A Coinductive Calculus for Asynchronous Side-effecting Processes

Sergey Goncharov*, Lutz Schröder*

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Martenstraße 3, 91058, Erlangen, Germany

Abstract

We present an abstract framework for concurrent processes in which atomic steps have generic side effects, handled according to the principle of monadic encapsulation of effects. Processes in this framework are potentially infinite resumptions, modelled using final coalgebras over the monadic base. As a calculus for such processes, we introduce a concurrent extension of Moggi's monadic meta-language of effects. We establish soundness and completeness of a natural equational axiomatization of this calculus. Our main result is a corecursion scheme that is explicitly definable over the base language and provides flexible expressive means for the definition of new operators on processes, such as parallel composition. Moreover, we present initial results on verification methods for generic side-effecting processes.

Keywords: corecursive scheme, computational monad, process algebra, coinduction, computational meta-language.

1. Introduction

Imperative programming languages work with many different side effects, such as I/O, state, exceptions, and others, which all moreover come with strong variations in detail. This variety is unified by the principle of monadic encapsulation of side-effects [33], which not only underlies extensive work in semantics (e.g. [25]) but, following Wadler [49], forms the basis for the treatment of side-effects in functional languages such as Haskell [37] and F# [46].

More generally, monads can be used to capture computations on a more abstract level, such as presented by various process algebras. The arising issue of handling concurrency is addressed by variants of the *resumption monad transformer* [7, 19], which lifts resumptions in the style of Hennessy and Plotkin [21] to the monadic level, and which has moreover been used in information flow security [20], semantics of functional logic programming languages such as Curry [47], modelling underspecification of compilers, e.g. for ANSI C [36], and the semantics of the π-calculus [14]. However, there is to date no concurrent correspondent to Moggi's *computational meta-language* [33], the underlying formalism of Haskell's do-notation: this language is essentially limited to linear sequential monadic programs, and does not include native support for concurrency.

The objective of the present work is therefore to develop an extension of the computational meta-language that can serve as a minimal common basis for generic concurrent programming and semantics. In our view a side-effecting process is a sequence of elementary steps, each of which can yield a side-effect. We wish to avoid detailing the precise specification of such a side-effect, which may include sending and receiving messages, updating the internal state, or creating names. To this end, our calculus is parametrized by

*Corresponding author
Email addresses: sergey.goncharov@fau.de (Sergey Goncharov), lutz.schroeder@fau.de (Lutz Schröder)

a monad $\mathbb{T}$ for elementary steps, which contrasts our approach to the work on fully-abstract models of π-calculus [14, 45] featuring a concrete choice of the process monad.

The monad structure of $\mathbb{T}$ ensures composability of steps: one can merge two neighbouring steps to obtain their composite. The effect of such a transformation is that the gap between the first and the second step, originally open to possible intermediate steps of the environment, is closed. We will exploit this structure for verification purposes in Section 7.

Since the approach we take is abstract w.r.t. the the underlying side-effect, we argue that our calculus provides a common basis for message-passing and shared-variable concurrency. While the latter is naturally modelled by the state monad, in the former case one can use, e.g., the monad $TA = \mu X.\ (A \times X + X^A + X)$, where A now plays the role of an alphabet (this monad has been used, e.g., in a monad-based model of the π-calculus [45]). A step of a process would be then a finite series of inputs, outputs, and silent actions (τ) in the spirit of process algebras such as CCS. The concept of packaging several actions into one process step may sound surprising initially, but is required to obtain the necessary monadic structure; it generalizes the expected setup of one action per step and thus allows for non-interruptible blocks of actions.

Unlike the notion of *process interleaving*, which is shared by message-passing and shared-variable concurrency, *message synchronization* is a specific feature of the former. In general, two effects can be performed asynchronously, i.e. one after another, but not synchronously. In this paper we therefore focus on the asynchronous processes. Adding the possibility of synchronization would amount to requesting corresponding additional structure on the base monad $\mathbb{T}$. We leave the development of this idea for future work.

