Flexible Verification Conditions with Continuations and Barriers

Andrei Paskevich

Flexible Verification Conditions with Continuations and Barriers

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.

1 Syntax

The building blocks of our language are *expressions* which perform effectful computations, and *terms* which represent pure data. Expressions and terms are distinct syntactic entities: a term can be passed as an argument to an expression, but an expression cannot evaluate to a term. Furthermore, terms can occur inside logical specifications and expressions cannot. We distinguish the identifiers that we bind to terms and expressions, and call them *variables* and *handlers*, respectively.

For our purposes in this document, we do not need comprehensive definitions of the syntax and semantics of terms. We expect terms to be composed of variables, constants, and pure total operations, so that any given term has the same meaning in a program code and in a logical specification. It is possible to restrict terms to variables and constants only, and delegate all computation to the *primitive handlers* that form the "standard library" of our language. For the sake of convenience however, we shall admit a handful of basic operations on unbounded integers, Booleans, and polymorphic lists:

term ::= *variable*
| `true` | `false` | `...` | `-1` | `0` | `1` | `2` | `...`
| *term* `+` *term* | *term* `-` *term* | *term* `*` *term*
| *term* `=` *term* | *term* `<` *term* | *term* `>` *term*
| `cons` *term* *term* | `nil` | `isCons` *term* | `...`

termType ::= *typeVariable*
| `bool` | `int` | `list` *termType* | `...`

typeBinding ::= *variable* `:` *termType*
| *variable* `: &` *termType*

Bindings with the & symbol before the type introduce mutable variables, or *references*. References are allocated and modified during program execution, and are automatically dereferenced when used inside terms. We denote immutable variables with letters x, y, z, references with p, q, r, terms with s and t, type variables with α and β, and term types with τ and θ.

A basic executable expression is a handler or an anonymous *closure* applied to a list of arguments. On top of that, we can put recursive *handler definitions*, *reference allocations*, logical *assertions*, and

two *barrier tags*, denoted ↑ and ↓. These tags guide the computation of verification conditions, and otherwise do not affect the program in any way.

$$
\begin{array}{rcl}
\mathit{expression} & ::= & \mathit{handler} \mid \mathit{closure} \\
& \mid & \mathit{expression}\ \ \mathit{argument} \\
& \mid & \{\ \mathit{formula}\ \}\ \ \mathit{expression} \\
& \mid & \uparrow\ \mathit{expression} \mid \downarrow\ \mathit{expression} \\
& \mid & \mathit{expression}\ \texttt{/}\ \texttt{\&}\ \mathit{variable}\ \texttt{=}\ \mathit{term} \\
& \mid & \mathit{expression}\ \texttt{/}\ \mathit{handler}\ \mathit{pre\text{-}write}\ \texttt{=}\ \mathit{closure} \\
\mathit{argument} & ::= & \mathit{term} \mid \texttt{\&}\ \mathit{variable} \mid \mathit{handler} \mid \mathit{closure} \\
\mathit{closure} & ::= & \mathit{parameter}{*}\ \texttt{→}\ \mathit{expression} \\
\mathit{parameter} & ::= & \mathit{typeBinding} \\
& \mid & \mathit{handler}\ \mathit{pre\text{-}write}\ \mathit{typeBinding}{*} \\
\mathit{prototype} & ::= & \mathit{handler}\ \mathit{pre\text{-}write}\ \mathit{typeVariable}{*}\ \mathit{parameter}{*} \\
\mathit{pre\text{-}write} & ::= & [\ \mathit{variable}{*}\ ]
\end{array}
$$

Handlers and closures accept *term*, *reference*, and *handler parameters*; the latter can be instantiated by both handlers and closures. Our language is limited to the first order: handler parameters cannot have handler parameters of their own. We generalize the syntax of handler parameters to *prototypes*, which admit the parameters of all three kinds, and also include the type variables in which they can be generalized. Handler prototypes do not appear in expressions, and are used only in the typing rules.

At every handler introduction, by definition or by abstraction, we give a *pre-write* annotation for the handler: the set of references that have it in their lexical scope and that are potentially modified between the moment the handler is introduced and the moment it is called. Thus, the pre-writes of a handler parameter in a closure are essentially the cumulative write effect of this term on all execution paths that lead to that particular continuation. In Section 5, we show how to compute the pre-writes for a given expression and to check that the pre-write annotations in it are correct.

We denote expressions with letters e and d, handlers with h, g, f, arguments with a, parameter lists with π and ϱ, formulas with φ and ψ. Formulas are propositions in some logical language, they can contain variables and terms, but not handlers. We consider as given an entailment relation $\Vdash$ for closed formulas.

We write $e\ \bar{a}$ for a series of nested applications, where argument list $\bar{a}$ can be empty. We write e // Λ for a series of nested handler definitions and reference allocations, where Λ is considered as a sequence, which also can be empty. Empty lists (of parameters, arguments, etc.) are denoted with $\square$. A handler definition $h\,[\bar{q}] = \pi \to d$ can be written in a shortened form $h\,[\bar{q}]\ \pi = d$.

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

$$\begin{aligned}
\mathrm{FV}(h) &\triangleq \varnothing & \mathrm{FH}(h) &\triangleq \{h\}\\
\mathrm{FV}(\square \to e) &\triangleq \mathrm{FV}(e) & \mathrm{FH}(\square \to e) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}((x:\tau)\,\pi \to e) &\triangleq \mathrm{FV}(\pi \to e) \setminus \{x\} & \mathrm{FH}((x:\tau)\,\pi \to e) &\triangleq \mathrm{FH}(\pi \to e)\\
\mathrm{FV}((p:\&\tau)\,\pi \to e) &\triangleq \mathrm{FV}(\pi \to e) \setminus \{p\} & \mathrm{FH}((p:\&\tau)\,\pi \to e) &\triangleq \mathrm{FH}(\pi \to e)\\
\mathrm{FV}((g\,[\bar{q}]\,\varrho)\,\pi \to e) &\triangleq \mathrm{FV}(\pi \to e) \cup \{\bar{q}\} & \mathrm{FH}((g\,[\bar{q}]\,\varrho)\,\pi \to e) &\triangleq \mathrm{FH}(\pi \to e) \setminus \{g\}\\
\mathrm{FV}(e\ s) &\triangleq \mathrm{FV}(e) \cup \mathrm{FV}(s) & \mathrm{FH}(e\ s) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}(e\ \&r) &\triangleq \mathrm{FV}(e) \cup \{r\} & \mathrm{FH}(e\ \&r) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}(e\ d) &\triangleq \mathrm{FV}(e) \cup \mathrm{FV}(d) & \mathrm{FH}(e\ d) &\triangleq \mathrm{FH}(e) \cup \mathrm{FH}(d)\\
\mathrm{FV}(\{\varphi\}\ e) &\triangleq \mathrm{FV}(\varphi) \cup \mathrm{FV}(e) & \mathrm{FH}(\{\varphi\}\ e) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}(\uparrow e) &\triangleq \mathrm{FV}(e) & \mathrm{FH}(\uparrow e) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}(\downarrow e) &\triangleq \mathrm{FV}(e) & \mathrm{FH}(\downarrow e) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}(e\ /\ \&r = s) &\triangleq (\mathrm{FV}(e) \setminus \{r\}) \cup \mathrm{FV}(s) & \mathrm{FH}(e\ /\ \&r = s) &\triangleq \mathrm{FH}(e)\\
\mathrm{FV}(e\ /\ h\,[\bar{q}] = d) &\triangleq \mathrm{FV}(e) \cup \{\bar{q}\} \cup \mathrm{FV}(d) & \mathrm{FH}(e\ /\ h\,[\bar{q}] = d) &\triangleq (\mathrm{FH}(e) \cup \mathrm{FH}(d)) \setminus \{h\}
\end{aligned}$$

