COMA, an Intermediate Verification Language with Explicit Abstraction Barriers

Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria,
Laboratoire Méthodes Formelles, F-91405 Gif-sur-Yvette

Abstract. We introduce COMA, a formally defined intermediate verification language. Specification annotations in COMA take the form of assertions mixed with the executable program code. A special programming construct representing the abstraction barrier is used to separate, inside a subroutine, the "interface" part of the code, which is verified at every call site, from the "implementation" part, which is verified only once, at the definition site. In comparison with traditional contract-based specification, this offers us an additional degree of freedom, as we can provide separate specification (or none at all) for different execution paths.
We define a verification condition generator for COMA and prove its correctness. For programs where specification is given in a traditional way, with abstraction barriers at the function entries and exits, our verification conditions are similar to the ones produced by a classical weakest-precondition calculus. For programs where abstraction barriers are placed in the middle of a function definition, the user-written specification is seamlessly completed with the verification conditions generated for the exposed part of the code. In addition, our procedure can factorize selected subgoals on the fly, which leads to more compact verification conditions.
We illustrate the use of COMA on two non-trivial examples, which have been formalized and verified using our implementation: a second-order regular expression engine and a sorting algorithm written in unstructured assembly code.

1 Introduction

Consider a simple function, written in some ML dialect, which eliminates the root node from a binary tree, using an existing library function that merges two trees in one:

```
type tree = Node tree elt tree | Empty

let removeRoot (t: tree) : tree
= match t with
  | Node l _ r → mergeTree l r
  | Empty      → fail
```

If we want to use `removeRoot` in a formally verified program, we need to provide this code with a specification. In a traditional contract-based approach, this means writing a precondition and a postcondition, and here is how they would usually look:

```
let removeRoot (t: tree) : tree
  requires { t ≠ Empty }
  ensures  { match t with
             | Node l _ r → ∀e:elt. e ∈ result ↔ e ∈ l ∨ e ∈ r
             | Empty      → false }
```

While this contract does its job, it is rather unpleasant. Not only does it take more space than the code it describes, it also basically repeats what is already written in the code. What is more, if we compute a verification condition (VC, for brevity) for the definition of `removeRoot`, it will take the form of one `match-with` formula implying another—or maybe two nested `match-with` formulas—and neither is easy to read and to prove.

Some programming languages, like Haskell and Agda, admit multiclause function definitions, and it is tempting to write our specification in this way, too:

```
removeRoot (Node l _ r)
  ensures { ∀e:elt. e ∈ result ↔ e ∈ l ∨ e ∈ r }
= mergeTree l r

removeRoot Empty = fail
```

This definition is much nicer. The postcondition in the first clause can refer to the results of the top-level pattern matching and does not need to do one itself. Furthermore, the second clause is self-explainable, so that we can omit the specification altogether.

However, from the verification point of view, something unusual is happening here. As we push the postcondition down the first branch of the pattern matching, we expose a part of the implementation (namely, the pattern matching itself) to the client code. Whenever `removeRoot` is called in our program, the VC for that call needs to perform the case analysis on the tree parameter in order to access the postcondition. Even more drastically, the second branch contains no specification at all, and so the caller's VC has to "inline" the entire second clause at the call site and prove that it is never reached.

What we did in this definition of `removeRoot`, is we moved the *abstraction barrier* inwards from the entry-exit boundary of a function, and even omitted it entirely on some of the execution paths. The question is, what are the rules of VC generation for programs with freely moving abstraction barriers? What if we do more in the exposed part of the code than just pattern matching or failure?

In this paper, we propose a formalism for computation and specification that intends to answer this question. We present a programming language called Coma that is small enough to comfortably study its properties, yet expressive enough to serve as a practical intermediate verification language for real-life applications. Coma programs are written in the continuation-passing style—the name Coma is short for Continuation Machine—which allows us to capture with just a few constructions the standard control structures: sequence, conditionals, loops, function calls, exception handling.

Specification annotations in Coma take the form of assertions mixed with executable code. Abstraction barriers are made explicit, as special tags that separate the "interface" part of a subroutine, which is verified at every call site, from the "implementation" part, which is verified only once, at the definition site.

Coma is a higher-order language, and the verification conditions for it are propositions in higher-order logic. These verification conditions can be reduced to first-order formulas, suitable for automated proving. For functions that are specified in a traditional manner, with the specification at the entry-exit boundary, the resulting formulas are close to those produced by a classical weakest-precondition calculus. However, we can also benefit from the initial higher-order form of our verification conditions, and factorize selected subformulas in the process of reduction. In this manner, we can curb the well-known exponential growth of classical weakest preconditions, and obtain proof obligations similar to the compact verification conditions of Flanagan and Saxe [7].

We implemented a VC generator for Coma programs and performed several case studies, two of which are presented below. While in this paper we focus on the pure fragment, our implementation also supports first-class alias-free mutable variables with effect inference and monadic translation into the pure core language. This implementation currently serves as a back-end for Creusot, a tool for deductive verification of Rust programs [5]. Of particular interest for Creusot is Coma's ability to automatically infer the contracts of simple Rust closures (anonymous functions).

To summarize, here are the main contributions of our work:

- an intermediate verification language with higher-order functions and explicit abstraction barriers (Section 2);
- a formally defined and proved verification condition generator (Sections 3 and 4);
- a working implementation with numerous added features, including alias-free mutable variables, rich function prototypes with specification annotations and variable binding, compact VC formulas via subgoal factorization, etc. (Section 5);
- two non-trivial case studies: a second-order regular expression engine (Section 6) and a sorting algorithm written in x86-64 assembly code (Section 7).

2 Syntax and Semantics of Coma

The building blocks of Coma are *expressions*, which perform computations, and *terms*, which represent data. Expressions and terms are distinct syntactic entities: a term can be passed as an argument to an expression, but an expression cannot reduce to a term. An expression can be encapsulated in a named or anonymous *handler* (which is what we call subroutines), 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. In theory, it would be possible to restrict the syntax of terms to variables and literal values, and delegate all computation to handlers, either predefined or introduced by the user. Still, for the sake of convenience, we admit in terms a handful of basic operations on unbounded integers, Booleans, and polymorphic finite sequences and binary trees. To handle type polymorphism, we treat types as a special kind of data: type expressions are considered to be terms of type `Type`, and we do not make a formal distinction between term and type variables. We denote variables with letters x, y, α, β (the latter two being reserved for types), and terms with s, t, τ, θ (again, the latter two being reserved for types). For specification, we use first-order formulas, denoted φ and ψ, which may contain variables and terms, but not handlers. By a slight abuse of notation, Boolean terms are accepted as atomic formulas.

type signature:	π, ϱ	::=	$\overline{x : \tau}\ \ \overline{g : \varrho}$	parameter list
handler:	k, o	::=	h	handler symbol
		|	$\pi \rightarrow d$	anonymous handler
expression:	e, d	::=	$k\ \bar{s}\ \bar{o}$	handler call
		|	$e\ /\ h\ \pi = d$	handler definition
		|	$\{\varphi\}\ e$	assertion
		|	$\uparrow e$	black-box barrier
		|	$\downarrow e$	white-box barrier

Fig. 1: Handlers and expressions.

Handlers accept term parameters (which includes type parameters) and handler parameters, also called continuation parameters or *outcomes*. The list of formal parameters of a handler is called its *type signature*. Since we adopt the continuation-passing style, handlers do not have return values. Handlers that have no parameters are said to have a *void type signature*, written with the symbol □. We use letters π and ϱ to denote type signatures, and letters h, g, f for handler names.

We assume to have access to a number of predefined *primitive handlers*, which form the "standard library" of Coma. Here are the type signatures of five primitive handlers that we use throughout this paper:

```
    if : (c:bool) (then:□) (else:□)
unTree : (α:Type) (t:tree α) (onNode:(l:tree α) (v:α) (r:tree α)) (onEmpty:□)
   get : (α:Type) (s:seq α) (i:int) (return:(v:α))
  halt : □
  fail : □
```

Handler `if` makes a choice between two continuations, represented by nullary outcomes `then` and `else`, depending on the condition `c`. Handler `unTree` inspects a binary tree `t`: if it is a node, then its datum and two subtrees are passed to the `onNode` continuation, otherwise `onEmpty` is called. Handler `get` retrieves the `i`-th element of sequence `s` and passes it to the continuation. This operation is allowed only when `i` is a valid index of `s`. Handler `halt` stops the computation. Finally, `fail` is an equivalent of `assert false`, it represents code that should never be reached in execution.

By allowing Coma computations to have multiple outcomes, 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 `halt` and `fail` are also noteworthy in this regard: as they do not accept continuation parameters, we know simply by looking at their signature that they cannot ever return control to the caller.

Type signatures are identified modulo parameter renaming. For example, the signature of `get` can be equivalently written as `(β:Type) (l:seq β) (k:int) (ret:(e:β))`. Each parameter binds the corresponding symbol in the types of subsequent parameters. This only matters for variables, as handler symbols cannot occur in type annotations.

$$\frac{h : \pi \in \Gamma}{\Gamma \vdash h : \pi}\ (\text{T-Sym}) \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \square \to e : \square}\ (\text{T-Void})$$

$$\frac{\Gamma \vdash k\ \bar{s} : (x : \tau)\, \pi \qquad \Gamma \vdash t : \tau}{\Gamma \vdash k\ \bar{s}\ t : \pi[x \mapsto t]}\ (\text{T-AppT}) \qquad \frac{\Gamma, x : \tau \vdash \pi \to e : \pi}{\Gamma \vdash (x : \tau)\, \pi \to e : (x : \tau)\, \pi}\ (\text{T-ParT})$$

$$\frac{\Gamma \vdash k\ \bar{s}\ \bar{o} : (g : \varrho)\, \pi \qquad \Gamma \vdash k' : \varrho}{\Gamma \vdash k\ \bar{s}\ \bar{o}\ k' : \pi}\ (\text{T-AppH}) \qquad \frac{\Gamma, g : \varrho \vdash \pi \to e : \pi}{\Gamma \vdash (g : \varrho)\, \pi \to e : (g : \varrho)\, \pi}\ (\text{T-ParH})$$

$$\frac{\Gamma \vdash \varphi : \text{Prop} \qquad \Gamma \vdash e : \square}{\Gamma \vdash \{\varphi\}\, e : \square}\ (\text{T-Prop}) \qquad \frac{\Gamma, h : \pi \vdash \pi \to d : \pi \qquad \Gamma, h : \pi \vdash e : \square}{\Gamma \vdash e\ /\ h\, \pi = d : \square}\ (\text{T-Defn})$$

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

Fig. 2: Typing rules for expressions.

The *order* of a type signature π (and, by extension, of any handler with that signature) is defined recursively: if π has no continuation parameters, it is of order zero; otherwise, the order of π is one plus the highest order of its outcomes. The length and order of a type signature are invariant with respect to type instantiation: handlers can be polymorphic only in the data types.

Figure 1 presents the syntax of Coma expressions. An expression is an application of a named or anonymous handler to a list of arguments, on top of which we can put recursive *handler definitions*, logical *assertions*, and two *barriers*, denoted $\uparrow$ and $\downarrow$, and called *black-box* and *white-box*, respectively. Handler definitions are placed to the right of the underlying expression; the slash symbol can be read as "where". The barriers guide the generation of verification conditions, and have no effect on execution. The black-box barrier is the abstraction barrier, which separates the exposed "interface" part of a handler definition from the hidden "implementation" part. The white-box barrier is an auxiliary construction that exposes the whole underlying expression. We use letters e and d to denote expressions, and letters k and o to denote handlers.

Type signatures serve as types for expressions, enumerating the expected arguments; in particular, a fully applied handler has type $\square$. *Typing contexts*, denoted Γ and Δ, are sequences of type bindings of the form $x : \tau$ and $g : \varrho$. A typing context is *well-formed* if no symbol is bound twice, and every variable type either is `Type` or has type `Type` with respect to the preceding bindings.

The typing rules for expressions are given in Fig. 2. In a judgement $\Gamma \vdash e : \pi$, the typing context is implicitly required to be well-formed. We consider as given the typing relations for terms and formulas, respectively denoted $\Gamma \vdash s : \tau$ and $\Gamma \vdash \varphi : \text{Prop}$; refer to Appendix A for the fragment used in this paper. Notice that bodies of handler definitions and anonymous handlers have to be fully applied. Thus, an anonymous handler $\pi \to d$ always has type π, modulo parameter renaming. As with type signatures, we identify expressions modulo renaming of bound variables and handler symbols.

The *initial typing context* Γ_{prim} binds the primitive handlers to their respective type signatures. An expression e is called a *program* when $\Gamma_{\text{prim}} \vdash e : \square$.

$$\frac{h\,\pi = d \in \Lambda}{h\,\bar{s}\,\bar{o} \mathbin{/\!/} \Lambda \longrightarrow (\pi \rightarrow d)\,\bar{s}\,\bar{o} \mathbin{/\!/} \Lambda} \quad \text{(E-Sym)}$$

$$((x:\tau)\,\pi \rightarrow e)\;t\;\bar{s}\,\bar{o} \mathbin{/\!/} \Lambda \longrightarrow (\pi \rightarrow e)[x \mapsto t]\;\bar{s}\,\bar{o} \mathbin{/\!/} \Lambda \quad \text{(E-AppT)}$$

$$((g:\varrho)\,\pi \rightarrow e)\;f\;\bar{o} \mathbin{/\!/} \Lambda \longrightarrow (\pi \rightarrow e)[g \mapsto f]\;\bar{o} \mathbin{/\!/} \Lambda \quad \text{(E-AppH)}$$

$$((g:\varrho)\,\pi \rightarrow e)\;(\varrho \rightarrow d)\;\bar{o} \mathbin{/\!/} \Lambda \longrightarrow (\pi \rightarrow e)\;\bar{o} \mathbin{/} g\,\varrho = \downarrow d \mathbin{/\!/} \Lambda \quad \text{(E-AppC)}$$

$$\square \rightarrow e \mathbin{/\!/} \Lambda \longrightarrow e \mathbin{/\!/} \Lambda \quad \text{(E-Void)} \qquad \frac{\vDash \varphi}{\{\varphi\}\;e \mathbin{/\!/} \Lambda \longrightarrow e \mathbin{/\!/} \Lambda} \quad \text{(E-Prop)}$$

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

$$\downarrow e \mathbin{/\!/} \Lambda \longrightarrow e \mathbin{/\!/} \Lambda \quad \text{(E-Wbox)} \qquad \frac{h \text{ is not free in } e}{e \mathbin{/} h\,\pi = d \mathbin{/\!/} \Lambda \longrightarrow e \mathbin{/\!/} \Lambda} \quad \text{(E-Gc)}$$

$$\frac{\vDash s}{\texttt{if}\;s\;k\;o \mathbin{/\!/} \Lambda \longrightarrow k \mathbin{/\!/} \Lambda} \qquad \frac{\vDash \neg s}{\texttt{if}\;s\;k\;o \mathbin{/\!/} \Lambda \longrightarrow o \mathbin{/\!/} \Lambda}$$

$$\frac{\vDash t = \texttt{Node}\;s_1\;s_2\;s_3}{\texttt{unTree}\;\tau\;t\;k\;o \mathbin{/\!/} \Lambda \longrightarrow k\;s_1\;s_2\;s_3 \mathbin{/\!/} \Lambda} \qquad \frac{\vDash t = \texttt{Empty}}{\texttt{unTree}\;\tau\;t\;k\;o \mathbin{/\!/} \Lambda \longrightarrow o \mathbin{/\!/} \Lambda}$$

$$\frac{\vDash \texttt{0} \le s_2 < \texttt{length}\;s_1 \qquad \vDash s_1[s_2..s_2\texttt{+1}] = [t]}{\texttt{get}\;\tau\;s_1\;s_2\;k \mathbin{/\!/} \Lambda \longrightarrow k\;t \mathbin{/\!/} \Lambda}$$

Fig. 3: Operational semantics.

We define a small-step operational semantics for Coma as a reduction relation $\longrightarrow$. The reduction rules are shown in Fig. 3. We write $e \mathbin{/\!/} \Lambda$ to denote series of nested handler definitions, where Λ is understood as a sequence of definitions, possibly empty.

The rule E-Sym expands handler definitions. We assume that no handler is defined in Λ twice, as we can always rename bound handlers. The rules E-AppT and E-AppH perform β-reduction. The rule E-AppC turns an anonymous handler argument into a local handler definition. This is done in a capture-safe manner: we expect that the handler symbol g does not occur freely in d or in $\bar{o}$. The barrier over d is needed to preserve the verification condition of the program, as we show later.

The rule E-Prop requires the asserted formula φ to be valid before proceeding with the execution. The validity judgement $\vDash \varphi$ is made within the standard model for our data types: integers, Booleans, sequences and binary trees. Of course, the validity of an arbitrary proposition cannot be effectively verified in a practical implementation. However, our purpose here is different: we define the operational semantics of Coma in order to state and prove the correctness of our verification procedure—in particular, that a program with a valid verification condition cannot get stuck during its execution because of a failed assertion.

The rule E-Void replaces a nullary anonymous handler by its body. The barriers are ignored during execution (rules E-Bbox and E-Wbox). Finally, the rule E-Gc prunes the context by removing unreachable handler definitions. This rule commutes with the rest of the rules, making Coma non-deterministic, yet still strongly confluent.[1]

[1] Modulo semantic equality of answer terms during evaluation of primitive handlers; see below.

We also postulate the evaluation rules for the primitive handlers. As for E-PROP, the application of these rules depends on validity of logical properties that express the pre- and postcondition of the primitives. For example, the Boolean condition of an `if` must be valid for the evaluation to progress along the first outcome (as mentioned in the beginning of the section, we admit Boolean terms as atomic formulas). In the rules for `unTree`, the function symbols `Node` and `Empty` are the constructors of the `tree` type. In the rule for `get`, we use the (total) slice operator on sequence s_1 to isolate the element at the position s_2, which must be a valid index in s_1. The answer terms in the rules for `unTree` and `get` can be any ground terms that validate the rule premises: for example, the expression `get int [42] 0 return` $/\!/\,\Lambda$ can reduce both to `return 42` $/\!/\,\Lambda$ and `return 6*7` $/\!/\,\Lambda$. While we could introduce some form of normalization to avoid this syntactical divergence, there is no need for that, since all conditions in our evaluation rules are expressed in terms of semantic validity. Finally, `halt` and `fail` represent the final states of a computation and cannot be evaluated.