As indicated above, the device we use to extend a monad to be able to capture processes and not only one-step computations is a variant of the resumption monad transformer, given specifically by

$$T^\nu A = \nu\gamma.\ T(\gamma + A) \tag{1}$$

where ν refers to the greatest fixed point operator. We call the elements of (1) *ν-resumptions* in contrast to the more customary μ-resumptions [7, 14, 19], which are obtained by replacing ν by the least fixed point operator μ. This choice is motivated by the fact that ν-resumptions can capture infinite processes even in the category of sets, while μ-resumptions are finite objects (in categories of domains, the difference typically disappears). Moreover, in contrast to μ-resumptions, ν-resumptions come with a powerful coinductive definition principle; e.g. a parallel composition operator can be defined simply as a terminal map of coalgebras $T^\nu A \times T^\nu B \to T^\nu(A \times B)$ once the coalgebra structure on $T^\nu A \times T^\nu B$ has been defined appropriately.

We define an abstract *meta-calculus for monadic processes* that is based on the resumption monad transformer (1). As indicated above, ν-resumptions bring tools from coalgebra into play, in particular corecursion and coinduction [41]. We present a complete equational axiomatization of our calculus which includes a simple loop construct (in coalgebraic terms, coiteration) and then derive a powerful *corecursion schema* for the definition of process combinators. It has a fully syntactic justification, i.e. one can explicitly construct a solution to a corecursive equation system by means of the basic term language. Even semantically, our corecursion scheme does not seem to follow from previous corecursion results (e.g. [2, 48, 31]), as it permits prefixing corecursive calls with monadic sequential composition.

We exemplify our corecursion scheme with the definition of a number of basic imperative programming and process-algebraic primitives including parallel composition. Moreover, we develop a more high-level verification logic on top of the basic equational calculus.

Further related work. Various forms of the resumption monad have been in use for many years, dating back at least to Hennessy and Plotkin's resumption semantics for processes [21], which, while not being the first work involving resumptions, seems to be the first where the abstract notion of resumption has been recognized and extensively used as a formal tool. Following [33], resumptions became part of the general monad-based paradigm, in particular in the shape of the *resumption monad transformer* [7, 6]. The latter has been applied, e.g., to obtain a fully abstract semantics of the π-calculus in [14], which is conceptually close to the present work and also implicitly introduces the main equations for semi-additive monads. In a more practically oriented setting, the resumption monad transformer has been used to model concurrency in functional languages such as Haskell [19, 18]; to be slightly more precise about a point raised above, while the resumptions discussed at the theoretical level in [19] are μ-resumptions, their implementation in

Haskell in fact leads to an identification of μ-resumptions with ν-resumptions through lazy evaluation, so that (potentially) infinite behaviours are captured in the implementation. A resumption monad without a base effect, the *delay monad*, has been studied by Capretta [5] with a view to capturing general recursion.

(Weakly) final coalgebras of I/O-trees have been considered in the context of dependent type theory for functional programming, without, however, following a fully parametrized monadic approach as pursued here [17]. A framework where infinite resumptions of a somewhat different type than considered here form the morphisms of a category of processes, which is thus enriched over coalgebras for a certain functor, is studied in [28], but no meta-language is provided for such processes. For specific technical comparison, morphisms from A to B in the framework of [28] are elements of $\nu\gamma.\, A \to T(\gamma \times B)$, while Kleisli morphisms $A \to B$ for the resumption monad considered here are morphisms $A \to \nu\gamma.\, T(\gamma + B)$. The latter type appears slightly simpler, as the greatest fixed point operator applies to a less complex expression; this may be a reason why a meta-language is easier to design and handle in the present framework (although, as indicated, no competing meta-language for the framework of [28] is available for comparison). A meta-language that essentially adds least fixed points, i.e. *inductive* data types as opposed to *coinductive* process types as used in the present work, to Moggi's base language is studied in [13], mainly with a view to proving equivalences between different denotational semantics for the same language; reasoning principles in this framework are necessarily of a rather different flavour than the ones employed here.

There is extensive work on effectful iteration and recursion, including recursive monadic binding [12], traced pre-monoidal categories [3], completely iterative algebras [31], Kleene monads [16], etc. These, however, address the question of axiomatizing an iteration operator for side-effecting programs from an entirely different perspective than pursued here. Specifically, they impose certain conditions on a monad $\mathbb{T}$ that enable it to support iteration. In contrast to this, we iterate the monad itself and hence obtain a new monad $\mathbb{T}^\nu$, the monad of processes over $\mathbb{T}$. We use the layered structure of $\mathbb{T}^\nu$ in order to define parallel composition, and this effectively distinguishes our approach from the others.

