \\
[\![(\bullet z = \circ s)\ \varrho \prec \bar{a}]\!] &\text{ is undefined} & [\![(\&\bullet q : \tau)\ \varrho \prec \&\circ r\ \bar{a}]\!] &\text{ is undefined}
\end{aligned}$$

$$\begin{aligned}
[\![[\circ p]\ \varrho]\!] &\triangleq [\![\varrho]\!] & [\![[\circ p]\ \varrho \prec \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!] \\
[\![(\circ x : \tau)\ \varrho]\!] &\triangleq [\![\varrho]\!] & [\![(\circ x : \tau)\ \varrho \prec s\ \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!] \\
[\![(\&\circ q : \tau)\ \varrho]\!] &\triangleq [\![\varrho]\!] & [\![(\&\circ q : \tau)\ \varrho \prec \&\circ r\ \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!] \\
[\![(\circ z = s)\ \varrho]\!] &\triangleq [\![\varrho]\!] & [\![(\circ z = s)\ \varrho \prec \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!] \\
[\![\{\varphi\}\ \varrho]\!] &\triangleq [\![\varrho]\!] & [\![\{\varphi\}\ \varrho \prec \bar{a}]\!] &\triangleq [\![\varrho \prec \bar{a}]\!]
\end{aligned}$$

$$[\![(\&\circ q : \tau)\ \varrho \prec \&\bullet r\ \bar{a}]\!] \triangleq \begin{cases} [\![\varrho \prec \bar{a}]\!] & \text{if pre-write } [q] \text{ does not occur anywhere in } \varrho, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

As before, ϱ_h is a specialized contract of h and $\overline{xqg}_\pi$ are the formal parameters of π. The first rule replaces a skippable expression with a call to an appropriate handler; if the expression can not be skipped, we analyse it piece by piece. The three rules for reference allocation demonstrate three fundamental cases in ghost code elimination: a material reference initialized with a material term is admitted, a material reference initialized with a ghost term is rejected, and, finally, a ghost reference is removed from the program regardless of how it is initialized.

We see these three cases again in the three groups of rules for contracts and applications. Material parameters and material arguments passed for them are admitted; and so are material pre-writes, as well as handler parameters and arguments. Ghost values supplied for material parameters or material auxiliary variables are rejected. Ghost pre-writes, parameters, and auxiliaries are removed; and so are preconditions which may depend on them.

One special case is passing a material reference argument for a ghost formal reference parameter. The called handler expects a ghost reference, and so whatever data it puts there should be considered ghost: we cannot let material code access this data. Thus, if anywhere in the called handler's contract, we see a pre-write into this reference parameter, we reject the program. Otherwise, if no modification is announced, we consider the application safe and simply remove the parameter and the argument. But what happens if the called handler modifies its ghost reference parameter and then escapes by calling some other handler in its context? This write effect would not appear anywhere in the contract: there we only find the pre-writes that are seen by the handler parameters.

The non-interference in this case is ensured by the alias safety. Indeed, neither the called handler nor the handlers in its context can be in the scope of the material reference that we pass as an argument. This means that if the called handler escapes, the subsequent computation loses access to the original material reference. On the other hand, if the called handler returns via one of the handler parameters, the write effect has to be accounted for in the contract, as hidden effects are not allowed. To sum up, we admit passing a material reference argument for a ghost reference parameter in exactly those cases where we can make a ghost copy of the original reference and pass the copy instead.

examples

Inference: add ghost annotations, allowing more handlers to be checkpoints and allowing more computations to be skipped (due to references becoming ghost). We do not aim for complete analysis, and various policies of convenience are possible. One similar to that of Why3 is as follows: — an allocated reference is marked ghost if its initialisation term is ghost — a parameter of a defined partial handler f is marked ghost if f ever receives a ghost argument in that position (either directly or via Impl) — all reference, term, and total handler parameters of a defined partial handler f are marked ghost, if f is ever passed as an argument in a call to a total handler that we will have to skip (due to a ghost argument passed for a non-ghost parameter, either directly or via Impl). — we never modify the signatures of handler parameters, library handlers or total defined handlers: these are expected to be correctly specified by the programmer.