COMA is a type-safe language. Type preservation is easy to establish, either through a direct proof or by embedding in a more expressive framework like System F_ω or CoC. As for the progress property, the blocking semantics of assertions limits it to programs with a valid verification condition; and so we defer this subject until Section 4.

We conclude with two examples. First, let us revisit the `removeRoot` function:

```
removeRoot (t: tree) (return (s: tree)) =
  unTree t ((l: tree) (_: elt) (r: tree) →
              ↑ mergeTree l r ret
              / ret (s: tree) = { ∀e:elt. e ∈ s ↔ e ∈ l ∨ e ∈ r }
                                 ↑ return s)
           fail
```

The normal outcome of `removeRoot` becomes a continuation parameter named `return`, and the argument of `return` is the tree produced by `removeRoot`. The implementation of `removeRoot` starts with a case analysis on the tree parameter `t`, using the primitive `unTree` hanlder. The two branches of the case analysis are represented, respectively, by an anonymous handler, which is called when `t` is a binary node, and the `fail` primitive, invoked when `t` is `Empty`. The anonymous handler contains a call of `mergeTree`, which we assume to be available. The result of `mergeTree` is passed to a wrapper handler `ret` which states the postcondition of the first branch, before calling `return`.

In this translation of `removeRoot`, the call of `mergeTree` is hidden behind the black-box barrier. The rest of the code—the case analysis by `unTree`, the failure on an empty node, and the postcondition describing the return value—is exposed, and will appear in the verification conditions at the call sites of `removeRoot`.

In Fig. 4, we show the Russian Peasant Multiplication algorithm written in COMA. This code is specified in a more traditional manner: the entire implementation of the `product` handler is put behind the abstraction barrier. Left in the interface part are the starting assertion `{b ≥ 0}`, which naturally becomes the precondition of `product`, and the wrapper handler `break`, which plays the same role as `ret` in `removeTree`, and whose precondition `{c = a · b}` is the postcondition of `product`.

The implementation defines and calls a recursive handler named `loop`. This handler does not return to the caller: to do that, it would need to receive a continuation parameter

```
product (a b: int) (return (c: int))
= { b ≥ 0 }
  ↑ loop a b 0
    / loop (p q r: int)
      = { p · q + r = a · b ∧ q ≥ 0 }
        ↑ if (q > 0) (→ if (q mod 2 = 1) (→ next (r + p)) (→ next r)
                        / next (s: int) = loop (p + p) (q div 2) s)
                     (→ break r)
  / break (c: int) = { c = a · b } ↑ return c
```

Fig. 4: Russian Peasant Multiplication in Coma.

and call it, like `removeTree` and `product` do. Instead, `loop` escapes by calling `break` at the end of computation. In this respect, `loop` behaves indeed rather like a loop than a recursive function: its continuation is determined statically, by its lexical context, rather than dynamically by its caller. Consequently, there is no distinct postcondition associated to `loop`: in Coma, postconditions are preconditions of continuation parameters (attached via wrapper handlers like `ret` and `break`), and `loop` has none thereof. And the precondition of `loop`, placed above the barrier, is just the loop invariant.

In practice, the majority of Coma programs would be generated by mechanical translation from existing languages like OCaml or Rust. Part of this translation would be a CPS transformation, required by our language. While in most cases, this transformation is not problematic, and allows us to reduce a large number of control structures to just two—definitions and calls—there are limits to what can be easily translated into Coma. Consider, for example, an OCaml exception that carries a closure:

```
exception E of (int → int)
```

In Coma, exception-raising functions are written as handlers that have multiple continuation parameters: one for the normal outcome, and one for each exception that might be raised in the handler code. However, if the closures passed with the exception `E` were themselves liable to raise `E`, we would not be able to give them a finite type in Coma. Incidentally, it is not a coincidence that higher-order exceptions can be used to realize fixed point computations without explicit recursion.

3 The Logic of Recipes

In their final form, verification conditions for Coma programs are first-order logical formulas, which we can handle with the usual methods of automated and interactive theorem proving. Their generation, however, goes through an intermediate stage, where a preliminary higher-order verification condition, called *recipe*, is constructed and then transformed, deterministically and in a finite number of steps, into the first-order form.

Recipes are formulas in a particular variety of higher-order logic, where bound predicate variables represent verification conditions of individual handlers and can only appear in a positive position. We denote recipes with letters Φ, Ψ, Υ. The syntax of

$$\begin{array}{rcl} \Phi, \Psi, \Upsilon & ::= & h \mid \Phi\ s \mid \lambda\, x : \tau\, .\, \Phi \mid \forall\, x : \tau\, .\, \Phi \mid \Phi \wedge \Psi \\ & \mid & \mathbf{0} \mid \Phi\ \Psi \mid \lambda\, g : \varrho\, .\, \Phi \mid \varphi \to \Phi \mid \natural\, \Phi \end{array}$$

$$\forall h : \pi\, .\, \Phi \triangleq (\lambda h : \pi\, .\, \Phi)\ \mathbf{0}_\pi$$

$$\mathbf{0}_\pi \triangleq \lambda\pi\, .\, \mathbf{0} \wedge \bigwedge_{(f : \overline{x : \tau}\ \overline{g : \varrho}) \in \pi} \forall \overline{x : \tau} . \forall \overline{g : \varrho} . f\ \bar{x}\ \bar{g}$$

Fig. 5: Preliminary verification conditions (recipes).

$$\frac{}{\Gamma \vdash \mathbf{0} : \square}\ (\text{TC-Fail}) \qquad \frac{\Gamma \vdash \Phi : \pi \qquad \Gamma \vdash \Psi : \pi}{\Gamma \vdash \Phi \wedge \Psi : \pi}\ (\text{TC-Conj})$$

$$\frac{h : \pi \in \Gamma}{\Gamma \vdash h : \pi}\ (\text{TC-Sym}) \qquad \frac{\Gamma \vdash \varphi : \text{Prop} \qquad \Gamma \vdash \Phi : \square}{\Gamma \vdash \varphi \to \Phi : \square}\ (\text{TC-Impl})$$

$$\frac{\Gamma \vdash \Phi : \pi}{\Gamma \vdash \natural\Phi : \pi}\ (\text{TC-Neu}) \qquad \frac{\Gamma, x : \tau \vdash \Phi : \square}{\Gamma \vdash \forall x : \tau . \Phi : \square}\ (\text{TC-AllT})$$

$$\frac{\Gamma \vdash \Phi : (x : \tau)\, \pi \qquad \Gamma \vdash s : \tau}{\Gamma \vdash \Phi\ s : \pi[x \mapsto s]}\ (\text{TC-AppT}) \qquad \frac{\Gamma, x : \tau \vdash \Phi : \pi}{\Gamma \vdash \lambda x : \tau . \Phi : (x : \tau)\, \pi}\ (\text{TC-ParT})$$

$$\frac{\Gamma \vdash \Phi : (g : \varrho)\, \pi \qquad \Gamma \vdash \Psi : \varrho}{\Gamma \vdash \Phi\ \Psi : \pi}\ (\text{TC-AppC}) \qquad \frac{\Gamma, g : \varrho \vdash \Phi : \pi}{\Gamma \vdash \lambda g : \varrho . \Phi : (g : \varrho)\, \pi}\ (\text{TC-ParC})$$

Fig. 6: Typing rules for recipes.

recipes is given in Fig. 5. In recipes, handler symbols become predicate variables of the same name and arity as the original handler. The symbol $\mathbf{0}$ is the verification condition of `fail`, a logical contradiction. The *neutralization operator*, denoted $\natural$, suppresses proof obligations in the underlying recipe. Finally, notice that the antecedent in an implication is not a recipe, but a first-order formula, which cannot have occurrences of handler symbols. We write $\lambda\pi.\Phi$ and $\forall\pi.\Phi$ to denote a series of nested λ-abstractions or quantifications. By convention, $\lambda\square.\Phi$ and $\forall\square.\Phi$ are the same as Φ.

Universal quantification over a predicate variable is defined recursively, as an instantiation with a *joker recipe* $\mathbf{0}_\pi$. A joker recipe is the verification condition of a handler of which nothing is known: on any input, the handler may fail or it may call any of its outcomes with arbitrary arguments. On a void type signature, the joker $\mathbf{0}_\square$ is simply $\mathbf{0}$.

A fully applied recipe is a logical proposition, which is why we disregard the result type (that is, Prop) and use type signatures once again as types for predicate variables and recipes. The typing rules are given in Fig. 6. The typing contexts are the same as for Coma expressions, and, like before, are implicitly required to be well-formed. Notice that conjunction applies to any two recipes of the same type, and not just to fully applied recipes. We identify recipes modulo renaming of bound symbols.

The semantics of recipes is given by means of a Krivine-style abstract machine [10] that converts a fully applied recipe into a first-order formula, where all bound predicate variables are eliminated. We have chosen this approach both for theoretical and practical reasons. First, the properties of recipes are naturally proved using logical relations [15], which are straightforward to express in this setting. Second, our implementation of a verification condition generator for Coma is based on the same abstract machine. In the rest of the section, we introduce this evaluator and establish some of its properties.

A *cell* is a triplet $\langle b, \Sigma, \Phi \rangle$, where Φ is a recipe, Σ a *cell context*, binding every free handler symbol in Φ to a cell, and b a Boolean value. Such a cell can be converted into a recipe by replacing the free handler symbols in Φ with the corresponding converted cells from Σ. We assign the type signature of the resulting recipe to the initial cell. In what follows, we denote cells with letters C and D, and assume that all cells and recipes under consideration are well-typed.

The *depth* of a cell $C = \langle b, \Sigma, \Phi \rangle$ is zero if its cell context Σ is empty; otherwise, it is one plus the maximum depth of the cells in Σ. The *neutralization* of C, denoted $\natural C$, is the cell $\langle \top, \natural\Sigma, \Phi \rangle$, where $\natural\Sigma$ is obtained by neutralizing every cell in Σ. Obviously, neutralization does not affect the type or depth of a cell.

We associate a specification recipe to each primitive handler:

$$
\begin{aligned}
\Psi_{\mathtt{if}} &\triangleq \lambda\mathtt{c}:\mathtt{bool}.\lambda\mathtt{then}:\square.\lambda\mathtt{else}:\square.\,(\mathtt{c} \to \mathtt{then}) \land (\neg\mathtt{c} \to \mathtt{else}) \\
\Psi_{\mathtt{unTree}} &\triangleq \lambda\alpha:\mathtt{Type}.\lambda\mathtt{t}:\mathtt{tree}\,\alpha.\lambda\mathtt{onNode}:(\mathtt{l}:\mathtt{tree}\,\alpha)(\mathtt{v}:\alpha)(\mathtt{r}:\mathtt{tree}\,\alpha).\lambda\mathtt{onEmpty}:\square. \\
&\qquad (\forall\mathtt{l}:\mathtt{tree}\,\alpha.\forall\mathtt{v}:\alpha.\forall\mathtt{r}:\mathtt{tree}\,\alpha.\,\mathtt{t} = \mathtt{Node}\,\mathtt{l}\,\mathtt{v}\,\mathtt{r} \to \mathtt{onNode}\,\mathtt{l}\,\mathtt{v}\,\mathtt{r}) \land \\
&\qquad (\mathtt{t} = \mathtt{Empty} \to \mathtt{onEmpty}) \\
\Psi_{\mathtt{get}} &\triangleq \lambda\alpha:\mathtt{Type}.\lambda\mathtt{s}:\mathtt{seq}\,\alpha.\lambda\mathtt{i}:\mathtt{int}.\lambda\mathtt{return}:(\mathtt{v}:\alpha). \\
&\qquad 0 \le \mathtt{i} < \mathtt{length}\,\mathtt{s} \land \forall\mathtt{v}:\alpha.\,\mathtt{s[i..i+1]} = [\mathtt{v}] \to \mathtt{return}\,\mathtt{v} \\
\Psi_{\mathtt{halt}} &\triangleq \natural\mathbf{0} \\
\Psi_{\mathtt{fail}} &\triangleq \mathbf{0} \\
\Sigma_{\mathrm{prim}} &\triangleq [\mathtt{if} \mapsto \langle\bot,\varnothing,\Psi_{\mathtt{if}}\rangle, \mathtt{unTree} \mapsto \langle\bot,\varnothing,\Psi_{\mathtt{unTree}}\rangle, \mathtt{get} \mapsto \langle\bot,\varnothing,\Psi_{\mathtt{get}}\rangle, \\
&\qquad\qquad \mathtt{halt} \mapsto \langle\bot,\varnothing,\Psi_{\mathtt{halt}}\rangle, \mathtt{fail} \mapsto \langle\bot,\varnothing,\Psi_{\mathtt{fail}}\rangle]
\end{aligned}
$$

The *initial cell context* Σ_{prim} binds primitive handlers to their respective specifications.

A *stack* is a mixed sequence of terms and cells. An empty stack is denoted ε. The neutralization of a stack ℓ, denoted $\natural\ell$, is obtained by neutralizing every cell in ℓ. We say that a stack is *aligned* with a cell, when the length of the stack and the types of its elements coincide with the cell's signature. In other words, an aligned stack contains appropriate arguments for the cell.

An n-ary relation R on same-typed cells holds on stacks $\ell_1, \ldots, \ell_n$ when they all have the same length and type signature, and for each position i, if $\ell_{1i}, \ldots, \ell_{ni}$ are terms, then they are all identical, and if they are cells, then both $R(\ell_{1i}, \ldots, \ell_{ni})$ and $R(\natural\ell_{1i}, \ldots, \natural\ell_{ni})$ are true. Similarly, R holds on cell contexts $\Sigma_1, \ldots, \Sigma_n$ when they bind the same handler names, and for every bound h, both $R(\Sigma_1(h), \ldots, \Sigma_n(h))$ and $R(\natural\Sigma_1(h), \ldots, \natural\Sigma_n(h))$ are true. It is easy to see that $R(\Sigma_1, \ldots, \Sigma_n)$ implies $R(\natural\Sigma_1, \ldots, \natural\Sigma_n)$, as $\natural\natural C = \natural C$.

As a special case of the above, any property of cells is said to hold for a stack ℓ or a cell context Σ whenever for every cell C in ℓ or Σ, both C and $\natural C$ have this property. Furthermore, if the property holds for a cell context Σ, then it also holds for $\natural\Sigma$.

The evaluation operator $\circ$, defined in Fig. 7, applies a cell to an aligned stack and produces a first-order logical formula. In the rule for $\forall x:\tau.\Phi$, we assume that x does not occur in the cell context Σ, to avoid collisions.

$$\langle b,\Sigma,\mathbf{0}\rangle\circ\varepsilon \triangleq b \qquad \langle b,\Sigma,\Phi\wedge\Psi\rangle\circ\ell \triangleq \langle b,\Sigma,\Phi\rangle\circ\ell\wedge\langle b,\Sigma,\Psi\rangle\circ\ell$$
$$\langle b,\Sigma,h\rangle\circ\ell \triangleq \Sigma(h)\circ\ell \qquad \langle b,\Sigma,\varphi\to\Phi\rangle\circ\varepsilon \triangleq \varphi\to\langle b,\Sigma,\Phi\rangle\circ\varepsilon$$
$$\langle b,\Sigma,\natural\Phi\rangle\circ\ell \triangleq \langle\top,\natural\Sigma,\Phi\rangle\circ\ell \qquad \langle b,\Sigma,\forall x\!:\!\tau\,.\,\Phi\rangle\circ\varepsilon \triangleq \forall x\!:\!\tau\,.\,\langle b,\Sigma,\Phi\rangle\circ\varepsilon$$
$$\langle b,\Sigma,\Phi\,s\rangle\circ\ell \triangleq \langle b,\Sigma,\Phi\rangle\circ s,\ell \qquad \langle b,\Sigma,\lambda x\!:\!\tau\,.\,\Phi\rangle\circ s,\ell \triangleq \langle b,\Sigma,\Phi[x\mapsto s]\rangle\circ\ell$$
$$\langle b,\Sigma,\Phi\,\Psi\rangle\circ\ell \triangleq \langle b,\Sigma,\Phi\rangle\circ\langle b,\Sigma,\Psi\rangle,\ell \qquad \langle b,\Sigma,\lambda h\!:\!\pi\,.\,\Phi\rangle\circ C,\ell \triangleq \langle b,\Sigma\uplus[h\mapsto C],\Phi\rangle\circ\ell$$

Fig. 7: Recipe evaluation.

Theorem 1. *The evaluation operator* $\circ$ *is defined on all aligned cells and stacks.*

Proof. We say that a cell C is *normalizing* if for any aligned normalizing stack ℓ, the evaluation $C\circ\ell$ is defined. This definition is well-founded, because every cell in ℓ is of lower order than C. Then we need to show that all cells (and, therefore, all stacks) are normalizing. In fact, it suffices to prove that every cell $\langle b,\Sigma,\Upsilon\rangle$ is normalizing, if its cell context Σ is normalizing. Once this is established, a simple induction over cell depth allows us to conclude.

We proceed by induction over the size of Υ, counting only the subrecipes, so that term substitutions do not affect the size. Take an arbitrary aligned normalizing stack ℓ.

Case Υ is $\mathbf{0}$. As $\mathbf{0}$ is $\square$-typed, ℓ has to be empty, and $\langle b,\Sigma,\mathbf{0}\rangle\circ\varepsilon$ is defined.

Case Υ is h. Since every cell in Σ is normalizing, the evaluation $\Sigma(h)\circ\ell$ is defined.

Case Υ is $\natural\Phi$. The context $\natural\Sigma$ is normalizing, and the induction hypothesis applies.

Case Υ is $\Phi\,\Psi$. Let D be $\langle b,\Sigma,\Psi\rangle$. Since $\natural\Sigma$ is normalizing, both cells, D and $\natural D$, are normalizing by the induction hypothesis. As the cell $\langle b,\Sigma,\Phi\rangle$ is also normalizing by the induction hypothesis, the evaluation $\langle b,\Sigma,\Phi\rangle\circ D,\ell$ is defined.