A different line of related research is coalgebra-based work on corecursion schemes such as [2]; we do not currently see, however, how our monadic corecursion scheme would be obtained from existing coalgebraic results, precisely due to the presence of monadic binding in our corecursive equations.

The theory of parametrized monads originated in [48] has slight parallels to our work. The functor $T(X, A) = T(X + A)$ is a parametrized monad and hence, e.g. Proposition 16, which shows that $\mathbb{T}^\nu$ is in fact a monad, becomes an instance of a theorem proved in [48] (in the partial case of interest it seems to be folklore knowledge anyway). Complete iterativity of $\nu\gamma.\, T(A + \gamma)$, proved in [48], can be reformulated in terms of corecursion and hence compared to our corecursive scheme, but does not seem to actually prove the latter, and in particular does not prove the syntactic part of our main result, expressivity of corecursive functions in terms of the basic loop primitive.

Finally, there are, of course, alternatives to using the resumption monad transformer when modelling concurrency in a monadic setting. The option that is maybe most clearly set apart from resumptions is to work with the continuation monad transformer, as demonstrated in [8]. From the point of view of monad-based verification logics, the continuation monad has the disadvantage of being hard to handle by generic methods; e.g. it has no so-called pure computations, which play a crucial role in the generic treatment of pre- and postconditions [42]. Also, it is not at all clear how corecursive definitions, the main topic of interest in the current work, can be handled within the continuation monad.

Plan of the paper. We start with an introduction to strong monads and Moggi's computational metalanguage in Section 2. In Section 3 we introduce semi-additive monads and ν-resumptions as abstraction devices for nondeterminism and process sequencing. In Sections 4 and 5 we present our calculus for side-effecting processes in two variants ME_ν and ME_ν^+ for deterministic and nondeterministic processes, respectively; we also prove our main result (Theorem 14) about unique solutions of mutual corecursive schemes. In Section 6 we show how various programming primitives including process interleaving can be expressed in our calculus. Finally, we analyse how standard techniques of proving safety properties [30] can be lifted to the level of side-effecting processes in Section 7.

A preliminary version of this work has appeared in FCT 2011 [15].

2. Strong monads and the meta-language of effects

Intuitively, a monad $\mathbb{T}$ associates to each type A a type TA of computations with results in A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. In other words, by means of a monad we can abstract from particular notions of computation by replacing non-pure functional programs with pure ones having a converted type profile.

One of the equivalent ways to define a monad $\mathbb{T}$ over a category $\mathbf{C}$ is by giving a *Kleisli triple*, i.e. a triple of the form $(T, \eta, \_^\star)$ where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, η is a family of morphisms $\eta_A : A \to TA$ called *unit*, and $\_^\star$, called *Kleisli star*, assigns to each morphism $f : A \to TB$ a morphism $f^\star : TA \to TB$ such that

$$\eta_A^\star = \mathrm{id}_{TA}, \qquad f^\star \circ \eta_A = f, \qquad \text{and} \qquad g^\star \circ f^\star = (g^\star \circ f)^\star.$$

This leads to the *Kleisli category* $\mathbf{C}_\mathbb{T}$ of $\mathbb{T}$, which is a category having the same objects as $\mathbf{C}$, and $\mathbf{C}$-morphisms $A \to TB$ as morphisms $A \to B$, with *Kleisli composition* $\diamond$ defined by $f \diamond g = f^\star \circ g$. A straightforward consequence of this definition is that T extends to an endofunctor on $\mathbf{C}$. We generally use blackboard bold characters (such as $\mathbb{T}$) to refer to monads and the corresponding standard characters (such as T) to refer to their functor parts. Computationally, η may be understood as promoting values to computations that happen to be effect-free, and Kleisli star encapsulates sequential chaining of computations.

When $\mathbf{C}$ is *Cartesian*, i.e. has finite products, then $\mathbb{T}$ is *strong* if it is equipped with a natural transformation $\tau_{A,B} : A \times TB \to T(A \times B)$ called *strength*, subject to certain coherence conditions (see e.g. [33]). Strength is known to be equivalent to enrichment [27], in particular any monad over **Set** is strong. All monads used in the further development will implicitly be assumed to be strong.

Strong monads over $\mathbf{C}$ form an (overlarge) category: a *monad morphism* is a natural transformation between the underlying monad functors satisfying expected coherence conditions w.r.t. the unit, Kleisli star and the strength, see e.g. [1] for details.