We consider the FV operator as given on terms and formulas. In the rules for $\mathrm{FV}(e\ d)$ and $\mathrm{FH}(e\ d)$, expression d stands for both handlers and closures. An expression is said to be *closed* if it contains no free variables and all free handlers in it are primitive.

We identify expressions modulo renaming of bound handlers and variables. In a given expression, a free immutable variable can be instantiated with a term (denoted $e[x \mapsto s]$), a free reference variable can be instantiated with another reference (denoted $e[p \mapsto r]$), and a free handler can be instantiated with another handler (denoted $e[g \mapsto f]$). All substitutions are capture-safe.

2 Semantics

Our language is designed to serve as an abstract representation in program verification, rather than in a compiler or an interpreter. Thus, its operational semantics is not expected to be efficient or even implementable on any real hardware. Its sole purpose is to provide a mathematical foundation for our verification procedures as well as a target for translation from source languages.

We identify a subset of ground terms as *values*: usually, these are constants and superpositions of constructors like `cons` and `nil`. We assume as given a normalization operator $[\![s]\!]_\Sigma$ that reduces s to its value in evaluation context Σ, which provides values for the free variables in s. For example, $[\![\mathtt{6 * x = 42}]\!]_{\mathtt{x}\mapsto 7}$ is `true`. We expect $\Vdash s = [\![s]\!]_\square$ to hold for any ground term s.

We extend the normalization operator to formulas, where $[\![\varphi]\!]_\Sigma$ simply instantiates the free variables in φ with their values in Σ. Then we define it on expressions as follows:

$$\begin{aligned}
[\![h]\!]_\Sigma &\triangleq h & [\![\pi \to e]\!]_\Sigma &\triangleq \pi \to e\\
[\![e\ s]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ [\![s]\!]_\Sigma & [\![\{\varphi\}\ e]\!]_\Sigma &\triangleq \{[\![\varphi]\!]_\Sigma\}\ e\\
[\![e\ \&r]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ \&r & [\![\uparrow e]\!]_\Sigma &\triangleq \uparrow e\\
[\![e\ d]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ d & [\![\downarrow e]\!]_\Sigma &\triangleq \downarrow e
\end{aligned}$$

$$[\![e\ /\ \&r = s]\!]_\Sigma \triangleq \begin{cases} [\![e]\!]_{\Sigma, r\mapsto[\![s]\!]_\Sigma}\ /\ \&r = [\![s]\!]_\Sigma & \text{when } r \in \mathrm{FV}([\![e]\!]_{\Sigma, r\mapsto[\![s]\!]_\Sigma}),\\ [\![e]\!]_{\Sigma, r\mapsto[\![s]\!]_\Sigma} & \text{otherwise.}\end{cases}$$

$$[\![e\ /\ h\,[\bar{q}] = d]\!]_\Sigma \triangleq \begin{cases} [\![e]\!]_\Sigma\ /\ h\,[\bar{q}] = d & \text{when } h \in \mathrm{FH}([\![e]\!]_\Sigma),\\ [\![e]\!]_\Sigma & \text{otherwise.}\end{cases}$$

Essentially, $[\![e]\!]_\Sigma$ normalizes terms in arguments and allocations, instantiates formulas in assertions, and drops unused allocations and handler definitions. Normalization does not descend under closures, assertions, and barriers.

We provide a small-step operational semantics for our language. We define a reduction relation $e \longrightarrow e'$ on normalized closed expressions, and then extend it to all closed expressions by performing a normalization step whenever it is appropriate. The reduction rules are given in Figure 1, and every rule, except E-Norm, has an implicit side condition that the evaluated expression is in normal form.

$$\frac{e \neq [\![e]\!]_\square}{e \longrightarrow [\![e]\!]_\square} \quad \text{(E-Norm)}$$

$$\frac{h\,[\bar{q}] = d \in \Lambda}{h\;\bar{a}\;/\!/\;\Lambda \longrightarrow d\;\bar{a}\;/\!/\;\Lambda} \quad \text{(E-Ctxt)}$$

$$((x:\tau)\;\pi \to e)\;s\;\bar{a}\;/\!/\;\Lambda \longrightarrow (\pi \to e)[x \mapsto s]\;\bar{a}\;/\!/\;\Lambda \quad \text{(E-AppT)}$$

$$((p:\&\tau)\;\pi \to e)\;\&r\;\bar{a}\;/\!/\;\Lambda \longrightarrow (\pi \to e)[p \mapsto r]\;\bar{a}\;/\!/\;\Lambda \quad \text{(E-AppR)}$$

$$((g\,[\bar{q}]\,\varrho)\;\pi \to e)\;f\;\bar{a}\;/\!/\;\Lambda \longrightarrow (\pi \to e)[g \mapsto f]\;\bar{a}\;/\!/\;\Lambda \quad \text{(E-AppH)}$$

$$((g\,[\bar{q}]\,\varrho)\;\pi \to e)\;(\varrho \to d)\;\bar{a}\;/\!/\;\Lambda \longrightarrow (\pi \to (e\;/\;g\,[\bar{q}]\,\varrho = \downarrow d))\;\bar{a}\;/\!/\;\Lambda \quad \text{(E-AppC)}$$