Case Υ is $\lambda h\!:\!\pi\,.\,\Phi$. Then the stack ℓ is of the form D,ℓ', where both D and $\natural D$ are normalizing. Thus, $\Sigma\uplus[h\mapsto D]$ is normalizing and the induction hypothesis applies.

In every other case, we pick a rule for $\circ$ and apply the induction hypothesis. ◻

A cell C is said to be *neutral*, if for any aligned neutral stack ℓ, the formula $C\circ\ell$ is valid. Just as above, this recursive definition is well-founded, because every cell in ℓ is of lower order than C.

Lemma 1. *Any neutralized cell* $\natural C$ *is neutral.*

Proof. We proceed in the same way as in Theorem 1, via an intermediate lemma stating that any cell of the form $\langle\top,\Sigma,\Upsilon\rangle$ is neutral, if its cell context Σ is neutral. ◻

A cell C_1 *entails* C_2, denoted $C_1 \Rrightarrow C_2$, when they have the same type and for any aligned stacks ℓ_1 and ℓ_2 such that $\ell_1 \Rrightarrow \ell_2$, we have $C_1\circ\ell_1 \Rightarrow C_2\circ\ell_2$. Here and below, the symbol $\Rightarrow$ stands for logical consequence, and $\Leftrightarrow$ for logical equivalence, under the same standard model used for assertions. Cell C_1 is *equivalent* to C_2, denoted $C_1\equiv C_2$, when $C_1 \Rrightarrow C_2$ and $C_2 \Rrightarrow C_1$.

Lemma 2. *Cell entailment is reflexive and transitive.*

Proof. We prove reflexivity as in Theorem 1 and Lemma 1, via an intermediate lemma stating that for any same-typed $C_1 = \langle b, \Sigma_1, \Upsilon \rangle$ and $C_2 = \langle b, \Sigma_2, \Upsilon \rangle$ where $\Sigma_1 \Rrightarrow \Sigma_2$, we have $C_1 \Rrightarrow C_2$. To establish transitivity, given $C_1 \Rrightarrow C_2$ and $C_2 \Rrightarrow C_3$, we consider aligned stacks ℓ_1 and ℓ_2 such that $\ell_1 \Rrightarrow \ell_2$. Then we have $C_1 \circ \ell_1 \Rightarrow C_2 \circ \ell_2$ by definition, and $C_2 \circ \ell_2 \Rightarrow C_3 \circ \ell_2$, since $\ell_2 \Rrightarrow \ell_2$. □

Lemma 3. *Consider two cells of the same type, C_1 and C_2, such that for any aligned stack ℓ, we have $C_1 \circ \ell \Rightarrow C_2 \circ \ell$. Then $C_1 \Rrightarrow C_2$.*

Proof. Let ℓ_1 and ℓ_2 be aligned stacks such that $\ell_1 \Rrightarrow \ell_2$. Then $C_1 \circ \ell_1 \Rightarrow C_1 \circ \ell_2$ by Lemma 2, and $C_1 \circ \ell_2 \Rightarrow C_2 \circ \ell_2$ by the lemma assumption. □

Note that $C_1 \Rrightarrow C_2$ does not imply $\natural C_1 \Rrightarrow \natural C_2$. For example, the cell $\langle \bot, \varnothing, \lambda g : \Box . \mathbf{0} \rangle$ entails $\langle \bot, \varnothing, \lambda g : \Box . g \rangle$, yet, when we apply their neutralizations to $\langle \bot, \varnothing, \mathbf{0} \rangle$, we obtain $\top$ and $\bot$, respectively. Thus, to establish $\ell_1 \Rrightarrow \ell_2$, we must show pairwise entailment not only for the cells in the two stacks, but also for their neutralizations. The same applies to cell contexts.

Given three cells $C_1 = \langle b_1, \Sigma_1, \Phi \rangle$, $C_2 = \langle b_2, \Sigma_2, \Phi \rangle$, and $C_3 = \langle b_3, \Sigma_3, \Phi \rangle$ that have the same type and the same recipe Φ, we say that C_1 is the *fusion* of C_2 and C_3 when $b_1 = b_2 \wedge b_3$ and Σ_1 is the fusion of Σ_2 and Σ_3. Quite obviously, if C_1 is the fusion of C_2 and C_3, then $\natural C_1$ is the fusion of $\natural C_2$ and $\natural C_3$ (all three are actually the same). Furthermore, any cell C is the fusion of itself and $\natural C$.

A cell C_1 is a *meet* of C_2 and C_3 if they all have the same type, the neutralized cells $\natural C_2$ and $\natural C_3$ are equivalent, and for any aligned stacks ℓ_1, ℓ_2, ℓ_3 such that ℓ_1 is a meet of ℓ_2 and ℓ_3, we have $C_1 \circ \ell_1 \Leftrightarrow C_2 \circ \ell_2 \wedge C_3 \circ \ell_3$.

Lemma 4. *If C_1 is the fusion of C_2 and C_3, then C_1 is a meet of C_2 and C_3.*

Proof. We proceed as in Theorem 1, via an intermediate lemma stating that for all same-typed $C_1 = \langle b_1, \Sigma_1, \Upsilon \rangle$, $C_2 = \langle b_2, \Sigma_2, \Upsilon \rangle$, and $C_3 = \langle b_3, \Sigma_3, \Upsilon \rangle$ where $b_1 = b_2 \wedge b_3$ and Σ_1 is a meet of Σ_2 and Σ_3, cell C_1 is a meet of C_2 and C_3. The equivalence of $\natural C_2$ and $\natural C_3$ directly follows from the intermediate lemma in the proof of Lemma 2. □

Corollary 1. *For any cell C and aligned stack ℓ, we have $C \circ \ell \Leftrightarrow \natural C \circ \ell \wedge C \circ \natural \ell$.*

Corollary 2. *Any cell C entails $\natural C$.*

Lemma 5. *Consider cells C_1, C_2, C_3 of the same type, such that $\natural C_2 \equiv \natural C_3$ and for any aligned stack ℓ, we have $C_1 \circ \ell \Leftrightarrow C_2 \circ \ell \wedge C_3 \circ \ell$. Then C_1 is a meet of C_2 and C_3.*

Proof. Consider aligned stacks ℓ_1, ℓ_2, ℓ_3 such that ℓ_1 is a meet of ℓ_2 and ℓ_3.

$$\begin{aligned}
C_1 \circ \ell_1 &\Leftrightarrow C_1 \circ \ell_2 \wedge C_1 \circ \ell_3 && \text{(Lemma 4)}\\
&\Leftrightarrow C_2 \circ \ell_2 \wedge C_3 \circ \ell_2 \wedge C_2 \circ \ell_3 \wedge C_3 \circ \ell_3 && \text{(assumption)}\\
&\Leftrightarrow \natural C_2 \circ \ell_2 \wedge C_2 \circ \natural \ell_2 \wedge \natural C_3 \circ \ell_2 \wedge C_3 \circ \natural \ell_2 \wedge && \text{(Corollary 1)}\\
&\qquad \natural C_2 \circ \ell_3 \wedge C_2 \circ \natural \ell_3 \wedge \natural C_3 \circ \ell_3 \wedge C_3 \circ \natural \ell_3 &&\\
&\Leftrightarrow \natural C_2 \circ \ell_2 \wedge C_2 \circ \natural \ell_2 \wedge \natural C_2 \circ \ell_2 \wedge C_3 \circ \natural \ell_3 \wedge && (\natural C_2 \equiv \natural C_3,\\
&\qquad \natural C_3 \circ \ell_3 \wedge C_2 \circ \natural \ell_2 \wedge \natural C_3 \circ \ell_3 \wedge C_3 \circ \natural \ell_3 && \natural \ell_2 \equiv \natural \ell_3)\\
&\Leftrightarrow \natural C_2 \circ \ell_2 \wedge C_2 \circ \natural \ell_2 \wedge \natural C_3 \circ \ell_3 \wedge C_3 \circ \natural \ell_3 &&\\
&\Leftrightarrow C_2 \circ \ell_2 \wedge C_3 \circ \ell_3 && \text{(Corollary 1)}
\end{aligned}$$

□

This leads to a surprising distributivity property. Consider a cell $D = \langle b, \Sigma, \Phi \wedge \Psi \rangle$ and its conjuncts $D_1 = \langle b, \Sigma, \Phi \rangle$ and $D_2 = \langle b, \Sigma, \Psi \rangle$. If $\natural D_1$ is equivalent to $\natural D_2$, then, by Lemma 5, D is a meet of D_1 and D_2, and $\natural D$ is a meet of $\natural D_1$ and $\natural D_2$. Then, for any appropriate cell C and stack ℓ, the formula $C \circ D, \ell$ is logically equivalent to the conjunction of $C \circ D_1, \ell$ and $C \circ D_2, \ell$. Informally speaking, we can split a recipe over any cell conjunction, no matter where it occurs inside the recipe, as long as the conjuncts have equivalent neutralizations.

Theorem 2. *Consider a type signature π and a cell $J = \langle \bot, \Sigma, \mathbf{0}_\pi \rangle$. For every cell C of type π, we have $J \Rrightarrow C$ and $\natural J \Rrightarrow \natural C$.*

The proof of Theorem 2 is given in Appendix B. This result justifies our use of joker recipes to represent universal quantification over predicate variables. Indeed, for any recipes $\Phi : \square$ and $\Psi : \varrho$, and a type-compatible cell context Σ, the cell $\langle \bot, \Sigma, \forall g : \varrho\,.\,\Phi \rangle$ entails $\langle \bot, \Sigma, (\lambda g : \varrho\,.\,\Phi)\, \Psi \rangle$, and the same holds for their neutralizations.

In the process of evaluation, we can factorize selected first-order monomorphic cells, that is, those that only have term parameters whose type is not `Type`.

Lemma 6. *Consider a cell $C = \langle b, \Sigma, \lambda g : (\overline{x : \tau})\,.\,\Phi \rangle$ and an aligned stack D, ℓ, such that none of the types τ_i is `Type`. Let D' be the cell $\langle \bot, \varnothing, \lambda \overline{x : \tau}\,.\, z_1 = x_1 \wedge \cdots \wedge z_n = x_n \rightarrow \mathbf{0} \rangle$ for some fresh variables $\bar{z}$. Then $C \circ D, \ell \Leftrightarrow C \circ \natural D, \ell \wedge \forall \overline{z : \tau}\,.\,(\natural C \circ D', \natural \ell) \vee (D \circ \bar{z})$.*

The proof of Lemma 6 is given in Appendix C. This lemma provides us with an alternative evaluation rule for cell arguments which are eligible and useful to factorize. The latter is a matter of heuristic choice: in our current implementation, we select non-neutral cells that are used multiple times in the final VC and are derived from executable code instead of just a sequence of assertions.

The new rule splits the formula $C \circ D, \ell$ into two parts. The first part, $C \circ \natural D, \ell$, erases all subgoals stemming from D. In the second part, the formula $\natural C \circ D', \natural \ell$ erases all subgoals that are *not* stemming from D, and replaces every occurrence of D with a "unification subgoal" D', which captures a term substitution in the answer variables $\bar{z}$. These substitutions are transferred to the single instance of D in the formula $D \circ \bar{z}$.

By rewriting the second part as an implication $\forall \overline{z : \tau}\,.\,\neg(\natural C \circ D', \natural \ell) \rightarrow (D \circ \bar{z})$, we can see the antecedent as the cumulated logical premise (or the strongest postcondition) of the context $C \circ [\,], \ell$ for the continuation in the hole. In the next section, we show how this rule allows us to produce more compact verification conditions.

4 Verification Condition Generation

Verification conditions for Coma expressions are computed by the operator $\mathsf{C}^{\mathfrak{p}}_{\mathfrak{d}}$, where Boolean flags $\mathfrak{p}$ and $\mathfrak{d}$ establish the mode:

- $\mathsf{C}^{\top}_{\bot}$: *caller verification condition*, to verify individual calls of a defined handler.
- $\mathsf{C}^{\bot}_{\top}$: *callee verification condition*, to prove the correctness of a handler definition.
- $\mathsf{C}^{\top}_{\top}$: *full verification condition*, which merges the proof goals of the first two modes.
- $\mathsf{C}^{\bot}_{\bot}$: *null verification condition*, which is always true on fully applied expressions.

$$
\begin{array}{rclrcl}
\mathbb{C}^{\top}_{\mathfrak{d}}(h) & \triangleq & h & \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(\pi \rightarrow e) & \triangleq & (\lambda\pi.\, \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e)) \wedge \natural(\lambda\pi.\, \mathbb{C}^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)) \\
\mathbb{C}^{\bot}_{\mathfrak{d}}(h) & \triangleq & \natural h & \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(k\ \bar{s}\ \bar{o}) & \triangleq & \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(k)\ \bar{s}\ \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(o_1)\ \ldots\ \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(o_n) \\
\mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(\uparrow e) & \triangleq & \mathbb{C}^{\mathfrak{d}}_{\mathfrak{d}}(e) & \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(\{\varphi\}\ e) & \triangleq & (\varphi \rightarrow \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e)) \wedge (\mathfrak{p} \rightarrow \neg\varphi \rightarrow \mathbf{0}) \\
\mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(\downarrow e) & \triangleq & \mathbb{C}^{\mathfrak{p}}_{\mathfrak{p}}(e) & \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e\ /\ h\ \pi = d) & \triangleq & \mathbf{let}\ h\ \pi = \mathbb{C}^{\top}_{\bot}(d)\ \mathbf{in}\ \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \forall\pi.\, \mathbb{C}^{\bot}_{\mathfrak{p}}(d)
\end{array}
$$

Fig. 8: Verification condition generation.

The caller mode extracts the specification (or the contract) of a defined handler from its definition. It treats every assertion as a precondition to verify at call sites, and it stops at the black-box barrier which separates the "interface" part of the definition from the hidden "implementation" part. The callee mode, on the contrary, treats every assertion as a precondition to assume, and verifies the correctness of the implementation part, after the black-box barrier, under those assumptions. In the full mode, which is the starting verification mode for Coma expressions, we prove assertions both before and after a barrier. In the null mode, which is equivalent to stopping verification, no proof obligations are generated at all. A Coma program e is said to be *correct*, when its fully evaluated verification condition $\langle \bot, \Sigma_{\text{prim}}, \mathbb{C}^{\top}_{\top}(e) \rangle \circ \varepsilon$ is valid.

Figure 8 shows the rules of VC generation. The notation **let** $h\ \pi = \Psi$ **in** Φ in the rule for handler definitions stands for $(\lambda h : \pi.\, \Phi)\, (\lambda\pi.\, \forall h : \pi.\, \Psi)$—notice the universal quantifier that covers the occurrences of h in Ψ and ensures that this symbol is bound in the resulting recipe, just as it is bound in the original Coma expression. In this rule, we assign the handler's specification $\lambda\pi.\, \forall h : \pi.\, \mathbb{C}^{\top}_{\bot}(d)$ to a predicate variable with the same name h. This recipe is verified every time h is called from the underlying expression e or recursively from the definition body d.

Informally, flag $\mathfrak{p}$ determines whether we should generate proof obligations—prove assertions, verify handler definitions, ensure the safety of handler calls—at the current position in the expression. For example, in the rule for $\{\varphi\}\ e$, we only generate a subgoal for φ (expressed as a double negation $\neg\varphi \rightarrow \mathbf{0}$), when $\mathfrak{p}$ is true. Similarly, in the rule for handler definitions, we verify the correctness of the implementation only when $\mathfrak{p}$ is true; otherwise, the formula $\mathbb{C}^{\bot}_{\bot}(d)$ always reduces to $\top$. Finally, on handler invocation, when $\mathfrak{p}$ is false, the corresponding predicate variable is neutralized, which effectively cancels all proof obligations in the handler's specification, as $\mathbf{0}$ becomes evaluated as $\top$.

When we pass through a black-box barrier $\uparrow$, the second flag $\mathfrak{d}$ takes the place of $\mathfrak{p}$. Thus, when we compute the specification of a handler by applying $\mathbb{C}^{\top}_{\bot}$ to the handler's body, we stop at the black-box barrier, where we switch to $\mathbb{C}^{\bot}_{\bot}$, which evaluates to $\top$. On the other hand, when we verify the correctness of a handler definition using $\mathbb{C}^{\bot}_{\top}$, we do not generate proof obligations for assertions and handler calls until we arrive at the black-box barrier, where we pass into the full mode $\mathbb{C}^{\top}_{\top}$ for the rest of the definition.

The white-box barrier $\downarrow$ replaces the second flag with $\mathfrak{p}$. This preserves the current value of $\mathfrak{p}$ for the rest of the expression, regardless of the subsequent barriers. White-box barriers are not needed in the source code, as we can simply avoid placing barriers in the underlying expression. However, they find their use in the E-AppC rule of the operational semantics of Coma, where an anonymous handler argument becomes a handler definition under a white-box barrier. This ensures that during the VC computation, the definition body is treated in the full mode, just like the anonymous handler was treated in the preceding state, preserving the validity of the overall verification condition.

