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

In our language, we separate code and data. It will not matter much for our purposes what kinds of data we work with. For now, we assume that the only data types we have are (unbounded) integers and Booleans. Later, we will show that our language can easily adopt algebraic types and first-rank type polymorphism, with the possibility of type inference in the Hindley-Milner style.

The code is structured in expressions and handlers. An *expression* is a computation that can be executed in an appropriate context, possibly altering the state along the way, and then passing control to some handler (continuation). A *handler* is a computation parametrized by a number of data parameters and a number of handler parameters.

Let us consider an example of a function that computes the factorial of a natural number. The concrete syntax we use in this and other examples serves the illustrative purposes only and is not meant to represent a realistic programming language.

```
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) then else
7       / then = fact (r * k) (k - 1)
8         else = return r
```

Lines 1 and 2 introduce the prototype of a handler. The handler is called `factorial`, 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 handler parameter, named `return`, which takes an integer argument `m`, and the precondition for `return` is that `m` is the factorial of `n`. Handler `return` takes no handler parameters, it is a true continuation for `factorial`.

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. Character `/` 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 prototype:

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

The `if` handler expects a single data parameter `c` with an empty (i.e., trivially true) precondition. It also takes two nullary handler parameters, named `then` and `else`. The precondition of `then` is that `c` is true, and the precondition of `else` is that `c` is false.

Going back to `fact`, the actual handlers `then` and `else` supplied to `if` are defined in lines 7 and 8, respectively. Whenever `k` is strictly positive, `if` passes control to `then`, which makes a recursive call of `fact`. Otherwise, if `k` is zero or negative, the `else` handler passes the parameter `r` to the `return` handler, leaving `factorial`. Notice that `fact` never returns to the caller: to do that, it would need to receive a handler parameter and call it, like `if` and `factorial` do. Instead, at the last call, `fact` escapes by calling `return`. 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`. Indeed, in our language, postconditions are preconditions of handler parameters, and `fact` has none thereof.

To verify a handler, 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 $\mathrm{WP}_{N\text{-}M}$ to denote the weakest precondition for the expression written in lines N to M):

$$
\begin{aligned}
\mathrm{VC}_{\texttt{factorial}} &= \forall \mathtt{n}.\ 0 \leqslant \mathtt{n} \rightarrow \mathrm{WP}_{3\text{-}8} \\
\mathrm{WP}_{3\text{-}8} &= \mathrm{WP}_3 \ \wedge\ \mathrm{VC}_{\texttt{fact}} \\
\mathrm{WP}_3 &= (0 \leqslant \mathtt{k} \leqslant \mathtt{n} \ \wedge\ \mathtt{r} \cdot \mathtt{k}! = \mathtt{n}!)\,[\mathtt{r} \mapsto 1, \mathtt{k} \mapsto \mathtt{n}] \\
&= 0 \leqslant \mathtt{n} \leqslant \mathtt{n} \ \wedge\ 1 \cdot \mathtt{n}! = \mathtt{n}! \\
\mathrm{VC}_{\texttt{fact}} &= \forall \mathtt{r}\,\mathtt{k}.\ (0 \leqslant \mathtt{k} \leqslant \mathtt{n} \ \wedge\ \mathtt{r} \cdot \mathtt{k}! = \mathtt{n}!) \ \rightarrow\ \mathrm{WP}_{6\text{-}8} \\
\mathrm{WP}_{6\text{-}8} &= \mathrm{WP}_6 \\
\mathrm{WP}_6 &= (\mathtt{k} > 0 \rightarrow \mathrm{WP}_7) \ \wedge\ (\neg(\mathtt{k} > 0) \rightarrow \mathrm{WP}_8) \\
\mathrm{WP}_7 &= (0 \leqslant \mathtt{k} \leqslant \mathtt{n} \ \wedge\ \mathtt{r} \cdot \mathtt{k}! = \mathtt{n}!)\,[\mathtt{r} \mapsto \mathtt{r} \cdot \mathtt{k}, \mathtt{k} \mapsto \mathtt{k} - 1] \\
&= 0 \leqslant \mathtt{k} - 1 \leqslant \mathtt{n} \ \wedge\ \mathtt{r} \cdot \mathtt{k} \cdot (\mathtt{k} - 1)! = \mathtt{n}! \\
\mathrm{WP}_8 &= (\mathtt{m} = \mathtt{n}!)\,[\mathtt{m} \mapsto \mathtt{r}] \\
&= \mathtt{r} = \mathtt{n}!
\end{aligned}
$$

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 that have explicitly written preconditions ($\mathrm{VC}_{\texttt{fact}}$). Calls without handler parameters (WP_3, WP_7, WP_8) require us to merely verify the instantiated precondition of the callee. Calls with handler parameters (WP_6) also require checking that the actual handler arguments satisfy the specification of the formal handler parameters. In our example, this means that the preconditions of the two handler parameters in the prototype of `if` must imply, respectively, the preconditions of the actual handlers `then` and `else` defined in lines 7 and 8. Since we have written no explicit preconditions for these handlers, the computed weakest preconditions (WP_7 and WP_8) are used instead.

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 and a polymorphic ML-style type system in Section 4. We modify this type system in Section 5 in order to ensure the alias safety of well-typed programs.

In Section 6, 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, as presented in Section 7.

In Section 8, we equip our handlers with functional specifications and show how to compute verification conditions for partial correctness. In particular, we demonstrate that classical Dijkstra-style weakest preconditions and efficient weakest preconditions as described by Flanagan and Saxe cite can both be derived as equivalent first-order forms of the same second-order formula. In Section 9, we add *auxiliary variables* that allow our specifications to refer to past values of modified mutable variables. Termination proofs, which give us full functional correctness, are treated in Section 10.

Finally, in Section 11, 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 typing and functional 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 The Pure Fragment

A program in our language is a collection of handlers calling each other. A computable *expression* consists of a single handler call followed by a number of handler definitions. A *handler call* consists of the name of the called handler and a list of data arguments and handler arguments. A *handler definition* consists of a *prototype* — the handler's name, its data parameters and handler parameters — and a *body*, an expression that implements the handler. Partial applications are not accepted: the actual data and handler arguments in a handler call must correspond exactly to the formal data and handler parameters in the prototype of the called handler. Here is the formal grammar:

$$
\begin{aligned}
\mathit{expression} &::= \mathit{handler}\ \ \mathit{term}{*}\ \ \mathit{handler}{*} \\
&\mid\ \ \mathit{expression}\ \texttt{/}\ \ \mathit{definition}{+} \\
\mathit{term} &::= \mathit{variable} \mid \mathit{constant} \\
\mathit{definition} &::= \mathit{prototype}\ \texttt{=}\ \mathit{expression} \\
\mathit{prototype} &::= \mathit{handler}\ \ \mathit{variable}{*}\ \ \mathit{prototype}{*}
\end{aligned}
$$

Lexemes *handler* and *variable* represent handler names and data variables, respectively. Lexeme *constant* represents data values such as integer numerals or propositional constants `true` and `false`. In our examples, we often use richer terms that contain some basic propositional and arithmetic operations, provided that they are total and effect-free and can be directly translated into specification. However, such operations can always be implemented as handlers and the restrictive definition of a term given above is sufficient for our purposes.

In the next section, we introduce mutable variables that can be allocated inside expressions and passed between handlers. After that, all changes in the syntax are limited to handler prototypes, as we add various annotations related to types, effects, preconditions, etc. We do not need to extend our language by adding dedicated control structures for conditionals, loops, pattern matching, exception handling: all of these can be conveniently expressed with handlers.

We use letters e and d for expressions, δ and γ for definitions, π and ϱ for prototypes, f, g, h for handlers, x, y, z for variables, and t and s for terms. In any prototype $f\,\bar{x}\,\bar{\pi}$, no variable can appear twice in $\bar{x}$ and no handler name can appear twice in $\bar{\pi}$. Likewise, in any expression $e\ \texttt{/}\ \bar{\gamma}$, no handler can be defined twice in $\bar{\gamma}$. We write $\natural\gamma$ or $\natural\varrho$ to denote the name of a handler defined by definition γ or described by prototype ϱ, and likewise for sequences of definitions and prototypes.

We define the *rank* of a handler according to its handler parameters. A handler whose prototype has no handler parameters has rank 0 and is called a *sink*. A handler whose handler parameters have

ranks up to and including r, has rank $r + 1$. In the example from the previous section, `fact`, `return`, `then`, and `else` are sinks, whereas `factorial` and `if` have rank 1.

Lexical scoping. Expressions and handler definitions obey the rules of lexical scoping. Assuming that the free variables of a term are simply all variables that occur in it (this works both for rich terms of our examples and restricted terms of our formal syntax), we compute the free variables of expressions and handler definitions as follows:

$$\mathrm{FV}(h\ \bar{s}\ \bar{g}) \triangleq \mathrm{FV}(\bar{s}) \qquad \mathrm{FV}(h\ \bar{x}\ \bar{\varrho} = d) \triangleq \mathrm{FV}(d) \setminus \{\bar{x}\}$$
$$\mathrm{FV}(e\ /\ \bar{\gamma}) \triangleq \mathrm{FV}(e) \cup \mathrm{FV}(\bar{\gamma})$$

When we write $\mathrm{FV}(\bar{s})$ or $\mathrm{FV}(\bar{\gamma})$, we mean the union of the free variable sets for each term in $\bar{s}$ and each definition in $\bar{\gamma}$, respectively. In the current syntax, variables can not occur freely on the left-hand side of a handler definition (i.e., in a prototype), but this will change when we start using annotations. An expression is called *closed* if it contains no free variables.

We compute the free handlers of expressions and handler definitions as follows:

$$\mathrm{FH}(h\ \bar{s}\ \bar{g}) \triangleq \{h, \bar{g}\} \qquad \mathrm{FH}(h\ \bar{x}\ \bar{\varrho} = d) \triangleq \mathrm{FH}(d) \setminus \{\natural\bar{\varrho}\}$$
$$\mathrm{FH}(e\ /\ \bar{\gamma}) \triangleq (\mathrm{FH}(e) \cup \mathrm{FH}(\bar{\gamma})) \setminus \{\natural\bar{\gamma}\}$$

Here, $\{\natural\bar{\gamma}\}$ and $\{\natural\bar{\varrho}\}$ represent the sets of handler names defined in $\bar{\gamma}$ or described in $\bar{\varrho}$, respectively. We do not define operator FH on prototypes, because handlers cannot occur freely inside a prototype (not even in an annotated one).

In our factorial example, the free variables and handlers in the body of `fact` (lines 6-8) are $\{\mathtt{r}, \mathtt{k}\}$ and $\{\mathtt{fact}, \mathtt{if}, \mathtt{return}\}$, respectively: the handlers `then` and `else` are defined inside this expression and do not escape. The only free variable in the definition of `fact` is `n`, as will be seen later, when we update the definition of FV to deal with annotations in prototypes. The free handlers in the definition of `fact` are again `if`, `return`, and `fact` itself. That a handler freely occurs in its own definition means that it is defined recursively. The definition of `factorial` does not contain free variables and the only free handler in it is `if`.