Index

$$
\begin{array}{rcl}
\mathit{term} & ::= & \mathit{variable} \\
& | & \texttt{true} \mid \texttt{false} \mid \ldots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
& | & \mathit{term}\ \texttt{+}\ \mathit{term} \mid \mathit{term}\ \texttt{-}\ \mathit{term} \mid \mathit{term}\ \texttt{*}\ \mathit{term} \\
& | & \mathit{term}\ \texttt{=}\ \mathit{term} \mid \mathit{term}\ \texttt{<}\ \mathit{term} \mid \mathit{term}\ \texttt{>}\ \mathit{term} \\
& | & \texttt{cons}\ \mathit{term}\ \mathit{term} \mid \texttt{nil} \mid \texttt{isCons}\ \mathit{term} \mid \ldots \\
\\
\mathit{termType} & ::= & \mathit{typeVariable} \\
& | & \texttt{bool} \mid \texttt{int} \mid \texttt{list}\ \mathit{termType} \mid \ldots
\end{array}
$$

Figure 7: Data terms and types.

$$
\begin{array}{rcl}
\mathit{expression} & ::= & \mathit{handler}\ \mathit{argument}* \\
& | & \mathit{expression}\ \texttt{/}\ \mathit{abbreviation} \\
& | & \mathit{expression}\ \texttt{/}\ \mathit{definition}+ \\
& | & \mathit{expression}\ \texttt{/}\ \texttt{\&}\ \mathit{variable}\ \texttt{=}\ \mathit{term} \\
\\
\mathit{argument} & ::= & \mathit{term} \mid \texttt{\&}\ \mathit{variable} \mid \mathit{handler} \\
\\
\mathit{abbreviation} & ::= & \mathit{handler}\ \mathit{contract}\ \texttt{->}\ \mathit{expression} \\
\\
\mathit{definition} & ::= & \mathit{handler}\ \mathit{contract}\ \texttt{=}\ \mathit{expression} \\
\\
\mathit{contract} & ::= & \mathit{pre\text{-}write}*\ \mathit{annotation}* \\
\\
\mathit{pre\text{-}write} & ::= & \texttt{[}\ \mathit{variable}\ \texttt{]} \\
\\
\mathit{annotation} & ::= & \mathit{variable}\ \texttt{:}\ \mathit{termType} \\
& | & \texttt{\&}\ \mathit{variable}\ \texttt{:}\ \mathit{termType} \\
& | & \mathit{handler}\ \texttt{:}\ \mathit{contract} \\
& | & \mathit{variable}\ \texttt{=}\ \mathit{term} \\
& | & \texttt{\{}\ \mathit{formula}\ \texttt{\}}
\end{array}
$$

Figure 8: Program expressions.

$$
\begin{array}{rcl}
\mathit{expressionType} & ::= & \bot \\
& | & \mathit{termType} \rightarrow \mathit{expressionType} \\
& | & \texttt{\&}\ \mathit{termType} \rightarrow \mathit{expressionType} \\
& | & \mathit{expressionType} \rightarrow \mathit{expressionType} \\
\\
\mathit{handlerType} & ::= & \langle\ \mathit{typeVariable}*\ \rangle\ \mathit{expressionType} \\
\\
\mathit{vc} & ::= & \top \quad | \quad \mathit{vc} \wedge \mathit{vc} \\
& | & \mathit{formula} \quad | \quad \mathit{formula} \rightarrow \mathit{vc} \\
& | & \forall\ \mathit{variable}\ :\ \mathit{termType}\ .\ \mathit{vc} \\
& | & \mathit{vc}\ \texttt{/}\ \mathit{handler}\ (\mathit{variable}\ :\ \mathit{termType})* \equiv \mathit{vc} \\
& | & \mathit{handler}\ \mathit{term}*
\end{array}
$$

Figure 9: Program types and verification conditions.

$$\frac{h \mapsto \varrho \succ d \in \Lambda \qquad \Sigma \vdash (\varrho \succ d) \prec \bar{a} \longrightarrow^* (\square \succ e) \prec \square}{\Lambda \vdash h\ \bar{a} \bullet \Sigma \longrightarrow e \bullet \Sigma} \text{ (E-Call)}$$

$$\frac{\Lambda, h \mapsto \varrho \succ d \vdash e \mathbin{/} \bar{\gamma} \bullet \Sigma \longrightarrow e' \mathbin{/} \bar{\gamma} \bullet \Sigma'}{\Lambda \vdash e \mathbin{/} \bar{\gamma}\ (h\ \varrho = d) \bullet \Sigma \longrightarrow e' \mathbin{/} \bar{\gamma}\ (h\ \varrho = d) \bullet \Sigma'} \text{ (E-Defn)}$$