The rule for handler calls simply propagates the VC operator down to the individual handlers without changing the mode. Similarly, the rule for anonymous handlers pushes the VC operator in its current mode under the λ-prefix—however, we must, in addition, verify the handler in the complementary mode, with both $\mathfrak{p}$ and $\mathfrak{d}$ negated. This secondary verification condition is only concerned with the continuation parameters of the handler, and not with its proper proof obligations, which is why we neutralize the corresponding recipe. To see why both conditions are necessary, consider the following Coma code:

$$\mathtt{crash} \,/\, \mathtt{crash} = ((\mathtt{f} : \square) \to {\uparrow}\mathtt{f})\ \mathtt{fail}$$

This program reduces to `fail`: we unfold `crash` (E-Sym), purge the now-unreachable definition (E-Gc), substitute `fail` into `f` (E-AppH), and drop the barrier (E-Bbox). Thus, we should not be able to prove it correct. Let us look at the full verification condition:

$$\begin{aligned}
& \mathbb{C}^{\top}_{\top}(\mathtt{crash} \,/\, \mathtt{crash} = (\mathtt{f} \to {\uparrow}\mathtt{f})\ \mathtt{fail}) \\
= \ & \mathbf{let}\ \mathtt{crash} = \mathbb{C}^{\top}_{\bot}((\mathtt{f} \to {\uparrow}\mathtt{f})\ \mathtt{fail})\ \mathbf{in}\ \mathbb{C}^{\top}_{\top}(\mathtt{crash}) \wedge \mathbb{C}^{\bot}_{\top}((\mathtt{f} \to {\uparrow}\mathtt{f})\ \mathtt{fail}) \\
= \ & \mathbf{let}\ \mathtt{crash} = \mathbb{C}^{\top}_{\bot}(\mathtt{f} \to {\uparrow}\mathtt{f})\ \mathbb{C}^{\top}_{\bot}(\mathtt{fail})\ \mathbf{in}\ \mathtt{crash} \wedge \mathbb{C}^{\bot}_{\top}(\mathtt{f} \to {\uparrow}\mathtt{f})\ \mathbb{C}^{\bot}_{\top}(\mathtt{fail}) \\
= \ & \mathbf{let}\ \mathtt{crash} = ((\lambda\mathtt{f}.\,\mathbb{C}^{\top}_{\bot}({\uparrow}\mathtt{f})) \wedge \natural(\lambda\mathtt{f}.\,\mathbb{C}^{\bot}_{\top}({\uparrow}\mathtt{f})))\ \mathtt{fail}\ \mathbf{in}\ \mathtt{crash} \wedge \mathbb{C}^{\bot}_{\top}(\mathtt{f} \to {\uparrow}\mathtt{f})\ (\natural\mathtt{fail}) \\
= \ & \mathbf{let}\ \mathtt{crash} = ((\lambda\mathtt{f}.\,\mathbb{C}^{\bot}_{\bot}(\mathtt{f})) \wedge \natural(\lambda\mathtt{f}.\,\mathbb{C}^{\top}_{\top}(\mathtt{f})))\ \mathtt{fail}\ \mathbf{in}\ \mathtt{crash} \wedge \mathbb{C}^{\bot}_{\top}(\mathtt{f} \to {\uparrow}\mathtt{f})\ (\natural\mathtt{fail}) \\
= \ & \mathbf{let}\ \mathtt{crash} = ((\lambda\mathtt{f}.\,\natural\mathtt{f}) \wedge \natural(\lambda\mathtt{f}.\,\mathtt{f}))\ \mathtt{fail}\ \mathbf{in}\ \mathtt{crash} \wedge \mathbb{C}^{\bot}_{\top}(\mathtt{f} \to {\uparrow}\mathtt{f})\ (\natural\mathtt{fail}) \\
= \ & \mathbf{let}\ \mathtt{crash} = ((\lambda\mathtt{f}.\,\natural\mathtt{f}) \wedge \natural(\lambda\mathtt{f}.\,\mathtt{f}))\ \mathtt{fail}\ \mathbf{in}\ \mathtt{crash} \wedge ((\lambda\mathtt{f}.\,\mathtt{f}) \wedge \natural(\lambda\mathtt{f}.\,\natural\mathtt{f}))\ (\natural\mathtt{fail}) \\
\approx \ & (\lambda\mathtt{f}.\,\natural\mathtt{f})\mathbf{0} \wedge (\natural\lambda\mathtt{f}.\,\mathtt{f})\mathbf{0} \wedge (\lambda\mathtt{f}.\,\mathtt{f})(\natural\mathbf{0}) \wedge (\natural\lambda\mathtt{f}.\,\natural\mathtt{f})(\natural\mathbf{0})
\end{aligned}$$

For the sake of readability, we perform several reductions directly on the recipe in the last step; it is easy to show that the resulting recipe leads to the same final VC formula. Out of the four conjuncts, only the second one evaluates to $\bot$, and the other three to $\top$. If the rule for anonymous handlers did not include the second condition $\natural\lambda\pi\,.\,\mathbb{C}^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(d)$, we would end up with only the first and the third conjunct, which both evaluate to $\top$.

Verification conditions produced by the $\mathbb{C}^{\top}_{\top}$ operator are for partial correctness: they do not ensure the termination of Coma programs. Here is one possible way to verify total correctness. Let us say that a handler definition $f\ \overline{x:\tau}\ \overline{g:\varrho} = d$ is equipped with a *variant*, if there exists an `int`-typed term $t[\bar{x}]$ such that every occurrence of f in d is in an expression of the form $\{\, t[\bar{s}] < t[\bar{x}] \wedge \mathtt{0} \le t[\bar{x}] \,\}\ f\ \bar{s}\ \bar{o}$. Here we assume that variables are never bound twice, and so the variables $\bar{x}$ in the assertion refer indeed to the formal parameters of f. The ordering relation in the assertion is well-founded, therefore, an infinite tower of recursive calls of f is impossible. A practical implementation would, of course, accept other well-founded relations such as structural decrease on binary trees, lexicographic orderings on tuples, etc. While the definition above does not allow us to use f as a handler argument inside d, this is not a limitation, as we can always move such occurrences into a local wrapper handler definition.

Below we list the main results about our VC generation procedure. The proofs are given in Appendix D.

Lemma 7. *For any* Coma *expression e, any cell of the form $\langle b, \Sigma, \complement^{\bot}_{\bot}(e)\rangle$ is neutral.*

Consequently, any VC of the form $\complement^{\bot}_{\bot}(e)$, where e is a fully applied expression, can be safely replaced with a tautological recipe such as $\natural\mathbf{0}$.

Lemma 8. *For any* Coma *expression e, any cell of the form $\langle b, \Sigma, \complement^{\top}_{\top}(e)\rangle$ is a meet of $\langle b, \Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle$ and $\langle b, \Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle$ for all $\mathfrak{p}$ and $\mathfrak{d}$.*

The fact that $\complement^{\top}_{\top}(e)$ can be split into $\complement^{\top}_{\bot}(e)$ and $\complement^{\bot}_{\top}(e)$ is the basis of the correctness preservation theorem:

Theorem 3 (Preservation of Correctness). *For any* Coma *programs e and e', if e is correct and $e \longrightarrow e'$, then e' is correct.*

Theorem 4 (Progress). *For any correct* Coma *program e, either e is* `halt`*, or $e \longrightarrow e'$ for some program e'.*

In conclusion, let us show some examples of verification conditions. For clarity, we omit type annotations, inline the specifications of primitive handlers, treat $\mathbf{0}$ as $\bot$ in the callee mode, and remove trivial subgoals coming from $\complement^{\bot}_{\bot}(\cdot)$ or $\bot \rightarrow \Phi$. The caller VC of `removeRoot` on page 7, for a given tree `t` and continuation `return`, is the recipe

$$\begin{aligned}&(\forall \mathtt{l\,v\,r}.\ \mathtt{t} = \mathtt{Node\ l\ v\ r} \rightarrow \forall \mathtt{s}.\,(\forall \mathtt{e}.\ \mathtt{e} \in \mathtt{s} \leftrightarrow \mathtt{e} \in \mathtt{l} \vee \mathtt{e} \in \mathtt{r}) \rightarrow \mathtt{return\ s}) \wedge \\ &\qquad (\mathtt{t} = \mathtt{Empty} \qquad \rightarrow \mathbf{0})\end{aligned}$$

This recipe is the specification of `removeRoot`, instantiated and proved at each call site. The subrecipe $(\forall \mathtt{e}.\ \mathtt{e} \in \mathtt{s} \leftrightarrow \mathtt{e} \in \mathtt{l} \vee \mathtt{e} \in \mathtt{r}) \rightarrow \mathtt{return\ s}$ is the callee VC for the `ret` handler. Notice that the assertion $\{\,\forall \mathtt{e{:}int}.\ \mathtt{e} \in \mathtt{s} \leftrightarrow \mathtt{e} \in \mathtt{l} \vee \mathtt{e} \in \mathtt{r}\,\}$ does not generate a subgoal here: as it occurs before the black-box barrier, it is treated as an assumption in the callee mode.

Here is the callee VC for `removeRoot`, to be proved for all values of `t`:

$$\forall \mathtt{l\,v\,r}.\ \mathtt{t} = \mathtt{Node\ l\ v\ r} \rightarrow \mathtt{mergeTree\ l\ r}\ (\lambda \mathtt{s}.\forall \mathtt{e}.\ \mathtt{e} \in \mathtt{s} \leftrightarrow \mathtt{e} \in \mathtt{l} \vee \mathtt{e} \in \mathtt{r})$$

There is no subgoal generated for the second outcome of `unTree`, as it does not contain an abstraction barrier. The predicate argument of `mergeTree` is the caller VC of `ret`; this time the assertion does generate a subgoal.

Here is the specification of the `product` handler in Fig. 4, for given integers `a`, `b` and a continuation `return`:

$$(\mathtt{b} \not\geq \mathtt{0} \rightarrow \mathbf{0}) \wedge (\forall \mathtt{c}.\ \mathtt{c} = \mathtt{a} \cdot \mathtt{b} \rightarrow \mathtt{return\ c})$$

The callee VC for `product`, to be proved for all values of `a` and `b`, is as follows:

$$\begin{aligned}&\mathtt{b} \geq \mathtt{0} \rightarrow \\ &\quad \mathtt{a} \cdot \mathtt{b} + \mathtt{0} = \mathtt{a} \cdot \mathtt{b} \wedge \mathtt{b} \geq \mathtt{0} \wedge \\ &\quad \forall \mathtt{p\,q\,r}.\ \mathtt{p} \cdot \mathtt{q} + \mathtt{r} = \mathtt{a} \cdot \mathtt{b} \wedge \mathtt{q} \geq \mathtt{0} \rightarrow \\ &\qquad (\mathtt{q} > \mathtt{0} \rightarrow \\ &\qquad\quad (\mathtt{q\ mod\ 2} = \mathtt{1} \rightarrow (\mathtt{p} + \mathtt{p}) \cdot (\mathtt{q\ div\ 2}) + \mathtt{r} + \mathtt{p} = \mathtt{a} \cdot \mathtt{b} \wedge \mathtt{q\ div\ 2} \geq \mathtt{0}) \wedge \\ &\qquad\quad (\mathtt{q\ mod\ 2} \neq \mathtt{1} \rightarrow (\mathtt{p} + \mathtt{p}) \cdot (\mathtt{q\ div\ 2}) + \mathtt{r} \qquad = \mathtt{a} \cdot \mathtt{b} \wedge \mathtt{q\ div\ 2} \geq \mathtt{0})) \wedge \\ &\qquad (\mathtt{q} \not> \mathtt{0} \rightarrow \mathtt{r} = \mathtt{a} \cdot \mathtt{b})\end{aligned}$$

```
product (a b: int) { b ≥ 0 } (return (c: int) { c = a · b })
= loop a b 0
  / loop (p q r: int) { p · q + r = a · b ∧ q ≥ 0 }
    = if (q > 0) (→ if (q mod 2 = 1) (→ next (r + p)) (→ next r)
                    / next (s: int) = loop (p + p) (q div 2) s)
                 (→ return r)
```

Fig. 9: Extended handler prototypes.

Notice how this formula coincides with the verification condition for the definition of `product` obtained by the traditional weakest-precondition calculus. Consider now the same VC, when we select the `next` handler for factorization, as described in Section 3:

$$
\begin{aligned}
&\mathtt{b} \geq 0 \longrightarrow \\
&\quad \mathtt{a}\cdot\mathtt{b} + 0 = \mathtt{a}\cdot\mathtt{b} \land \mathtt{b} \geq 0 \land \\
&\quad \forall \mathtt{pqr}.\ \mathtt{p}\cdot\mathtt{q} + \mathtt{r} = \mathtt{a}\cdot\mathtt{b} \land \mathtt{q} \geq 0 \longrightarrow \\
&\qquad (\mathtt{q} > 0 \longrightarrow \\
&\qquad\quad \forall \mathtt{s}.\ ((\mathtt{q}\ \mathtt{mod}\ 2 = 1 \land \mathtt{s} = \mathtt{r} + \mathtt{p}) \lor (\mathtt{q}\ \mathtt{mod}\ 2 \neq 1 \land \mathtt{s} = \mathtt{r})) \longrightarrow \\
&\qquad\qquad (\mathtt{p} + \mathtt{p})\cdot(\mathtt{q}\ \mathtt{div}\ 2) + \mathtt{s} = \mathtt{a}\cdot\mathtt{b} \land \mathtt{q}\ \mathtt{div}\ 2 \geq 0) \land \\
&\qquad (\mathtt{q} \not> 0 \longrightarrow \mathtt{r} = \mathtt{a}\cdot\mathtt{b})
\end{aligned}
$$

The formula $\lambda \mathtt{s}.(\mathtt{q}\ \mathtt{mod}\ 2 = 1 \land \mathtt{s} = \mathtt{r} + \mathtt{p}) \lor (\mathtt{q}\ \mathtt{mod}\ 2 \neq 1 \land \mathtt{s} = \mathtt{r})$ is the strongest postcondition of the expression `if (q mod 2 = 1) (→ next (r + p)) (→ next r)` with respect to the continuation `next`. The method of compact verification conditions proposed by Flanagan and Saxe [7] aggregates in a similar way the strongest postconditions across alternative execution paths. The connection between the compact verification conditions and the classical weakest-precondition calculus was studied by Leino [12]. Our approach makes this connection even more prominent, as it allows us to derive both forms from the common precursor verification condition.

5 Implementation

We have implemented the COMA language and its VC generator on top of the WHY3 platform [6]. The terms and formulas of COMA are written in the logical language of WHY3. This way, we can make use of logical theories from the WHY3 standard library, and we readily benefit from WHY3's interface with many automated theorem provers. In addition to what is presented in the previous sections, our implementation offers a few extensions, described below.

Extended handler prototypes. To facilitate writing and understanding of COMA programs, we define a suitable syntax for writing pre- and postconditions directly in the handler prototype, as shown in Fig. 9. This notation is desugared into assertions, black-box barriers, and wrapper handlers of the core COMA language; in particular, the code in Fig. 9 is translated into what is shown in Fig. 4. The precondition `{ b ≥ 0 }` in the prototype of `product` becomes an assertion put on top of the definition body, now hidden under a black-box barrier. The same transformation is applied to the precondition of the inner `loop` handler. The postcondition { c = a · b } attached to the `return`, forces creation of a wrapper handler above the main black-box barrier and becomes the precondition in the body of this wrapper handler.

Notice that in this syntax we do not put a colon between a handler parameter and its type signature: the parameters of an outcome are listed directly after the handler's name.

Let-binding for variables. We added a proper syntax for binding a variable to a term, to avoid writing anonymous handler applications $((x : \tau) \to e)\ s$. The new construction is written $e\ /\ x : \tau = s$. We show its use in the example in Fig. 11 in the next section.

Mutable state. Our implementation supports mutable variables (*references*) that can be allocated, modified, and passed as arguments to handlers. References are alias-free, which means that is forbidden to pass a statically accessible reference as an argument or to pass the same reference argument twice. Each handler is annotated with a *pre-write* clause, which lists the references in its lexical scope that might be modified before the handler is executed. For example, here is the prototype of a handler that increments an integer reference received as argument and returns its previous value:

```
incr (&r: int) (return [r] (p: int))
```

The pre-write annotation `[r]` for the `return` outcome signifies that the code that calls `return`—namely, the `incr` handler—may change the value of `r` before the call. Pre-write annotations are automatically inferred for defined handlers and their continuation parameters. However, we do not infer them for the higher-order outcomes (i.e., continuation parameters of continuation parameter).

The code with references is translated into pure Coma via a fine-grained monadic transformation, during which the references in the pre-write annotations become additional term parameters. In the example above, after translation, `incr` would return to the caller the updated value of `r` along with its previous value in `p`.

To capture the pre-state of references, we admit let-bindings in handler prototypes. The full prototype of `incr`, together with its specification, is as follows:

```
incr (&r: int) [o: int = r] (return [r] (p: int) { r = o + 1 ∧ p = o })
```

Specification extraction. Given a first-order handler, Coma can produce, on request, the logical predicates that represent its pre- and postconditions. These predicates are computed in a similar way to subgoal factorization discussed in the previous section.

6 Case Study: Regular Expression Processing

In this section, we demonstrate the use of Coma by verifying a small but non-trivial OCaml program, that uses higher-order functions, exceptions, and requires giving specification to closures in order to be verified. Figure 10 shows the code for a function `accept` that checks if a string `s` is recognized by a regular expression `r`. The type

```
type regexp =  Empty         |  Alt    of regexp * regexp
            |  Epsilon       |  Concat of regexp * regexp
            |  Char of char  |  Star   of regexp

let accept (r: regexp) (s: string): bool =
  let n = String.length s in
  let rec a (r: regexp) (i: int) (k: int → unit): unit = match r with
    | Empty          → ()
    | Epsilon        → k i
    | Char c         → if i < n && s.[i] = c then k (i + 1)
    | Alt (r1, r2)   → a r1 i k; a r2 i k
    | Concat (r1, r2) → a r1 i (fun j → a r2 j k)
    | Star r1        → k i; a r1 i (fun j → if i < j then a r j k) in
  try a r 0 (fun j → if j = n then raise Exit); false with Exit → true
```

Fig. 10: Regular expression engine in OCaml.

`regexp` of regular expressions is given on top on Fig. 10. The code traverses the string with a recursive function `a`, which takes three parameters: a current regexp `r`, an integer index `i`, and a continuation `k`. This function tries to match a substring `s[i..j)` with `r`, and then applies the continuation `k` to index `j` to proceed with the matching of `s[j..)`. If no such `j` exists, function `a` returns the unit value `()`. The initial continuation passed to function `a` signals a success by raising the predefined exception `Exit`.

Figure 11 contains a Coma translation for the `accept` function, which can be obtained by a mechanical CPS-translation of the OCaml code. The `accept` handler has a `return` outcome that receives the Boolean results of the computation. Second, the internal handler `a` has a continuation `o`, that corresponds to the normal outcome of the original OCaml function. Finally, the continuation `k` is transformed itself into CPS-style, and thus has its own outcome, named `h`. Another way to look at this code is to interpret `k` and `o` as success and error/backtrack continuations, respectively, as in a double-barreled CPS [17]. The pattern-matching on the regular expression `r` is performed using a handler `unRe` similar to the `unTree` handler.