Since handler definitions are a binding point both for variables and handlers, the definitions above do not give a clear-cut notion of the scope relationship between these two entities. We will say that in a handler prototype $h\ \bar{x}\ \bar{\varrho}$, each handler in $\bar{\varrho}$ (and every data and handler parameter of those handlers, and every parameter of those handler parameters, etc.) is *in the scope* of the variables $\bar{x}$. We will say that in a handler definition $h\ \bar{x}\ \bar{\varrho} = e$, every variable and handler bound inside e is *in the scope* of the variables $\bar{x}$ and handlers $\bar{\varrho}$. We will say that in an expression $e\ /\ \bar{\gamma}$, every variable and handler bound inside e or inside $\bar{\gamma}$ is *in the scope* of the handlers $\bar{\gamma}$. We will say "A is *accessible* to B" as as a synonym for "B is in the scope of A".

Operational semantics. We define a small-step operational semantics on closed expressions. An *evaluation step* consists in evaluating the leftmost handler call. We write $e\ //\ \Delta$ as an abbreviation for $e\,/\,\bar{\gamma}_1 \ldots /\,\bar{\gamma}_n$, an expression followed by several (possibly zero) definition blocks. Here, Δ is the syntactic context of subexpression e, which contains all definitions $\bar{\gamma}_1, \ldots, \bar{\gamma}_n$ accessible to e. In an expression $h\ \bar{s}\ \bar{g}\ //\ \Delta$ that is subject to evaluation, Δ is called *execution context* and the handlers in Δ are *top-level handlers*. We assume that there are no name collisions in Δ.

When the called handler is a top-level handler, the evaluation rule is as follows:

$$h\ \bar{s}\ \bar{g}\ //\ \Delta \longrightarrow d[\bar{x}, \natural\bar{\varrho} \mapsto \bar{s}, \bar{g}]\ //\ \Delta \qquad \text{where } \Delta \text{ contains definition } h\ \bar{x}\ \bar{\varrho} = d$$

The substitution $\natural\bar{\varrho} \mapsto \bar{g}$ means that we replace the names of handlers described in prototypes $\bar{\varrho}$ with the handlers $\bar{g}$ that are passed to h as actual handler arguments. To avoid name capture, we perform an implicit alpha-conversion on the definition of h and give fresh names to all handlers defined in d.

Of course, the evaluation rule above is not enough. Expressions and handlers are mere control structures, and all actual work has to be done by predefined *library handlers*. The semantics of a library handler is axiomatized by evaluation rules like these:

$$\texttt{if true}\ f\ g\ /\!/\ \Delta \longrightarrow f\ /\!/\ \Delta \qquad \texttt{plus}\ n\ m\ h\ /\!/\ \Delta \longrightarrow h\ (n+m)\ /\!/\ \Delta$$
$$\texttt{if false}\ f\ g\ /\!/\ \Delta \longrightarrow g\ /\!/\ \Delta \qquad \texttt{less}\ n\ m\ h\ /\!/\ \Delta \longrightarrow h\ (n<m)\ /\!/\ \Delta$$

where $(n+m)$ is the integer constant equal to the sum of integer constants n and m, and $(n<m)$ is the Boolean constant `true` if n is less than m, and `false` otherwise. For a slightly more involved example, we can consider the two constructors and destructor for singly-linked lists:

$$\texttt{cons}\ a\ l\ h\ /\!/\ \Delta \longrightarrow h\ (a :: l)\ /\!/\ \Delta \qquad \texttt{unList}\ (a :: l)\ f\ g\ /\!/\ \Delta \longrightarrow f\ a\ l\ /\!/\ \Delta$$
$$\texttt{nil}\ h\ /\!/\ \Delta \longrightarrow h\ \Diamond\ /\!/\ \Delta \qquad \texttt{unList}\ \Diamond\ f\ g\ /\!/\ \Delta \longrightarrow g\ /\!/\ \Delta$$

where $\Diamond$ is the empty list and $(a :: l)$ is the list that consists of a head element a and a tail list l. Handler `nil` is, generally speaking, redundant, as we can use constant $\Diamond$ directly.

Let us see how evaluation works on our running example. Let $\delta_{\texttt{fact}}$, $\delta_{\texttt{then}}[r,k]$, and $\delta_{\texttt{else}}[r]$ be the definitions of `fact`, `then`, and `else`, respectively (here, we ignore the precondition of `fact` and thus do not consider n as a free variable in $\delta_{\texttt{fact}}$).

Computing `fact 3 2` / $\delta_{\texttt{fact}}$ produces the following evaluation chain:

$$\begin{aligned}
&\texttt{fact}\ 3\ 2\ /\ \delta_{\texttt{fact}} \longrightarrow \texttt{if}\ (2>0)\ \texttt{then else}\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad \texttt{then}\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad \texttt{fact}\ (3\cdot 2)\ (2-1)\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\quad \texttt{if}\ (1>0)\ \texttt{then}'\ \texttt{else}'\ /\ \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad \texttt{then}'\ /\ \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\quad \texttt{fact}\ (6\cdot 1)\ (1-1)\ /\ \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\quad \texttt{if}\ (0>0)\ \texttt{then}''\ \texttt{else}''\ /\ \delta_{\texttt{then}''}[6,0]\ \delta_{\texttt{else}''}[6]\ / \\
&\qquad\qquad \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\texttt{else}''\ /\ \delta_{\texttt{then}''}[6,0]\ \delta_{\texttt{else}''}[6]\ /\ \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\quad \texttt{return}\ 6\ /\ \delta_{\texttt{then}''}[6,0]\ \delta_{\texttt{else}''}[6]\ /\ \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}}
\end{aligned}$$

Handler `return` has no top-level definition, nor it is a library handler, and so evaluation stops.

Reachable code. While theoretically sound, our evaluation rule is not practical. Indeed, the list of top-level definitions keeps growing with each call, which is akin to a never-shrinking stack. We can remedy this defect by evicting the handlers that become unreachable:

$$\begin{aligned}
&\texttt{fact}\ 3\ 2\ /\ \delta_{\texttt{fact}} \longrightarrow \texttt{if}\ (2>0)\ \texttt{then else}\ /\ \delta_{\texttt{then}}[3,2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad \texttt{then}\ /\ \delta_{\texttt{then}}[3,2]\ /\ \delta_{\texttt{fact}} \longrightarrow \texttt{fact}\ (3\cdot 2)\ (2-1)\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad\quad \texttt{if}\ (1>0)\ \texttt{then}'\ \texttt{else}'\ /\ \delta_{\texttt{then}'}[6,1]\ \delta_{\texttt{else}'}[6]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad \texttt{then}'\ /\ \delta_{\texttt{then}'}[6,1]\ /\ \delta_{\texttt{fact}} \longrightarrow \texttt{fact}\ (6\cdot 1)\ (1-1)\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad\quad \texttt{if}\ (0>0)\ \texttt{then}''\ \texttt{else}''\ /\ \delta_{\texttt{then}''}[6,0]\ \delta_{\texttt{else}''}[6]\ /\ \delta_{\texttt{fact}} \longrightarrow \\
&\qquad\qquad\qquad\qquad \texttt{else}''\ /\ \delta_{\texttt{else}''}[6] \longrightarrow \texttt{return}\ 6
\end{aligned}$$

This notion of reachability will play an important role in computing the effects of expressions, and thus merits a proper definition. The set of *requested handlers* in an expression or a handler definition

```
let rec find_greater (n: int) (l: int list) : int
= match l with
  | [] -> raise Not_found
  | h :: t -> if n < h 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 on_nil
  / on_nil = not_found
    on_cons h t = less n h test
      / test b = if b then else
          / then = return h
            else = 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 _ = return true
    not_found = return false
```

Figure 2: The program from Fig. 1, in our language.

is computed by the following rules:

$$
\begin{aligned}
\mathrm{RH}(h\ \bar{s}\ \bar{g}) &\triangleq \{h, \bar{g}\} \\
\mathrm{RH}(e\ /\ \bar{\gamma}) &\triangleq \mathrm{RH}(e, \{\bar{\gamma}\}) \setminus \{\natural\bar{\gamma}\} \\
\mathrm{RH}(e,\ D) &\triangleq \text{the smallest set } H \text{ such that } \mathrm{RH}(e) \subseteq H \\
&\qquad \text{and } \forall \gamma \in D.\, \natural\gamma \in H \to \mathrm{RH}(\gamma) \subseteq H \\
\mathrm{RH}(h\ \bar{x}\ \bar{\varrho} = d) &\triangleq \mathrm{RH}(d) \setminus \{\natural\bar{\varrho}\}
\end{aligned}
$$

The binary form of the RH operator, applied to an expression e and a set of handler definitions D, *absorbs* these definitions into $\mathrm{RH}(e)$: for each requested handler h that is defined in D, we add the handlers requested by the definition of h and continue until reaching the fixed point. This process eventually terminates, since the working set is bounded above by $\mathrm{RH}(e) \cup \bigcup_{\gamma \in D} \mathrm{RH}(\gamma)$. It is easy to see that for any expression or handler definition X, $\mathrm{RH}(X) \subseteq \mathrm{FH}(X)$.

In an expression $e\ /\ \bar{\gamma}$, a definition of a handler h in $\bar{\gamma}$, as well as the handler h itself, is said to be *reachable*, if and only if h belongs to $\mathrm{RH}(e, \{\bar{\gamma}\})$. Unreachable handler definitions can be safely removed from the program, as they will never be used in an evaluation.

Going back to our example, consider an intermediate state $(\texttt{then}\ /\ \delta_{\texttt{then}}[3, 2]\ \delta_{\texttt{else}}[3]\ /\ \delta_{\texttt{fact}})$ in the first evaluation chain above. In the left-most call, the only requested handler is `then`. Absorbing the definitions of `then` and `else` brings in one new handler: $\mathrm{RH}(\texttt{then}, \{\delta_{\texttt{then}}, \delta_{\texttt{else}}\}) = \{\texttt{then}, \texttt{fact}\}$. Since `else` does not appear in this set, the definition $\delta_{\texttt{else}}[3]$ is unreachable and can be dropped.

Control structures. We conclude this section with another example showing how usual programming constructs — sequence, recursion, pattern matching and exception handling — are translated 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 handlers: `unList` to deconstruct a list, and `less` to compare two integers. Exceptions are translated using additional handler parameters: `find_greater` receives one handler parameter named `return` for the normal termination, and another, named `not_found`, for an exceptional termination. The caller, `check_greater`, defines a handler for the exceptional outcome, which corresponds to catching the exception.

