Flexible Verification Conditions with Continuations and Barriers

Andrei Paskevich

▶ **To cite this version:**

Andrei Paskevich. Flexible Verification Conditions with Continuations and Barriers. 2023. hal-04115885v3

HAL Id: hal-04115885

https://inria.hal.science/hal-04115885v3

Preprint submitted on 12 Sep 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

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, 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 Introduction

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

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

This does not seem to be a universal perception: after all, the foundational tool of deductive verification, the Hoare triple, comes with a single postcondition. The notion that a program expression can only end in a single way and has to satisfy this postcondition on exit is a mental habit that we have to overcome when we start dealing with `break`, `continue`, `return`, and exceptions, both in code and in specification. We recognize the same habit in the type systems of ML-like languages, where exception-raising instructions are given the type $\forall\alpha.\alpha$, which is a formal way to say "whatever". The type system simply cannot express that the expression does not produce any value (nor that it raises an exception), and the best we can do is to say that the produced value can have any type at all. The usual way to explain this inadequacy is to say that exceptions are side effects, and the type system, with its noble lambda pedigree, cares not for such shenanigans. However, we believe that it is more useful to see exceptions not as a side effect but as a pure control structure, a sequence step towards a different code point. In fact, from the standpoint of specification and verification (as opposed to compilation and actual execution), there is not much reason to distinguish a "normal" and an "exceptional" way to leave a block of code: `raise` is just a different kind of `return`. From the

same standpoint, and perhaps more radically, there is not much reason to distinguish function calls and function returns. Indeed, we describe them in a very similar manner: both are parametrized by data (received and returned), and both carry functional properties (pre- and postconditions) to verify when execution reaches the corresponding point. The difference is that the code guarded by the precondition is given once, during the function definition, and the code guarded by the postcondition comes after individual function calls. However, the verification condition has the same form in both cases: the weakest precondition of the underlying code must hold in every program state that satisfies the guarding property.

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

In this document, we develop a formal programming language named Coma (an abbreviation for Continuation Machine), intended to serve as an intermediate language in deductive verification tools. The syntax of Coma contains just a handful of operations: definition and invocation of subroutines, memory allocation, and some specification facilities. Despite this austerity, the language can readily express the usual control structures of imperative languages: conditionals, loops, pattern matching, exceptions, etc. The translation is straightforward and preserves the structure of the original program. We equip Coma with a simple alias-free memory model, which allows us to write specifications and compute verification conditions without resorting to ownership discipline or separation logic. This setting suffices to specify and verify programs that work with separated memory regions, allocated and destroyed on the fly; for example, the internal memory models of Why3 and Frama-C can be represented in Coma. We expect our approach to be compatible with more expressive specification languages and effect systems; however, the primary focus of this work is on not state management but on control flow.

One peculiar aspect of Coma is that specification annotations can be mixed with executable code. For example, it is possible to specify the precondition of a subroutine not at the entry point but in the middle of the computation. In this way, given a function defined by pattern matching, Haskell-style, we can provide each branch with its own contract, without redoing the same pattern matching in the common pre- and postcondition. The same mechanism allows us to omit specification entirely for some (or all) control paths in a subroutine, and let the verification condition generator to effectively inline the code.

Verification conditions for Coma programs are expressed in a higher-order logical language. We demonstrate that both classical Dijkstra-style weakest preconditions and efficient weakest preconditions, as described by Flanagan and Saxe [3], can be mechanically obtained as equivalent forms of the same verification condition. Besides theoretical interest, this also has practical value, as we can use a single implementation of the verification condition generator, producing both flavors of verification conditions as the result.

Outline. We present the syntax of our language in Section 2, its operational semantics in Section 3, and its type system with the no-alias restriction in Section 4. In Section 5, we show how idiomatic code in a traditional procedural language can be translated into our language, while preserving its lexical structure and computational behaviour.

We discuss effect annotations and show how they can be verified (of, if need be, automatically computed) in Section 6. With the help of these annotations, we can translate effectful code into an equivalent pure form via a fine-grained monadic encoding.

In Section 7, we show how to compute verification conditions for partial and total correctness. We also show how different flavors of verification conditions (classical weakest preconditions or so-called efficient weakest preconditions) can be derived from the initial verification condition.

Finally, in Section 8, we introduce the notion of ghost data: program variables that do not affect the main computation and are used 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.

2 Syntax

The building blocks of Coma 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 be reduced to a term. Furthermore, terms can occur inside logical specifications and expressions cannot. An expression can be encapsulated in a named *handler* (which is what we call subroutines) or in an anonymous *closure*, and either invoked directly or passed as a continuation argument to another expression.