$$\square \to e\;/\!/\;\Lambda \longrightarrow e\;/\!/\;\Lambda \quad \text{(E-Void)}$$

$$\uparrow e\;/\!/\;\Lambda \longrightarrow e\;/\!/\;\Lambda \quad \text{(E-TagU)}$$

$$\downarrow e\;/\!/\;\Lambda \longrightarrow e\;/\!/\;\Lambda \quad \text{(E-TagD)}$$

$$\frac{\Vdash \varphi}{\{\varphi\}\;e\;/\!/\;\Lambda \longrightarrow e\;/\!/\;\Lambda} \quad \text{(E-Assert)}$$

$$\texttt{if true}\;d\;e\;/\!/\;\Lambda \longrightarrow d\;/\!/\;\Lambda \qquad \texttt{unList}\;(\texttt{cons}\;t_1\;t_2)\;d\;e\;/\!/\;\Lambda \longrightarrow d\;t_1\;t_2\;/\!/\;\Lambda$$

$$\texttt{if false}\;d\;e\;/\!/\;\Lambda \longrightarrow e\;/\!/\;\Lambda \qquad \texttt{unList nil}\;d\;e\;/\!/\;\Lambda \longrightarrow e\;/\!/\;\Lambda$$

$$\frac{t \neq 0 \qquad s \operatorname{div} t = n}{\texttt{div}\;s\;t\;e\;/\!/\;\Lambda \longrightarrow e\;n\;/\!/\;\Lambda} \qquad \texttt{assign}\;\&r\;s\;e\;/\!/\;\Lambda, \&r = t, \mathrm{M} \longrightarrow e\;/\!/\;\Lambda, \&r = s, \mathrm{M}$$

Figure 1: Operational semantics.

Rule E-Ctxt expands handler definitions by replacing the handler with the closure on the right side of the definition. Rules E-AppT, E-AppR, and E-AppH perform β-reduction. As noted above, we expect that substitution renames bound variables and handlers as necessary to avoid illegal capture. Rule E-AppC converts a closure argument into a handler definition and pushes it inside the closure on the left side. This operation is also performed in a capture-safe manner: we expect that parameter list π does not rebind g and does not capture any variable or handler in the new definition.

Rule E-Assert requires the asserted formula φ to be valid before proceeding with the execution. Of course, the validity of an arbitrary logical formula (e.g., in first-order logic) cannot be effectively verified in a practical implementation, but this is not our purpose here. We want to state and prove the correctness of our verification procedures — in particular, that a program with a valid verification condition cannot get stuck during its execution because of a failed assertion.

We also postulate the evaluation rules for the *primitive handlers*. Of note is the rule for `assign`, which modifies the context of the call: the allocation clauses in the evaluated expression effectively act as the mutable store. Also notice the rule for `div`, which progresses only when the denominator is non-zero: like assertions, partial primitives have the blocking semantics.

cite Fell.

3 Typing rules

Our language is compatible with various type systems, and here we equip it here with ML-style polymorphic rank-1 types. Type polymorphism is restricted to data types: the arity of handlers is invariant. The types of handlers are given through their prototypes. For example, here are the prototypes of the four primitive handlers shown above:

```
if     []   (b: bool) (then []) (else [])
unList [] α (l: list α) (onCons [] (lh: α) (lt: list α)) (onNil [])
div    []   (m: int) (n: int) (return [] (q: int))
assign [] α (r: &α) (v: α) (return [r])
```

These handlers have no pre-writes, as there are no globally defined references that would be visible to the primitive handlers. The `unList` and `assign` handlers are polymorphic in α; generally speaking, primitive handlers are always polymorphic in all type variables that appear in their prototypes. The handler parameter of `assign` has a non-empty pre-write annotation `[r]`, indicating that reference `r` is potentially modified during the execution of `assign`, before it passes control to the continuation. Specifying the pre-writes of handler parameters when they refer to previous reference parameters is the reason we keep parameter names in the prototypes. Handler prototypes are identified modulo α-conversion, that is, `assign [] β (p: &β) (z: β) (out [p])` is the same prototype as above.

We use parameter lists, denoted π and ϱ, as expression types. Expressions that do not expect arguments have the *void type* $\square$. Expression types are also identified modulo parameter renaming, which allows us to match the types of parameters with the types of arguments. A typing judgement $\Gamma \vdash e : \pi$ is made in context Γ, composed of handler prototypes and type bindings for variables and references. We assume that Γ contains the prototype of every primitive handler, and we consider as given the typing relations for terms and formulas, written as $\Gamma \vdash s : \tau$ and $\Gamma \vdash \varphi : \mathrm{Prop}$, respectively. Typing rules for expressions are shown in Figure 2.

$$\frac{h\,[\bar{q}]\,\bar{\alpha}\,\pi \in \Gamma}{\Gamma \vdash h : \pi[\bar{\alpha} \mapsto \bar{\tau}]}\ \text{(T-Ctxt)} \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \square \to e : \square}\ \text{(T-Void)}$$

$$\frac{\Gamma \vdash e : (x : \tau)\,\pi \qquad \Gamma \vdash s : \tau}{\Gamma \vdash e\ s : \pi}\ \text{(T-AppT)} \qquad \frac{\Gamma,\, x : \tau \vdash \pi \to e : \pi}{\Gamma \vdash (x : \tau)\,\pi \to e : (x : \tau)\,\pi}\ \text{(T-ParT)}$$

$$\frac{\Gamma \vdash e : (g\,[\bar{q}]\,\varrho)\,\pi \qquad \Gamma \vdash d : \varrho}{\Gamma \vdash e\ d : \pi}\ \text{(T-AppE)} \qquad \frac{\Gamma,\, p : \&\tau \vdash \pi \to e : \pi}{\Gamma \vdash (p : \&\tau)\,\pi \to e : (p : \&\tau)\,\pi}\ \text{(T-ParR)}$$

$$\frac{\Gamma, \Delta' \vdash e : (r : \&\tau)\,\pi \qquad \Delta' \text{ is } \Delta \text{ with all handler prototypes removed}}{\Gamma, r : \&\tau, \Delta \vdash e\ \&r : \pi}\ \text{(T-AppR)}$$

$$\frac{\overline{q : \&\tau} \in \Gamma \qquad \Gamma,\, g\,[\bar{q}]\,\varrho \vdash \pi \to e : \pi \qquad \pi \text{ contains no term or reference parameters}}{\Gamma \vdash (g\,[\bar{q}]\,\varrho)\,\pi \to e : (g\,[\bar{q}]\,\varrho)\,\pi}\ \text{(T-ParH)}$$