$$\frac{\Lambda, h \mapsto \varrho \succ d \vdash e \bullet \Sigma \longrightarrow e' \bullet \Sigma'}{\Lambda \vdash e \mathbin{/} h\ \varrho \twoheadrightarrow d \bullet \Sigma \longrightarrow e' \mathbin{/} h\ \varrho \twoheadrightarrow d \bullet \Sigma'} \text{ (E-Abbr)}$$

$$\frac{\Lambda \vdash e \bullet \Sigma, r \mapsto [\![s]\!]_\Sigma \longrightarrow e' \bullet \Sigma', r \mapsto t}{\Lambda \vdash e \mathbin{/} \&r = s \bullet \Sigma \longrightarrow e' \mathbin{/} \&r = t \bullet \Sigma'} \text{ (E-Alloc)}$$

$$\Sigma \vdash ([p]\ \varrho \succ e) \prec \bar{a} \longrightarrow (\varrho \succ e) \prec \bar{a} \text{ (E-Eff)}$$

$$\Sigma \vdash ((x : \tau)\ \varrho \succ e) \prec s\ \bar{a} \longrightarrow (\varrho \succ e)[x \mapsto [\![s]\!]_\Sigma] \prec \bar{a} \text{ (E-AppT)}$$

$$\Sigma \vdash ((\&q : \tau)\ \varrho \succ e) \prec \&r\ \bar{a} \longrightarrow (\varrho \succ e)[q \mapsto r] \prec \bar{a} \text{ (E-AppR)}$$

$$\Sigma \vdash ((g : \pi)\ \varrho \succ e) \prec f\ \bar{a} \longrightarrow (\varrho \succ e)[g \mapsto f] \prec \bar{a} \text{ (E-AppH)}$$

$$\Sigma \vdash ((z = s)\ \varrho \succ e) \prec \bar{a} \longrightarrow (\varrho \succ e)[z \mapsto [\![s]\!]_\Sigma] \prec \bar{a} \text{ (E-Aux)}$$

$$\Sigma \vdash (\{\varphi\}\ \varrho \succ e) \prec \bar{a} \longrightarrow (\varrho \succ e) \prec \bar{a} \text{ (E-Pre)}$$

Figure 10: Operational semantics.

$$\frac{h : \langle \bar{\alpha} \rangle \xi \in \Gamma}{\Gamma \vdash h\ \square : \xi[\bar{\alpha} \mapsto \bar{\tau}]} \text{ (T-Call)} \qquad \frac{\Gamma \vdash h\ \bar{a} : \tau \to \xi \qquad \Gamma \vdash s : \tau}{\Gamma \vdash h\ \bar{a}\ s : \xi} \text{ (T-AppT)}$$

$$\left[\frac{\Gamma \vdash h\ \bar{a} : \&\tau \to \xi \qquad \&r : \tau \in \Gamma}{\Gamma \vdash h\ \bar{a}\ \&r : \xi} \text{ (T-AppR}_0\text{)}\right] \qquad \frac{\Gamma \vdash h\ \bar{a} : \zeta \to \xi \qquad \Gamma \vdash g\ \square : \zeta}{\Gamma \vdash h\ \bar{a}\ g : \xi} \text{ (T-AppH)}$$

$$\frac{\Gamma, \Delta' \vdash h\ \bar{a} : \&\tau \to \xi \qquad \Delta' \text{ is } \Delta \text{ with all handler bindings removed}}{\Gamma, \&r : \tau, \Delta \vdash h\ \bar{a}\ \&r : \xi} \text{ (T-AppR)}$$

$$\frac{\Gamma \vdash \varrho : \xi \qquad \Gamma, h : \langle \bar{\alpha} \rangle \xi \vdash e \mathbin{/} (\_\ \varrho \twoheadrightarrow d) \mathbin{/} \bar{\gamma} : \bot \qquad \text{no type variable in } \bar{\alpha} \text{ is free in } \Gamma}{\Gamma \vdash e \mathbin{/} \bar{\gamma}\ (h\ \varrho = d) : \bot} \text{ (T-Defn)}$$