Terms are composed of variables, constants, and pure total operations, provided that they have the same meaning in executable code and in specification. It is possible to restrict the syntax of terms to variables and literal values, and delegate all computation to handlers, either predefined or introduced by the programmer. For the sake of convenience however, we shall admit in terms a handful of basic operations on unbounded integers, Booleans, and polymorphic lists. Of course, any practical implementation may choose its own set of types and operations, and may also allow the programmer to extend it when desired.

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* | `...`

binding ::= *variable* `:` *termType*

refBinding ::= `&` *variable* `:` *termType*

Figure 1: Terms and term types.

prototype ::= *handler* *pre-write* *signature*

signature ::= (*typeVariable* | *binding* | *refBinding*)$*$ *prototype*$*$

pre-write ::= `[` *variable*$*$ `]`

Figure 2: Handler prototypes and type signatures.

The grammar of terms and term types is shown in Figure 1. Type bindings with the ampersand symbol before the variable 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.

Handlers are characterized by their *prototypes*, whose syntax is given in Figure 2. Handler names are lexically distinct from variables; for brevity, we often use the word "handler" meaning a handler

identifier, when this does not introduce an ambiguity. The *type signature* of a handler is the list of its formal parameters (as we adopt the continuation-passing style, handlers do not have return values).

Handlers admit type parameters, variable and reference parameters, and, finally, handler parameters, also called continuation parameters or *outcomes*. Outcomes must be placed at the end of the parameter list: this allows for a simpler formulation of the alias-free typing rules in Section 4.

Every handler in the program comes with a *pre-write annotation*: the set of references that are visible to that handler and are potentially modified between the moment the handler is introduced and any moment it is invoked during program execution. In particular, the pre-writes associated to a given outcome in the prototype of a handler represent the cumulative write effect of this handler on all execution paths that lead to that particular continuation. In Section 6, we show how to compute the pre-writes for a given program expression and to check that all pre-write annotations in it are correct.

We assume to have at our disposal a number of predefined *primitive handlers*, which constitute the "standard library" of Coma. Here are the prototypes of six primitive handlers that we use throughout this paper (for convenience, we add parentheses where needed and omit empty pre-write annotations):

```
if       (c: 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])
fail
halt
```

Handler `if` makes a choice between two continuations (represented by handler parameters named `then` and `else`, which take no parameters), depending on the Boolean condition `c`. Handler `unList` deconstructs a list value passed in variable parameter `l`: if `l` is non-empty, the head and the tail of the list are passed to the `onCons` outcome, otherwise control passes to `onNil`. Handler `div` performs Euclidean division of two integers and passes the result to its continuation. This operation is allowed only when the divisor is non-zero. Handler `assign` writes value `v` into reference `r`. The outcome of `assign` has a non-empty pre-write annotation `[r]`, indicating that `r` may change during the execution of `assign`. Handler `fail` is an equivalent of `assert false`, it represents code that should never be reached in execution. Finally, handler `halt` stops the computation. Type signatures of handlers are identified modulo renaming of parameters; for instance, `assign β (&p: β) (z: β) (ret [p])` is the same prototype as above.

By allowing Coma computations to have multiple outcomes (or none at all), we can represent as first-class entities what usually has to be hardwired into the core syntax of programming languages: conditionals and pattern matching. Handlers `fail` and `halt` are also noteworthy in this regard: as they do not accept continuation parameters, we know simply by looking at their type signature that they cannot ever return control to the calling computation.[1]

The syntax of Coma expressions is given in Figure 3. A basic executable expression is a handler or an anonymous closure applied to a list of arguments. Type parameters are instantiated by types, variable parameters by terms, reference parameters by references, and handler parameters by handlers and closures. On top of that, we can put recursive *handler definitions*, *reference allocations*, logical *assertions*, and, finally, two *barrier tags*, denoted ↑ and ↓, and called *black-box* and *white-box* tag, respectively. Definitions and allocations are written after the underlying expression; the slash symbol can be read as "where". Barrier tags guide the computation of verification conditions, and have no effect otherwise.

We denote expressions with letters e and d, handlers with h, g, f, arguments with a, type signatures with π and ϱ, and formulas with φ and ψ. Formulas are propositions in some logical language, they can contain variables and terms, but not handlers. We write $e\,\bar{a}$ for a series of nested applications, where argument list $\bar{a}$ can be empty. We write $e \mathbin{/\!/} \Lambda$ for a series of nested handler definitions and reference allocations, where Λ is considered as a sequence, which also can be empty. Empty lists (in

[1] Compare this to a rather brilliant hack in OCaml, where the `assert` function has type `bool → unit`, but the application `assert false` can assume an arbitrary type, so that it can be used in contexts where a return value is expected.