$$\frac{\overline{q : \&\tau} \in \Gamma \qquad \Gamma, h\,[\bar{q}]\,\bar{\alpha}\,\pi \vdash d : \pi \qquad \Gamma, h\,[\bar{q}]\,\bar{\alpha}\,\pi \vdash e : \square \qquad \text{none of } \bar{\alpha} \text{ is free in } \Gamma}{\Gamma \vdash e \,/\, h\,[\bar{q}] = d : \square}\ \text{(T-Defn)}$$

$$\frac{\Gamma \vdash s : \tau \qquad \Gamma, r : \&\tau \vdash e : \square}{\Gamma \vdash e \,/\, \&r = s : \square}\ \text{(T-Alloc)} \qquad \frac{\Gamma \vdash \varphi : \mathrm{Prop} \qquad \Gamma \vdash e : \square}{\Gamma \vdash \{\varphi\}\,e : \square}\ \text{(T-Assert)}$$

$$\frac{\Gamma \vdash e : \square}{\Gamma \vdash \uparrow e : \square}\ \text{(T-TagU)} \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \downarrow e : \square}\ \text{(T-TagD)}$$

Figure 2: Typing rules.

Rule T-AppR ensures that our code does not contain memory aliases, which means that at no

point it is possible to access the same memory location through two different references. The only way to create a memory alias in our language is to give a new name to a reference by passing it as a reference argument to an expression. To ensure the alias safety of an application e `&`r, rule T-AppR type-checks the applicand e in a restricted typing context, from which reference r and any handler in its scope are expunged. To see how this works, consider two typical cases of aliasing (for the sake of readability, we omit the empty pre-writes):

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

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

Rule T-AppR rejects both of these programs. Indeed, `f &r &r` does not type-check, because the typing context for `f &r` does not contain a binding for `&r`. Application `g &r` is also ill-typed, because the typing context for `g` does not contain a binding for `g` itself.

When a handler parameter is instantiated with a closure, evaluation rule E-AppC creates a local handler definition. This may produce ill-typed code when in the same parameter list, a reference is bound after the handler, as the newly created definition is pushed under the reference parameter, which changes the order of introduction, making the T-AppR rule inapplicable. To avoid this, a side condition in the T-ParH rule requires handler parameters to always come after term and reference parameters in parameter lists and prototypes.[1]

Handlers introduced by a definition (rule T-Defn) are generalized in the type variables that are not fixed by the typing context. The types of handler parameters (rule T-ParH) are not generalized, in agreement with the rules of prenex polymorphism.

4 Translation

Rewrite with discussion of automated translation from While and ML, some syntactic sugar (spec in prototypes, like in leiden?), and more examples.

Let us demonstrate how usual programming constructs — recursion, pattern matching, and exception handling — are expressed in our language. In Figure 3, 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 4, we translate these functions into our language. Exceptions are translated using additional handler parameters: `find_greater` receives a handler parameter named `return` for normal termination, and another, named `not_found`, for exceptional termination. The caller, `check_greater`, provides a continuation for both outcomes. If the OCaml version of `check_greater` did not catch and handle the exceptional outcome of `find_greater`, then its prototype in our language would have an additional handler parameter to reflect that `check_greater` can raise an exception, and this parameter would be passed verbatim to `find_greater`.

[1] It is possible to forgo this restriction by making T-AppR keep in Δ' the handlers that do not depend, directly or through other handlers, on reference r. To keep track of these dependencies, we would need an additional "hide" annotation in handlers' prototypes which would require their implementations to be typeable without the hidden references. Here, we opt for a simpler, albeit stricter, approach.

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

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

Figure 3: Simple OCaml program.

```
find_greater (n: int) (l: list int) (return (_: int)) not_found
= unList l ((h: int) (t: list int) →
    if (h > n) (→ return h) (→ find_greater n t return not_found)) not_found

check_greater (n: int) (l: list int) (return (_: bool))
= find_greater n l ((_: int) → return true) (→ return false)
```

Figure 4: The program from Fig. 3, translated.

5 Effect analysis

The pre-write annotations in a program must be exhaustive in the sense that every actual write effect that may happen during evaluation is accounted for. In order to ensure this, we compute an overapproximation of the pre-writes for each handler in our code, and we check that each reference in this overapproximation is listed in the pre-write annotation for that handler.

An *effect set*, or simply an *effect*, is a set of pairs (r, h) such that reference r is a potential pre-write for handler h. For example, if we consider an application `assign &p 42 g`, which writes `42` into reference `p` and then passes control to handler `g`, then `p` is a pre-write for `g`, and the effect of the call is the singleton set $\{(\mathtt{p}, \mathtt{g})\}$.

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

$$\begin{aligned}
\mathcal{E}(h) &\triangleq \varnothing & \mathcal{E}(\square \to e) &\triangleq \mathcal{E}(e)\\
\mathcal{E}(e\ s) &\triangleq \mathcal{E}(e) & \mathcal{E}((x:\tau)\,\pi \to e) &\triangleq \mathcal{E}(\pi \to e)\\
\mathcal{E}(e\ \&r) &\triangleq \mathcal{E}(e) & \mathcal{E}((p:\&\tau)\,\pi \to e) &\triangleq \mathcal{E}(\pi \to e) \ominus p\\
\mathcal{E}((e : (g\,[\bar{q}]\,\varrho)\,\pi)\ d) &\triangleq \mathcal{E}(e) \cup (\bar{q} \otimes d) & \mathcal{E}((g\,[\bar{q}]\,\varrho)\,\pi \to e) &\triangleq \mathcal{E}(\pi \to e) \ominus g\\
\mathcal{E}(e\ /\ h\,[\bar{q}] = d) &\triangleq (\mathcal{E}(e) \cup (\bar{q} \otimes d)) \ominus h & \mathcal{E}(\uparrow e) \triangleq \mathcal{E}(\downarrow e) &\triangleq \mathcal{E}(e)\\
\mathcal{E}(e\ /\ \&r = s) &\triangleq \mathcal{E}(e) \ominus r & \mathcal{E}(\{\varphi\}\,e) &\triangleq \mathcal{E}(e)
\end{aligned}$$

$$E \ominus r \triangleq \{\,(q,g) \in E \mid q \neq r\,\} \qquad \bar{q} \otimes d \triangleq \mathcal{E}(d) \cup (\{\bar{q}\} \times \mathrm{FH}(d))$$

$$E \ominus h \triangleq \begin{cases} \{\,(q,g) \in E \mid g \neq h\,\} & \text{if the pre-write annotation of } h \text{ covers every } (p,h) \text{ in } E,\\ \text{undefined} & \text{otherwise.}\end{cases}$$

In the rule for $\mathcal{E}(e\ d)$, we refer to the inferred type of expression e. Also, in the definition of $E \ominus h$, we assume the prototype of handler h to be known from the context. In other words, the operations above are performed on specific occurrences of expressions inside the program, so that their lexical and typing context is available.