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 prototype 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 first language extension consists in adding mutable variables. To avoid complexities related to state handling (which is tangential for this work), we 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 purpose, we redefine the syntax of expressions and handler prototypes as follows:

$$
\begin{array}{rcl}
\textit{expression} & ::= & \textit{handler}\ \ \textit{reference}{*}\ \ \textit{term}{*}\ \ \textit{handler}{*} \\
& | & \textit{expression}\ \texttt{/}\ \textit{definition}{+} \\
& | & \textit{expression}\ \texttt{/}\ \textit{allocation} \\
\textit{allocation} & ::= & \texttt{ref}\ \textit{reference}\ \texttt{=}\ \textit{term} \\
\textit{prototype} & ::= & \textit{handler}\ \ \textit{referenceParameter}{*}\ \ \textit{dataParameter}{*}\ \ \textit{prototype}{*} \\
\textit{referenceParameter} & ::= & \texttt{ref}\ \textit{reference} \\
\textit{dataParameter} & ::= & \textit{variable}
\end{array}
$$

Lexeme *reference* represents reference variables. We put references in a separate lexical category to simplify the presentation, so that references are not mixed with terms, and reference parameters with ordinary data parameters. In a more realistic language, data values would contain both mutable and immutable components, and separation between them would be expressed through types. We denote references with letters p, q, and r.

The elementary operations of dereference and assignment are made available via library handlers `access` and `assign`, respectively, which have the following annotated prototypes:

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

Library handler `access` takes a reference parameter `r` of arbitrary type α — in Section 4, we equip our language with an ML-style type system to accommodate such generic types. Then `access` takes a single handler parameter `return`, which allows `access` to pass control back to the caller and

has a single data parameter (i.e., the return value of `access`), named `v`, of the same type α. The precondition of `return`, which corresponds to the postcondition of `access`, specifies that the value passed to `return` is the one that is stored in `r`. Notice that dereference in preconditions is implicit.

Library handler `assign` takes a reference parameter `r` and a data 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. The pre-write annotations are part of the functional specification and will be formally introduced in Section 6.

The set of free references in an expression or handler definition is computed as follows:

$$\mathrm{FR}(h\ \bar{q}\ \bar{s}\ \bar{g}) \triangleq \{\bar{q}\} \qquad \mathrm{FR}(e\ \texttt{/ ref}\ p\ \texttt{=}\ t) \triangleq \mathrm{FR}(e) \setminus \{p\}$$
$$\mathrm{FR}(e\ \texttt{/}\ \bar{\gamma}) \triangleq \mathrm{FR}(e) \cup \mathrm{FR}(\bar{\gamma}) \qquad \mathrm{FR}(h\ \bar{r}\ \bar{x}\ \bar{\varrho}\ \texttt{=}\ e) \triangleq \mathrm{FR}(d) \setminus \{\bar{r}\}$$

Here and below, we reuse letters x, y, z for data parameters, and p, q, r for reference parameters.

Operators FV, FH, and RH ignore reference parameters in prototypes and reference arguments in handler calls. We add the following rules to their definitions to handle the allocation construction:

$$\mathrm{FV}(e\ \texttt{/ ref}\ p\ \texttt{=}\ t) \triangleq \mathrm{FV}(e) \cup \mathrm{FV}(t)$$
$$\mathrm{FH}(e\ \texttt{/ ref}\ p\ \texttt{=}\ t) \triangleq \mathrm{FH}(e)$$
$$\mathrm{RH}(e\ \texttt{/ ref}\ p\ \texttt{=}\ t) \triangleq \mathrm{RH}(e)$$

Operational semantics. The evaluation rule for top-level handler calls does not change much:

$$h\ \bar{q}\ \bar{s}\ \bar{g}\ \texttt{//}\ \Delta \longrightarrow d[\bar{r}, \bar{x}, \natural\bar{\varrho} \mapsto \bar{q}, \bar{s}, \bar{g}]\ \texttt{//}\ \Delta \quad \text{where } \Delta \text{ contains definition } h\ \bar{r}\ \bar{x}\ \bar{\varrho}\ \texttt{=}\ d$$

To avoid name collisions with the references allocated in Δ, we perform an alpha-conversion on the definition of h and give fresh names to all references allocated in d (just like we do for handlers).

Let us now see the evaluation rules for library handlers that manipulate references:

$$\texttt{access}\ q\ f\ \texttt{//}\ \Delta_1\ \texttt{/ ref}\ q\ \texttt{=}\ k\ \texttt{//}\ \Delta_2 \longrightarrow f\ k\ \texttt{//}\ \Delta_1\ \texttt{/ ref}\ q\ \texttt{=}\ k\ \texttt{//}\ \Delta_2$$
$$\texttt{assign}\ q\ k\ f\ \texttt{//}\ \Delta_1\ \texttt{/ ref}\ q\ \texttt{=}\ k_0\ \texttt{//}\ \Delta_2 \longrightarrow f\ \texttt{//}\ \Delta_1\ \texttt{/ ref}\ q\ \texttt{=}\ k\ \texttt{//}\ \Delta_2$$

The execution context acts as a mutable store. Expression `access` $q\ f$ // Δ retrieves the data value associated to q in Δ and passes it to f, without modifying Δ. Expression `assign` $q\ k\ f$ // Δ replaces the value associated to q in Δ with k and passes control to f. Remember that evaluated expressions cannot contain free variables, and hence the terms in allocations in Δ are necessarily constants.

Blending the store into program syntax may seem unorthodox. A more traditional approach would be to use a separate store and a library handler for allocation. The advantage of our formalism is that it allows for more natural type checking and effect computation rules for programs generated during evaluation. Indeed, by putting the allocated reference at a specific position among top-level definitions, we preserve the scoping relationship between handlers and references after allocation takes place. The importance of this relationship will become clear in Section 5.

Still, it is perhaps a better style to perform all allocations in the source code through a dedicated handler, and let the allocation construction appear in the code only through the calls to this handler during execution. Here is how we can define such a handler in our formalism:

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

Handler `alloc` takes a data 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, so we do

not need to state this explicitly in the prototype. The implementation of `alloc` simply returns a freshly allocated reference initialized to `v`. This does not create a "dangling pointer", since there is no stack shrinking on return. However, if we adopt an operational semantics that evicts unreachable handlers from the execution context Δ, we can additionally remove from Δ the allocations for references that are not used in the context anymore.

4 Typing

Our language can be seen as a fragment of an ML-like language. Indeed, handlers can be represented by functions whose result type is some fixed abstract type for which we provide no inhabitant or simply an empty type (also, nullary handlers must be given an extra unit parameter, to ensure the correct order of execution). Thus, the Hindley-Milner type system can be adapted to our language, with the usual perquisite of effective type inference. Let us spell out the details of this type system.

We start with another syntax extension — type annotations for data and reference parameters:

$$\begin{aligned} \mathit{dataParameter} &::= \mathit{variable}\ \texttt{:}\ \mathit{type} \\ \mathit{referenceParameter} &::= \texttt{ref}\ \mathit{reference}\ \texttt{:}\ \mathit{type} \\ \mathit{type} &::= \mathit{typeSymbol}\ \ \mathit{type}* \\ &\mid \ \mathit{typeVariable} \end{aligned}$$

Types characterize values passed via data arguments and stored in memory. In our language, handlers are not data, and so we do not consider the function type (though it is possible to add it and even to have some form of conversion between pure handlers and these first-class values). Data types are described by non-terminal *type* and denoted with letters τ and ξ. Type symbols of a fixed arity are described by non-terminal *typeSymbol* and denoted with letter T. Type variables are described by non-terminal *typeVariable* and denoted with letters α and β. Type substitutions instantiate type variables with types and are denoted with letter θ. *Ground types* do not contain type variables.

Typing judgements are made in the context Γ, a list of data parameters, reference parameters, handler parameters (prototypes), and handler definitions. A type variable α is said to be *fixed* in Γ if it occurs inside a data, reference or handler parameter listed in Γ. Handler definitions are implicitly generalized and thus do not fix type variables. The order of elements in Γ reflects the scoping relationship between variables and handlers: this comes in handy in the rule for type generalization.

We infer typing judgements for terms ($\Gamma \vdash s : \tau$), references ($\Gamma \vdash \texttt{ref}\ r : \tau$), expressions ($\Gamma \vdash_{\mathrm{wt}} e$), handler prototypes ($\Gamma \vdash_{\mathrm{wt}} \varrho$), handler definitions ($\Gamma \vdash_{\mathrm{wt}} \gamma$), and so-called *specialized prototypes* written inside angle brackets ($\Gamma \vdash_{\mathrm{wt}} \langle\varrho\rangle$). Expressions, handler prototypes, and handler definitions do not have proper types, thus the typing judgement for these objects simply means being well-typed in the given context. The typing rules are given in Figure 3. We assume that no variable or handler in the program is bound twice.

The typing rules for terms and references — T-Const, T-Var, and T-Ref — are self-explanatory. Notice that constants may have multiple types including non-ground ones: for example, the empty list constant $\Diamond$ can assume any type of the form $\texttt{list}\ \tau$.

Rule T-Proto says that a handler prototype is well-typed if its handler parameters are well-typed in the scope of its data and reference parameters. Right now, handler prototypes do not contain freely occurring elements to type-check, and thus any handler prototype is well-typed. In the following sections, when we start equipping prototypes with annotations, we will extend the T-Proto rule with well-typedness conditions for these annotations. Rule T-Defn says that a handler definition is well-typed if the handler prototype on the left-hand side is well-typed and the definition body is well-typed in the scope of the formal parameters.