To verify this Coma program, we need to extend it with barriers and specifications. Figure 12 contains a version of `accept` with added preconditions (in `cyan`) and ghost parameters (in `gray`). As explained in Section 5, specification annotations inside handler prototypes are automatically desugared into assertions, black-box barriers, and wrapper handlers. The postcondition of `accept` (i.e., the precondition of `return`) uses a built-in logical predicate `mem`, where `mem s r` holds if and only if the string `s` belongs to the language of `r`.

We add a ghost parameter `ck` to handler `a` for the purpose of its specification. Note that the current implementation of Coma does not provide any special treatment for ghost code and data. In future, we plan to introduce the necessary checks that ensure that ghost computations do not interfere with the observable part of the program. As in the OCaml code, handler `a` tries to match a substring `s[i..j)` with `r`, and then applies the continuation `k` to index `j` to proceed with the matching of `s[j..)`. The `ck` parameter is

```
accept (r: regexp) (s: string) (return (b: bool))
= a r 0 (j h → if (j = n) (→ return true) h) (→ return false)
  / a (r: regexp) (i: int) (k (j: int) (h)) (o)
    = unRe r empty eps char alt cat star
      / empty     = o
      / eps       = k i o
      / char c    = if (i < n && s[i] = c) (k (i + 1) o) o
      / alt r1 r2 = a r1 i k (→ a r2 i k o)
      / cat r1 r2 = a r1 i (j h → a r2 j k h) o
      / star r1   = k i (→ a r1 i (j h → if (i < j) (→ a r j k h) h) o)
  / n: int = length s
```

Fig. 11: Regular expression engine in Coma.

a first-class predicate which characterizes the index `j` passed to `k`. The `cons` predicate, declared on top of the figure, is a shortcut to simplify annotations.

In addition to the annotations given in Fig. 12, we have also instrumented the Coma code to verify the termination of handler `a`, as described in Section 4. In this case, the variant is a pair (|`s`| − `i`, `r`), ordered lexicographically: namely, we either progress in string `s` or we move to a smaller regular expression. When the VC for handler `accept` is sent to Why3, it is split into 44 individual proof tasks which are easily discharged by the SMT solvers Z3 [4] and Alt-Ergo [3].

7 Case Study: Verified Assembly Code

We believe that Coma is a suitable intermediate language for the verification of unstructured programs. As a proof of concept, we have built a prototype tool for the deductive verification of x86-64 assembly programs. The input of the tool is assembly code annotated with assumptions, assertions, and loop invariants. Figure 13 shows the x86-64 assembly code for a function `sortbits` that sorts the bits of a 64-bit integer using the "I can't believe it can sort" algorithm by Fung [8]. We use the AT&T syntax, with the destination operand on the right. For instance, `mov` A, B copies the register A into B, and `andn` A, B, C computes $A \land \neg B$ and stores it in C. Integer literals start with a `$` sign. The code contains unnecessary labels (e.g., `test2`), which are only introduced to simplify the forthcoming explanations.

Function `sortbits` receives an integer in the `rdi` register and returns an integer in the `rax` register, with the same number of 1 bits, which are moved to the least significant positions. The code iterates over all pairs of bits $0 \le i, j < 64$, using registers `rdi` and `rsi`, with two nested loops. Whenever the bit i is clear and the bit j is set, the two bits are swapped. (It is not obvious why this sorting procedure is correct; see Fung's paper for an explanation.) The code contains logical annotations as special comments: namely, two loop invariants and one assertion before the function end. Here, we only show that the population count remains constant (using a logical function `pop`), but we do not show that bits are indeed sorted. We use the notation `rdi@sortbits` to refer to the value of the `rdi` register at function entry.

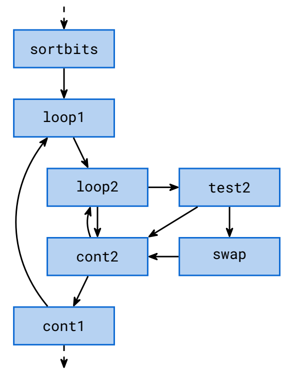
```
predicate cons (s: string) (r: regexp) (ck: int → bool) (i: int) =
  exists j. i ≤ j ≤ length s ∧ mem s[i..j] r ∧ ck j

accept (r: regexp) (s: string) (return (b: bool) { b ↔ mem s r })
= a r 0 (j ↦ j = n) (j h → if (j = n) (→ return true) h) (→ return false)
  / a (r: regexp) (i: int) (ck: int → bool) { 0 ≤ i ≤ n }
      (k (j: int) { mem s[i..j] r ∧ i ≤ j ≤ n } (h { not (ck j) }))
      (o { not (cons s r ck i) })
    = unRe r empty eps char alt cat star
      / empty     = o
      / eps       = k i o
      / char c    = if (i < n && s[i] = c) (k (i + 1) o) o
      / alt r1 r2 = a r1 i ck k (→ a r2 i ck k o)
      / cat r1 r2 = a r1 i (j ↦ cons s r2 ck j) (j h → a r2 j ck k h) o
      / star r1   = k i (→ a r1 i (j ↦ i < j ∧ cons s r ck j)
                            (j h → if (i < j) (→ a r j ck k h) h) o)
  / n: int = length s
```

Fig. 12: Regular expression engine in Coma, with specification.

```
# sort the bits of %rdi using ``I can't believe it can sort''
sortbits:
        mov     %rdi, %rax
        mov     $0x8000000000000000, %rdi
loop1:  #@ invariant pop(rdi) = 1 ∧ pop(rax) = pop(rdi@sortbits)
        mov     $0x8000000000000000, %rsi
loop2:  #@ invariant pop(rsi) = 1 ∧ pop(rax) = pop(rdi@sortbits)
        mov     %rax, %rcx                  # if !(rax & rdi)
        and     %rdi, %rcx
        jnz     cont2
test2:  mov     %rax, %rcx                  # and (rax & rsi)
        and     %rsi, %rcx
        jz      cont2
swap:   or      %rdi, %rax                  # then swap bits
        andn    %rax, %rsi, %rax
cont2:  shr     $1, %rsi
        jnz     loop2
cont1:  shr     $1, %rdi
        jnz     loop1
        #@ assert pop(rax) = pop(rdi@sortbits)
        ret
```

Fig. 13: Example of verified x86-64 code.



(a) Control flow of `sortbits`.

```
sortbits &rax &rcx &rsi &rdi &flags [rdi0=rdi]
= [...]
  loop1
  / loop1 =
    { pop(rdi) = 1 ∧ pop(rax) = pop(rdi0) }
    ↑ [...] loop2
      / loop2 =
        { pop(rsi) = 1 ∧ pop(rax) = pop(rdi0) }
        ↑ [...] if b0 cont2 test2
          / test2 = [...] if b1 cont2 swap
            / swap = [...] cont2
          / cont2 = [...] if b2 loop2 cont1
            / cont1 = [...] if b3 loop1 exit
              / exit = { pop(rax) = pop(rdi0) }
                halt
```

(b) Structure of the Coma code for `sortbits`.

Fig. 14: Compilation passes for `sortbits`.

Our tool parses the code and its annotations, and starts with building its control-flow graph (depicted in Fig. 14a). Then, it computes the dominator tree, the entry point being the function entry. A basic block A dominates a block B whenever any path from the entry to B traverses A. For instance, block `loop2` dominates block `test2`, which itself dominates `swap2`. Finally, our tool builds a Coma code that follows the structure of the dominator tree. In this way, we do not need to repeat the outer invariant in the inner loop for variables that are not modified. For instance, the handler `swap2` is a local definition in handler `test2`, which is itself a local definition in handler `loop2`. Each invariant is translated into an assertion followed by a barrier. Figure 14b shows the structure of the Coma code for `sortbits`. For an easier reading, we omit type annotations and we have left only what relates to control-flow and specification. Conditional jumps are translated using the primitive `if` (and suitable Boolean conditions b_i), and unconditional jumps are handler calls. Parts where references are modified are written "`[...]`" for clarity. The translated code relies on our Why3 model of a fragment of the x86-64 instruction set. For instance, the instruction `andn` is translated into a Coma reference assignment `&rax` ← `andn rax rsi`, where the logical function `andn` is defined in the accompanying Why3 library.

The Coma backend computes the VC for the code in Fig. 14b, and sends it to Why3. There it is split into 5 proof tasks—two instances of invariant initialization, two instances of invariant preservation, and the final postcondition—which are automatically proved by Z3 and Alt-Ergo.

8 Related work

To our knowledge, no deductive verification system features explicit abstraction barriers in the style of Coma. Compared to the widely used intermediate verification languages like WhyML [6], Viper [14], and Boogie [13], Coma is a smaller language, with fewer constructs and a simpler VC generator, yet offering the same control structures.

There is a natural connection between the weakest-precondition calculus and the CPS transformation, the former being a predicate transformer and the latter a structurally similar code transformer. This connection was first studied on a minimal imperative language by Jensen [9]. This work was later extended with exception handling and `goto` statement by Audebaud and Zucca [1], and furthermore, with recursion, higher-order functions, and side effects by Kura [11]. A predicate transformer called the Dijkstra monad, introduced by Swamy et al. [16] and used to verify higher-order and effectful $F^\star$ programs, also highlights this connection. Coma exploits in a similar manner the relation between the continuation-based style and the WP computation. Explicit abstraction barriers allow us to verify recursive code without computing a fixed point of its verification condition.

The CFML tool developed by Charguéraud [2] enables the interactive verification of higher-order stateful programs, written in a subset of OCaml. Programs are translated into so-called characteristic formulas, which essentially capture the weakest precondition of the programs, with respect to a shallow embedding of separation logic in Coq. Specifications are proved in the form of lemmas derived from characteristic formulae. Unlike Coma, or other VC-based program verifiers, the program logic rules of CFML have to be applied manually in the course of an interactive Coq proof. Assertions and invariants are provided as the proof progresses, which limits the possibilities for automation. On the other hand, CFML offers a greater flexibility in stating properties of program code, such as verifying a given function against two different contracts.

9 Conclusion

We presented Coma, an intermediate verification language with explicit abstraction barriers that can be placed inside function definitions in order to make the exposed part of the computation appear in the specification. This allows us to write specifications in a more concise and flexible way, without having to manually translate executable code into logical specification.

Our future work is planned along three main axes. We plan to add more powerful mechanisms to handle the mutable state, by including the notions of ownership and borrowing, and using prophecy variables in verification conditions. It is also interesting to provide support for advanced control structures, such as iterators, coroutines, and algebraic effects. Finally, we continue to improve the efficiency of our implementation, in particular the recipe evaluation engine and the subgoal factorization heuristics.

References

1. Audebaud, P., Zucca, E.: Deriving proof rules from continuation semantics. Formal aspects of computing **11**, 426–447 (1999)
2. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP). pp. 418–430. ACM, Tokyo, Japan (September 2011)

3. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018)
4. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
5. Denis, X., Jourdan, J.H., Marché, C.: Creusot: a foundry for the deductive verification of rust programs. In: International Conference on Formal Engineering Methods. pp. 90–105. Springer (2022)
6. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
7. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: Principles Of Programming Languages. pp. 193–205. ACM (2001)
8. Fung, S.P.Y.: Is this the simplest (and most surprising) sorting algorithm ever? (2021), https://arxiv.org/abs/2110.01111
9. Jensen, K.: Connection between Dijkstra's predicate-transformers and denotational continuation-semantics. DAIMI Report Series (86) (1978)
10. Krivine, J.L.: Un interprète du λ-calcul (1985)
11. Kura, S.: Higher-Order Weakest Precondition Transformers via a CPS transformation. arXiv preprint arXiv:2301.09997 (2023)
12. Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93**(6), 281–288 (2005)
13. Leino, K.R.M.: This is Boogie 2. manuscript KRML **178**(131), 9 (2008)
14. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17. pp. 41–62. Springer (2016)
15. Statman, R.: Logical relations and the typed λ-calculus. Information and control **65**(2-3), 85–97 (1985)
16. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. ACM SIGPLAN Notices **48**(6), 387–398 (2013)
17. Thielecke, H.: Comparing control constructs by double-barrelled CPS. Higher-Order and Symbolic Computation **15**, 141–160 (2002)

A Terms and formulas

The grammar of types, terms and logical formulas, as used in this paper, is presented below. Of course, any practical implementation of COMA will choose its own set of types and operations, and may also allow the programmer to extend it when desired.

$$
\begin{array}{llll}
\text{variable:} & x, y, \alpha, \beta & & \\
\text{term:} & s, t, \tau, \theta & ::= & \texttt{Type} \mid \texttt{bool} \mid \texttt{int} \\
& & \mid & x \mid \texttt{tree}\ \tau \mid \texttt{seq}\ \tau \\
& & \mid & \top \mid \bot \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
& & \mid & s \texttt{ + } t \mid s \texttt{ - } t \mid s \texttt{ * } t \\
& & \mid & s \texttt{ = } t \mid s \texttt{ < } t \mid s \texttt{ > } t \\
& & \mid & \texttt{Node}\ s_1\ s_2\ s_3 \mid \texttt{Empty} \\
& & \mid & \texttt{[}\ \bar{s}\ \texttt{]} \mid s_1\ \texttt{[}\ s_2\ \texttt{..}\ s_3\ \texttt{]} \\
& & \mid & \texttt{concat}\ s\ t \mid \texttt{length}\ s \\
\text{formula:} & \varphi, \psi & ::= & s \mid \neg\, \varphi \mid \varphi \wedge \psi \\
& & \mid & \varphi \rightarrow \psi \mid \varphi \vee \psi \\
& & \mid & \forall\, x : \tau\, .\, \varphi \mid \exists\, x : \tau\, .\, \varphi
\end{array}
$$

As explained in Section 2, type expressions are treated as terms, so that a type annotation $s : \tau$ matches both "`42:int`" and "`int:Type`". We do not include an access operator for sequences; instead, we use the ternary slice operator $s[i\,..\,j]$, which is defined for all sequences s and integers i, j, producing the sequence (possibly empty) of all elements s_k such that $i \leq k < j$. Finally, we use Boolean terms as atomic formulas; however, a non-atomic formula cannot be used as a Boolean term.

Figure 15 shows the typing rules for terms. Since we do not distinguish type and term variables, the rule for $x : \tau$ also applies to α : `Type`. Notice, however, that the symbol `Type` itself does not have any type assigned to it.

Figure 16 shows the typing rules for logical formulas. The quantifiers in user-written formulas are restricted to data variables, whose type is not `Type`. This restriction is relaxed in fully evaluated verification conditions, as produced by the $\circ$ operator from Section 3: in those, type variables can be universally quantified. Such quantifiers may only appear in positive positions, and never under an existential quantifier, avoiding the need in dependent types.

B Undefined Behaviour (Proof of Theorem 2)

A stack ℓ *covers* a cell $C = \langle b, \Sigma, \Phi \rangle$ if one of the two condition holds: either C appears in ℓ, or $b = \top$ and ℓ covers Σ. As with any property of cells extended to cell contexts, the last condition means that ℓ covers every cell in Σ as well as its neutralization; the latter requirement is trivially satisfied, since any neutralized cell admits an arbitrary cover.

We say that a cell $C = \langle b, \Sigma, \Phi \rangle$ *submits* to a type signature π if for every aligned stack ℓ that submits to π, for every cell $C_0 = \langle b_0, \Sigma_0, \mathbf{0}_\pi \rangle$ and aligned stack ℓ_0, such that b_0 implies b and ℓ_0 covers both Σ and ℓ, we have $C_0 \circ \ell_0 \Rightarrow C \circ \ell$.

$$\frac{}{\Gamma \vdash \texttt{bool} : \texttt{Type}} \qquad \frac{}{\Gamma \vdash \texttt{int} : \texttt{Type}} \qquad \frac{\Gamma \vdash \tau : \texttt{Type}}{\Gamma \vdash \texttt{tree}\ \tau : \texttt{Type}} \qquad \frac{\Gamma \vdash \tau : \texttt{Type}}{\Gamma \vdash \texttt{seq}\ \tau : \texttt{Type}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \top : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \bot : \texttt{bool}} \qquad \frac{s \in \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots\}}{\Gamma \vdash s : \texttt{int}}$$

$$\frac{\Gamma \vdash s : \texttt{int} \qquad \Gamma \vdash t : \texttt{int}}{\Gamma \vdash s \texttt{ + } t : \texttt{int}} \qquad \frac{\Gamma \vdash s : \texttt{int} \qquad \Gamma \vdash t : \texttt{int}}{\Gamma \vdash s \texttt{ - } t : \texttt{int}}$$

$$\frac{\Gamma \vdash s : \texttt{int} \qquad \Gamma \vdash t : \texttt{int}}{\Gamma \vdash s \texttt{ * } t : \texttt{int}} \qquad \frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma \vdash s : \tau \qquad \Gamma \vdash t : \tau}{\Gamma \vdash s \texttt{ = } t : \texttt{bool}}$$

$$\frac{\Gamma \vdash s : \texttt{int} \qquad \Gamma \vdash t : \texttt{int}}{\Gamma \vdash s \texttt{ < } t : \texttt{bool}} \qquad \frac{\Gamma \vdash s : \texttt{int} \qquad \Gamma \vdash t : \texttt{int}}{\Gamma \vdash s \texttt{ > } t : \texttt{bool}}$$

$$\frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma \vdash s_1 : \texttt{tree}\ \tau \qquad \Gamma \vdash s_2 : \tau \qquad \Gamma \vdash s_3 : \texttt{tree}\ \tau}{\Gamma \vdash \texttt{Node}\ s_1\ s_2\ s_3 : \texttt{tree}\ \tau}$$

$$\frac{\Gamma \vdash \tau : \texttt{Type}}{\Gamma \vdash \texttt{Empty} : \texttt{tree}\ \tau} \qquad \frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma \vdash s_1 : \tau \qquad \ldots \qquad \Gamma \vdash s_n : \tau}{\Gamma \vdash \texttt{[}\ s_1\ \ldots\ s_n\ \texttt{]} : \texttt{seq}\ \tau}$$

$$\frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma \vdash s_1 : \texttt{seq}\ \tau \qquad \Gamma \vdash s_2 : \texttt{int} \qquad \Gamma \vdash s_3 : \texttt{int}}{\Gamma \vdash s_1\ \texttt{[}\ s_2\ \texttt{..}\ s_3\ \texttt{]} : \texttt{seq}\ \tau}$$

$$\frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma \vdash s_1 : \texttt{seq}\ \tau \qquad \Gamma \vdash s_2 : \texttt{seq}\ \tau}{\Gamma \vdash \texttt{concat}\ s_1\ s_2 : \texttt{seq}\ \tau} \qquad \frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma \vdash s : \texttt{seq}\ \tau}{\Gamma \vdash \texttt{length}\ s : \texttt{seq}\ \tau}$$

Fig. 15: Typing rules for terms.

$$\frac{\Gamma \vdash s : \texttt{bool}}{\Gamma \vdash s : \mathrm{Prop}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{Prop}}{\Gamma \vdash \neg\varphi : \mathrm{Prop}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{Prop} \qquad \Gamma \vdash \psi : \mathrm{Prop}}{\Gamma \vdash \varphi \wedge \psi : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash \varphi : \mathrm{Prop} \qquad \Gamma \vdash \psi : \mathrm{Prop}}{\Gamma \vdash \varphi \rightarrow \psi : \mathrm{Prop}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{Prop} \qquad \Gamma \vdash \psi : \mathrm{Prop}}{\Gamma \vdash \varphi \vee \psi : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma, x : \tau \vdash \psi : \mathrm{Prop}}{\Gamma \vdash \forall x : \tau . \varphi : \mathrm{Prop}} \qquad \frac{\Gamma \vdash \tau : \texttt{Type} \qquad \Gamma, x : \tau \vdash \psi : \mathrm{Prop}}{\Gamma \vdash \exists x : \tau . \varphi : \mathrm{Prop}}$$

Fig. 16: Typing rules for formulas.