When we instantiate a handler parameter $g\,[\bar{q}]\,\varrho$ with an expression d (which may be a handler or a closure) or consider a handler definition $h\,[\bar{q}] = d$ (here, d has to be a closure), references $\bar{q}$ are

considered as potential pre-writes for every handler occurring in d, and the corresponding pairs are added to the proper effect of d. The resulting effect set is denoted $\bar{q} \otimes d$.

When a reference r is bound in an expression, it is evicted from the underlying effect set E, which we denote $E \ominus r$. Similarly, when a handler h is bound, we remove from E the pre-writes computed for h (denoted $E \ominus h$), on condition that they are all duly listed in the handler's pre-write annotation. The effect $\mathcal{E}(e)$ is undefined when a pre-write computed for a handler introduced in e does not appear in the pre-write annotation of this handler. A program whose effect set is undefined is considered ill-specified and is rejected. In what follows, we assume that all programs we consider have a well-defined effect.

Let us go back to the application `assign &p 42 g`. The type of `assign` in this expression, according to its prototype and the T-Ctxt rule, is `(r: &int) (v: int) (return [r])`. The pre-write annotation of `return` is bound by the reference parameter, so that the type of the partial application `assign &p 42` is `return [p]`. The effect of the whole expression is then computed as follows:

$$\mathcal{E}(\texttt{(assign \&p 42 : return [p]) g}) = \mathcal{E}(\texttt{assign \&p 42}) \cup (\texttt{p} \otimes \texttt{g}) =$$
$$\mathcal{E}(\texttt{assign \&p}) \cup \mathcal{E}(\texttt{g}) \cup (\{\texttt{p}\} \times \{\texttt{g}\}) = \mathcal{E}(\texttt{assign}) \cup \varnothing \cup \{(\texttt{p}, \texttt{g})\} = \{(\texttt{p}, \texttt{g})\}$$

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

Notice that the space of solutions is finite. Indeed, the typing rules require the references in the pre-write annotation of a handler to be present in the typing context of that handler. Thus, there is only a finite number of references that can appear in the pre-write annotation of any given handler. The computed effect sets are similarly bounded. Indeed, any handler that appears in $\mathcal{E}(e)$ must freely occur in expression e. The same is not necessarily true for references, since a pre-write may be introduced via the prototype of a called handler, without occurring explicitly in the expression. However, since this handler must be in the scope of the concerned reference, and e must be in the scope of the handler, we can conclude that any reference in $\mathcal{E}(e)$ must be visible in e.

6 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. The pre-write annotation of a handler h indicates which references accessible to it may have changed their value between the introduction and the execution of the handler: it is those references (or rather their current values) that must be passed to h as extra parameters. The current value of any other reference can be accessed directly from the lexical context of h, since the reference has not been modified since the introduction of h.

Operator $[\![\cdot]\!]$ translates expressions into stateless form and eliminates reference parameters and

pre-writes from parameter lists:

$$\begin{array}{rcl@{\qquad}rcl}
[\![h]\!] & \triangleq & h\ \bar{q}_h & [\![\square]\!] & \triangleq & \square \\
[\![e\ s]\!] & \triangleq & [\![e]\!]\ s & [\![(x:\tau)\,\pi]\!] & \triangleq & (x:\tau)\,[\![\pi]\!] \\
[\![e\ \&r]\!] & \triangleq & [\![e]\!]\ r & [\![(p:\&\tau)\,\pi]\!] & \triangleq & (p:\tau)\,[\![\pi]\!] \\
[\![(e:(g\,[\bar{q}]\,\varrho)\,\pi)\ f]\!] & \triangleq & [\![e]\!]\ ((\overline{q:\tau_q})\ [\![\varrho]\!] \to [\![f\ \bar{a}_\varrho]\!]) & [\![(g\,[\bar{q}]\,\varrho)\,\pi]\!] & \triangleq & (g\ (\overline{q:\tau_q})\ [\![\varrho]\!])\ [\![\pi]\!] \\
[\![(e:(g\,[\bar{q}]\,\varrho)\,\pi)\ (\varrho \to d)]\!] & \triangleq & [\![e]\!]\ ((\overline{q:\tau_q})\ [\![\varrho]\!] \to [\![d]\!]) & [\![\pi \to e]\!] & \triangleq & [\![\pi]\!] \to [\![e]\!] \\
[\![e\ /\ h\,[\bar{q}]\,\pi = d]\!] & \triangleq & [\![e]\!]\ /\ h\ (\overline{q:\tau_q})\ [\![\pi]\!] = [\![d]\!] & [\![\{\varphi\}\ e]\!] & \triangleq & \{\varphi\}\ [\![e]\!] \\
[\![e\ /\ \&r = s]\!] & \triangleq & [\![e]\!][r \mapsto s] & [\![\uparrow e]\!] \triangleq \uparrow [\![e]\!] & \quad & [\![\downarrow e]\!] \triangleq \downarrow [\![e]\!]
\end{array}$$

In the rule for $[\![h]\!]$, we assume handler h to have prototype $h\,[\bar{q}_h]\,\bar{\alpha}\,\pi$. In the rules for $[\![e\ f]\!]$ and $[\![e\ (\varrho \to d)]\!]$, we refer to the inferred type of expression e. Given a parameter list ϱ, we denote with $\bar{a}_\varrho$ the corresponding list of arguments: a term parameter $x:\tau$ becomes a term argument x, a reference parameter $r:\&\theta$ becomes a reference argument $\&r$, and a handler parameter $g\,[\bar{q}]\,\varrho$ becomes a handler argument g. We write τ_p to denote the type of reference p in the original program.

Our transformation converts reference parameters and pre-written references into term parameters. When a handler f is passed as a handler argument to expression e, the pre-write annotation of f may not be exactly the same as the one of the corresponding handler parameter. To preserve the well-typedness of the transformed code, we η-expand this occurrence of f into an appropriate closure. Notice that converting pre-writes into term parameters captures the corresponding references in the underlying expression. This is what allows us to instantiate them with correct values during execution.