$$
\begin{array}{rcl}
\textit{expression} & ::= & \textit{handler} \mid \textit{closure} \\
& \mid & \textit{expression}\ \ \textit{argument} \\
& \mid & \texttt{\{}\ \textit{formula}\ \texttt{\}}\ \textit{expression} \\
& \mid & \texttt{↑}\ \textit{expression} \mid \texttt{↓}\ \textit{expression} \\
& \mid & \textit{expression}\ \texttt{/}\ \textit{handler pre-write}\ \texttt{=}\ \textit{closure} \\
& \mid & \textit{expression}\ \texttt{/ \&}\ \textit{variable}\ \texttt{:}\ \textit{termType}\ \texttt{=}\ \textit{term} \\
\textit{argument} & ::= & \textit{termType} \mid \textit{term} \mid \texttt{\&}\ \textit{variable} \mid \textit{handler} \mid \textit{closure} \\
\textit{closure} & ::= & \textit{signature}\ \texttt{→}\ \textit{expression}
\end{array}
$$

Figure 3: Expressions.

particular, empty type signatures) are denoted with symbol $\square$. 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 symbols (type variables, variables, and handler symbols) in expressions and type signatures as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{FS}(h) & \triangleq & \{h\} & \mathrm{FS}(\square \to e) & \triangleq & \mathrm{FS}(e) \\
\mathrm{FS}(e\ \theta) & \triangleq & \mathrm{FS}(e) \cup \mathrm{FS}(\theta) & \mathrm{FS}(\alpha\,\pi \to e) & \triangleq & \mathrm{FS}(\pi \to e) \setminus \{\alpha\} \\
\mathrm{FS}(e\ s) & \triangleq & \mathrm{FS}(e) \cup \mathrm{FS}(s) & \mathrm{FS}((x:\tau)\,\pi \to e) & \triangleq & (\mathrm{FS}(\pi \to e) \setminus \{x\}) \cup \mathrm{FS}(\tau) \\
\mathrm{FS}(e\ \&r) & \triangleq & \mathrm{FS}(e) \cup \{r\} & \mathrm{FS}((\&p:\tau)\,\pi \to e) & \triangleq & (\mathrm{FS}(\pi \to e) \setminus \{p\}) \cup \mathrm{FS}(\tau) \\
\mathrm{FS}(e\ d) & \triangleq & \mathrm{FS}(e) \cup \mathrm{FS}(d) & \mathrm{FS}((g\,[\bar{q}]\,\varrho)\,\pi \to e) & \triangleq & (\mathrm{FS}(\pi \to e) \setminus \{g\}) \cup \mathrm{FS}(\varrho) \cup \{\bar{q}\} \\
\mathrm{FS}(\{\varphi\}\,e) & \triangleq & \mathrm{FS}(e) \cup \mathrm{FS}(\varphi) & \mathrm{FS}(e\ /\ \&r:\tau = s) & \triangleq & (\mathrm{FS}(e) \setminus \{r\}) \cup \mathrm{FS}(\tau) \cup \mathrm{FS}(s) \\
\mathrm{FS}(\uparrow e) & \triangleq & \mathrm{FS}(e) & \mathrm{FS}(e\ /\ h\,[\bar{q}] = d) & \triangleq & (\mathrm{FS}(e) \cup \mathrm{FS}(d) \cup \{\bar{q}\}) \setminus \{h\} \\
\mathrm{FS}(\downarrow e) & \triangleq & \mathrm{FS}(e) & \mathrm{FS}(\varrho) & \triangleq & \mathrm{FS}(\varrho \to \_) \setminus \{\_\}
\end{array}
$$

We consider the FS operator as given on term types, terms, and formulas. In the rule for $\mathrm{FS}(e\,d)$, expression d stands for both handlers and closures. In the rule for $\mathrm{FS}(\varrho)$, the underscore symbol stands for a fresh handler symbol, which we use to create an ad hoc closure out of the signature ϱ. An expression is said to be *closed* if it contains no free variables and all free handlers in it are primitive. Free type variables are allowed in closed expressions: they are treated as abstract types.

We identify expressions modulo renaming of bound handlers, variables, and type variables. In a given expression, a free type variable can be instantiated with a term type (denoted $e[\alpha \mapsto \theta]$), 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.

As an example, let us define a handler that computes the factorial of a natural number:

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

The handler is named `factorial`, and it takes an integer parameter `n` and a continuation parameter `return`. The latter also expects an integer parameter, which represents the result of `factorial`. The implementation begins with asserting that `n` is non-negative. This assertion guards the expression in lines 3–6, which is placed under the *black-box tag* `↑`. The tag does not affect the execution of

`factorial` and serves only in the computation of its verification condition. Specifically, a black-box tag delimits a portion of the definition body which is to be verified once, for all acceptable inputs.