Remark 1. Here and in the following, we work over an arbitrary base category $\mathbf{C}$, which may for better understanding be thought of as the category of sets, but may equally well be a category of domains. Most existing completeness results for monadic meta-languages, including Moggi's original completeness result for the computational meta-language and the completeness result proved here, are w.r.t. variable base categories. Restricting to a specific base category will, of course, preserve soundness, while completeness may break down due to particular properties of the base category becoming observable in the calculus. A point in case is the fact that least and greatest fixed points typically coincide in categories of domains, which means that a domain semantics for our calculus will need to include an induction principle to complement the coinduction principle studied here (this, we expect, would relate the resulting calculus with the FIX-logic of Crole and Pitts [11]). The standard set-up using a variable base category thus amounts to a separation of concerns in that it allows for a study of coinductive principles in isolation.

Example 2. [33] Computationally relevant strong monads on **Set** (and, mutatis mutandis, on other categories with sufficient structure) include the following.

1. *The identity monad* is the trivial one: $TA = A$.
2. *Stateful computations:* $TA = S \to (S \times A)$, where S is a fixed set of states.
3. *Exceptions*: $TA = A + E$, where E is a fixed object of exceptions.
4. *Nondeterminism:* $TA = \mathcal{P}(A)$, where $\mathcal{P}$ is the covariant powerset functor. Moreover, for any cardinal α, $\mathcal{P}_\alpha(A) = \{B \in \mathcal{P}(A) \mid |B| < \alpha\}$ yields a monad of α-bounded nondeterminism; e.g. $\mathcal{P}_\omega$ and $\mathcal{P}_{\omega_1}$ model finite and countable nondeterminism respectively.
5. *Interactive input:* TA is the smallest fixed point of $\gamma \mapsto A + (U \to \gamma)$, where U is an object of input values.
6. *Interactive output:* TA is the smallest fixed point of $\gamma \mapsto A + (U \times \gamma)$, where U is an object of output values.

7. *Nondeterministic stateful computation:* $TA = S \to \mathcal{P}(A \times S)$ and the variations $TA = S \to \mathcal{P}_\alpha(A \times S)$ for all α.

8. *Continuations:* $TA = (A \to O) \to O$ where O is an object of global answers.

As originally observed by Moggi [33], strong monads support a *computational meta-language*, i.e. essentially a generic imperative programming language, which we shall refer to as the *meta-language of effects*. We consider a first-order version of this language here, i.e. we do not include function types (as we are mainly interested in the common imperative programming basis, which does not include functional abstraction). We do however include coproducts. Adding new expressive features to the language, as long as they admit a clear set-theoretic semantics (e.g. function types), seems to be just a technical exercise, not having a bearing on our results, expect that the new constructions might actually facilitate the proof of our main corecursion result (Theorem 14). Therefore, it appears that adhering to a minimal syntax as outlined here ensures stronger results.

Our language is parametrized over a countable signature Σ including a set of atomic types W, from which the type system is generated by the grammar

$$P ::= W \mid 1 \mid P \times P \mid P + P \mid TP.$$

Moreover, Σ includes basic programs $f : A \to B$ where A and B are types. We call the annotation $A \to B$ at f its *type profile*.

The terms of the language, which we will here also refer to as *programs*, and their types are then determined by the term formation rules shown in Fig. 1 (top section); judgements $\Gamma \rhd t : A$ read 'term t has type A in context Γ', where a *context* is a list $\Gamma = (x_1 : A_1, \dots, x_n : A_n)$ of typed variables. The notions of free and bound variables are defined accordingly in a standard way (which is apparent from the typing rules; e.g., x is bound in $\mathsf{do}\, x \leftarrow p; q$), along with an appropriate notion of capture-avoiding substitution.

The intuitive meaning of the various language constructs is as follows.

- The notation for products and coproducts is standard: $\langle \_, \_ \rangle$ denotes pairing, fst and snd are projections, $\star$ is the unique element of the unit type 1, inl and inr map the components A and B into their disjoint union $A + B$, and the case construct combines two functions on A and B, respectively, into a function on $A + B$ by pattern matching.

- ret promotes a value to a computation (and, semantically, represents the unit of the monad); it can be understood as terminating a computation by returning a value.

- do is sequential composition with propagation of results as parameters: $\mathsf{do}\, x \leftarrow p; q$ executes the computation p, binds its result to x, and then executes the computation q, which may depend on x. Semantically, it is interpreted by means of Kleisli composition as detailed below.