$$\frac{\Gamma \vdash \varrho \succ d : \xi \qquad \Gamma, h : \xi \vdash e : \bot}{\Gamma \vdash e \mathbin{/} h\ \varrho \twoheadrightarrow d : \bot} \text{ (T-Abbr)} \qquad \frac{\Gamma \vdash s : \tau \qquad \Gamma, \&r : \tau \vdash e : \bot}{\Gamma \vdash e \mathbin{/} \&r = s : \bot} \text{ (T-Alloc)}$$

$$\frac{\Gamma, \_ : \bot \vdash \pi \succ \_\ \square : \zeta}{\Gamma \vdash \pi : \zeta} \text{ (T-Contract)} \qquad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash \square \succ e : \bot} \text{ (T-NullSpec)}$$

$$\frac{\Gamma, x : \tau \vdash \varrho \succ e : \xi}{\Gamma \vdash (x : \tau)\ \varrho \succ e : \tau \to \xi} \text{ (T-ParT)} \qquad \frac{\&p : \tau \in \Gamma \qquad \Gamma \vdash \varrho \succ e : \xi}{\Gamma \vdash [p]\ \varrho \succ e : \xi} \text{ (T-Eff)}$$

$$\frac{\Gamma, \&q : \tau \vdash \varrho \succ e : \xi}{\Gamma \vdash (\&q : \tau)\ \varrho \succ e : \&\tau \to \xi} \text{ (T-ParR)} \qquad \frac{\Gamma \vdash s : \tau \qquad \Gamma, z : \tau \vdash \varrho \succ e : \xi}{\Gamma \vdash (z = s)\ \varrho \succ e : \xi} \text{ (T-Aux)}$$

$$\frac{\Gamma \vdash \pi : \zeta \qquad \Gamma, g : \zeta \vdash \varrho \succ e : \xi}{\Gamma \vdash (g : \pi)\ \varrho \succ e : \zeta \to \xi} \text{ (T-ParH)} \qquad \frac{\Gamma \vdash \varphi : \texttt{bool} \qquad \Gamma \vdash \varrho \succ e : \xi}{\Gamma \vdash \{\varphi\}\ \varrho \succ e : \xi} \text{ (T-Pre)}$$

Above, the "non-rule" T-AppR$_0$, included for comparison with T-AppR, does not prevent aliases.

Figure 11: Typing rules.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{E}(h\ \bar{a}) & \triangleq & \mathcal{E}(\varrho_h \prec \bar{a}) & \mathcal{E}(e\ /\ \bar{\gamma}\ (h\ \varrho = d)) & \triangleq & \mathcal{E}(e\ /\ (\_\ \varrho \twoheadrightarrow d)\ /\ \bar{\gamma}) \ominus h \\
\mathcal{E}(e\ /\ \&r = s) & \triangleq & \mathcal{E}(e) \ominus r & \mathcal{E}(e\ /\ h\ \varrho \twoheadrightarrow d) & \triangleq & (\mathcal{E}(e) \ominus h) \cup \mathcal{E}(\varrho \succ d) \\
\mathcal{E}([p]\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) \cup (p \otimes \varrho \succ e) & \mathcal{E}([p]\ \varrho \prec \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\
\mathcal{E}((x:\tau)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) & \mathcal{E}((x:\tau)\ \varrho \prec s\ \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\
\mathcal{E}((\&q:\tau)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) \ominus q & \mathcal{E}((\&q:\tau)\ \varrho \prec \&r\ \bar{a}) & \triangleq & \mathcal{E}(\varrho[q \mapsto r] \prec \bar{a}) \\
\mathcal{E}((f:\pi)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) \ominus f & \mathcal{E}((f:\pi)\ \varrho \prec h\ \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \cup \mathcal{E}(\pi \succ h\ \overline{xqg}_\pi) \\
\mathcal{E}((z = s)\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) & \mathcal{E}((z = s)\ \varrho \prec \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\
\mathcal{E}(\{\varphi\}\ \varrho \succ e) & \triangleq & \mathcal{E}(\varrho \succ e) & \mathcal{E}(\{\varphi\}\ \varrho \prec \bar{a}) & \triangleq & \mathcal{E}(\varrho \prec \bar{a}) \\
\mathcal{E}(\square \succ e) & \triangleq & \mathcal{E}(e) & \mathcal{E}(\square \prec \square) & \triangleq & \varnothing \\
E \ominus r & \triangleq & \{\,(q,g) \in E \mid q \neq r\,\} & p \otimes \varrho \succ e & \triangleq & \{p\} \times \mathrm{FH}(\varrho \succ e)
\end{array}
$$

$$
E \ominus h \triangleq \begin{cases} \{\,(q,g) \in E \mid g \neq h\,\} & \text{if for each } (p,h) \text{ in } E,\ p \text{ is a formal pre-write of } h, \\ \text{undefined} & \text{otherwise.} \end{cases}
$$

Contract ϱ_h is a specialized (type-instantiated and with all bound variables refreshed) contract of h and $\overline{xqg}_\pi$ stands for the list of all parameters of π in the order of their appearance in the contract.