The parts of the definition that are not hidden behind the tag — the assertion on line 2 and the definition of the `break` handler on line 7 — determine the specification of `factorial`. This specification must be instantiated and verified whenever the handler is invoked. It takes form of a parametrized logical formula[2] $\lambda n{:}\mathtt{int}.\,\lambda(\mathtt{return}\ (m{:}\mathtt{int})).\,n \geqslant 0 \wedge (\forall m{:}\mathtt{int}.\,m = n! \rightarrow \mathtt{return}\ m)$, where `return` is a unary predicate that stands for the verification condition of the continuation parameter of `factorial`. The right side of the conjunction is actually the proof of the handler definition on line 7. Indeed, we show that the continuation of `factorial` is correct, assuming that its parameter `m` is the factorial of `n`. This is what we expect to prove regarding the return from `factorial`, and thus the assertion $m = n!$ represents the postcondition of this handler, whereas the assertion $n \geqslant 0$ acts as its precondition.

Let us go back to the expression under the tag. There we define a recursive handler `fact` with two parameters, `r` and `k` (lines 4-6), and call it with arguments `1` and `n` (line 3). The precondition of `fact` is given in line 5. The subsequent implementation tests whether `k` is positive, using a predefined handler `if`. This handler receives a Boolean parameter and two continuations, and calls the one or the other depending on whether the condition is true or false. The specification of `if` is the logical formula[2] $\lambda c{:}\mathtt{bool}.\lambda \mathtt{then}.\lambda \mathtt{else}.(c \rightarrow \mathtt{then}) \wedge (\neg c \rightarrow \mathtt{else})$, where `then` and `else` are nullary predicates standing for the respective verification conditions of the two continuation parameters. It is worth emphasizing that `if` is just an ordinary handler, like `factorial` or `fact`, and the specification above is sufficient for verification purposes.

The actual continuations passed to `if` in line 6 are anonymous *closures*. Whenever `k` is positive, `if` makes a recursive call of `fact`. Otherwise, if `k` is zero or negative, `if` calls the `break` handler, leaving `fact`. Since the bodies of `fact` and `break` are hidden under black-box tags, we only need to prove their respective preconditions ($0 \leqslant k - 1 \leqslant n \wedge (r * k) * (k - 1)! = n!$ and $m = n!$) at each call.

Note that `fact` does not return to the caller: to do that, it would need to receive a continuation parameter and invoke it, like `if` and `factorial` do. Instead, `fact` escapes by calling `break` at the end of computation. In this respect, `fact` behaves rather like a loop than a recursive function. Indeed, its continuation is determined statically, by its lexical context, rather than dynamically by its caller. Consequently, there is no distinct postcondition associated to `fact`: in our language, postconditions are preconditions of continuation parameters, and `fact` has none thereof.

demonstrate the piecewise specification

In conclusion, let us reimplement the `factorial` handler using references:

```
1  factorial (n: int) (return (m: int))
2  = { n ⩾ 0 }
3    ↑ fact
4      / fact [r k]
5        = { 0 ⩽ k ⩽ n ∧ r * k! = n! }
6          ↑ if (k > 0) (→ assign int &r (r * k) (→ assign int &k (k - 1) fact))
7                       break
8      / &k: int = n
9    / break [r] = { r = n! } ↑ return r
10   / &r: int = 1
```

References `r` and `k` are allocated in lines 10 and 8, respectively. Handlers `fact` and `break` can access their current values directly, and thus do not require parameters anymore. The first branch of the conditional in line 6 calls `assign` twice in succession, modifying `r` and `k`, before passing to the next iteration of `fact`. Consequently, both references appear in the pre-write annotation of `fact`, and `r` in the pre-write annotation of `break` (as `k` is introduced later).

[2]The actual formula, as defined in Section 7, is slightly more complex due to controllable failure handling. Here, we show a simplified version to give a first intuition of verification conditions in Coma.

3 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, $[\![\texttt{6} * \texttt{x} = \texttt{42}]\!]_{\texttt{x}\mapsto\texttt{7}}$ is `true`. We expect $\vDash s = [\![s]\!]_\square$ to hold for any ground term s, where $\vDash$ denotes logical entailment.

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:

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

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

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

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 4, and every rule, except E-Norm, has an implicit side condition that the evaluated expression is in normal form.

Rule E-Defn expands handler definitions by replacing the handler with the closure on the right side of the definition. Rules E-AppT, E-AppV, 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 type signature π does not rebind g and does not capture any handler in the new definition (notice that according to the syntax of signatures, π may only contain handler parameters at this point).

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. Handlers `fail` and `halt` cannot be evaluated: the former has an unsatisfiable precondition, and the latter represents the final state of computation.