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

```
assign [] α (r: α) (v: α) (return [] (r: α))
```

The operational semantics of this pure version is as follows: `assign` $t\ s\ e\ /\!/\ \Lambda \longrightarrow e\ s\ /\!/\ \Lambda$.

7 Functional correctness

Verification conditions, also called VC formulas, are written in a restricted fragment of the first-order language with local predicate definitions. This fragment can be described by the following grammar:

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

In this syntax, user-written formulas act as atoms, and handlers as locally defined predicate symbols. Predicate definitions are not recursive and are generalized in every type variable that is not fixed by their lexical context. Note that *vc* does not appear on the left-hand side of an implication. Moreover, since handlers can not occur in user-written formulas, all occurrences of locally defined predicates and, therefore, of VC subformulas in a verification condition are necessarily positive. We denote verification conditions with Φ and Ψ.

Given a verification condition Φ and a parameter list π, we define:

$\Phi|^-_\pi$ is Φ where every VC subformula $h\ \bar{s}$, such that h is bound by π, is replaced with $\bot$.

$\Phi|^+_\pi$ is Φ where every VC subformula $h\ \bar{s}$, such that h is bound by π, is replaced with $\top$.

$\Phi|^\circ_\pi$ is Φ where we replace with $\top$ every VC subformula that has no free occurrences of handlers (acting as predicate symbols) defined inside Φ or bound by π.

It is easy to see that any Φ is equivalent to $\Phi|^+_\pi \wedge \Phi|^\circ_\pi$ for all π. Also, $\Phi|^+_\square$ is just Φ and $\Phi|^\circ_\square$ is just $\top$.

$$\mathcal{S}(h) \triangleq \begin{cases} \lambda\pi.\Phi & \text{when } h \text{ is a primitive with contract } \lambda\pi.\Phi, \\ \lambda\pi.\, h\,\bar{q}\,\bar{z}_\pi & \text{when } h \text{ is a handler parameter } h\,[\bar{q}]\,\pi, \\ \lambda\pi.\,\bot & \text{when } h \text{ is defined recursively by } h\,[\bar{q}]\,\pi = d, \\ & \quad \text{the current invocation of } h \text{ is inside } d, \\ & \quad \text{and the result of } \mathcal{S}(h) \text{ is used in } \complement^{\top}_{\bot}(d), \\ \lambda\pi.\, h\,\bar{q}\,\bar{z}_\pi \wedge \complement^{\top}_{\bot}(d)|^{\circ}_{\pi} & \text{otherwise, when } h \text{ is defined by } h\,[\bar{q}]\,\pi = d. \end{cases}$$

$$\complement^{\top}_{\mathfrak{d}}(h) \triangleq \lambda\varrho.\Psi \quad \text{where } \lambda\varrho.\Psi \text{ is an appropriate type instance of } \mathcal{S}(h)$$
$$\complement^{\bot}_{\mathfrak{d}}(h) \triangleq \lambda\varrho.\Psi|^{\circ}_{\varrho} \quad \text{where } \lambda\varrho.\Psi \text{ is an appropriate type instance of } \mathcal{S}(h)$$

$$\begin{aligned}
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ s) &\triangleq (\lambda\pi.\Phi)[x \mapsto s] && \text{when } \complement^{\mathfrak{p}}_{\mathfrak{d}}(e) = \lambda(x:\tau)\,\pi.\Phi \\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ \&r) &\triangleq (\lambda\pi.\Phi)[p \mapsto r] && \text{when } \complement^{\mathfrak{p}}_{\mathfrak{d}}(e) = \lambda(p:\&\tau)\,\pi.\Phi \\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ d) &\triangleq \lambda\pi.(\Phi \mathbin{/} g\ (\overline{q:\tau_q})\ (\overline{z_\varrho:\tau_{z_\varrho}}) \equiv \Psi) && \text{when } \complement^{\mathfrak{p}}_{\mathfrak{d}}(e) = \lambda(g\,[\bar{q}]\,\varrho)\,\pi.\Phi \\
& && \quad \text{and } \complement^{\mathfrak{p}}_{\mathfrak{d}}(d) = \lambda\varrho.\Psi \\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e \mathbin{/} \&r = s) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)[r \mapsto s] \\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e \mathbin{/} h\,[\bar{q}]\,\pi = d) &\triangleq (\complement^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \forall\,\overline{q:\tau_q}.\forall\,\overline{z_\pi:\tau_{z_\pi}}.\complement^{\bot}_{\mathfrak{p}}(d)|^{-}_{\pi}) \mathbin{/} h\ (\overline{q:\tau_q})\ (\overline{z_\pi:\tau_{z_\pi}}) \equiv \complement^{\top}_{\bot}(d)|^{+}_{\pi}
\end{aligned}$$

$$\begin{aligned}
\complement^{\mathfrak{p}}_{\mathfrak{d}}(\pi \to e) &\triangleq \lambda\pi.\,\complement^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)|^{\circ}_{\pi} & \complement^{\mathfrak{p}}_{\mathfrak{d}}(\uparrow e) &\triangleq \complement^{\mathfrak{d}}_{\mathfrak{d}}(e) \\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(\{\varphi\}\,e) &\triangleq (\mathfrak{p} \to \varphi) \wedge (\varphi \to \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)) & \complement^{\mathfrak{p}}_{\mathfrak{d}}(\downarrow e) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{p}}(e)
\end{aligned}$$

Figure 5: Verification condition generation.

An expression of type π produces parametrized verification conditions of the form $\lambda\pi.\Phi$. A term or a reference parameter in π binds in Φ a first-order variable of the same type. A handler parameter $g\,[\bar{r}]\,\varrho$ in π binds in Φ a predicate symbol, whose parameters correspond to $\bar{r}$ and ϱ combined. By convention, $\lambda\square.\Phi$ is the same as Φ.

To each handler in our program we assign a *contract*: a parametrized verification condition that describes the invocations of this handler. A handler with prototype $h\,[\bar{q}]\,\bar{\alpha}\,\pi$ has the contract of the form $\lambda\pi.\Phi$, which can be type-instantiated in $\bar{\alpha}$. The contracts of primitive handlers are predefined:

```
if :      λ(c : bool) (then [ ]) (else [ ]) . (c → then) ∧ (¬c → else)
unList :  λ(l : list α) (onCons [ ] (_ : α) (_ : list α)) (onNil [ ]) .
            (∀ h : α. ∀ t : list α. l = cons h t → onCons h t) ∧ (l = nil → onNil)
div :     λ(n : int) (m : int) (return [ ] (_ : int)) . m ≠ 0 ∧ return (n div m)
assign :  λ(r : &α) (v : α) (return [r]) . return v
```

The contracts of handler parameters and user-defined handlers are produced by operator $\mathcal{S}$ below.