The next three rules — T-Mono, T-Gen, and T-Lib — define the appropriate type instances of called handlers. Rule T-Mono applies to the handlers that appear as non-defined prototypes in Γ. Such prototypes represent formal handler parameters (see the second premise of T-Defn) and are

$$\frac{c \text{ has type } \tau}{\Gamma \vdash c : \tau}\ \text{(T-Const)} \qquad \frac{}{\Gamma_1,\, x : \tau,\, \Gamma_2 \vdash x : \tau}\ \text{(T-Var)}$$

$$\frac{\Gamma, \bar{r}, \bar{x} \vdash_{\text{wt}} \varrho_1 \quad \ldots \quad \Gamma, \bar{r}, \bar{x} \vdash_{\text{wt}} \varrho_n}{\Gamma \vdash_{\text{wt}} h\ \bar{r}\ \bar{x}\ \varrho_1 \ldots \varrho_n}\ \text{(T-Proto)} \qquad \frac{}{\Gamma_1,\, \texttt{ref}\ r : \tau,\, \Gamma_2 \vdash \texttt{ref}\ r : \tau}\ \text{(T-Ref)}$$

$$\frac{\Gamma \vdash_{\text{wt}} h\ \bar{r}\ \bar{x}\ \bar{\varrho} \qquad \Gamma, \bar{r}, \bar{x}, \bar{\varrho} \vdash_{\text{wt}} d}{\Gamma \vdash_{\text{wt}} h\ \bar{r}\ \bar{x}\ \bar{\varrho} = d}\ \text{(T-Defn)} \qquad \frac{}{\Gamma_1,\, \varrho,\, \Gamma_2 \vdash_{\text{wt}} \langle \varrho \rangle}\ \text{(T-Mono)}$$

$$\frac{\text{for every } \alpha \text{ fixed in } \Gamma_1,\ \theta(\alpha) = \alpha}{\Gamma_1,\, \varrho = d,\, \Gamma_2 \vdash_{\text{wt}} \langle \varrho\theta \rangle}\ \text{(T-Gen)} \qquad \frac{\varrho \text{ is a library handler}}{\Gamma \vdash_{\text{wt}} \langle \varrho\theta \rangle}\ \text{(T-Lib)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash s_m : \tau_m \\ \Gamma \vdash \texttt{ref}\ q_1 : \xi_1 \qquad \ldots \qquad \Gamma \vdash \texttt{ref}\ q_k : \xi_k \\ \Gamma \vdash_{\text{wt}} \text{Impl}(\varrho_1, g_1) \quad \ldots \quad \Gamma \vdash_{\text{wt}} \text{Impl}(\varrho_n, g_n) \\ \Gamma \vdash_{\text{wt}} \langle h\ (\texttt{ref}\ r_1 : \xi_1) \ldots (\texttt{ref}\ r_k : \xi_k)\ (x_1 : \tau_1) \ldots (x_m : \tau_m)\ \varrho_1 \ldots \varrho_n \rangle\end{array}}{\Gamma \vdash_{\text{wt}} h\ q_1 \ldots q_k\ s_1 \ldots s_m\ g_1 \ldots g_n}\ \text{(T-Call)}$$

$$\frac{\Gamma, \gamma_1, \ldots, \gamma_n \vdash_{\text{wt}} \gamma_1 \quad \ldots \quad \Gamma, \gamma_1, \ldots, \gamma_n \vdash_{\text{wt}} \gamma_n \qquad \Gamma, \gamma_1, \ldots, \gamma_n \vdash_{\text{wt}} e}{\Gamma \vdash_{\text{wt}} e\ /\ \gamma_1\ \ldots\ \gamma_n}\ \text{(T-Rec)}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma, \texttt{ref}\ p : \tau \vdash_{\text{wt}} e}{\Gamma \vdash_{\text{wt}} e\ /\ \texttt{ref}\ p = t}\ \text{(T-Alloc)}$$

Figure 3: Typing rules.

not generalized, in agreement with the rules of prenex polymorphism. This is why the specialized prototype $\langle \varrho \rangle$ in the conclusion of T-Mono is actually not type-instantiated. Rule T-Gen implements let-polymorphism: every handler introduced by a handler definition is generalized in all type variables of its prototype that are not fixed in the typing context of that definition. Rule T-Lib applies to library handlers, which are always fully generalized. We do not put the original prototypes of library handlers in the typing context: just like with the operational semantics, these handlers form an axiomatic basis of our type system.

Typing rule T-Call verifies that each actual argument in a handler call has the type prescribed by the corresponding parameter in an appropriate specialized prototype of the called handler. We write $\text{Impl}(\varrho, g)$ to denote the handler definition $\varrho = g\ \bar{p}\ \bar{y}\ \bar{f}$, where $\bar{p}$, $\bar{y}$, and $\bar{f}$ are the names of the reference, data, and handler parameters in ϱ, respectively. Well-typedness of this handler definition means that g can be used as an implementation of ϱ.

Finally, rules T-Rec and T-Alloc treat handler definitions and reference allocations.

5 Alias Safety

It is possible to refine our type system in a way that ensures that well-typed programs are alias-free. Before presenting the modified rules, let us see how an alias — a possibility to access the same memory location under two different names — may appear in our programs. Let us consider two typical cases:

```
f r r / ref r = 0
      / f (ref p) (ref q) = ...
      // Δ
```

```
g r / g (ref p) = ...
    / ref r = 0
    // Δ
```

In the program on the left, the call `f r r` will evaluate to the body of `f`, where both `p` and `q` are instantiated with `r`, creating an alias. In the program on the right, the call `g r` will evaluate 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`.

To exclude such programs we only need to change the T-Call rule:

$$\frac{\begin{array}{l}\Gamma_1, \Gamma_2 \vdash s_1 : \tau_1 \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash s_m : \tau_m \\ \quad \Gamma_2 \vdash \texttt{ref}\ q_1 : \xi_1 \quad \ldots \quad \Gamma_2 \vdash \texttt{ref}\ q_k : \xi_k \qquad q_1, \ldots, q_k \text{ are distinct} \\ \Gamma_1, \Gamma_2 \vdash_{\text{wt}} \text{Impl}(\varrho_1, g_1) \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash_{\text{wt}} \text{Impl}(\varrho_n, g_n) \\ \quad \Gamma_1 \vdash_{\text{wt}} \langle h\ (\texttt{ref}\ r_1 : \xi_1) \ldots (\texttt{ref}\ r_k : \xi_k)\ (x_1 : \tau_1) \ldots (x_m : \tau_m)\ \varrho_1 \ldots \varrho_n \rangle\end{array}}{\Gamma_1, \Gamma_2 \vdash_{\text{wt}} h\ q_1 \ldots q_k\ s_1 \ldots s_m\ g_1 \ldots g_n}\ \text{(T-Call)}$$

Two changes distinguish this rule from the one in Figure 3. They correspond to the two scenarios of alias creation shown above. First, all reference arguments must be pairwise distinct: no reference can be passed to a handler twice. Second, all reference arguments are type-checked under a restricted context: any reference accessible to h (and thus declared in Γ_1) cannot be passed as an argument to h.

From now on, we assume that all programs we consider are well-typed and alias-safe, which means that all handler calls in them are checked using the updated T-Call rule.

As a small exercise, consider the following handler definitions:

```
id1 (x: int) {}                    id2 (ref x: int) {}
  return (y: int) { y = x }          return (ref y: int) { y = x }
= return x                         = return x
```

Are these handlers alias-safe? Are these definitions well-typed?

6 Effect Computation

The next element of specification we add to our language are the *pre-write annotations* that list the references that are potentially modified during the execution of a parent handler before it calls one of its handler parameters or locally defined sub-handlers. We extend the syntax of handler prototypes:

$$\begin{array}{lcl} \textit{prototype} & ::= & \textit{handler}\ \ \textit{pre-write}\ \ \textit{referenceParameter}*\ \ \textit{dataParameter}*\ \ \textit{prototype}* \\ \textit{pre-write} & ::= & [\ \ \textit{reference}*\ \ ] \end{array}$$

In our examples, when we omit the pre-write annotation in a prototype, we mean that the annotation is an empty list. We assume that all references in the pre-write annotation are distinct.

Handler prototypes may now contain free references, which must be type-checked. We update the rules for handler prototypes, definitions, and calls as follows:

$$\frac{\Gamma \vdash_{\text{wt}} h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho} \qquad \Gamma, \bar{r}, \bar{x}, \bar{\varrho} \vdash_{\text{wt}} d}{\Gamma \vdash_{\text{wt}} h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho} = d}\ \text{(T-Defn)}$$

$$\frac{\Gamma \vdash \texttt{ref}\ p_1 : \tau_1 \quad \ldots \quad \Gamma \vdash \texttt{ref}\ p_m : \tau_m \qquad \Gamma, \bar{r}, \bar{x} \vdash_{\text{wt}} \varrho_1 \quad \ldots \quad \Gamma, \bar{r}, \bar{x} \vdash_{\text{wt}} \varrho_n}{\Gamma \vdash_{\text{wt}} h\ [p_1 \ldots p_m]\ \bar{r}\ \bar{x}\ \varrho_1 \ldots \varrho_n}\ \text{(T-Proto)}$$

$$\frac{\begin{array}{l}\Gamma_1, \Gamma_2 \vdash s_1 : \tau_1 \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash s_m : \tau_m \\ \quad \Gamma_2 \vdash \texttt{ref}\ q_1 : \xi_1 \quad \ldots \quad \Gamma_2 \vdash \texttt{ref}\ q_k : \xi_k \qquad q_1, \ldots, q_k \text{ are distinct} \\ \Gamma_1, \Gamma_2 \vdash_{\text{wt}} \text{Impl}(\varrho_1[\bar{r} \mapsto \bar{q}], g_1) \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash_{\text{wt}} \text{Impl}(\varrho_n[\bar{r} \mapsto \bar{q}], g_n) \\ \Gamma_1 \vdash_{\text{wt}} \langle h\ [\bar{p}]\ (\texttt{ref}\ r_1 : \xi_1) \ldots (\texttt{ref}\ r_k : \xi_k)\ (x_1 : \tau_1) \ldots (x_m : \tau_m)\ \varrho_1 \ldots \varrho_n \rangle\end{array}}{\Gamma_1, \Gamma_2 \vdash_{\text{wt}} h\ q_1 \ldots q_k\ s_1 \ldots s_m\ g_1 \ldots g_n}\ \text{(T-Call)}$$

Rule T-Call ignores the pre-write annotation $[\bar{p}]$ of the called handler, but instantiates the handler parameters $\varrho_1, \ldots, \varrho_n$ to ensure that the definitions $\text{Impl}(\varrho_i[\bar{r} \mapsto \bar{q}], g_i)$ are well-typed.

$$
\begin{aligned}
\mathcal{E}(h\ \bar{q}\ \bar{s}\ g_1\ \ldots\ g_n) &\triangleq \{(p, g_i) \mid p \text{ is a pre-write of } \varrho_i[\bar{r} \mapsto \bar{q}]\} \\
&\quad \text{where } \langle h\ [\cdot]\ \bar{r}\ \bar{x}\ \varrho_1 \ldots \varrho_n\rangle \text{ is the specialized prototype of } h \\
&\quad\ \text{and for each } \varrho_i \text{ and } g_i,\ \mathcal{E}(\mathrm{Impl}(\varrho_i[\bar{r} \mapsto \bar{q}], g_i)) = \varnothing \\
\mathcal{E}(e\ /\ \bar{\gamma}) &\triangleq \{(r, f) \in \mathcal{E}(e, \{\bar{\gamma}\}) \mid f \notin \{\natural\bar{\gamma}\}\} \\
&\quad \text{where for each } g\ [\bar{p}]\ \ldots \text{ in } \bar{\gamma},\ \mathcal{E}(e, \{\bar{\gamma}\})^{-1}(g) \subseteq \{\bar{p}\} \\
\mathcal{E}(e,\ D) &\triangleq \text{the smallest set } W \text{ such that } \mathcal{E}(e) \subseteq W \text{ and} \\
&\quad \forall \gamma \in D.\, \natural\gamma \in \mathrm{RH}(e, D) \rightarrow (W^{-1}(\natural\gamma) \times \mathrm{RH}(\gamma)) \cup \mathcal{E}(\gamma) \subseteq W \\
\mathcal{E}(e\ /\ \texttt{ref}\ p = t) &\triangleq \{(r, f) \in \mathcal{E}(e) \mid r \neq p\} \\
\mathcal{E}(h\ [\cdot]\ \bar{r}\ \bar{x}\ \bar{\varrho} = d) &\triangleq \{(q, f) \in \mathcal{E}(d) \mid q \notin \{\bar{r}\} \wedge f \notin \{\natural\bar{\varrho}\}\} \\
&\quad \text{where for each } g\ [\bar{p}]\ \ldots \text{ in } \bar{\varrho},\ \mathcal{E}(d)^{-1}(g) \subseteq \{\bar{p}\}
\end{aligned}
$$

Figure 4: Effect computation.