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

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

$$(\alpha\,\pi \to e)\;\theta\;\bar{a} \mathbin{//} \Lambda \longrightarrow (\pi \to e)[\alpha \mapsto \theta]\;\bar{a} \mathbin{//} \Lambda \quad \text{(E-AppT)}$$

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

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

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

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

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

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

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

$$\frac{\vDash \varphi}{\{\varphi\}\,e \mathbin{//} \Lambda \longrightarrow e \mathbin{//} \Lambda} \quad \text{(E-Assert)}$$

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

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

$$\frac{s = n \cdot t + m \qquad 0 \leq m < |t|}{\texttt{div}\;s\;t\;e \mathbin{//} \Lambda \longrightarrow e\;n \mathbin{//} \Lambda} \qquad \texttt{assign}\;\tau\;\&r\;s\;e \mathbin{//} \Lambda, \&r:\tau = t, \mathrm{M} \longrightarrow e \mathbin{//} \Lambda, \&r:\tau = s, \mathrm{M}$$

Figure 4: Operational semantics.

4 Typing rules

In Coma, expression types are type signatures, that is, parameter lists. Expressions that do not take arguments have the *void type* $\square$. The arity is invariant: expressions are polymorphic only in the data types. A typing judgement $\Gamma \vdash e : \pi$ is made in context Γ, composed of handler prototypes and type bindings for variables and references. We do not put type variables in the typing context: every free type variable in e is considered arbitrary but fixed. 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 5.

Rule T-ParT forbids to generalize a closure in a type variable that is fixed by the typing context. In rule T-ParH, we typecheck the signature ϱ by creating an ad hoc closure $\varrho \to \_$, where $\_$ is a fresh nullary handler symbol with the empty pre-write annotation (cf. the definition of $\mathrm{FS}(\varrho)$).

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):

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

$$\frac{\Gamma \vdash e : \alpha\,\pi}{\Gamma \vdash e\ \theta : \pi[\alpha \mapsto \theta]}\ \text{(T-AppT)} \qquad \frac{\Gamma \vdash \pi \to e : \pi \qquad \alpha \text{ is not free in } \Gamma}{\Gamma \vdash \alpha\,\pi \to e : \alpha\,\pi}\ \text{(T-ParT)}$$

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

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

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

$$\frac{\overline{\&q:\tau} \in \Gamma \qquad \Gamma, \_\,[\square]\square \vdash \varrho \to \_ : \varrho \qquad \Gamma, g\,[\bar{q}]\,\varrho \vdash \pi \to e : \pi}{\Gamma \vdash (g\,[\bar{q}]\,\varrho)\,\pi \to e : (g\,[\bar{q}]\,\varrho)\,\pi}\ \text{(T-ParH)}$$

$$\frac{\overline{\&q:\tau} \in \Gamma \qquad \Gamma, h\,[\bar{q}]\,\pi \vdash d : \pi \qquad \Gamma, h\,[\bar{q}]\,\pi \vdash e : \square}{\Gamma \vdash e\ /\ h\,[\bar{q}] = d : \square}\ \text{(T-Defn)}$$

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

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

Figure 5: Typing rules.

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

Notice that considering type signatures modulo parameter renaming allows us to match precisely the type of the left-hand side expression with the applicand in rules T-AppR and T-AppH.