Lemma 9. *All cells submit to all type signatures.*

Proof. By induction over the order of the type signature. Let us pick a type signature π and assume, by the induction hypothesis, that all cells submit to every type signature of a lower order than π. Now it suffices to prove that for any cell $C = \langle b, \Sigma, \Upsilon\rangle$, if Σ submits to π, then so does C. Once this is established, a simple induction over cell depth will allow us to conclude. We proceed by induction over the size of Υ, counting only the subrecipes, so that term substitutions do not affect the size.

Consider a stack ℓ aligned with C that submits to π. Consider a cell $C_0 = \langle b_0, \Sigma_0, \mathbf{0}_\pi\rangle$ such that b_0 implies b, and an aligned stack ℓ_0 that covers Σ and ℓ. We need to show $C_0 \circ \ell_0 \Rightarrow C \circ \ell$. If b_0 is $\bot$, then $C_0 \circ \ell_0$ evaluates to $\bot$, which trivially entails $C \circ \ell$. Otherwise, both b_0 and b are $\top$, and we have the following cases to consider.

Case Υ is $\mathbf{0}$. Since b is $\top$, the formula $C \circ \ell$ is also $\top$.

Case Υ is h. Let $\Sigma(h)$ be a cell $C_h = \langle b_h, \Sigma_h, \Phi_h\rangle$. As ℓ_0 covers C_h, either C_h is in ℓ_0 or $b_h = \top$ and ℓ_0 covers Σ_h. If the latter, we have $C_0 \circ \ell_0 \Rightarrow C_h \circ \ell$, as C_h submits to π. Otherwise, C_h appears in ℓ_0. By Corollary 1, we have $C_h \circ \ell \Leftrightarrow \natural C_h \circ \ell \wedge C_h \circ \natural\ell$. The cell $\natural C_h$ submits to π and ℓ_0 covers the neutralized context $\natural\Sigma_h$. Then $C_0 \circ \ell_0 \Rightarrow \natural C_h \circ \ell$. To prove $C_0 \circ \ell_0 \Rightarrow C_h \circ \natural\ell$, we note that $C_0 \circ \ell_0$ evaluates to a conjunction, where one part is of the form $\langle \top, \Sigma_0', \forall \overline{x:\tau}. \forall \overline{g:\varrho}. f\ \bar{x}\ \bar{g}\rangle \circ \varepsilon$ such that $\Sigma_0'(f)$ is C_h. By evaluating this formula further, we obtain $\forall \overline{x:\tau}. C_h \circ \bar{x}, \langle \top, \Sigma_0'', g_1\rangle, \dots, \langle \top, \Sigma_0'', g_n\rangle$, where Σ_0'' is $\Sigma_0' \uplus [g_1 \mapsto \langle \top, \Sigma_0', \mathbf{0}_{\varrho_1}\rangle, \dots, g_n \mapsto \langle \top, \Sigma_0', \mathbf{0}_{\varrho_n}\rangle]$. Consider any cell $\langle \top, \Sigma_0'', g_i\rangle$ along with the i-th cell in $\natural\ell$, which we shall denote $\natural D_i$. The two cells have the same type ϱ_i. Since the order of ϱ_i is two less than the order of π, all cells submit to ϱ_i. Therefore, for any aligned stack ℓ'', we obtain $\langle \top, \Sigma_0'', g_i\rangle \circ \ell'' = \langle \top, \Sigma_0', \mathbf{0}_{\varrho_i}\rangle \circ \ell'' \Rightarrow \natural D_i \circ \ell''$, since ℓ'' covers itself as well as the neutralized cell context of $\natural D_i$. Then $\langle \top, \Sigma_0'', g_i\rangle \Rrightarrow \natural D_i$ by Lemma 3. By the same reasoning, we also have $\natural\langle \top, \Sigma_0'', g_i\rangle \Rrightarrow \natural\natural D_i$. Consequently, $\forall \overline{x:\tau}. C_h \circ \bar{x}, \langle \top, \Sigma_0'', g_1\rangle, \dots, \langle \top, \Sigma_0'', g_n\rangle \Rightarrow \forall \overline{x:\tau}. C_h \circ \bar{x}, \natural D_1, \dots, \natural D_n \Rightarrow C_h \circ \natural\ell$. Thus, $C_0 \circ \ell_0$ implies $C_h \circ \ell$.

Case Υ is $\natural\Phi$. As neutralized cells admit any cover, ℓ_0 covers $\natural\Sigma$. By the induction hypothesis, $\langle \top, \natural\Sigma, \Phi\rangle$ submits to π, and $C_0 \circ \ell_0 \Rightarrow \langle \top, \natural\Sigma, \Phi\rangle \circ \ell$.

Case Υ is $\Phi\,\Psi$. Let C_1 be $\langle \top, \Sigma, \Phi\rangle$ and C_2 be $\langle \top, \Sigma, \Psi\rangle$. By the induction hypothesis, $C_1, C_2, \natural C_2$ submit to π. Also, ℓ_0 covers C_2 and $\natural C_2$, and so $C_0 \circ \ell_0 \Rightarrow C_1 \circ C_2, \ell$.

Case Υ is $\lambda g : \varrho. \Phi$. The stack ℓ is of the form C', ℓ', where the cells C' and $\natural C'$ submit to π and are covered by ℓ_0. Then $\Sigma \uplus [g \mapsto C']$ also submits to π and is covered by ℓ_0. By the induction hypothesis, $C_0 \circ \ell_0 \Rightarrow \langle \top, \Sigma \uplus [g \mapsto C'], \Phi\rangle \circ \ell'$.

Case Υ is $\Phi \wedge \Psi$. Let C_1 be $\langle \top, \Sigma, \Phi\rangle$ and C_2 be $\langle \top, \Sigma, \Psi\rangle$. By the induction hypothesis, $C_0 \circ \ell_0$ entails both $C_1 \circ \ell$ and $C_2 \circ \ell$.

Case Υ is $\forall x : \tau. \Phi$. By the induction hypothesis, $C_0 \circ \ell_0$ implies $\langle \top, \Sigma, \Phi\rangle \circ \varepsilon$. We can safely assume that the variable x does not occur in $C_0 \circ \ell_0$, and thus, $C_0 \circ \ell_0$ entails $\forall x : \tau. \langle \top, \Sigma, \Phi\rangle \circ \varepsilon$.

The rest of the cases are handled in a similar way. □

Corollary 3 (Theorem 2). *Consider a type signature π and a cell $J = \langle \bot, \Sigma, \mathbf{0}_\pi\rangle$. For every cell C of type π, we have $J \Rrightarrow C$ and $\natural J \Rrightarrow \natural C$.*

Proof. By Lemma 3. For any aligned stack ℓ, we have $J \circ \ell = \bot \Rightarrow C \circ \ell$. For the second part, we note that ℓ covers itself and the cell context of $\natural C$, hence $\natural J \circ \ell \Rightarrow \natural C \circ \ell$. □

C Cell factorization (Proof of Lemma 6)

Consider a cell D of type $\overline{x:\tau}$, where none of the term types τ_i is equal to $\texttt{Type}$. We say that a cell $C = \langle b, \Sigma, \Phi \rangle$ *wraps around* D when either C is equal to D or $b = \top$ and Σ wraps around D. This is a special case of a broader notion from the previous section: C wraps around D exactly when the singleton stack D covers C. We say that C *releases* D, when C wraps around D and for any stack ℓ that is aligned with C and releases D, we have $C \circ \ell \Leftrightarrow \forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z})$, where C' and ℓ' are obtained by replacing every occurrence of D in Σ and ℓ, respectively (including recursively inside nested cell contexts), with the cell $D' = \langle \bot, \varnothing, \lambda \overline{x:\tau}.\, z_1 = x_1 \wedge \cdots \wedge z_n = x_n \rightarrow \mathbf{0} \rangle$.

Lemma 10. *Any cell D of type $\overline{x:\tau}$, where none of the types τ_i is $\texttt{Type}$, releases itself.*

Proof. Since D cannot appear in its own cell context, and any aligned stack is a sequence of terms, we obtain $\forall \overline{z:\tau}.(D \circ \bar{s}) \vee (D \circ \bar{z}) \Leftrightarrow (D \circ \bar{s}) \vee (\forall \overline{z:\tau}.D \circ \bar{z}) \Leftrightarrow D \circ \bar{s}$. □

Lemma 11. *Consider a cell D of type $\overline{x:\tau}$, where none of the term types τ_i is $\texttt{Type}$. Any cell C that wraps around D, releases D.*

Proof. It is sufficient to prove that every cell $C = \langle \top, \Sigma, \Upsilon \rangle$, where Σ releases D, also releases D. Once this is established, a simple induction over cell depth together with Lemma 10 allow us to conclude. We proceed by induction over the size of Υ, counting only the subrecipes, so that term substitutions do not affect the size. Since first component of C is $\top$ and Σ wraps around D, the cell C also wraps around D. Consider a stack ℓ that is aligned with C and releases D. Let Σ' and ℓ' denote the result of replacing all occurrences of D throughout Σ and ℓ, respectively, with the cell $D' = \langle \bot, \varnothing, \lambda \overline{x:\tau}.\, z_1 = x_1 \wedge \cdots \wedge z_n = x_n \rightarrow \mathbf{0} \rangle$. Let C' be the cell $\langle \top, \Sigma', \Upsilon \rangle$. We need to prove that $C \circ \ell \Leftrightarrow \forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z})$.

Case Υ is h and $\Sigma(h) = D$. Then ℓ is a sequence of terms $\bar{s}$, and $\ell' = \ell$. We have

$$
\begin{aligned}
\forall \overline{z:\tau}.(C' \circ \bar{s}) \vee (D \circ \bar{z}) &= \forall \overline{z:\tau}.(D' \circ \bar{s}) \vee (D \circ \bar{z}) \\
&= \forall \overline{z:\tau}.(z_1 = s_1 \wedge \cdots \wedge z_n = s_n \rightarrow \bot) \vee (D \circ \bar{z}) \\
&\Leftrightarrow \forall \overline{z:\tau}.z_1 = s_1 \wedge \cdots \wedge z_n = s_n \rightarrow D \circ \bar{z} \\
&\Leftrightarrow D \circ \bar{s} \;=\; C \circ \bar{s}
\end{aligned}
$$

Case Υ is h and $\Sigma(h) \neq D$. By hypothesis, $\Sigma(h)$ releases D. Let C'' denote $\Sigma(h)$ after replacing all occurrences of D in its cell context with D'. We have $C \circ \ell = \Sigma(h) \circ \ell \Leftrightarrow \forall \overline{z:\tau}.(C'' \circ \ell') \vee (D \circ \bar{z}) = \forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z})$.

Case Υ is $\mathbf{0}$. Since the first component of C is $\top$, we have $C \circ \varepsilon = C' \circ \varepsilon = \top$.

Case Υ is $\natural\Phi$. Let Σ'' be the result of replacing all occurrences of D in $\natural\Sigma$ with D'. By induction hypothesis, $C \circ \ell = \langle \top, \natural\Sigma, \Phi \rangle \circ \ell \Leftrightarrow \forall \overline{z:\tau}.(\langle \top, \Sigma'', \Phi \rangle \circ \ell') \vee (D \circ \bar{z})$. Also, $\forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z}) = \forall \overline{z:\tau}.(\langle \top, \natural\Sigma', \Phi \rangle \circ \ell') \vee (D \circ \bar{z})$. If D is neutralized (that is, $D = \natural D$), then $D \circ \bar{z} \Leftrightarrow \top$. If D is not neutralized, then it cannot occur in $\natural\Sigma$, and so $\Sigma'' = \natural\Sigma$. Furthermore, $\natural D \equiv \natural D'$, and consequently, $\natural\Sigma' \equiv \natural\Sigma$.

Case Υ is $\Phi\,\Psi$ and $\langle \top, \Sigma, \Psi \rangle = D$. Then Σ cannot contain D, and $\Sigma' = \Sigma$. As the first component of D is $\top$ and Σ wraps around D, the cell D is neutralized. Consequently, $\natural D = D$ and both cells release D by Lemma 10. We obtain $C \circ \ell = \langle \top, \Sigma, \Phi \rangle \circ D, \ell \Leftrightarrow$

$\forall \overline{z:\tau}.(\langle\top,\Sigma,\Phi\rangle \circ D',\ell') \vee (D \circ \bar{z}) \Leftrightarrow \top \Leftrightarrow \forall \overline{z:\tau}.(\langle\top,\Sigma,\Phi\rangle \circ D,\ell') \vee (D \circ \bar{z}) \Leftrightarrow \forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z})$.

Case Υ is $\Phi\,\Psi$ and $\langle\top,\Sigma,\Psi\rangle \neq D$. The cell $\langle\top,\Sigma,\Psi\rangle$ releases D by the induction hypothesis, and so does its neutralization. Then $C \circ \ell = \langle\top,\Sigma,\Phi\rangle \circ \langle\top,\Sigma,\Psi\rangle,\ell \Leftrightarrow \forall \overline{z:\tau}.(\langle\top,\Sigma',\Phi\rangle \circ \langle\top,\Sigma',\Psi\rangle,\ell') \vee (D \circ \bar{z}) \Leftrightarrow \forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z})$.

Case Υ is $\Phi \wedge \Psi$. Then we have

$$\begin{aligned}
C \circ \ell &= (\langle\top,\Sigma,\Phi\rangle \circ \ell) \wedge (\langle\top,\Sigma,\Psi\rangle \circ \ell)\\
&\Leftrightarrow (\forall \overline{z:\tau}.(\langle\top,\Sigma',\Phi\rangle \circ \ell') \vee (D \circ \bar{z})) \wedge (\forall \overline{z:\tau}.(\langle\top,\Sigma',\Psi\rangle \circ \ell') \vee (D \circ \bar{z}))\\
&\Leftrightarrow \forall \overline{z:\tau}.(\langle\top,\Sigma',\Phi\rangle \circ \ell' \wedge \langle\top,\Sigma',\Psi\rangle \circ \ell') \vee (D \circ \bar{z})\\
&\Leftrightarrow \forall \overline{z:\tau}.(C' \circ \ell') \vee (D \circ \bar{z})
\end{aligned}$$

The rest of the cases are handled in a similar way. □

Lemma 12 (Lemma 6). *Consider a cell* $C = \langle b,\Sigma,\lambda g:(\overline{x:\tau}).\Phi\rangle$ *and an aligned stack* D,ℓ*, such that none of* τ_i *is* Type*. Let* D' *be* $\langle\bot,\varnothing,\lambda\overline{x:\tau}.z_1 = x_1 \wedge \cdots \wedge z_n = x_n \rightarrow \mathbf{0}\rangle$ *for some fresh variables* $\bar{z}$*. Then* $C \circ D,\ell \Leftrightarrow C \circ \natural D,\ell \wedge \forall \overline{z:\tau}.(\natural C \circ D',\natural\ell) \vee (D \circ \bar{z})$.

Proof. By Lemma 4, $C \circ D,\ell \Leftrightarrow C \circ \natural D,\ell \wedge \natural C \circ D,\natural\ell$. If D is neutralized, then $C \circ D,\ell = C \circ \natural D,\ell$ and $\natural C \circ D,\natural\ell \Leftrightarrow \top$. Otherwise, D does not occur in $\natural C$ or $\natural\ell$. We conclude by Lemma 11, as neutralized cells wrap around any cell. □

D Verification Condition Generation

Lemma 13. *For any expression e and term substitution σ, we have $\mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e)\sigma = \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e\sigma)$.*

Proof. By structural induction over e. □

Lemma 14. *For any expression e and handler symbols f, g, we have $\mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e)[g \mapsto f] = \mathbb{C}^{\mathfrak{p}}_{\mathfrak{d}}(e[g \mapsto f])$.*

Proof. By structural induction over e. □

Lemma 15. *Consider a cell C of the form $\langle b, \Sigma, \Phi[g \mapsto f]\rangle$. Let b' be an arbitrary Boolean value, and Σ' a cell context such that $\Sigma'(f) = \Sigma(f)$. Then C is equivalent to $\langle b, \Sigma \uplus [g \mapsto \langle b', \Sigma', f\rangle], \Phi\rangle$.*

Proof. By structural induction over Φ. □

Lemma 16 (Lemma 7). *For any expression e_0, any cell $D = \langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(e_0)\rangle$ is neutral.*

Proof. We proceed by induction over the size of e_0, counting only the subexpressions, so that term substitutions do not affect the size. Let ℓ be an aligned neutral stack. We need to prove that the formula $D \circ \ell$ is valid.

Case e_0 is h. Then $D \circ \ell = \langle b, \Sigma, \natural h\rangle \circ \ell = \langle \top, \natural\Sigma, h\rangle \circ \ell$ and Lemma 1 applies.