Verification conditions are computed by operator $\complement^{\mathfrak{p}}_{\mathfrak{d}}$. Boolean parameters $\mathfrak{p}$ and $\mathfrak{d}$ set the mode:

- $\complement^{\bot}_{\top}$ generates a *callee verification condition* to prove the correctness of handler definitions.
- $\complement^{\top}_{\bot}$ generates a *caller verification condition* for individual invocations of defined handlers.
- $\complement^{\top}_{\top}$ generates a *full verification condition*, equivalent to the conjunction of the first two modes.
- $\complement^{\bot}_{\bot}$ generates a *null verification condition*, which is always true on fully applied expressions.

The rules of VC generation are given in Figure 5. Every expression is considered within its lexical and typing context in the program. We denote with $\bar{z}_\pi$ the list of variables bound by parameter list π, and we write τ_v to denote the type of variable v. In the rule for $\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ d)$, we assume that π does not rebind g and does not capture any symbol in the new predicate definition.

The third clause in the definition of $\mathcal{S}(h)$ breaks a circular dependency that arises whenever we compute the contract of a recursively defined handler. When a handler h is introduced by a definition, the contract of h is the caller VC of the definition body (a part of this VC is moved into a locally defined predicate, but the meaning stays the same). When the definition is recursive, this computation refers to $\mathcal{S}(h)$, creating a circularity. To avoid this problem, we overapproximate the contract of h with $\lambda\pi.\bot$ whenever it is used in the computation of $\complement_\bot^\top$ for the body of h. In more formal terms, we equip the operators $\complement_\flat^\mathrm{p}$ and $\mathcal{S}$ with an implicit auxiliary parameter that tracks the invocations of $\complement_\bot^\top$ on handler definition bodies.

Termination. At the syntactic level, we identify a certain subset of variables as *variant variables*, and denote them using letters with circumflex: $\hat{u}$, $\hat{v}$, etc. We say that a recursive definition $h\,[\bar{q}]\,\pi = d$ has a *variant term* s (under π), whenever the definition body d is a β-redex $((\hat{v} : \theta) \to e)\ s$ and type θ is equipped with a well-founded order $\prec$. Then, to verify the termination of such functions, we extend the definition of the $\mathcal{S}$ operator as follows:

$$\mathcal{S}(h) \triangleq \begin{cases} \lambda\pi.\Phi & \text{when } h \text{ is a primitive with contract } \lambda\pi.\Phi, \\ \lambda\pi.\,h\,\bar{q}\,\bar{z}_\pi & \text{when } h \text{ is a handler parameter } h\,[\bar{q}]\,\pi, \\ \lambda\pi.\bot & \text{when } h \text{ is defined recursively by } h\,[\bar{q}]\,\pi = d, \\ & \quad \text{the current invocation of } h \text{ is inside } d, \\ & \quad \text{and the result of } \mathcal{S}(h) \text{ is used in } \complement_\bot^\top(d), \\ \lambda\pi.\,h\,\bar{q}\,\bar{z}_\pi \wedge \complement_\bot^\top(d)|_\pi^\circ \wedge s \prec \hat{v} & \text{otherwise, when } h \text{ is defined by } h\,[\bar{q}]\,\pi = d, \\ & \quad \text{expression } d \text{ is of the form } ((\hat{v} : \theta) \to e)\ s, \\ & \quad \text{and the current invocation of } h \text{ is inside } e, \\ \lambda\pi.\,h\,\bar{q}\,\bar{z}_\pi \wedge \complement_\bot^\top(d)|_\pi^\circ & \text{otherwise, when } h \text{ is defined by } h\,[\bar{q}]\,\pi = d. \end{cases}$$

The added goal $s \prec \hat{v}$ states the value of the variant computed for the current call of h (notice that s may refer to the parameters bound by π) decreases with respect to the value of the variant computed at the beginning of the body of h and stored in $\hat{v}$. Since $\prec$ is well-founded, this ensures that an infinite tower of recursive calls of h is impossible.

8 Ghost code

In the practice of deductive verification, the term "ghost code" refers to the parts of our program that do not participate in the computation of the final result but help writing adequate specification and verifying the program correctness.

Ghost code is a versatile and powerful instrument. Local ghost variables and ghost function parameters allow us to better describe the progression of an algorithm; for example, when looking for a maximal element in an array, we may use a ghost variable to store the index of the current best candidate, eliminating the need in an existential quantifier in the loop invariant. Ghost fields in data structures can provide a mathematical model of the data, independent of the implementation details, so that the operations on the data structure can be specified in terms of the abstract model rather than concrete implementation. One can even write an entire ghost reimplementation of a program, to be executed in lockstep with the original code, so that the correctness proof of one version could be transferred to the other [1].

From the point of view of type-checking, evaluation, effect analysis or VC generation, ghost code and ghost data are treated in exactly the same way as non-ghost, or *material*, parts of our program. Moreover, we want to use the same data types and operations over material and ghost data, without introducing a parallel standard library for ghost integers, ghost references, ghost arrays, etc. We must however ensure that ghost code in our program does not affect the material computation, that it can be erased from the program without changing its flow or outcome. One can see ghost data as a sort of classified information that should never leak to the lower-security material code. One way to verify

this *non-interference* property is to recognize ghost data at the type level and use the typing rules to track dependencies between values, so that the material computation is never contaminated by ghost data. Filliâtre et al. used this approach in their formal framework for ghost code in a functional language with polymorphic types and mutable state [2].

In this work, we explore a slightly different method. Assuming that every variable in our code is explicitly marked as material or ghost, we determine which parts of our program can be safely skipped, as they produce no material data for subsequent computation nor make a choice between different continuations. Such expressions are said to be *transparent* up to the point at which the computation resumes. After performing this analysis, we "deghostify" our program: we erase all ghost parameters and ghost values, and we replace all transparent expressions with their non-skippable continuations. If the resulting program is well-formed, we know that the ghost data does not leak into the material computation, and the original program is admissible. If, on the other hand, we are left with code where a handler expecting a material parameter is given a ghost argument, and this handler call can not be skipped — then we have found an interference and must reject the program.