5 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 6, 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.

```
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 6: 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 7: The program from Fig. 6, translated.

In Figure 7, 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`.

6 Effect analysis

Pre-write annotations 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 int &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:

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

$$E \ominus r \triangleq \{(q,g) \in E \mid q \neq r\} \qquad Q \otimes d \triangleq \mathcal{E}(d) \cup \{(q,g) \mid q \in Q \wedge g \in \mathrm{FS}(d)\}$$

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

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 known.

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 potential pre-writes for every handler freely occurring in d, therefore we add the corresponding pairs 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 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 int &p 42 g`. The type signature of `assign` is `α (&r:α) (v:α) (return [r])`. The pre-write annotation of `return` is bound by the reference parameter, so that the type of the partial application `assign int &p 42` is `return [p]`. The effect of the whole expression is then computed as follows:

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

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 preferable to look for minimally 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.

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 $\llbracket\cdot\rrbracket$ translates expressions into stateless form and eliminates reference parameters and

pre-writes from type signatures:

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

In the first rule, we denote with $\bar{q}_h$ the references in the pre-write annotation of handler h. In the rules for $\llbracket e\ (\varrho \to d) \rrbracket$ and $\llbracket e\ f \rrbracket$, we refer to the inferred type of expression e. Given an outcome prototype $g\,[\bar{q}]\,\varrho$, we write $\bar{a}_\varrho$ to denote the corresponding list of arguments. We write τ_q to denote the type of reference q in the original program.

Our transformation converts reference parameters and pre-written references into term parameters. When a handler f is passed as an 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` $\tau\ t\ s\ e\ //\ \Lambda \longrightarrow e\ s\ //\ \Lambda$.

7 Functional correctness

Verification conditions, also called VC formulas, are written in the following grammar:

$$\begin{aligned}
vc \ ::= \ & \mathit{handler} \mid vc\ (\mathit{termType} \mid \mathit{term} \mid vc) \\
\mid \ & \lambda\ (\mathit{typeVariable} \mid \mathit{binding} \mid \mathit{prototype})\ .\ vc \\
\mid \ & \forall\ (\mathit{typeVariable} \mid \mathit{binding})\ .\ vc \\
\mid \ & \bot \mid \mathit{formula} \to vc \mid vc \wedge vc
\end{aligned}$$

Verification conditions are higher-order propositions, and are denoted with letters Φ and Ψ. Logical operations of implication, conjunction, and universal quantification can only be used with the fully applied VC formulas. Handler symbols act as predicate symbols, and can be bound by a λ-abstraction. A handler symbol with prototype $h\,[\bar{q}]\,\pi$ corresponds to a predicate symbol whose parameters are:

- a Boolean "safety flag", which determines whether the precondition of h should be checked,
- the "current state" parameters $\overline{q : \tau_q}$, as shown in the previous section,
- all parameters in π, where references are treated as normal variables, and handlers as predicates.

Note that all VC subformulas in a verification condition are positive: only user-written formulas may appear on the left-hand side of an implication, and they cannot contain handler symbols. For the sake of simplicity, we admit Boolean terms in the positions of formulas and vice versa.

Given a verification condition Φ, a Boolean term s, and a signature ϱ, we can add s to the safety flags of predicate invocations in Φ as follows:

$$\Phi|^s_\varrho \triangleq (\lambda h_1\,[\bar{q}_1]\,\pi_1 . \ldots \lambda h_n\,[\bar{q}_n]\,\pi_n . \Phi)\ (\lambda c : \mathtt{bool} . h_1\ (s \wedge c)) \cdots (\lambda c : \mathtt{bool} . h_n\ (s \wedge c))$$

where $h_1\,[\bar{q}_1]\,\pi_1, \ldots, h_n\,[\bar{q}_n]\,\pi_n$ are the free handlers in Φ that are not bound by ϱ. This operation allows us to put a new safety flag on top of a verification condition:

$$\sharp\Phi \triangleq \lambda b\!:\!\mathtt{bool}.\,\Phi|^{b}_{\square}$$

Assuming that all safety conditions in Φ are expressed through external handlers, such as `fail`, the VC formula $\Phi|^{\perp}_{\square}$ does not produce any proof obligations by itself, and only sets the context for its predicate parameters.