Figure 12: Effect computation.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{V}(h\ \bar{a}) & \triangleq & \mathcal{V}(\varrho_h^\star \prec \bar{a}) & \mathcal{V}(e\ /\ \bar{\gamma}\ (h\ \varrho = d)) & \triangleq & \mathcal{V}(e\ /\ \bar{\gamma}) \wedge \mathcal{V}(\varrho \succ d) \\
\mathcal{V}(e\ /\ \&r = s) & \triangleq & \mathcal{V}(e)[r \mapsto s] & \mathcal{V}(e\ /\ h\ \varrho \twoheadrightarrow d) & \triangleq & \mathcal{V}(e)\ /\ h\ \overline{pxqz}_\varrho = \mathcal{V}(d) \\
\mathcal{V}([p]\ \varrho \succ e) & \triangleq & \forall p:\tau_p\,.\,\mathcal{V}(\varrho \succ e) & \mathcal{V}([p]\ \varrho \prec \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a}) \\
\mathcal{V}((x:\tau)\ \varrho \succ e) & \triangleq & \forall x:\tau\,.\,\mathcal{V}(\varrho \succ e) & \mathcal{V}((x:\tau)\ \varrho \prec s\ \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a})[x \mapsto s] \\
\mathcal{V}((\&q:\tau)\ \varrho \succ e) & \triangleq & \forall q:\tau\,.\,\mathcal{V}(\varrho \succ e) & \mathcal{V}((\&q:\tau)\ \varrho \prec \&r\ \bar{a}) & \triangleq & \mathcal{V}(\varrho[q \mapsto r] \prec \bar{a}) \\
\mathcal{V}((f:\pi)\ \varrho \succ e) & \triangleq & \mathcal{V}(\varrho \succ e) & \mathcal{V}((f:\pi)\ \varrho \prec h\ \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a}) \wedge \mathcal{V}(\pi \succ h\ \overline{xqg}_\pi) \\
\mathcal{V}((z = s)\ \varrho \succ e) & \triangleq & \mathcal{V}(\varrho \succ e)[z \mapsto s] & \mathcal{V}((z = s)\ \varrho \prec \bar{a}) & \triangleq & \mathcal{V}(\varrho \prec \bar{a})[z \mapsto s] \\
\mathcal{V}(\{\varphi\}\ \varrho \succ e) & \triangleq & \varphi \rightarrow \mathcal{V}(\varrho \succ e) & \mathcal{V}(\{\varphi\}\ \varrho \prec \bar{a}) & \triangleq & \varphi \wedge \mathcal{V}(\varrho \prec \bar{a}) \\
\mathcal{V}(\square \succ e) & \triangleq & \mathcal{V}(e) & \mathcal{V}(\square \prec \square) & \triangleq & \top
\end{array}
$$

$$
\varrho_h^\star \triangleq \begin{cases} \varrho_h\ \{\,h\ \overline{pxqz}_{\varrho_h}\,\} & \text{if } h \text{ is an abbreviation,} \\ \varrho_h\ \{\,\hat{u} \prec \hat{v}\,\} & \text{if } \hat{u} \text{ is a variant auxiliary in } \varrho_h, \\ & \quad \hat{v} \text{ appears in the current typing context,} \\ & \quad \hat{v} \text{ is a variant auxiliary of handler } g, \text{ and} \\ & \quad g \text{ is } h \text{ or defined in the same group as } h, \\ \varrho_h & \text{otherwise.} \end{cases}
$$

Contract ϱ_h is a specialized (type-instantiated and with all bound variables refreshed) contract of h, $\overline{pxqz}_\varrho$ stands for the list of pre-writes, term and reference parameters, and auxiliary variables of ϱ, $\overline{xqg}_\pi$ stands for the list of all parameters of π in the order of their appearance in the contract, and τ_p is the type of reference p. Relation $\prec$ is well-founded and used for all handlers in the group of h and g.