- Summation $+$ represents binary nondeterministic choice, and $\varnothing$ is deadlock.

We use the notation $\mathsf{do}\, \langle x, y \rangle \leftarrow p; q$ as an abbreviation for

$$\mathsf{do}\, z \leftarrow p; q[\mathsf{fst}\, z/x, \mathsf{snd}\, z/y]$$

where z is not free in q.

Let $\mathbf{C}$ be a distributive category [9], e.g. a Cartesian category with binary coproducts such that the canonical map

$$[\mathrm{id} \times \kappa_1, \mathrm{id} \times \kappa_2] : A \times B + A \times C \to A \times (B + C)$$

is an isomorphism. We denote by *dist* the corresponding inverse from $A \times (B + C)$ to $A \times B + A \times C$. Given an interpretation of atomic types W as objects $[\![W]\!]$ in $\mathbf{C}$ and a strong monad $\mathbb{T}$ on $\mathbf{C}$, we obtain obvious interpretations $[\![A]\!]$, $[\![\Gamma]\!]$ of types A and contexts Γ as objects in C. Then an interpretation of basic programs $f : A \to B$ as morphisms $[\![f]\!] : [\![A]\!] \to [\![B]\!]$ induces an interpretation $[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$ of terms $\Gamma \rhd t : A$, given by the following clauses:

$$\textbf{(var)}\ \frac{x : A \in \Gamma}{\Gamma \rhd x : A} \qquad \textbf{(app)}\ \frac{f : A \to B \in \Sigma \quad \Gamma \rhd t : A}{\Gamma \rhd f(t) : B} \qquad \textbf{(1)}\ \frac{}{\Gamma \rhd \star : 1}$$

$$\textbf{(pair)}\ \frac{\Gamma \rhd t : A \quad \Gamma \rhd u : B}{\Gamma \rhd \langle t, u\rangle : A \times B} \qquad \textbf{(fst)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathsf{fst}\, t : A} \qquad \textbf{(snd)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathsf{snd}\, t : B}$$

$$\textbf{(case)}\ \frac{\Gamma \rhd s : A + B \quad \Gamma, x : A \rhd t : C \quad \Gamma, y : B \rhd u : C}{\Gamma \rhd \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t;\ \mathsf{inr}\, y \mapsto u : C} \qquad \textbf{(inl)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \mathsf{inl}\, t : A + B}$$

$$\textbf{(inr)}\ \frac{\Gamma \rhd t : B}{\Gamma \rhd \mathsf{inr}\, t : A + B} \qquad \textbf{(ret)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \mathsf{ret}\, t : TA} \qquad \textbf{(do)}\ \frac{\Gamma \rhd p : TA \quad \Gamma, x : A \rhd q : TB}{\Gamma \rhd \mathsf{do}\ x \leftarrow p; q : TB}$$

$$\textbf{(nil)}\ \frac{}{\Gamma \rhd \varnothing : TA} \qquad \textbf{(plus)}\ \frac{\Gamma \rhd p : TA \quad \Gamma \rhd q : TA}{\Gamma \rhd p + q : TA}$$

Figure 1: General term formation rules for the meta-language of effects (top); additional term formation rules for semi-additive monads (bottom).

- $[\![x_1 : A_1, \ldots, x_n : A_n \rhd x_i : A_i]\!] = \pi_i^n$,
- $[\![\Gamma \rhd f(t) : B]\!] = [\![f]\!] \circ [\![\Gamma \rhd t : A]\!]$, $f : A \to B \in \Sigma$,
- $[\![\Gamma \rhd \star : 1]\!] = !_{[\![\Gamma]\!]}$,
- $[\![\Gamma \rhd \langle t, u\rangle : A \times B]\!] = \langle [\![\Gamma \rhd t : A]\!], [\![\Gamma \rhd u : B]\!]\rangle$,
- $[\![\Gamma \rhd \mathsf{fst}\, t : A]\!] = \pi_1 \circ [\![\Gamma \rhd t : A \times B]\!]$ and $[\![\Gamma \rhd \mathsf{snd}\, t : A]\!] = \pi_2 \circ [\![\Gamma \rhd t : A \times B]\!]$,
- $[\![\Gamma \rhd \mathsf{inl}\, t : A + B]\!] = \kappa_1 \circ [\![\Gamma \rhd t : A]\!]$ and $[\![\Gamma \rhd \mathsf{inr}\, t : A + B]\!] = \kappa_2 \circ [\![\Gamma \rhd t : B]\!]$,
- $[\![\Gamma \rhd \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t;\ \mathsf{inr}\, y \mapsto u : C]\!] = \big[[\![\Gamma, x : A \rhd t : C]\!], [\![\Gamma, y : B \rhd u : C]\!]\big] \circ dist \circ \big\langle \mathrm{id}, [\![\Gamma \rhd s : A + B]\!]\big\rangle$,
- $[\![\Gamma \rhd \mathsf{do}\ x \leftarrow p; q : TB]\!] = [\![\Gamma, x : A \rhd q : TB]\!] \diamond (\tau_{[\![\Gamma]\!],[\![A]\!]} \circ \langle \mathrm{id}, [\![\Gamma \rhd p : TA]\!]\rangle)$,
- $[\![\Gamma \rhd \mathsf{ret}\, t : TA]\!] = \eta_A \circ [\![\Gamma \rhd t : A]\!]$