The formal definition of a free reference needs to be updated accordingly:

$$
\begin{aligned}
\mathrm{FR}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho} = d) &\triangleq \mathrm{FR}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho}) \cup (\mathrm{FR}(d) \setminus \{\bar{r}\}) \\
\mathrm{FR}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho}) &\triangleq \{\bar{p}\} \cup (\mathrm{FR}(\bar{\varrho}) \setminus \{\bar{r}\})
\end{aligned}
$$

As in earlier definitions, we write $\mathrm{FR}(\bar{\varrho})$ to denote the union of the sets of free references in each prototype in $\bar{\varrho}$.

Pre-writes can be effectively inferred for defined handlers and for the formal handler parameters of defined handlers. Thus it is possible to compare the inferred effect with the user-supplied annotations to check that the latter are correct — or to allow the user to omit writing annotations where they can be computed by the tool.

Given an expression or a handler definition X, the *effect operator* $\mathcal{E}(X)$ produces the set of all pairs (r, h), where handler h belongs to $\mathrm{RH}(X)$ and reference r is potentially modified during the execution of X before a call to h. We write $\mathcal{E}(X)^{-1}(h)$ to denote the pre-write set $\{r \mid (r, h) \in \mathcal{E}(X)\}$.

The effect operator is defined in Figure 4 in a similar manner to the RH operator, using definition absorption. This requires access to the specialized prototypes of the called handlers, and thus the rules for $\mathcal{E}$ are implicitly parametrized by the typing context Γ as used in the typing rules.

The effect of a handler call is derived from the instantiation of the called handler's prototype: all references in the pre-write annotation of each formal handler parameter ϱ_i, become — after we substitute the actual reference arguments $\bar{q}$ for the formal reference parameters $\bar{r}$ — the pre-writes for the corresponding actual handler argument g_i. For example, $\mathcal{E}(\texttt{assign q k f}) = \{(\texttt{q}, \texttt{f})\}$. Indeed, `assign` has a single handler parameter, whose only pre-write is the reference parameter of `assign`.

The second condition requires each handler argument g_i to conform to the specification of the corresponding formal handler parameter ϱ_i. Specifically, this means that g_i does not perform any reference modifications that are not authorized by the pre-write annotations of the outcomes of ϱ_i. We check this by verifying that $\mathcal{E}$ is well-defined on the definition $\mathrm{Impl}(\varrho_i[\bar{r} \mapsto \bar{q}], g_i)$, in which case the effect is guaranteed to be empty.

The effect of an expression with local definitions $e\ /\ \bar{\gamma}$ is the combined effect $\mathcal{E}(e, \{\bar{\gamma}\})$, from which we remove all pre-writes for the handlers defined in $\bar{\gamma}$, since they are bound in the expression. Additionally, we check that the pre-writes computed for each handler g defined in $\bar{\gamma}$ are included in the pre-write annotation of g.

The combined effect $\mathcal{E}(e, D)$ is computed, as was the case for the RH operator, by a fixed point computation. We start with the effect computed for e and for each reachable definition γ in set D,

```
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 on_nil
  / on_nil = 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 [?] (i: int) {} ret_add [r] {}
    = assign r (i + !r) ret_add
  / out [?] {} = return !r
  / ref r = 0
```

Figure 5: Illicit hidden side effects.

add the effect $\mathcal{E}(\gamma)$ of the definition itself along with the currently known pre-writes for $\natural\gamma$ attached to every handler in $\mathrm{RH}(\gamma)$ (since these handlers would be called by $\natural\gamma$, hence after those pre-writes). This may add new pre-writes for the handlers defined in D, and so we must continue until reaching the fixed point.

The effect of an expression with a reference allocation e `/ ref` p `=` t is simply the effect of e with the bound reference p filtered out.

Finally, the effect of a handler definition is the effect of the definition body, from which we remove all reference and handler parameters. Additionally, we check that the pre-writes computed for each formal handler parameter g in $\bar{\varrho}$ are included in the pre-write annotation of g.

In what follows, we assume that all programs we consider have a well-defined effect.

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 5, 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. For the sake of clarity, we use the dereference operator `(!)` in our terms rather than the `access` handler; this is sound, since dereference does not have any preconditions or side effects.

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 prototype of `assign`, the singleton $\{(\texttt{r}, \texttt{ret\_add})\}$. The effect of the definition of `add` is empty, since `ret_add` is bound in the definition. However, the rule for $\mathcal{E}$ on handler definitions will check that `r` is properly indicated as a pre-write of `ret_add` in the prototype of `add`. This pre-write is the write effect of `add` when it returns via `ret_add`.

But what are the pre-writes of handlers `add` and `out`: what are the writes that may be performed during an execution of `sum` before `add` or `out` are called? The correct answer for both handlers is `[r]`, since `iter` may possibly call `add` several times (and so `r` will have changed before the second call) before calling `out`. Unfortunately, our effect system cannot infer these pre-writes. Indeed, `iter` does not announce any pre-writes for its outcomes `fn` and `return`, and thus the effect of the handler call `iter l add out`, if defined, would be an empty set. Absorbing the definitions of `add` or `out` does not change anything, since the effect of these definitions is also null. In absence of effect polymorphism,

the only sound choice in this situation is to reject the problematic code, and this is indeed what happens here — let us see why.

Let $\varrho_{\texttt{fn}}$ be the prototype `fn (i: int) {} ret_fn {}` of the first handler parameter of `iter`. Computing $\mathcal{E}$(`iter l add out`) requires the effect of the handler definition Impl($\varrho_{\texttt{fn}}$, `add`) to be well-defined and null. This definition is as follows (we fill in the empty pre-write annotations):

```
fn [] (i: int) {}
  ret_fn [] {}
= add i ret_fn
```

The effect of the body, `add i ret_fn`, is the singleton set {(`r`, `ret_fn`)}, according to the pre-write annotation of `ret_add` in the prototype of `add`. However, `r` does not appear in the empty pre-write annotation of `ret_fn` in the prototype of `fn`, and thus the effect of Impl($\varrho_{\texttt{fn}}$, `add`) is undefined. Informally speaking, we cannot pass `add` as an argument to `iter`, because `add` modifies the program state (as reflected in the non-empty pre-write of `ret_add`), whereas `iter` expects a handler without side effects (as reflected in the empty pre-write of `ret_fn`).

To make this code accepted, the potential side effects of handler parameter `fn` in the prototype of `iter` must be indicated in the pre-write annotation of `ret_fn`. This requires the concerned references to be accessible to `iter`. One way of doing it is to move the definition of `iter` into the scope of reference `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 corresponding prototype would look like: cite?

```
iter [] (ref 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.

7 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 annotation of a handler h indicates 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.

We say that a well-typed expression or handler definition is in *canonical form*, if in every handler call $h\,\bar{q}\,\bar{s}\,\bar{g}$, where the specialized prototype of h is $\langle h\,[\bar{p}]\,\bar{r}\,\bar{x}\,\bar{\varrho}\rangle$, each handler argument g_i has the same prototype modulo alpha-conversion as the instantiated prototype of the corresponding handler parameter $\varrho_i[\bar{r} \mapsto \bar{q}]$. (Later, when we introduce preconditions, we will also need to instantiate the data parameters: $\varrho_i[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}]$.)

It is easy to convert an expression to a canonical form. Indeed, we only need to hide the actual handler arguments g_i under the "refinement definitions", that is, replace each call $h\,\bar{q}\,\bar{s}\,g_1 \ldots g_n$ with $h\,\bar{q}\,\bar{s}\,f_1 \ldots f_n$ / Impl($\varrho_1[\bar{r} \mapsto \bar{q}], g_1) \ldots$ Impl($\varrho_n[\bar{r} \mapsto \bar{q}], g_n$), where $f_1, \ldots, f_n$ are the names of the handlers described by prototypes $\varrho_1, \ldots, \varrho_n$, respectively. Then the added definitions must be also converted to a canonical form. This process eventually terminates, since each sub-definition defines a handler of a lower rank.

```
iter [] (ref q: int) (l: list int) {}
  fn [q] (i: int) {}
    ret_fn [q] {}
  return [q] {}
= unList l on_cons on_nil
  / on_nil [] = return
    on_cons [] h t = fn 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
  / ref r = 0
```

```
iter (q: int) (l: list int) {}
  fn (q': int) (i: int) {}
    ret_fn (q'': int) {}
  return (q': int) {}
= unList l on_cons on_nil
  / on_nil = 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 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 6: Program with side effects (left) and its stateless encoding (right)

Operator Pure, defined below, applies to expressions and handler definitions in canonical form, as well as to handler prototypes. For simplicity, we reuse reference names as data variables in the resulting expression.

$$\begin{aligned}
\text{Pure}(h\ \bar{q}\ \bar{s}\ \bar{g}) &\triangleq h\ \bar{p}\bar{q}\bar{s}\ \bar{g} \\
&\quad \text{where } [\bar{p}] \text{ is the pre-write of } h \\
\text{Pure}(e\ \texttt{/}\ \gamma_1 \ldots \gamma_n) &\triangleq \text{Pure}(e)\ \texttt{/}\ \text{Pure}(\gamma_1)\ \ldots\ \text{Pure}(\gamma_n) \\
\text{Pure}(e\ \texttt{/ ref}\ p\ \texttt{=}\ t) &\triangleq \text{Pure}(e)[p \mapsto t] \\
\text{Pure}(\varrho\ \texttt{=}\ d) &\triangleq \text{Pure}(\varrho)\ \texttt{=}\ \text{Pure}(d) \\
\text{Pure}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \varrho_1 \ldots \varrho_n) &\triangleq h\ \bar{p}\bar{r}\bar{x}\ \text{Pure}(\varrho_1)\ \ldots\ \text{Pure}(\varrho_n)
\end{aligned}$$

Informally speaking, we simply treat reference parameters and pre-written references as additional data parameters. The canonical form is required to ensure that all handler arguments still have the same arity as the handler parameters they are passed for. On the right-hand side of the last rule, we assume that the keyword `ref` is dropped from $\bar{r}$ and appropriate types are filled in for $\bar{p}$.