We write $\lambda\pi.\Phi$ and $\forall\pi.\Phi$ to denote a series of nested λ-abstractions or universal quantifications. By convention, $\lambda\square.\Phi$ and $\forall\square.\Phi$ are the same as Φ. We define $\lambda\&r:\tau.\Phi$ and $\forall\&r:\tau.\Phi$ as syntactic sugar for $\lambda r:\tau.\Phi$ and $\forall r:\tau.\Phi$, respectively. An expression $\forall h\,[\bar{q}]\,\pi.\Phi$ is understood as a verification condition where we have no knowledge about handler symbol h, and is defined recursively:

$$\begin{aligned}\forall h\,[\bar{q}]\,\pi.\Phi \triangleq\ & (\lambda h\,[\bar{q}]\,\pi.\Phi)\ (\sharp\lambda\overline{q:\tau_q}.\lambda\pi.\\ & \qquad \mathtt{fail}\ \top \wedge (\forall\overline{r_1:\tau_{r_1}}.\forall\varrho_1.g_1\top\bar{r}_1\ \bar{a}_{\varrho_1}) \wedge \cdots \wedge (\forall\overline{r_n:\tau_{r_n}}.\forall\varrho_n.g_n\top\bar{r}_n\ \bar{a}_{\varrho_n}))\end{aligned}$$

where $g_1\,[\bar{r}_1]\,\varrho_1, \ldots, g_n\,[\bar{r}_n]\,\varrho_n$ are the handler parameters in π. The VC formula on the right-hand side of the application describes a handler with an undefined behaviour: it may fail or it may pass control to any of its handler parameters with arbitrary arguments. When the safety flag introduced by the $\sharp$ operator is set to $\perp$, the predicate is invoked in a weakened form, where we ignore the possibility of failure and only consider the outcomes.

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

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

Informally speaking, $\mathfrak{p}$ says whether we should prove the assertions and the safety of handler calls at the current position, and $\mathfrak{d}$ says whether we shall be proving it after the black-box barrier $\uparrow$.

Here is the definition of the verification condition operator:

$$\begin{aligned}
\complement^{\mathfrak{p}}_{\mathfrak{d}}(h) &\triangleq h\ \mathfrak{p}\ \bar{q}_h & \complement^{\mathfrak{p}}_{\mathfrak{d}}(\uparrow e) &\triangleq \complement^{\mathfrak{d}}_{\mathfrak{d}}(e)\\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ \theta) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\ \theta & \complement^{\mathfrak{p}}_{\mathfrak{d}}(\downarrow e) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{p}}(e)\\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ s) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\ s & \complement^{\mathfrak{p}}_{\mathfrak{d}}(\pi \twoheadrightarrow e) &\triangleq \lambda\pi.\,\complement^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)|^{\perp}_{\pi}\\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ \&r) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\ r & \complement^{\mathfrak{p}}_{\mathfrak{d}}((e : (g\,[\bar{q}]\,\varrho)\,\pi)\ d) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\ (\sharp\lambda\overline{q:\tau_q}.\complement^{\mathfrak{p}}_{\mathfrak{d}}(d))\\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ /\ \&r:\tau = s) &\triangleq \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)[r \mapsto s] & \complement^{\mathfrak{p}}_{\mathfrak{d}}(\{\varphi\}\,e) &\triangleq (\neg\varphi \to \mathtt{fail}\ \mathfrak{p}) \wedge (\varphi \to \complement^{\mathfrak{p}}_{\mathfrak{d}}(e))\\
\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ /\ h\,[\bar{q}]\,\pi = d) &\triangleq (\lambda h\,[\bar{q}]\,\pi.\,\complement^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \forall\overline{q:\tau_q}.\forall\pi.\,\complement^{\perp}_{\mathfrak{p}}(d))\ (\sharp\lambda\overline{q:\tau_q}.\lambda\pi.\forall h\,[\bar{q}]\,\pi.\,\complement^{\top}_{\perp}(d))
\end{aligned}$$

In the rule for $\complement^{\mathfrak{p}}_{\mathfrak{d}}(h)$, we denote with $\bar{q}_h$ the references in the pre-write annotation of handler h. In the rule for $\complement^{\mathfrak{p}}_{\mathfrak{d}}(e\ d)$, we refer to the inferred type of expression e. Given a reference q, we write τ_q to denote its type in the program.

The closed verification condition for a closed expression e is obtained by supplying a contract for each primitive handler (for brevity, we omit the type signatures in the λ-prefix):

$$\begin{aligned}\complement(e) \triangleq\ & (\lambda\,\mathtt{if}\ \mathtt{unlist}\ \mathtt{div}\ \mathtt{assign}\ \mathtt{fail}\ \mathtt{halt}\ldots.\,\complement^{\top}_{\top}(e))\\ & \qquad \Phi_{\mathtt{if}}\ \Phi_{\mathtt{unlist}}\ \Phi_{\mathtt{div}}\ \Phi_{\mathtt{assign}}\ \Phi_{\mathtt{fail}}\ \Phi_{\mathtt{halt}}\cdots\end{aligned}$$

where

$$\begin{aligned}
\Phi_{\texttt{if}} &\triangleq \lambda \texttt{b}:\texttt{bool}.\,\lambda \texttt{c}:\texttt{bool}.\,\lambda \texttt{then}.\,\lambda \texttt{else}.\,(\texttt{c} \longrightarrow \texttt{then}\ \top) \wedge (\neg \texttt{c} \longrightarrow \texttt{else}\ \top) \\
\Phi_{\texttt{unList}} &\triangleq \lambda \texttt{b}:\texttt{bool}.\,\lambda \alpha.\,\lambda \texttt{l}:\texttt{list}\,\alpha.\,\lambda \texttt{onCons}\ (_:\alpha)\ (_:\texttt{list}\,\alpha).\,\lambda \texttt{onNil}. \\