Figure 13: Verification condition generation.

$$
\begin{array}{rcl@{\qquad}rcl}
\llbracket h\ \bar{a} \rrbracket & \triangleq & h\ \llbracket \varrho_h \prec \bar{a} \rrbracket & \llbracket e\ /\ \bar{\gamma}\ (h\ \varrho = d) \rrbracket & \triangleq & \llbracket e\ /\ \bar{\gamma} \rrbracket\ (h\ \llbracket \varrho \rrbracket = \llbracket d \rrbracket) \\
\llbracket e\ /\ \&r = s \rrbracket & \triangleq & \llbracket e \rrbracket [r \mapsto s] & \llbracket e\ /\ h\ \varrho \twoheadrightarrow d \rrbracket & \triangleq & \llbracket e \rrbracket\ /\ h\ \llbracket \varrho \rrbracket \twoheadrightarrow \llbracket d \rrbracket \\
\llbracket [p]\ \varrho \rrbracket & \triangleq & (p : \tau_p)\ \llbracket \varrho \rrbracket & \llbracket [p]\ \varrho \prec \bar{a} \rrbracket & \triangleq & p\ \llbracket \varrho \prec \bar{a} \rrbracket \\
\llbracket (x : \tau)\ \varrho \rrbracket & \triangleq & (x : \tau)\ \llbracket \varrho \rrbracket & \llbracket (x : \tau)\ \varrho \prec s\ \bar{a} \rrbracket & \triangleq & s\ \llbracket \varrho \prec \bar{a} \rrbracket \\
\llbracket (\&q : \tau)\ \varrho \rrbracket & \triangleq & (q : \tau)\ \llbracket \varrho \rrbracket & \llbracket (\&q : \tau)\ \varrho \prec \&r\ \bar{a} \rrbracket & \triangleq & r\ \llbracket \varrho [q \mapsto r] \prec \bar{a} \rrbracket \\
\llbracket (f : \pi)\ \varrho \rrbracket & \triangleq & (f : \llbracket \pi \rrbracket)\ \llbracket \varrho \rrbracket & \llbracket (f : \pi)\ \varrho \prec h\ \bar{a} \rrbracket & \triangleq & f\ \llbracket \varrho \prec \bar{a} \rrbracket\ /\ f\ \llbracket \pi \rrbracket \twoheadrightarrow \llbracket h\ \overline{xqg}_\pi \rrbracket \\
\llbracket (z = s)\ \varrho \rrbracket & \triangleq & (z = s)\ \llbracket \varrho \rrbracket & \llbracket (z = s)\ \varrho \prec \bar{a} \rrbracket & \triangleq & \llbracket \varrho \prec \bar{a} \rrbracket \\
\llbracket \{\varphi\}\ \varrho \rrbracket & \triangleq & \{\varphi\}\ \llbracket \varrho \rrbracket & \llbracket \{\varphi\}\ \varrho \prec \bar{a} \rrbracket & \triangleq & \llbracket \varrho \prec \bar{a} \rrbracket \\
\llbracket \square \rrbracket & \triangleq & \square & \llbracket \square \prec \square \rrbracket & \triangleq & \square
\end{array}
$$

Contract ϱ_h is a specialized (type-instantiated and with all bound variables refreshed) contract of h, $\overline{xqg}_\pi$ stands for the list of all parameters of π in the order of their appearance in the contract, and τ_p is the type of reference p.

Figure 14: State elimination.