Obviously, we must also update the semantics of library handlers that manipulate the state. First of all, let us see the converted prototypes of `access` and `assign`:

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

Now, the evaluation rules for the converted handlers can be simply obtained by applying Pure to the both sides of the original rules:

$$\begin{aligned}
\texttt{access}\ k\ f\ \texttt{//}\ \Delta &\longrightarrow f\ k\ \texttt{//}\ \Delta \\
\texttt{assign}\ k_0\ k\ f\ \texttt{//}\ \Delta &\longrightarrow f\ k\ \texttt{//}\ \Delta
\end{aligned}$$

In Figure 6, we come back to the example from Figure 5, in which we add a reference parameter to `iter`, as suggested at the end of Section 6, to ensure that the effects are computed correctly. On the left side of Figure 6 we show the stateful definitions of `iter` and `sum`, and on the right side, their stateless encoding. The dereference operator `(!)` becomes an identity (just like `access`), and we simply remove it.

8 Partial Correctness

We endow handler prototypes with an optional functional specification that describes the expected program state at the beginning of the handler execution:

$$\begin{array}{rcl} \textit{prototype} & ::= & \textit{handler}\ \ \textit{pre-write}\ \ \textit{referenceParameter}*\ \ \textit{dataParameter}* \\ & & \qquad\qquad \textit{precondition}?\ \ \textit{prototype}* \\ \textit{precondition} & ::= & \texttt{\{}\ \ \textit{formula}\ \ \texttt{\}} \end{array}$$

The non-terminal *formula* represents first-order formulas in any suitable syntax. We denote formulas with letters φ and ψ. An empty precondition `{}` — not to be confused with an omitted precondition — is an abbreviation for `{true}`. Variables and references may freely occur in formulas.

Our method of VC generation requires that all handlers without a definition (i.e., formal handler parameters and library handlers) carry an explicitly written precondition. As for defined handlers, given an expression $e \mathbin{/} \gamma_1 \ldots \gamma_n$, we require the handler defined in γ_i to have a precondition only if it is called from some γ_j, where $j \geqslant i$. A handler that is only called from the code to the left of its definition does not need to have a precondition.

Here are the updated rules to compute the free variables of handler definitions and prototypes:

$$\begin{array}{rcl} \mathrm{FV}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho} = d) & \triangleq & \mathrm{FV}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho}) \cup (\mathrm{FV}(d) \setminus \{\bar{x}\}) \\ \mathrm{FV}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho} = d) & \triangleq & \mathrm{FV}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho}) \cup (\mathrm{FV}(d) \setminus \{\bar{x}\}) \\ \mathrm{FV}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho}) & \triangleq & (\mathrm{FV}(\varphi) \cup \mathrm{FV}(\bar{\varrho})) \setminus \{\bar{x}\} \\ \mathrm{FV}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho}) & \triangleq & \mathrm{FV}(\bar{\varrho}) \setminus \{\bar{x}\} \end{array}$$

We also need to complement the definition of the FR operator:

$$\begin{array}{rcl} \mathrm{FR}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho} = d) & \triangleq & \mathrm{FR}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho}) \cup (\mathrm{FR}(d) \setminus \{\bar{r}\}) \\ \mathrm{FR}(h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho}) & \triangleq & \{\bar{p}\} \cup ((\mathrm{FR}(\varphi) \cup \mathrm{FR}(\bar{\varrho})) \setminus \{\bar{r}\}) \end{array}$$

Let us show the updated typing rules. In the rules T-Defn and T-Call, $\{\varphi\}?$ denotes a possibly absent precondition. Notice that rule T-Proto requires the precondition to be always present: this ensures that all handler parameters have an explicit precondition. Since preconditions can be omitted in the handler definitions, the T-Defn rule allows us to supply a phony precondition (e.g., `true`) to typecheck the prototype on the left-hand side of the definition.

$$\frac{\Gamma \vdash_{\mathrm{wt}} h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho} \qquad \Gamma, \bar{r}, \bar{x}, \bar{\varrho} \vdash_{\mathrm{wt}} d}{\Gamma \vdash_{\mathrm{wt}} h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}?\ \bar{\varrho} = d}\ (\text{T-Defn})$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathtt{ref}\ p_1 : \tau_1 \quad \ldots \quad \Gamma \vdash \mathtt{ref}\ p_m : \tau_m \\ \Gamma, \bar{r}, \bar{x} \vdash_{\mathrm{wt}} \varrho_1 \quad \ldots \quad \Gamma, \bar{r}, \bar{x} \vdash_{\mathrm{wt}} \varrho_n \qquad \Gamma, \bar{r}, \bar{x} \vdash_{\mathrm{wt}} \varphi \end{array}}{\Gamma \vdash_{\mathrm{wt}} h\ [p_1 \ldots p_m]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \varrho_1 \ldots \varrho_n}\ (\text{T-Proto})$$

$$\frac{\begin{array}{c} \Gamma_1, \Gamma_2 \vdash s_1 : \tau_1 \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash s_m : \tau_m \\ \Gamma_2 \vdash \mathtt{ref}\ q_1 : \xi_1 \quad \ldots \quad \Gamma_2 \vdash \mathtt{ref}\ q_k : \xi_k \qquad q_1, \ldots, q_k \text{ are distinct} \\ \Gamma_1, \Gamma_2 \vdash_{\mathrm{wt}} \mathrm{Impl}(\varrho_1[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_1) \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash_{\mathrm{wt}} \mathrm{Impl}(\varrho_n[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_n) \\ \Gamma_1 \vdash_{\mathrm{wt}} \langle h\ [\bar{p}]\ (\mathtt{ref}\ r_1 : \xi_1) \ldots (\mathtt{ref}\ r_k : \xi_k)\ (x_1 : \tau_1) \ldots (x_m : \tau_m)\ \{\varphi\}?\ \varrho_1 \ldots \varrho_n \rangle \end{array}}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{wt}} h\ q_1 \ldots q_k\ s_1 \ldots s_m\ g_1 \ldots g_n}\ (\text{T-Call})$$

Since prototypes can now have free variables, we need to instantiate the handler parameters ϱ_i in the T-Call rule, to ensure that the definitions $\mathrm{Impl}(\varrho_i[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_i)$ are well-typed.

$$
\begin{aligned}
\mathrm{VC}(h\ \bar{q}\ \bar{s}\ g_1 \ldots g_n) &\triangleq \Phi \blacktriangleleft \mathrm{Impl}(\varrho_1[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_1) \blacktriangleleft \cdots \blacktriangleleft \mathrm{Impl}(\varrho_n[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_n) \\
&\quad \text{where } \langle h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}^?\ \varrho_1 \ldots \varrho_n \rangle \text{ is the specialized prototype of } h \\
&\quad \text{and } \Phi = \begin{cases} \textbf{assert}\ \varphi[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}] & \text{if precondition } \varphi \text{ is given,} \\ \textbf{apply}\ h\ \bar{p}\bar{q}\bar{s} & \text{otherwise} \end{cases} \\
\mathrm{VC}(e\ /\ \gamma_1 \ldots \gamma_n) &\triangleq \mathrm{VC}(e) \blacktriangleleft \gamma_1 \blacktriangleleft \cdots \blacktriangleleft \gamma_n \\
\mathrm{VC}(e\ /\ \texttt{ref}\ p = t) &\triangleq \mathrm{VC}(e)[p \mapsto t] \\
\\
\Phi \blacktriangleleft h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \{\varphi\}\ \bar{\varrho} = d &\triangleq \Phi\ \textbf{and forall}\ \bar{p}\bar{r}\bar{x}\ (\varphi\ \textbf{implies}\ \mathrm{VC}(d)) \\
\Phi \blacktriangleleft h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{\varrho} = d &\triangleq \Phi\ \textbf{with}\ h\ \bar{p}\bar{r}\bar{x} = \mathrm{VC}(d)
\end{aligned}
$$

Figure 7: Verification condition generation.

Verification conditions are produced in a restricted fragment of the second-order language, defined by the following grammar:

$$
\begin{array}{rcl}
vc & ::= & \textbf{assert}\ formula \\
& | & formula\ \textbf{implies}\ vc \\
& | & \textbf{forall}\ binder{*}\ vc \\
& | & \textbf{apply}\ handler\ vcTerm{*} \\
& | & vc\ \textbf{with}\ handler\ binder{*}\ =\ vc \\
& | & vc\ \textbf{and}\ vc
\end{array}
\qquad
\begin{array}{rcl}
binder & ::= & variable\ :\ type \\
& | & reference\ :\ type \\
\\
vcTerm & ::= & term \mid reference
\end{array}
$$

In this syntax, references are treated as ordinary first-order variables. Handler names act as second-order variables assigned to predicates: verification conditions, parametrized by a number of first-order values. Verification conditions obey the usual rules of lexical scoping, both for first- and second-order variables. In particular, predicate definitions introduced by the **with** construction are not recursive: each second-order variable applied inside such a definition must be bound in the outer context.

We assume that predicate definitions under **with** are type-generalized as much as possible: that is, to the extent that the free occurrences of first- and second-order variables in the definition body do not prevent generalization. We denote verification conditions with letters Φ and Ψ.

Semantically, verification conditions are just logical formulas: **and**, **implies**, and **forall** have the usual meaning; **assert** and **apply** mark an occurrence of a first-order formula or a second-order variable application, respectively; **with** is a second-order let-binder. Notice that all occurrences of *vc* are positive: only first-order formulas may appear in the antecedent of an implication.

The rules for verification condition generation are given in Figure 7. As we need to know the specialized prototypes of the called handlers as well as the types of variables and references, these rules are implicitly parametrized by the typing context Γ. We assume that no variable, reference or handler in the source code is bound twice.

Handler definitions are joined to the verification condition of the subordinate expression using left-associative binary operator $\blacktriangleleft$. Verification of handler arguments in handler calls is expressed as verification of the virtual definitions $\mathrm{Impl}(\cdot, \cdot)$, just as in the type-checking rules. Whenever we refer to the specialized prototype of a called handler, we assume that all formal parameters in that prototype are given fresh names by alpha-conversion, in order to avoid name collisions with the outer context of the verification condition.

Verification condition formulas produced by the rules in Figure 7 closely follow the structure of the verified code. It is easy to see that these formulas are well-formed and well-typed. First of all, whenever a first-order name (a variable or a reference) is bound in an expression, it is bound

in its verification condition. Also, all external references in the pre-write annotations are treated as additional parameters, just as in the stateless encoding in Section 7.

Handlers with preconditions — which includes all handler parameters and library handlers — do not translate into predicate variables. Instead, a definition of such a handler produces an additional proof obligation: for all values of formal parameters and pre-written external references, the precondition implies the verification condition of the body. A call to such a handler is then translated as a simple assertion of the correspondingly instantiated precondition.

Handlers without a precondition — which must have a definition and cannot be called from their own body or from handlers defined after them — are translated into predicate variables. A call to such a handler is translated as a predicate application, and the "only-call-me-from-the-left" restriction ensures that this application lies in the scope of the corresponding predicate definition.

We extend the VC operator to definitions: $\mathrm{VC}(\gamma_1, \ldots, \gamma_n) \triangleq$ **assert** `true` $\blacktriangleleft \gamma_1 \blacktriangleleft \cdots \blacktriangleleft \gamma_n$.