Case e_0 is $\pi \rightarrow e$. Then, for some cell context Σ' and substitution σ, we have

$$\begin{aligned}
D \circ \ell &= \langle b, \Sigma, (\lambda\pi.\mathbb{C}^{\bot}_{\bot}(e)) \wedge \natural(\lambda\pi.\mathbb{C}^{\top}_{\top}(e))\rangle \circ \ell \\
&= \langle b, \Sigma, \lambda\pi.\mathbb{C}^{\bot}_{\bot}(e)\rangle \circ \ell \wedge \langle \top, \natural\Sigma, \lambda\pi.\mathbb{C}^{\top}_{\top}(e)\rangle \circ \ell \\
&\Leftrightarrow \langle b, \Sigma, \lambda\pi.\mathbb{C}^{\bot}_{\bot}(e)\rangle \circ \ell \wedge \top && \text{(Lem. 1)} \\
&\Leftrightarrow \langle b, \Sigma', \mathbb{C}^{\bot}_{\bot}(e)\sigma\rangle \circ \varepsilon \\
&= \langle b, \Sigma', \mathbb{C}^{\bot}_{\bot}(e\sigma)\rangle \circ \varepsilon && \text{(Lem. 13)} \\
&\Leftrightarrow \top && \text{(IH)}
\end{aligned}$$

Case e_0 is $k\ \bar{s}\ \bar{o}$. Then we have

$$\begin{aligned}
D \circ \ell &= \langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(k)\ \bar{s}\ \mathbb{C}^{\bot}_{\bot}(o_1)\ \ldots\ \mathbb{C}^{\bot}_{\bot}(o_n)\rangle \circ \ell \\
&= \langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(k)\rangle \circ \bar{s}, \langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(o_1)\rangle, \ldots, \langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(o_n)\rangle, \ell
\end{aligned}$$

Each cell $\langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(o_i)\rangle$ added to the stack is neutral by the induction hypothesis and the neutralization of this cell is neutral by Lemma 1. Thus the new stack is neutral and we conclude by the induction hypothesis.

Case e_0 is $\{\varphi\}\ e$. The stack ℓ is empty and we have

$$\begin{aligned}
D \circ \varepsilon &= \langle b, \Sigma, (\varphi \rightarrow \mathbb{C}^{\bot}_{\bot}(e)) \wedge (\bot \rightarrow \neg\varphi \rightarrow \mathbf{0})\rangle \circ \varepsilon \\
&= \langle b, \Sigma, \varphi \rightarrow \mathbb{C}^{\bot}_{\bot}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma, \bot \rightarrow \neg\varphi \rightarrow \mathbf{0}\rangle \circ \varepsilon \\
&= (\varphi \rightarrow \langle b, \Sigma, \mathbb{C}^{\bot}_{\bot}(e)\rangle \circ \varepsilon) \wedge (\bot \rightarrow \neg\varphi \rightarrow \langle b, \Sigma, \mathbf{0}\rangle \circ \varepsilon) \\
&\Leftrightarrow (\varphi \rightarrow \top) \wedge (\bot \rightarrow \neg\varphi \rightarrow b) && \text{(IH)} \\
&\Leftrightarrow \top \wedge \top
\end{aligned}$$

Case e_0 is e / $h\ \pi = d$. Let Σ_h be $\Sigma \uplus [h \mapsto \langle b, \Sigma, \lambda\pi.\forall h{:}\pi.\complement^{\top}_{\bot}(d)\rangle]$ and $\overline{x:\tau}$ the term parameters in π. The stack ℓ is empty and, for some cell context Σ', we have

$$\begin{aligned} D \circ \varepsilon &= \langle b, \Sigma, \textbf{let}\ h\ \pi = \complement^{\top}_{\bot}(d)\ \textbf{in}\ \complement^{\bot}_{\bot}(e) \wedge \forall\pi.\complement^{\bot}_{\bot}(d)\rangle \circ \varepsilon \\ &= \langle b, \Sigma_h, \complement^{\bot}_{\bot}(e) \wedge \forall\pi.\complement^{\bot}_{\bot}(d)\rangle \circ \varepsilon \\ &= \langle b, \Sigma_h, \complement^{\bot}_{\bot}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma_h, \forall\pi.\complement^{\bot}_{\bot}(d)\rangle \circ \varepsilon \\ &= \langle b, \Sigma_h, \complement^{\bot}_{\bot}(e)\rangle \circ \varepsilon \wedge \forall\overline{x:\tau}.\langle b, \Sigma', \complement^{\bot}_{\bot}(d)\rangle \circ \varepsilon \\ &\Leftrightarrow \top \wedge \top && \text{(IH)} \end{aligned}$$

Case e_0 is $\uparrow e$ or $\downarrow e$. Then $\complement^{\bot}_{\bot}(e_0)$ is $\complement^{\bot}_{\bot}(e)$, and the induction hypothesis applies. □

Corollary 4. *Any cell of the form* $\langle b, \Sigma, \complement^{\top}_{\top}(\pi \to e)\rangle$ *is equivalent to* $\langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle$.

Proof. For any aligned stack ℓ, there exist a cell context Σ' and a term substitution σ such that

$$\begin{aligned} \langle b, \Sigma, \complement^{\top}_{\top}(\pi \to e)\rangle \circ \ell &= \langle \top, \natural\Sigma, (\lambda\pi.\complement^{\top}_{\top}(e)) \wedge \lambda\pi.\complement^{\bot}_{\bot}(e)\rangle \circ \ell \\ &= \langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle \circ \ell \wedge \langle b, \Sigma, \natural\lambda\pi.\complement^{\bot}_{\bot}(e)\rangle \circ \ell \\ &= \langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle \circ \ell \wedge \langle \top, \Sigma', \complement^{\bot}_{\bot}(e)\sigma\rangle \circ \varepsilon \\ &= \langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle \circ \ell \wedge \langle \top, \Sigma', \complement^{\bot}_{\bot}(e\sigma)\rangle \circ \varepsilon && \text{(Lem. 13)} \\ &\Leftrightarrow \langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle \circ \ell \wedge \top && \text{(Lem. 7)} \\ &\Leftrightarrow \langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle \circ \ell && \square \end{aligned}$$

Lemma 17. *For any* Coma *expression* e_0 *and cell context* Σ, *the neutralized cells* $D_1 = \langle \top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e_0)\rangle$ *and* $D_2 = \langle \top, \natural\Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e_0)\rangle$ *are equivalent.*

Proof. We proceed by structural induction over e_0. By Lemma 3, it suffices to prove that $D_1 \circ \ell \Leftrightarrow D_2 \circ \ell$ for any aligned stack ℓ.

Case e_0 is h. We can assume, without loss of generality, that $\mathfrak{p}$ is $\top$. Then we have $D_2 \circ \ell = \langle \top, \natural\Sigma, \natural h\rangle \circ \ell = \langle \top, \natural\natural\Sigma, h\rangle \circ \ell = \langle \top, \natural\Sigma, h\rangle \circ \ell = D_1 \circ \ell$, by the idempotence of neutralization.

Case e_0 is $\pi \to e$. Given that $\natural\natural\Sigma = \natural\Sigma$, we have

$$\begin{aligned} D_1 \circ \ell &= \langle \top, \natural\Sigma, (\lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)) \wedge \natural(\lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e))\rangle \circ \ell \\ &= \langle \top, \natural\Sigma, \lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle \top, \natural\Sigma, \natural\lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \\ &= \langle \top, \natural\natural\Sigma, \lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle \top, \natural\natural\Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \\ &= \langle \top, \natural\Sigma, \natural\lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle \top, \natural\Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \\ &\Leftrightarrow \langle \top, \natural\Sigma, \natural\lambda\pi.\complement^{\neg\neg\mathfrak{p}}_{\neg\neg\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle \top, \natural\Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \\ &= \langle \top, \natural\Sigma, (\lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)) \wedge \natural(\lambda\pi.\complement^{\neg\neg\mathfrak{p}}_{\neg\neg\mathfrak{d}}(e))\rangle \circ \ell \\ &= D_2 \circ \ell \end{aligned}$$

Case e_0 is $k\ \bar{s}\ \bar{o}$. Then we have $D_1 \circ \ell = \langle \top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(k)\ \bar{s}\ \complement^{\mathfrak{p}}_{\mathfrak{d}}(o_1)\ \ldots\ \complement^{\mathfrak{p}}_{\mathfrak{d}}(o_n)\rangle \circ \ell = \langle \top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(k)\rangle \circ \bar{s}, \langle \top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(o_1)\rangle, \ldots, \langle \top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(o_n)\rangle, \ell$. Each cell $\langle \top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(o_i)\rangle$

is equivalent to $\langle\top, \natural\Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(o_i)\rangle$ by induction hypothesis. These cells are neutralized, hence the new stack is equivalent to $\bar{s}, \langle\top, \natural\Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(o_1)\rangle, \ldots, \langle\top, \natural\Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(o_n)\rangle, \ell$. Since the cells $\langle\top, \natural\Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(k)\rangle$ and $\langle\top, \natural\Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(k)\rangle$ are also equivalent by induction hypothesis, we obtain $D_1 \circ \ell \Leftrightarrow D_2 \circ \ell$ by definition of cell equivalence.

In all other cases, the expression e_0 is fully applied, and so are the cells D_1 and D_2. Then the stack ℓ is empty, and $D_1 \circ \varepsilon \Leftrightarrow \top \Leftrightarrow D_2 \circ \varepsilon$ by Lemma 1. □

Lemma 18 (Lemma 8). *Given any expression e_0 and three cells $D = \langle b, \Sigma, \complement^{\top}_{\top}(e_0)\rangle$, $D_1 = \langle b, \Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e_0)\rangle$, and $D_2 = \langle b, \Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e_0)\rangle$, the cell D is a meet of D_1 and D_2.*

Proof. By Lemma 17, we have $\natural D_1 \equiv \natural D_2$. We proceed by induction on the size of e_0. By Lemma 5, it suffices to prove that $D \circ \ell \Leftrightarrow D_1 \circ \ell \wedge D_2 \circ \ell$ for any aligned stack ℓ.

Case e_0 is h. We can assume, without loss of generality, that $\mathfrak{p}$ is $\top$. Then D is the same as D_1, and $D_2 \circ \ell$ is equal to $\natural D_1 \circ \ell$. Then $D \circ \ell \Leftrightarrow D_1 \circ \ell \wedge D_2 \circ \ell$ by Lemma 4.

Case e_0 is $\pi \twoheadrightarrow e$. Then, for some cell contexts Σ', Σ'' and term substitution σ, which are determined by π and ℓ, we have

$$\begin{aligned}
D \circ \ell \;&=\; \langle b, \Sigma, \complement^{\top}_{\top}(\pi \twoheadrightarrow e)\rangle \circ \ell \Leftrightarrow \langle b, \Sigma, \lambda\pi.\complement^{\top}_{\top}(e)\rangle \circ \ell && \text{(Cor. 4)}\\
&=\; \langle b, \Sigma', \complement^{\top}_{\top}(e)\sigma\rangle \circ \varepsilon \;=\; \langle b, \Sigma', \complement^{\top}_{\top}(e\sigma)\rangle \circ \varepsilon && \text{(Lem. 13)}\\
&\Leftrightarrow \langle b, \Sigma', \complement^{\mathfrak{p}}_{\mathfrak{d}}(e\sigma)\rangle \circ \varepsilon \wedge \langle b, \Sigma', \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e\sigma)\rangle \circ \varepsilon && \text{(IH)}\\
&=\; \langle b, \Sigma', \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\sigma\rangle \circ \varepsilon \wedge \langle b, \Sigma', \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\sigma\rangle \circ \varepsilon && \text{(Lem. 13)}\\
&=\; \langle b, \Sigma, \lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle b, \Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell\\
&\Leftrightarrow \langle b, \Sigma, \lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge \natural\langle b, \Sigma, \lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge && (C \Rrightarrow \natural C)\\
&\quad\;\; \langle b, \Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \wedge \natural\langle b, \Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell\\
&\Leftrightarrow \langle b, \Sigma, \lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle b, \Sigma, \natural\lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \wedge\\
&\quad\;\; \langle b, \Sigma, \lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \ell \wedge \langle b, \Sigma, \natural\lambda\pi.\complement^{\neg\neg\mathfrak{p}}_{\neg\neg\mathfrak{d}}(e)\rangle \circ \ell\\
&=\; \langle b, \Sigma, (\lambda\pi.\complement^{\mathfrak{p}}_{\mathfrak{d}}(e)) \wedge \natural(\lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e))\rangle \circ \ell \wedge\\
&\quad\;\; \langle b, \Sigma, (\lambda\pi.\complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)) \wedge \natural(\lambda\pi.\complement^{\neg\neg\mathfrak{p}}_{\neg\neg\mathfrak{d}}(e))\rangle \circ \ell\\
&=\; D_1 \circ \ell \wedge D_2 \circ \ell
\end{aligned}$$

Case e_0 is $\{\varphi\}\, e$. Then the stack ℓ is empty and we have

$$\begin{aligned}
D \circ \varepsilon \;&=\; \langle b, \Sigma, (\varphi \to \complement^{\top}_{\top}(e)) \wedge (\top \to \neg\varphi \to \mathbf{0})\rangle \circ \varepsilon\\
&=\; (\varphi \to \langle b, \Sigma, \complement^{\top}_{\top}(e)\rangle \circ \varepsilon) \wedge (\top \to \neg\varphi \to b)\\
&\Leftrightarrow (\varphi \to (\langle b, \Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \varepsilon)) \wedge (\neg\varphi \to b) && \text{(IH)}\\
&\Leftrightarrow (\varphi \to \langle b, \Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \varepsilon) \wedge (\mathfrak{p} \to \neg\varphi \to b) \wedge\\
&\quad\;\; (\varphi \to \langle b, \Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \varepsilon) \wedge (\neg\mathfrak{p} \to \neg\varphi \to b)\\
&=\; \langle b, \Sigma, (\varphi \to \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)) \wedge (\mathfrak{p} \to \neg\varphi \to \mathbf{0})\rangle \circ \varepsilon \wedge\\
&\quad\;\; \langle b, \Sigma, (\varphi \to \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)) \wedge (\neg\mathfrak{p} \to \neg\varphi \to \mathbf{0})\rangle \circ \varepsilon\\
&=\; D_1 \circ \varepsilon \wedge D_2 \circ \varepsilon
\end{aligned}$$

Case e_0 is $k\ \bar{s}\ \bar{o}$. Then we have $D \circ \ell = \langle b, \Sigma, \complement^{\top}_{\top}(k)\ \bar{s}\ \complement^{\top}_{\top}(o_1)\ \ldots\ \complement^{\top}_{\top}(o_n)\rangle \circ \ell = \langle b, \Sigma, \complement^{\top}_{\top}(k)\rangle \circ \bar{s}, \langle b, \Sigma, \complement^{\top}_{\top}(o_1)\rangle, \ldots, \langle b, \Sigma, \complement^{\top}_{\top}(o_n)\rangle, \ell$, and similarly for $D_1 \circ \ell$ and $D_2 \circ \ell$. Each cell $\langle b, \Sigma, \complement^{\top}_{\top}(o_i)\rangle$ is a meet of the cells $\langle b, \Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(o_i)\rangle$ and $\langle b, \Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(o_i)\rangle$ by induction hypothesis, and the same is true for their respective neutralizations. Furthermore, the cell $\langle b, \Sigma, \complement^{\top}_{\top}(k)\rangle$ is also a meet of $\langle b, \Sigma, \complement^{\mathfrak{p}}_{\mathfrak{d}}(k)\rangle$ and $\langle b, \Sigma, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(k)\rangle$ by induction hypothesis. Then $D \circ \ell \Leftrightarrow D_1 \circ \ell \wedge D_2 \circ \ell$.