$$
\lfloor\!\lfloor e \rfloor\!\rfloor \triangleq \begin{cases} h\ \square & \text{if } e \text{ is skippable up to handler } h, \\ \lfloor e \rfloor & \text{otherwise.} \end{cases}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\lfloor h\ \bar{a} \rfloor & \triangleq & h\ \lfloor\!\lfloor \varrho_h \prec \bar{a} \rfloor\!\rfloor & \lfloor e\ /\ \bar{\gamma}\ (h\ \varrho = d) \rfloor & \triangleq & \lfloor\!\lfloor e\ /\ \bar{\gamma} \rfloor\!\rfloor\ (h\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor = \lfloor\!\lfloor d \rfloor\!\rfloor) \\
\lfloor e\ /\ \&\bullet r = \bullet s \rfloor & \triangleq & \lfloor\!\lfloor e \rfloor\!\rfloor\ /\ \&r = s & \lfloor e\ /\ h\ \varrho \twoheadrightarrow d \rfloor & \triangleq & \lfloor\!\lfloor e \rfloor\!\rfloor\ /\ h\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor \twoheadrightarrow \lfloor\!\lfloor d \rfloor\!\rfloor \\
\lfloor e\ /\ \&\bullet r = \circ s \rfloor & \text{is undefined} & & \lfloor e\ /\ \&\circ r = s \rfloor & \triangleq & \lfloor\!\lfloor e \rfloor\!\rfloor \\
\lfloor\!\lfloor [\bullet p]\ \varrho \rfloor\!\rfloor & \triangleq & [p]\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor [\bullet p]\ \varrho \prec \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor (\bullet x : \tau)\ \varrho \rfloor\!\rfloor & \triangleq & (x : \tau)\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (\bullet x : \tau)\ \varrho \prec \bullet s\ \bar{a} \rfloor\!\rfloor & \triangleq & s\ \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor (\&\bullet q : \tau)\ \varrho \rfloor\!\rfloor & \triangleq & (\&q : \tau)\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (\&\bullet q : \tau)\ \varrho \prec \&\bullet r\ \bar{a} \rfloor\!\rfloor & \triangleq & \&r\ \lfloor\!\lfloor \varrho [q \mapsto r] \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor (f : \pi)\ \varrho \rfloor\!\rfloor & \triangleq & (f : \lfloor\!\lfloor \pi \rfloor\!\rfloor)\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (f : \pi)\ \varrho \prec h\ \bar{a} \rfloor\!\rfloor & \triangleq & f\ \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor\ /\ f\ \lfloor\!\lfloor \pi \rfloor\!\rfloor \twoheadrightarrow \lfloor\!\lfloor h\ \overline{xqg}_\pi \rfloor\!\rfloor \\
\lfloor\!\lfloor (\bullet z = \bullet s)\ \varrho \rfloor\!\rfloor & \triangleq & (z = s)\ \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (\bullet z = \bullet s)\ \varrho \prec \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor \square \rfloor\!\rfloor & \triangleq & \square & \lfloor\!\lfloor \square \prec \square \rfloor\!\rfloor & \triangleq & \square \\
\lfloor\!\lfloor (\bullet z = \circ s)\ \varrho \rfloor\!\rfloor & \text{is undefined} & & \lfloor\!\lfloor (\bullet x : \tau)\ \varrho \prec \circ s\ \bar{a} \rfloor\!\rfloor & \text{is undefined} & \\
\lfloor\!\lfloor (\bullet z = \circ s)\ \varrho \prec \bar{a} \rfloor\!\rfloor & \text{is undefined} & & \lfloor\!\lfloor (\&\bullet q : \tau)\ \varrho \prec \&\circ r\ \bar{a} \rfloor\!\rfloor & \text{is undefined} & \\
\lfloor\!\lfloor [\circ p]\ \varrho \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor [\circ p]\ \varrho \prec \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor (\circ x : \tau)\ \varrho \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (\circ x : \tau)\ \varrho \prec s\ \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor (\&\circ q : \tau)\ \varrho \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (\&\circ q : \tau)\ \varrho \prec \&\circ r\ \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor (\circ z = s)\ \varrho \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor (\circ z = s)\ \varrho \prec \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor \\
\lfloor\!\lfloor \{\varphi\}\ \varrho \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \rfloor\!\rfloor & \lfloor\!\lfloor \{\varphi\}\ \varrho \prec \bar{a} \rfloor\!\rfloor & \triangleq & \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor
\end{array}
$$

$$
\lfloor\!\lfloor (\&\circ q : \tau)\ \varrho \prec \&\bullet r\ \bar{a} \rfloor\!\rfloor \triangleq \begin{cases} \lfloor\!\lfloor \varrho \prec \bar{a} \rfloor\!\rfloor & \text{if pre-write } [q] \text{ does not occur anywhere in } \varrho, \\ \text{undefined} & \text{otherwise.} \end{cases}
$$

Notation $\bullet X$ means that variable or term X is material and $\circ X$ means that variable or term X is ghost. Contract ϱ_h is a specialized (type-instantiated and with all bound variables refreshed) contract of h, and $\overline{xqg}_\pi$ stands for the list of all parameters of π in the order of their appearance in the contract.

Figure 15: Ghost code elimination.