In what follows, we always simplify the occurrences of `true` in conjunctions and implications.

Example: Russian Peasant Multiplication. The code below implements the multiplication algorithm called Russian Peasant Multiplication. For simplicity, we define the algorithm on the natural numbers and use Euclidean division and remainder operators in terms.

```
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 skip
      / skip [] = 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
  / ref p = a
  / ref q = b
  / ref r = 0
```

This code produces the following verification condition:

forall (a : nat) (b : nat)
  assert a · b + 0 = a · b **and**
  forall (p : nat) (q : nat) (r : nat)
    p · q + r = a · b **implies**
      (q > 0 **implies apply** next) **and**
      (q ≯ 0 **implies apply** skip)
      with skip = **assert** r = a · b
      with next =
        (q mod 2 = 1 **implies apply** write_r) **and**
        (q mod 2 ≠ 1 **implies apply** write_p r)
        with write_r = **forall** (v : nat) (v = r + p **implies apply** write_p v)
        with write_p r = **forall** (v : nat) (v = p + p **implies apply** write_q v r)
        with write_q p r = **forall** (v : nat) (v = q div 2 **implies assert** p · v + r = a · b)

Notice that the first **assert** statement generated by the initial call to `loop` corresponds exactly to the initialisation of a loop invariant. Similarly, the assertion coming from the second, 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 post-condition. In our language, however, verification conditions are pretty much the same as those generated by the usual WP calculus for loops.

First-order representation. Despite the use of second-order predicate variables, our verification conditions never quantify over them, and thus are essentially first-order. Indeed, we can easily translate any second-order VC formula to the first-order language by inlining all predicate definitions. 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, here is the verification condition for the Russian Peasant Multiplication algorithm, obtained in this fashion (in addition to inlining, we also simplify occurrences of **forall** x $(x = t$ **implies** $\Phi)$ to $\Phi[x \mapsto t]$, when term t does not contain x):

forall (a : nat) (b : nat)
  assert a · b + 0 = a · b **and**
  forall (p : nat) (q : nat) (r : nat)
    p · q + r = a · b **implies**
      (q > 0 **implies**
        (q mod 2 = 1 **implies assert** (p + p) · (q div 2) + (r + p) = a · b) **and**
        (q mod 2 ≠ 1 **implies assert** (p + p) · (q div 2) + r = a · b)) **and**
      (q ≯ 0 **implies assert** r = a · b)

One well-known inconvenience of this method is that whenever some non-annotated fragment of code is reachable via two different code paths (which in our language is expressed by a locally defined handler without a precondition that is called from two different points in the subordinate expression) its weakest precondition will appear twice in the verification condition. By chaining such constructions, we can easily come up with examples that demonstrate exponential growth of such verification conditions compared to the size of the original program. Indeed, a simple sequence of conditional statements suffices to produce an oversized verification condition.

An alternative method of verification condition generation was suggested 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.

cite

cite

Below we show that these "compact" verification conditions can be also obtained directly from second-order VC formulas by a chain of truth-preserving transformations. This allows us to split VC generation into two stages: first, from source code to a linear-sized second-order formula, as defined by the rules in Figure 7, and then to a first-order formula, where 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 $(\Phi$ **with** $h\ x_1 \ldots x_n = \Phi_h)$. We want to rearrange formula Φ in such a way that multiple applications of h in it are merged into one. If we obtain an equivalent verification condition $(\Phi'$ **with** $h\ x_1 \ldots x_n = \Phi_h)$, where h is only applied once inside Φ', then we can inline the predicate definition without increasing the size of the formula. We assume that Φ contains no occurrences of **with**: all predicate definitions in Φ must be eliminated beforehand.

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:

$$\textbf{forall}\ z_1 \ldots z_n\ (\Phi_0\ \textbf{with}\ h\ x_1 \ldots x_n = \Phi_h),$$

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

$$(z_1 = s_1 \wedge \cdots \wedge z_n = s_n)\ \textbf{implies apply}\ h\ z_1 \ldots z_n.$$

It is obvious that the new verification condition is equivalent to the initial one. Indeed, the universal quantifier over $z_1, \ldots, z_n$ can be pushed down in Φ_0, and **forall** $\bar{z}$ ($\bar{z} = \bar{s}$ **implies apply** $h\ \bar{z}$) is equivalent to **apply** $h\ \bar{s}$ — since variables $\bar{z}$ are fresh, they cannot occur in terms $\bar{s}$.

Now we can move the occurrences of **apply** $h\ \bar{z}$ 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 deepest subformulas:

$$(\varphi_1 \textbf{ implies apply } h\ \bar{z}) \textbf{ and } (\varphi_2 \textbf{ implies apply } h\ \bar{z}) \implies (\varphi_1 \vee \varphi_2) \textbf{ implies apply } h\ \bar{z}$$

$$\varphi_1 \textbf{ implies } (\varphi_2 \textbf{ implies apply } h\ \bar{z}) \implies (\varphi_1 \wedge \varphi_2) \textbf{ implies apply } h\ \bar{z}$$

$$\varphi_1 \textbf{ implies } ((\varphi_2 \textbf{ implies apply } h\ \bar{z}) \textbf{ and } \Psi) \implies ((\varphi_1 \wedge \varphi_2) \textbf{ implies apply } h\ \bar{z}) \textbf{ and } (\varphi_1 \textbf{ implies } \Psi)$$

$$\textbf{forall } \bar{y}\ (\varphi \textbf{ implies apply } h\ \bar{z}) \implies (\exists \bar{y}.\, \varphi) \textbf{ implies apply } h\ \bar{z}$$

$$\textbf{forall } \bar{y}\ ((\varphi \textbf{ implies apply } h\ \bar{z}) \textbf{ and } \Psi) \implies ((\exists \bar{y}.\, \varphi) \textbf{ implies apply } h\ \bar{z}) \textbf{ and forall } \bar{y}.\, \Psi$$

Here we treat the conjunction connective **and** modulo associativity and commutativity. We also assume that Ψ does not contain occurrences of **apply** $h\ \bar{z}$. 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 **apply** $h\ \bar{z}$. Again, it is easy to see that each rewriting step produces an equivalent formula.

We stop rewriting when the VC formula under the definition of h contains only one application of h. Then we simply inline the definition. Of course, it is 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.

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

forall (a : nat) (b : nat)
  assert a · b + 0 = a · b **and**
  forall (p : nat) (q : nat) (r : nat)
    p · q + r = a · b **implies**
      (q > 0 **implies**
        forall (z : nat)
          (q mod 2 = 1 ∧ z = r + p) ∨ (q mod 2 ≠ 1 ∧ z = r) **implies**
            assert (p + p) · (q div 2) + z = a · b) **and**
      (q ≯ 0 **implies assert** r = a · b)

Notice that the third rewriting rule above duplicates formula φ_1. This may lead to combinatorial explosion, when predicate definitions, such as Φ_h, contain multiple applications of predicate variables defined above h.

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

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

9 Auxiliary Variables

When specifying the behaviour of code with side effects, we often need to refer to past states of the execution. For this purpose, we introduce a possibility to capture the state of a reference at the handler entry and store it in an *auxiliary variable* which can only be used in specifications but not in the actual code. Here is how we would specify a handler that increments an integer reference:

```
incr (ref r: int) (old_r = r) {}
  return [r] { r = old_r + 1 }
```

The precondition of `return` says that the value stored in reference `r` at the exit of `incr` is equal to the value of `r` at the start of `incr` (which we preserved in the auxiliary variable `old_r`), plus one.

The syntax of handler prototypes is extended as follows:

$$\begin{aligned} \mathit{prototype} \;&::=\; \mathit{handler}\ \mathit{pre\text{-}write}\ \mathit{referenceParameter}{*}\ \mathit{dataParameter}{*} \\ &\qquad\qquad \mathit{auxiliary}{*}\ \mathit{precondition}^{?}\ \mathit{prototype}{*} \\ \mathit{auxiliary} \;&::=\; \mathit{variable}\ \texttt{=}\ \mathit{reference} \end{aligned}$$

We call the definitions of auxiliary variables simply *auxiliaries* and denote them with letter o.

The definitions of free variables and references are easily adapted: auxiliaries bind variables in the same scope as data parameters, and captured references are bound in the same way as reference occurrences in the precondition. This is reflected in the updated typing rules:

$$\frac{\Gamma \vdash_{\mathrm{wt}} h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{o}\ \{\varphi\}\ \bar{\varrho} \qquad \Gamma, \bar{r}, \bar{x}, \bar{o}, \bar{\varrho} \vdash_{\mathrm{wt}} d}{\Gamma \vdash_{\mathrm{wt}} h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{o}\ \{\varphi\}^{?}\ \bar{\varrho} = d}\ (\textsc{T-Defn}) \qquad \frac{\Gamma \vdash \mathtt{ref}\ p : \tau}{\Gamma \vdash_{\mathrm{wt}} z = p}\ (\textsc{T-Aux})$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathtt{ref}\ p_1 : \tau_1 \quad \ldots \quad \Gamma \vdash \mathtt{ref}\ p_m : \tau_m \\ \Gamma, \bar{r}, \bar{x} \vdash_{\mathrm{wt}} o_1 \quad \ldots \quad \Gamma, \bar{r}, \bar{x} \vdash_{\mathrm{wt}} o_k \\ \Gamma, \bar{r}, \bar{x}, \bar{o} \vdash_{\mathrm{wt}} \varrho_1 \quad \ldots \quad \Gamma, \bar{r}, \bar{x}, \bar{o} \vdash_{\mathrm{wt}} \varrho_n \qquad \Gamma, \bar{r}, \bar{x}, \bar{o} \vdash_{\mathrm{wt}} \varphi \end{array}}{\Gamma \vdash_{\mathrm{wt}} h\ [p_1 \ldots p_m]\ \bar{r}\ \bar{x}\ o_1 \ldots o_k\ \{\varphi\}\ \varrho_1 \ldots \varrho_n}\ (\textsc{T-Proto})$$

$$\frac{\begin{array}{c} \Gamma_1, \Gamma_2 \vdash s_1 : \tau_1 \quad \ldots \quad \Gamma_1, \Gamma_2 \vdash s_m : \tau_m \qquad \Gamma' = \Gamma_1, \Gamma_2, \bar{o}[\bar{r} \mapsto \bar{q}] \\ \Gamma_2 \vdash \mathtt{ref}\ q_1 : \xi_1 \quad \ldots \quad \Gamma_2 \vdash \mathtt{ref}\ q_k : \xi_k \qquad q_1, \ldots, q_k \text{ are distinct} \\ \Gamma' \vdash_{\mathrm{wt}} \mathrm{Impl}(\varrho_1[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_1) \quad \ldots \quad \Gamma' \vdash_{\mathrm{wt}} \mathrm{Impl}(\varrho_n[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_n) \\ \Gamma_1 \vdash_{\mathrm{wt}} \langle h\ [\bar{p}]\ (\mathtt{ref}\ r_1 : \xi_1) \ldots (\mathtt{ref}\ r_k : \xi_k)\ (x_1 : \tau_1) \ldots (x_m : \tau_m)\ \bar{o}\ \{\varphi\}^{?}\ \varrho_1 \ldots \varrho_n \rangle \end{array}}{\Gamma_1, \Gamma_2 \vdash_{\mathrm{wt}} h\ q_1 \ldots q_k\ s_1 \ldots s_m\ g_1 \ldots g_n}\ (\textsc{T-Call})$$

We put auxiliaries straight into the typing context in order to distinguish auxiliary variables from data parameters and make them inadmissible in the program code (as the T-Var rule would not apply). In precondition formulas, the occurrences of auxiliary variables are type-checked using the typing axiom $\Gamma_1, \mathtt{ref}\ p : \tau, \Gamma_2, z = p, \Gamma_3 \vdash z : \tau$. In the T-Call rule, we check the handler arguments $\bar{g}$ in the typing context Γ' which includes the instantiated auxiliaries, where the reference parameters $\bar{r}$ of handler h are replaced with the actual reference arguments $\bar{q}$.