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

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

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

- h is a primitive handler whose evaluation may diverge or halt;
- h is a handler parameter;
- h is defined by $h\,[\bar{q}] = d$, where d is unrestrained or $\mathrm{FH}(d) \setminus \{h\}$ contains an escaping handler.

Identifying escaping handlers is important for code flow analysis, as they might not return to the caller via one of their handler parameters. Instead, they may escape either by halting the program or by entering a non-terminating computation, or by passing control to some other escaping handler.

A handler without handler parameters ought to be escaping, with one important exception: we can define an endlessly looping handler, whose termination can be "proved" due to a contradictory precondition: `absurd = ((v̂:int) → {⊥} ↑ absurd) 0`. The definition of `absurd` contains a variant, and therefore is restrained. Since it contains no other handlers, `absurd` is non-escaping. The callee verification condition for `absurd` is equivalent to $\bot \to (\bot \wedge \texttt{0} < \texttt{0})$, which is provable. The contract of `absurd` is equivalent to $\bot$, which means that it can be used only in unreachable code.

An expression e is *transparent* up to handler f whenever the following conditions are met:

1. e has type $\square$ and is inside the scope of f;
2. f does not have handler parameters, and all term and reference parameters of f are ghost;
3. every reference r in the pre-write annotation of f, such that $(r, f) \in \mathcal{E}(e)$, is ghost;
4. e is restrained and every escaping handler h in $\mathrm{FH}(e) \setminus \{f\}$ is introduced through a definition $h\,[\bar{q}]\,\pi = d$, where expression d either coincides with e or is transparent up to f.

An expression that is transparent up to a handler f can be replaced with f when we erase ghost data from our program. Indeed, f is accessible from that position and would not require arguments. The skipped expression does not modify material references accessible to f and does not diverge, meaning that no observable effects are lost when we skip it. The last condition ensures that the target handler cannot be bypassed: every escaping handler in the skipped expression is either f itself or can be replaced with f. We only need to consider escaping handlers, as the non-escaping ones must necessarily return to the caller via one of their handler parameters (or represent unreachable code). Recursive handler invocations are discounted in order to avoid circularity in the definition.

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

There is also a weaker form of transparency, where an application is replaced with its continuation. An expression e is *transparent* up to closure $\varrho \to d$ whenever the following conditions are met:

1. e has type $\square$ and is of the form $e_0\ (\varrho \to d)\ \bar{a}$;
2. ϱ does not contain handler parameters, and all term and reference parameters in ϱ are ghost;
3. assuming expression e_0 has type $(g\ [\bar{q}]\ \varrho)\ \pi$, every reference in $\bar{q}$ is ghost;
4. e_0 and every expression in $\bar{a}$ are restrained and do not contain escaping handlers.

Operators $\|\cdot\|$ and $\lfloor\cdot\rfloor$ apply to expressions and parameter lists. They eliminate assertions, barriers, transparent subexpressions, ghost terms, and ghost parameters.

$$\|e\| \triangleq \begin{cases} f & \text{if } e \text{ is transparent up to handler } f, \\ \lfloor \varrho \to d \rfloor & \text{otherwise, if } e \text{ is transparent up to closure } \varrho \to d, \\ \lfloor e \rfloor & \text{otherwise.} \end{cases}$$

$$\begin{aligned}
\lfloor h \rfloor &\triangleq h \\
\lfloor (e : (\bullet x : \tau)\ \pi)\ \bullet s \rfloor &\triangleq \|e\|\ s \\
\lfloor (e : (\circ x : \tau)\ \pi)\ s \rfloor &\triangleq \|e\| \\
\lfloor (e : (\bullet p : \&\tau)\ \pi)\ \&r \rfloor &\triangleq \|e\|\ \&r \\
\lfloor (e : (\circ p : \&\tau)\ \pi)\ \&r \rfloor &\triangleq \|e\| \\
\lfloor e\ d \rfloor &\triangleq \|e\|\ \|d\| \\
\lfloor e\ /\ h\ [\bar{q}] = d \rfloor &\triangleq \|e\|\ /\ h\ [\|\bar{q}\|] = \|d\| \\
\lfloor e\ /\ \&\bullet r = \bullet s \rfloor &\triangleq \|e\|\ /\ \&r = s \\
\lfloor e\ /\ \&\circ r = s \rfloor &\triangleq \|e\|
\end{aligned}$$

$$\begin{aligned}
\|\square\| &\triangleq \square \\
\|(\bullet x : \tau)\ \pi\| &\triangleq (x : \tau)\ \|\pi\| \\
\|(\circ x : \tau)\ \pi\| &\triangleq \|\pi\| \\
\|(\bullet p : \&\tau)\ \pi\| &\triangleq (p : \&\tau)\ \|\pi\| \\
\|(\circ p : \&\tau)\ \pi\| &\triangleq \|\pi\| \\
\|(g\ [\bar{q}]\ \varrho)\ \pi\| &\triangleq (g\ [\|\bar{q}\|]\ \|\varrho\|)\ \|\pi\| \\
\lfloor \pi \to e \rfloor &\triangleq \|\pi\| \to \|e\| \\
\lfloor \{\varphi\}\ e \rfloor &\triangleq \|e\| \\
\lfloor \uparrow e \rfloor \triangleq \|e\| \qquad \lfloor \downarrow e \rfloor &\triangleq \|e\|
\end{aligned}$$

$\lfloor (e : (\bullet x : \tau)\ \pi) \circ s \rfloor$ is undefined

$\lfloor e\ /\ \&\bullet r = \circ s \rfloor$ is undefined

$\|\bar{q}\|$ is $\bar{q}$ with all ghost references removed

The operations above are undefined whenever ghost data is put under a material name. In this case, the original program is considered invalid and is rejected. In the rules for application, we refer to the type of the expression on the left-hand side to know the ghost status of the expected parameter. Renaming of parameters preserves their ghost status, which ensures that reference arguments always have the same ghost status as the corresponding parameters, and that the rule for $\lfloor e\ d \rfloor$ preserves the well-typedness of the program.

Index

full, $\complement_{\top}^{\top}$, 10
null, $\complement_{\bot}^{\bot}$, 10
operator, $\complement_{\mathfrak{d}}^{\mathfrak{p}}$, 10
void type, $\square$, 5

References

[1] Martin Clochard, Claude Marché, and Andrei Paskevich. Deductive verification with ghost monitors. In *Principles of Programming Languages*, New Orleans, United States, 2020.

[2] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.