&\qquad (\forall \texttt{h}:\alpha.\,\forall \texttt{t}:\texttt{list}\,\alpha.\,\texttt{l} = \texttt{cons h t} \longrightarrow \texttt{onCons}\ \top\ \texttt{h t}) \wedge \\
&\qquad (\texttt{l} = \texttt{nil} \longrightarrow \texttt{onNil}\ \top) \\
\Phi_{\texttt{div}} &\triangleq \lambda \texttt{b}:\texttt{bool}.\,\lambda \texttt{n}:\texttt{int}.\,\lambda \texttt{m}:\texttt{int}.\,\lambda \texttt{return}\ (_:\texttt{int}).\,(\texttt{m} = 0 \longrightarrow \texttt{b} \longrightarrow \bot) \wedge \\
&\qquad (\forall \texttt{q}:\texttt{int}.\,\forall \texttt{r}:\texttt{int}.\,\texttt{n} = \texttt{m} * \texttt{q} + \texttt{r} \wedge 0 \leq \texttt{r} < |\texttt{m}| \longrightarrow \texttt{return}\ \top\ \texttt{q}) \\
\Phi_{\texttt{assign}} &\triangleq \lambda \texttt{b}:\texttt{bool}.\,\lambda \alpha.\,\lambda \texttt{r}:\alpha.\,\lambda \texttt{v}:\alpha.\,\lambda \texttt{return}\,[\texttt{r}].\,\texttt{return}\ \top\ \texttt{v} \\
\Phi_{\texttt{fail}} &\triangleq \lambda \texttt{b}:\texttt{bool}.\,\texttt{b} \longrightarrow \bot \\
\Phi_{\texttt{halt}} &\triangleq \lambda \texttt{b}:\texttt{bool}.\,\bot \longrightarrow \bot
\end{aligned}$$

As for termination proofs, we say that a handler definition $h\,[\bar{q}]\,\pi = d$ has a *variant term* s (under signature π), whenever the definition body d is of the form

$$((v:\theta) \to e \;/\; h_{\text{rec}}\,[\bar{q}]\,\pi = \{s \prec v\}\ h\ \bar{a}_\pi)\ s$$

where h does not occur in e and $\prec$ is a well-founded order on the type of s, denoted θ. Here, we assign to variable v the value of the variant term at the start of the handler (as a function of the parameters in π and the references in $\bar{q}$), and we make every recursive invocation of h made through h_{rec} compare its respective variant to v and check that it decreases with respect to a chosen well-founded order. This effectively ensures that an infinite tower of recursive calls of h is impossible, and a program where every recursive handler definition has a variant must terminate.

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 $\mathrm{FS}(d) \setminus \{h\}$ 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: $\texttt{absurd} = ((\texttt{v:int}) \to (\{\bot\} \uparrow \texttt{absurd}_{\mathrm{rec}})\ /\ \texttt{absurd}_{\mathrm{rec}} = \{\texttt{0} \prec \texttt{v}\}\,\texttt{absurd})\ \texttt{0}$. The definition of `absurd` contains a variant, and therefore is restrained. Since it contains no other handlers, `absurd` is non-escaping. The contract of `absurd` is equivalent to $\bot$, which means that it can be used only in unreachable code. The callee VC is equivalent to $\bot \to (\texttt{0} \prec \texttt{0} \wedge \bot)$, and thus is provable.

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 has no type or 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 $\mathrm{FS}(e) \setminus \{f\}$ 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. 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.

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

$$\begin{aligned}
\lfloor h \rfloor &\triangleq h & \|\square\| &\triangleq \square \\
\lfloor e\ \theta \rfloor &\triangleq \|e\|\ \theta & \|\alpha\ \pi\| &\triangleq \alpha\ \|\pi\| \\
\lfloor (e : (\bullet x : \tau)\ \pi)\ \bullet s \rfloor &\triangleq \|e\|\ s & \|(\bullet x : \tau)\ \pi\| &\triangleq (x : \tau)\ \|\pi\| \\
\lfloor (e : (\circ x : \tau)\ \pi)\ s \rfloor &\triangleq \|e\| & \|(\circ x : \tau)\ \pi\| &\triangleq \|\pi\| \\
\lfloor (e : (\&\bullet p : \tau)\ \pi)\ \&r \rfloor &\triangleq \|e\|\ \&r & \|(\&\bullet p : \tau)\ \pi\| &\triangleq (\&p : \tau)\ \|\pi\| \\
\lfloor (e : (\&\circ p : \tau)\ \pi)\ \&r \rfloor &\triangleq \|e\| & \|(\&\circ p : \tau)\ \pi\| &\triangleq \|\pi\| \\
\lfloor e\ d \rfloor &\triangleq \|e\|\ \|d\| & \|(g\ [\bar{q}]\ \varrho)\ \pi\| &\triangleq (g\ [\|\bar{q}\|]\ \|\varrho\|)\ \|\pi\| \\
\lfloor e\ /\ h\ [\bar{q}] = d \rfloor &\triangleq \|e\|\ /\ h\ [\|\bar{q}\|] = \|d\| & \lfloor \pi \to e \rfloor &\triangleq \|\pi\| \to \|e\| \\
\lfloor e\ /\ \&\bullet r : \tau = \bullet s \rfloor &\triangleq \|e\|\ /\ \&r : \tau = s & \lfloor \{\varphi\}\ e \rfloor &\triangleq \|e\| \\
\lfloor e\ /\ \&\circ r : \tau = s \rfloor &\triangleq \|e\| & \lfloor \uparrow e \rfloor \triangleq \|e\| \quad \lfloor \downarrow e \rfloor &\triangleq \|e\|
\end{aligned}$$

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

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

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

Figure 8: Ghost code elimination.

Operators $\|\cdot\|$ and $\lfloor\cdot\rfloor$, defined in Figure 8, apply to expressions and type signatures. They eliminate assertions, barriers, transparent subexpressions, ghost terms, and ghost parameters. These operations 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

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.

[3] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles Of Programming Languages*, pages 193–205. ACM, 2001.