Since auxiliary variables can only appear in specifications, they do not affect program execution in any way. Nevertheless, we still must take them into account in our evaluation rules, in order to maintain the well-typedness and the provability of verification conditions during evaluation:

$$h\ \bar{q}\ \bar{s}\ \bar{g}\ /\!/\ \Delta \;\longrightarrow\; d[\bar{r}, \bar{x}, \bar{z}, \natural\bar{\varrho} \mapsto \bar{q}, \bar{s}, \bar{k}, \bar{g}]\ /\!/\ \Delta \quad \text{where } \Delta \text{ contains definition } h\ [\cdot]\ \bar{r}\ \bar{x}\ \bar{o}\ \{\cdot\}^{?}\ \bar{\varrho} = d \text{ and for each } z_i = p_i \text{ in } \bar{o}[\bar{r} \mapsto \bar{q}], \; k_i \text{ is the value stored in } p_i \text{ in } \Delta$$

A similar adjustment must be made for every library handler that has auxiliaries in its prototype.

$$\mathrm{VC}(h\ \bar{q}\ \bar{s}\ g_1 \ldots g_n) \triangleq (\Phi \blacktriangleleft \mathrm{Impl}(\varrho_1[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_1) \blacktriangleleft \cdots \blacktriangleleft \mathrm{Impl}(\varrho_n[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_n))\sigma$$

where $\langle h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{o}\ \{\varphi\}^?\ \varrho_1 \ldots \varrho_n \rangle$ is the specialized prototype of h,
σ is the characteristic substitution of $\bar{o}[\bar{r} \mapsto \bar{q}]$, and

$$\Phi = \begin{cases} \textbf{assert}\ \varphi[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}] & \text{if precondition } \varphi \text{ is given,} \\ \textbf{apply}\ h\ \bar{p}\,\bar{q}\,\bar{s} & \text{otherwise} \end{cases}$$

$$\mathrm{VC}(e\ /\ \gamma_1 \ldots \gamma_n) \triangleq \mathrm{VC}(e) \blacktriangleleft \gamma_1 \blacktriangleleft \cdots \blacktriangleleft \gamma_n$$

$$\mathrm{VC}(e\ /\ \texttt{ref}\ p = t) \triangleq \mathrm{VC}(e)[p \mapsto t]$$

$$\Phi \blacktriangleleft h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{o}\ \{\varphi\}\ \bar{\varrho} = d \triangleq \Phi\ \textbf{and forall}\ \bar{p}\bar{r}\bar{x}\ (\varphi\ \textbf{implies}\ \mathrm{VC}(d))\sigma$$

where σ is the characteristic substitution of $\bar{o}$

$$\Phi \blacktriangleleft h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{o}\ \bar{\varrho} = d \triangleq \Phi\ \textbf{with}\ h\ \bar{p}\bar{r}\bar{x} = \mathrm{VC}(d)\sigma$$

where σ is the characteristic substitution of $\bar{o}$

Figure 8: Verification conditions with auxiliary variables.

Given a list of auxiliaries $z_1 = p_1, \ldots, z_n = p_n$, we call $[\bar{z} \mapsto \bar{p}]$ the *characteristic substitution* of that list. We use characteristic substitutions to eliminate auxiliary variables in verification conditions, as shown in Figure 8. Notice that we can only apply these substitutions to a verification condition formula, and never to a program expression or handler definition. Indeed, replacing an auxiliary variable with the corresponding reference in the specifications of handler parameters and locally defined handlers would completely destroy its intended meaning: for example, the postcondition of `incr` would become an unsatisfiable formula `r = r + 1`.

Consider the following expression:

```
start
/ start (old_q = q) {} = assign q 1 cont
  / cont [q] { q > old_q } = incr q cont
/ ref q = 0
```

It represents a non-terminating computation, where reference `q`, initially set to 0, is first set to 1, and then is repeatedly incremented in an infinite loop. The invariant of the loop, represented by the precondition of handler `cont`, states that the value of `q` at the beginning of each loop iteration is strictly greater than 0, which is the value of `q` at the start of the computation.

Let us compute the verification condition for this expression. The updated rules produce the following formula, which we show without applying substitutions for the auxiliary variables and reference allocations. As before, we remove the trivial **assert** `true` subformulas.

```
((forall (q : int) (q = 1 implies q > old_q)) and
 (forall (q : int) (q > old_q implies
   (forall (q : int) (q = old_r + 1 implies q > old_q))[old_r ↦ q]))
)[old_q ↦ q][q ↦ 0]
```

The first line is the verification condition of the call `assign q 1 cont`, which reduces to the VC of $\mathrm{Impl}(\texttt{return}_{\texttt{assign}}[\texttt{r}, \texttt{v} \mapsto \texttt{q}, 1], \texttt{cont})$: whatever the new value of `q`, the instantiated postcondition of `assign` implies the precondition of `cont`. The rest of the formula is the verification condition for the definition of `cont`. The third line is the VC of the call `incr q cont`, which reduces to the VC of $\mathrm{Impl}(\texttt{return}_{\texttt{incr}}[\texttt{r} \mapsto \texttt{q}], \texttt{cont})$: whatever the new value of `q`, the instantiated postcondition of

`incr` implies the precondition of `cont`. Unlike `assign`, handler `incr` captures the initial value of its reference parameter in an auxiliary variable `old_r`. Thus, once we finished computing the VC of the call to `incr`, we replace `old_r` with the actual reference argument `q`. Likewise, once we finished computing the verification condition for the definition of `start`, we replace auxiliary variable `old_q` with `q`, and instantiate `q` with 0.

After substitution and alpha-conversion, we obtain the following formula:

$$
\begin{array}{l}
(\textbf{forall}\ (\mathtt{q} : \mathtt{int})\ (\mathtt{q} = 1\ \textbf{implies}\ \mathtt{q} > 0))\ \textbf{and} \\
(\textbf{forall}\ (\mathtt{q} : \mathtt{int})\ (\mathtt{q} > 0\ \textbf{implies} \\
\quad (\textbf{forall}\ (\mathtt{q}' : \mathtt{int})\ (\mathtt{q}' = \mathtt{q} + 1\ \textbf{implies}\ \mathtt{q}' > 0))))
\end{array}
$$

10 Termination

$$\mathrm{VC}(h\ \bar{q}\ \bar{s}\ g_1 \ldots g_n) \triangleq (\Psi \blacktriangleleft \mathrm{Impl}(\varrho_1[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_1) \blacktriangleleft \cdots \blacktriangleleft \mathrm{Impl}(\varrho_n[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}], g_n))\sigma$$

where $\langle h\ [\bar{p}]\ \bar{r}\ \bar{x}\ \bar{o}\ \{\varphi\}^?\ \varrho_1 \ldots \varrho_n\rangle$ is the specialized prototype of h,

σ is the characteristic substitution of $\bar{o}[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}]$,

$$\Psi = \begin{cases} \Phi\ \textbf{and assert}\ \mathfrak{u} \prec \mathfrak{v} & \text{if } \mathfrak{u} \text{ is a variant variable defined in } \bar{o}, \\ & \quad \mathfrak{v} \text{ appears in the typing context of the call,} \\ & \quad \mathfrak{v} \text{ is the variant variable of } h \text{ or a sibling of } h, \\ \Phi & \text{otherwise} \end{cases}$$

$$\Phi = \begin{cases} \textbf{assert}\ \varphi[\bar{r}, \bar{x} \mapsto \bar{q}, \bar{s}] & \text{if precondition } \varphi \text{ is given,} \\ \textbf{apply}\ h\ \bar{p}\,\bar{q}\,\bar{s} & \text{otherwise} \end{cases}$$

11 Ghost Code

1. 'partial/total' - handler annotation, derivable for defined handlers. A defined handler is partial if RH' of its definition contains a partial handler (which is why it is forbidden to pass a partial handler argument for a total handler parameter). Recursive definitions whose termination is not proved, contain 'infty' in their RH', and thus force the handler to be partial. Otherwise, the handler can be considered total. For handler parameters, by convention: a handler parameter is partial if all of its handler parameters are total. The same convention applies to library handlers. The only exception is 'absurd' (terminating recursive definition).

2. 'ghost/material' - annotation for data and reference variables, and for total handlers. Partially derivable by contamination (described at the end). Does not influence typing or operational semantics.

3. A handler is a 'checkpoint' iff it is partial, has no partial handler parameters and all of its data, reference and total handler parameters are ghost.

4. 'skips to' is the maximal relation on handlers that satisfies the following equivalence:

f skips to h iff - f is a defined partial handler with definition d - h is a checkpoint handler accessible to f - for every partial handler g in RH'(d), - g is either h or skips to h - all references in E(d)(g) are ghost.

We compute this relation bottom-up. We ignore the total handlers in RH'(d) and the effects associated to them, because the total handlers will always return to the caller (that is, d), and the computation of d will continue up to some partial handler, in which case, we will see the same

5. Erasure operator removes from the program all ghost data and all skippable computations. If some ghost variable or reference is used in an unskippable part of code, then the resulting program will be ill-typed, meaning that the original code is inadmissible.

Erase(e) = h if - h is a checkpoint handler and e is in the scope of h - for every partial handler g in RH'(d), - g is either h or skips to h - all references in E(d)(g) are ghost.

if several h are possible, use the rightmost non-skippable, and if all are skippable, then use the rightmost one.

Erase(f q* s* g*) - drop the arguments for ghost parameters, - canonicalize via Impl the handler parameters that have incompatible signatures after erasure.

Erase(h r* x* p*) - drop ghost parameters, pre-writes, pre-conditions

Erase(e / d*) - drop the definitions of ghost total handlers

Erase(e / ref r) - drop ghost allocations.

6. Contamination - adds 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, data, 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