Case e_0 is $e\ /\ h\ \pi = d$. Let Σ_h be $\Sigma \uplus [h \mapsto \langle b, \Sigma, \lambda\pi.\forall h{:}\pi.\complement^{\top}_{\bot}(d)\rangle]$ and $\overline{x:\tau}$ be the term parameters in π. The stack ℓ is empty and, for some cell context Σ', which is determined by π, we have

$$
\begin{aligned}
D \circ \varepsilon\ &=\ \langle b, \Sigma, \mathbf{let}\ h\ \pi = \complement^{\top}_{\bot}(d)\ \mathbf{in}\ \complement^{\top}_{\top}(e) \wedge \forall\pi.\complement^{\bot}_{\top}(d)\rangle \circ \varepsilon \\
&=\ \langle b, \Sigma_h, \complement^{\top}_{\top}(e) \wedge \forall\pi.\complement^{\bot}_{\top}(d)\rangle \circ \varepsilon \\
&=\ \langle b, \Sigma_h, \complement^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma_h, \forall\pi.\complement^{\bot}_{\top}(d)\rangle \circ \varepsilon \\
&=\ \langle b, \Sigma_h, \complement^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \forall\overline{x:\tau}.\langle b, \Sigma', \complement^{\bot}_{\top}(d)\rangle \circ \varepsilon \\
&\Leftrightarrow\ \langle b, \Sigma_h, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma_h, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \forall\overline{x:\tau}.\langle b, \Sigma', \complement^{\bot}_{\top}(d)\rangle \circ \varepsilon \qquad \text{(IH)} \\
&\Leftrightarrow\ \langle b, \Sigma_h, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \forall\overline{x:\tau}.\langle b, \Sigma', \complement^{\bot}_{\mathfrak{p}}(d)\rangle \circ \varepsilon\ \wedge \qquad \text{(Lem. 7)} \\
&\quad\ \ \langle b, \Sigma_h, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \forall\overline{x:\tau}.\langle b, \Sigma', \complement^{\bot}_{\neg\mathfrak{p}}(d)\rangle \circ \varepsilon \\
&=\ \langle b, \Sigma_h, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma_h, \forall\pi.\complement^{\bot}_{\mathfrak{p}}(d)\rangle \circ \varepsilon\ \wedge \\
&\quad\ \ \langle b, \Sigma_h, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e)\rangle \circ \varepsilon \wedge \langle b, \Sigma_h, \forall\pi.\complement^{\bot}_{\neg\mathfrak{p}}(d)\rangle \circ \varepsilon \\
&=\ \langle b, \Sigma_h, \complement^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \forall\pi.\complement^{\bot}_{\mathfrak{p}}(d)\rangle \circ \varepsilon\ \wedge \\
&\quad\ \ \langle b, \Sigma_h, \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e) \wedge \forall\pi.\complement^{\bot}_{\neg\mathfrak{p}}(d)\rangle \circ \varepsilon \\
&=\ \langle b, \Sigma, \mathbf{let}\ h\ \pi = \complement^{\top}_{\bot}(d)\ \mathbf{in}\ \complement^{\mathfrak{p}}_{\mathfrak{d}}(e) \wedge \forall\pi.\complement^{\bot}_{\mathfrak{p}}(d)\rangle \circ \varepsilon\ \wedge \\
&\quad\ \ \langle b, \Sigma, \mathbf{let}\ h\ \pi = \complement^{\top}_{\bot}(d)\ \mathbf{in}\ \complement^{\neg\mathfrak{p}}_{\neg\mathfrak{d}}(e) \wedge \forall\pi.\complement^{\bot}_{\neg\mathfrak{p}}(d)\rangle \circ \varepsilon \\
&=\ D_1 \circ \varepsilon \wedge D_2 \circ \varepsilon
\end{aligned}
$$

Case e_0 is $\uparrow e$ or $\downarrow e$. Straightforward by the induction hypothesis. □

Theorem 5 (Theorem 3). *For any Coma programs e_1 and e_2, if e_1 is correct and $e_1 \longrightarrow e_2$, then e_2 is correct.*

Proof. Given a sequence of handler definitions Λ, we define a cell context Σ_Λ and a formula ψ_Λ recursively as follows:

$$
\begin{aligned}
\Sigma_\varepsilon &\triangleq \Sigma_{\text{prim}} & \Sigma_{h\,\pi=d,\Lambda} &\triangleq \Sigma_\Lambda \uplus [h \mapsto \langle \bot, \Sigma_\Lambda, \lambda\pi.\forall h{:}\pi.\complement^{\top}_{\bot}(d)\rangle] \\
\psi_\varepsilon &\triangleq \top & \psi_{h\,\pi=d,\Lambda} &\triangleq \langle \bot, \Sigma_{h\,\pi=d,\Lambda}, \forall\pi.\complement^{\bot}_{\top}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda
\end{aligned}
$$

Informally, Σ_Λ results from adding to Σ_{prim} the specifications of handlers defined in Λ, and ψ_Λ is a conjunction of proof obligations for these handler definitions. We can show by structural induction that for any program $e\ /\!/\ \Lambda$, the formula $\langle \bot, \Sigma_{\text{prim}}, \complement^{\top}_{\top}(e\ /\!/\ \Lambda)\rangle \circ \varepsilon$ is logically equivalent to $\langle \bot, \Sigma_\Lambda, \complement^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \psi_\Lambda$.

We proceed by case analysis of the evaluation step $e_1 \longrightarrow e_2$. Provided that e_1 is of the form $e\ /\!/\ \Lambda$ and e_2 of the form $e'\ /\!/\ \Lambda$, we need to show that $\langle \bot, \Sigma_\Lambda, \complement^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \psi_\Lambda$ entails $\langle \bot, \Sigma_\Lambda, \complement^{\top}_{\top}(e')\rangle \circ \varepsilon \wedge \psi_\Lambda$.

Case E-SYM. Then e_1 is of the form $h\ \bar{s}\ \bar{o} \mathbin{/\!/} \Lambda$ and e_2 of the form $(\pi \rightarrow d)\ \bar{s}\ \bar{o} \mathbin{/\!/} \Lambda$. Let ℓ denote the stack $\bar{s}, \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_1)\rangle, \ldots, \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_n)\rangle$. Assuming that the type signature π is of the form $\overline{x:\tau}\ \overline{g:\varrho}$, let σ be the term substitution $[\bar{x} \mapsto \bar{s}]$ and Σ' the cell context $\Sigma_\Lambda \uplus [g_1 \mapsto \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_1)\rangle, \ldots, g_n \mapsto \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_n)\rangle]$. Assuming that Λ is of the form $\Lambda_1, h\,\pi = d, \Lambda_2$, we obtain

$$\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(h\ \bar{s}\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(h)\rangle \circ \ell \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_{\Lambda_2}, \lambda\pi.\forall h{:}\pi.\mathbb{C}^{\top}_{\bot}(d)\rangle \circ \ell \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_{\Lambda_2}, \lambda\pi.\forall h{:}\pi.\mathbb{C}^{\top}_{\bot}(d)\rangle \circ \ell \wedge \langle \bot, \Sigma_{h\,\pi = d, \Lambda_2}, \forall\pi.\mathbb{C}^{\bot}_{\top}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_{\Lambda}, \lambda\pi.\forall h{:}\pi.\mathbb{C}^{\top}_{\bot}(d)\rangle \circ \ell \wedge \langle \bot, \Sigma_{\Lambda}, \forall\pi.\mathbb{C}^{\bot}_{\top}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad \Rightarrow \langle \bot, \Sigma_{\Lambda}, \lambda\pi.\forall h{:}\pi.\mathbb{C}^{\top}_{\bot}(d)\rangle \circ \ell \wedge \langle \bot, \Sigma_{\Lambda}, \lambda\pi.\mathbb{C}^{\bot}_{\top}(d)\rangle \circ \ell \wedge \psi_\Lambda && \text{(Thm. 2)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma', \forall h{:}\pi.\mathbb{C}^{\top}_{\bot}(d\sigma)\rangle \circ \varepsilon \wedge \langle \bot, \Sigma', \mathbb{C}^{\bot}_{\top}(d\sigma)\rangle \circ \varepsilon \wedge \psi_\Lambda && \text{(Lem. 13)} \\
&\quad \Rightarrow \langle \bot, \Sigma', \mathbb{C}^{\top}_{\bot}(d\sigma)\rangle \circ \varepsilon \wedge \langle \bot, \Sigma', \mathbb{C}^{\bot}_{\top}(d\sigma)\rangle \circ \varepsilon \wedge \psi_\Lambda && \text{(Thm. 2)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma', \mathbb{C}^{\top}_{\top}(d\sigma)\rangle \circ \varepsilon \wedge \psi_\Lambda && \text{(Lem. 8)} \\
&\quad = \langle \bot, \Sigma_\Lambda, \lambda\pi.\mathbb{C}^{\top}_{\top}(d)\rangle \circ \ell \wedge \psi_\Lambda && \text{(Lem. 13)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(\pi \rightarrow d)\rangle \circ \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((\pi \rightarrow d)\ \bar{s}\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda
\end{aligned}$$

Case E-APPT. Then e_1 is of the form $((x:\tau)\,\pi \rightarrow e)\ t\ \bar{s}\ \bar{o} \mathbin{/\!/} \Lambda$ and e_2 is of the form $(\pi \rightarrow e)[x \mapsto t]\ \bar{s}\ \bar{o} \mathbin{/\!/} \Lambda$. Let ℓ stand for $\bar{s}, \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_1)\rangle, \ldots, \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_n)\rangle$. Then

$$\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(((x:\tau)\,\pi \rightarrow e)\ t\ \bar{s}\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((x:\tau)\,\pi \rightarrow e)\rangle \circ t, \ell \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \lambda x:\tau.\lambda\pi.\mathbb{C}^{\top}_{\top}(e)\rangle \circ t, \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad = \langle \bot, \Sigma_\Lambda, (\lambda\pi.\mathbb{C}^{\top}_{\top}(e))[x \mapsto t]\rangle \circ \ell \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \lambda\pi[x \mapsto t].\mathbb{C}^{\top}_{\top}(e[x \mapsto t])\rangle \circ \ell \wedge \psi_\Lambda && \text{(Lem. 13)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(\pi \rightarrow e)[x \mapsto t]\rangle \circ \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((\pi \rightarrow e)[x \mapsto t])\rangle \circ \ell \wedge \psi_\Lambda && \text{(Lem. 13)} \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((\pi \rightarrow e)[x \mapsto t]\ \bar{s}\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda
\end{aligned}$$

Case E-APPH. Then e_1 is of the form $((g:\varrho)\,\pi \rightarrow e)\ f\ \bar{o} \mathbin{/\!/} \Lambda$ and e_2 is of the form $(\pi \rightarrow e)[g \mapsto f]\ \bar{o} \mathbin{/\!/} \Lambda$. Let ℓ stand for $\langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_1)\rangle, \ldots, \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(o_n)\rangle$. Then

$$\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(((g:\varrho)\,\pi \rightarrow e)\ f\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((g:\varrho)\,\pi \rightarrow e)\rangle \circ \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(f)\rangle, \ell \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \lambda g:\varrho.\lambda\pi.\mathbb{C}^{\top}_{\top}(e)\rangle \circ \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(f)\rangle, \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad = \langle \bot, \Sigma_\Lambda \uplus [g \mapsto \langle \bot, \Sigma_\Lambda, f\rangle], \lambda\pi.\mathbb{C}^{\top}_{\top}(e)\rangle \circ \ell \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda \uplus [g \mapsto \langle \bot, \Sigma_\Lambda, f\rangle], \mathbb{C}^{\top}_{\top}(\pi \rightarrow e)\rangle \circ \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}(\pi \rightarrow e)[g \mapsto f]\rangle \circ \ell \wedge \psi_\Lambda && \text{(Lem. 15)} \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((\pi \rightarrow e)[g \mapsto f])\rangle \circ \ell \wedge \psi_\Lambda && \text{(Lem. 14)} \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbb{C}^{\top}_{\top}((\pi \rightarrow e)[g \mapsto f]\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda
\end{aligned}$$

Case E-APPC. Then e_1 is of the form $((g : \varrho)\, \pi \to e)\ (\varrho \to d)\ \bar{o}\ /\!/\ \Lambda$ and e_2 of the form $(\pi \to e)\ \bar{o}\ /\ g\ \varrho = {\downarrow} d\ /\!/\ \Lambda$, assuming that g does not occur freely in d or in $\bar{o}$. Let Σ' denote the cell context $\Sigma_{g\varrho={\downarrow}d,\Lambda} = \Sigma_\Lambda \uplus [g \mapsto \langle \bot, \Sigma_\Lambda, \lambda\varrho.\forall g : \varrho.\mathcal{C}^{\top}_{\bot}({\downarrow} d)\rangle]$. It is easy to show that the stack $\ell = \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(o_1)\rangle, \ldots, \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(o_n)\rangle$ is equivalent to the stack $\ell' = \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(o_1)\rangle, \ldots, \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(o_n)\rangle$. Then we have

$$
\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(((g : \varrho)\, \pi \to e)\ (\varrho \to d)\ \bar{o})\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}((g : \varrho)\, \pi \to e)\rangle \circ \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(\varrho \to d)\rangle, \ell \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \lambda g : \varrho.\lambda\pi.\mathcal{C}^{\top}_{\top}(e)\rangle \circ \langle \bot, \Sigma_\Lambda, \lambda\varrho.\mathcal{C}^{\top}_{\top}(d)\rangle, \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \lambda g : \varrho.\lambda\pi.\mathcal{C}^{\top}_{\top}(e)\rangle \circ \langle \bot, \Sigma_\Lambda, \lambda\varrho.\forall g : \varrho.\mathcal{C}^{\top}_{\bot}({\downarrow} d)\rangle, \ell \wedge \psi_\Lambda && (*) \\
&\quad = \langle \bot, \Sigma', \lambda\pi.\mathcal{C}^{\top}_{\top}(e)\rangle \circ \ell \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(\pi \to e)\rangle \circ \ell \wedge \psi_\Lambda && \text{(Cor. 4)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(\pi \to e)\rangle \circ \ell \wedge \langle \bot, \Sigma', \forall\varrho.\mathcal{C}^{\bot}_{\bot}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda && \text{(Lem. 7)} \\
&\quad \Leftrightarrow \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(\pi \to e)\rangle \circ \ell' \wedge \langle \bot, \Sigma', \forall\varrho.\mathcal{C}^{\bot}_{\top}({\downarrow} d)\rangle \circ \varepsilon \wedge \psi_\Lambda && (\ell \equiv \ell') \\
&\quad = \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(\pi \to e)\ \bar{o} \wedge \forall\varrho.\mathcal{C}^{\bot}_{\top}({\downarrow} d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbf{let}\ g\ \varrho = \mathcal{C}^{\top}_{\bot}({\downarrow} d)\ \mathbf{in}\ \mathcal{C}^{\top}_{\top}(\pi \to e)\ \bar{o} \wedge \forall\varrho.\mathcal{C}^{\bot}_{\top}({\downarrow} d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}((\pi \to e)\ \bar{o}\ /\ g\ \varrho = {\downarrow} d)\rangle \circ \varepsilon \wedge \psi_\Lambda
\end{aligned}
$$

On the step marked (*), we can safely introduce the universal quantifier over g, since this handler symbol does not occur in $\mathcal{C}^{\top}_{\top}(d)$.

Case E-VOID. Then e_1 is of the form $\square \to e\ /\!/\ \Lambda$ and e_2 of the form $e\ /\!/\ \Lambda$. We have $\langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(\square \to e)\rangle \circ \varepsilon \wedge \psi_\Lambda \Leftrightarrow \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \psi_\Lambda$ by Corollary 4.

Case E-PROP. Then e_1 is of the form $\{\varphi\}\ e\ /\!/\ \Lambda$, where the formula φ is valid, and e_2 is of the form $e\ /\!/\ \Lambda$. We obtain

$$
\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(\{\varphi\}\ e)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, (\varphi \to \mathcal{C}^{\top}_{\top}(e)) \wedge (\top \to \neg\varphi \to \mathbf{0})\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = (\varphi \to \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon) \wedge (\top \to \neg\varphi \to \bot) \wedge \psi_\Lambda \\
&\quad \Leftrightarrow (\varphi \to \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon) \wedge \varphi \wedge \psi_\Lambda \\
&\quad \Rightarrow \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \psi_\Lambda
\end{aligned}
$$

Case E-GC. Then e_1 is of the form $e\ /\ h\,\pi = d\ /\!/\ \Lambda$, where h does not occur in e, and e_2 is $e\ /\!/\ \Lambda$. Let Σ' stand for $\Sigma_{h\pi=d,\Lambda} = \Sigma_\Lambda \uplus [h \mapsto \langle \bot, \Sigma_\Lambda, \lambda\pi.\forall h : \pi.\mathcal{C}^{\top}_{\bot}(d)\rangle]$. Then

$$
\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(e\ /\ h\,\pi = d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma_\Lambda, \mathbf{let}\ h\ \pi = \mathcal{C}^{\top}_{\bot}(d)\ \mathbf{in}\ \mathcal{C}^{\top}_{\top}(e) \wedge \forall\pi.\mathcal{C}^{\bot}_{\top}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(e) \wedge \forall\pi.\mathcal{C}^{\bot}_{\top}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \langle \bot, \Sigma', \forall\pi.\mathcal{C}^{\bot}_{\top}(d)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad \Rightarrow \langle \bot, \Sigma', \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad \Leftrightarrow \langle \bot, \Sigma_\Lambda, \mathcal{C}^{\top}_{\top}(e)\rangle \circ \varepsilon \wedge \psi_\Lambda && (h \text{ does not occur in } e)
\end{aligned}
$$

Cases E-BBOX and E-WBOX are trivial.

The rest of the cases correspond to the use of primitive handlers. For example, let expression e_1 be of the form $\texttt{if}\ s\ k\ o \mathbin{/\!/} \Lambda$. Then we have

$$\begin{aligned}
&\langle \bot, \Sigma_\Lambda, \complement^\top_\top(\texttt{if}\ s\ k\ o)\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = \langle \bot, \varnothing, \Psi_{\texttt{if}}\rangle \circ s, \langle \bot, \Sigma_\Lambda, \complement^\top_\top(k)\rangle, \langle \bot, \Sigma_\Lambda, \complement^\top_\top(o)\rangle \wedge \psi_\Lambda \\
&\quad = \langle \bot, [\texttt{then} \mapsto \langle \bot, \Sigma_\Lambda, \complement^\top_\top(k)\rangle, \texttt{else} \mapsto \langle \bot, \Sigma_\Lambda, \complement^\top_\top(o)\rangle], \\
&\qquad\quad (s \to \texttt{then}) \wedge (\neg s \to \texttt{else})\rangle \circ \varepsilon \wedge \psi_\Lambda \\
&\quad = (s \to \langle \bot, \Sigma_\Lambda, \complement^\top_\top(k)\rangle \circ \varepsilon) \wedge (\neg s \to \langle \bot, \Sigma_\Lambda, \complement^\top_\top(o)\rangle \circ \varepsilon) \wedge \psi_\Lambda
\end{aligned}$$

Depending on whether we progressed along the first or the second rule for `if`, the last formula entails either $\langle \bot, \Sigma_\Lambda, \complement^\top_\top(k)\rangle \circ \varepsilon \wedge \psi_\Lambda$ or $\langle \bot, \Sigma_\Lambda, \complement^\top_\top(o)\rangle \circ \varepsilon \wedge \psi_\Lambda$.

The applications of `unTree` and `get` are treated in the similar way. □

Theorem 6 (Theorem 4). *For any correct* Coma *program* e_1*, either* e_1 *is* `halt`*, or* $e_1 \longrightarrow e_2$ *for some program* e_2*.*

Proof. Here we only consider the premise of the E-Prop rule, as well as the premises of the evaluation rules for the primitive handlers—the rest of the proof is straightforward. Just like in the proof of Theorem 3, we will use the fact that for any program $e \mathbin{/\!/} \Lambda$, the formula $\langle \bot, \Sigma_{\text{prim}}, \complement^\top_\top(e \mathbin{/\!/} \Lambda)\rangle \circ \varepsilon$ is logically equivalent to $\langle \bot, \Sigma_\Lambda, \complement^\top_\top(e)\rangle \circ \varepsilon \wedge \psi_\Lambda$. Then we can easily show that the premise of E-Prop, as well as the first premise of the evaluation rule for `get`, are logical consequences of the verification condition of e_1. Furthermore, the standard model for Booleans, integers, binary trees, and sequences, which we adopt for our semantics, guarantees that every ground Boolean term is either $\top$ or $\bot$, every ground term of type `tree` τ is either a `Node` or an `Empty`, every sequence contains an element for every valid index, etc. □