where as usual the π_i^n are the projections $A_1 \times \cdots \times A_n \to A_i$, $!_A : A \to 1$ is the unique morphism into the terminal object, $\langle f, g\rangle : A \to B \times C$ denotes pairing of morphisms $f : A \to B$, $g : A \to C$, the κ_i with $i = 1, 2$ denote coproduct injections $A \to A + B$ and $B \to A + B$, and $[f, g] : A + B \to C$ denotes copairing of $f : A \to C$ and $g : B \to C$. We say that an *equation* $\Gamma \rhd t = s$, where $\Gamma \rhd t : A$ and $\Gamma \rhd s : A$, is *satisfied* by the interpretation over $\mathbb{T}$, and shortly write $\mathbb{T} \models \Gamma \rhd t = s$, if $[\![\Gamma \rhd t : A]\!] = [\![\Gamma \rhd s : A]\!]$.

Of course, any category with binary coproducts has n-ary coproducts, whose coproduct injections we denote as $\mathsf{inj}_i^n : A_i \to A_1 + \cdots + A_n$. A special case are n-fold coproducts $n = 1 + \cdots + 1$. We shall occasionally denote the elements of n as numerals $\bar{1} = \mathsf{inj}_1^n \star, \ldots, \bar{n} = \mathsf{inj}_n^n \star$. Similarly, one can define the type of Booleans with the usual structure as $2 = 1 + 1$. We write $\mathsf{if}\, b\, \mathsf{then}\, p\, \mathsf{else}\, q$ as an abbreviation for $\mathsf{case}\ b\ \mathsf{of}\ \mathsf{inl}\, x \mapsto p;\ \mathsf{inr}\, x \mapsto q$ where $b : 2$. Boolean-valued equality on numerals can then be defined in the obvious way as a case term.

3. Effect combination, semi-additive monads and ν-resumptions

In practice, computational effects most often are combinations of several primitive effects; e.g. the nondeterministic state monad from Example 2.7 is a combination of the state monad (Example 2.2) and the nondeterminism monad (Example 2.4). Thus there arises the challenge to find a suitable formalization of effect combination. One practically accepted solution is via so-called *monad transformers*. Most generally, a monad transformer is an endomap defined over the class of all monads over the given category (see [29] for further categorical justifications of this notion). A more advanced approach to combining effects, proposed in [23, 22], is to use symmetric binary operations on monads, most prominently sum and tensor.

Definition 3 (Sum, Tensor). Given two monads $\mathbb{T}_1$ and $\mathbb{T}_2$ over $\mathbf{C}$, the *sum* $\mathbb{T}_1 + \mathbb{T}_2$, if it exists, is the categorical coproduct of $\mathbb{T}_1$ and $\mathbb{T}_2$ in the category of monads over $\mathbf{C}$. Their *tensor product*, if it exists, is a strong monad $T = T_1 \otimes T_2$ defined by the universal property of having monad morphisms $\sigma_1 : T_1 \to T$ and $\sigma_2 : T_2 \to T$ such that every two programs of the form $\sigma_1(p)$ and $\sigma_2(q)$ commute.

The universality property here can be spelled out as follows: for any strong monad R and any two monad morphisms $\alpha_1 : T_1 \to R$, $\alpha_2 : T_2 \to R$ such that every $\alpha_1(p)$ commutes with every $\alpha_2(q)$, there is a unique monad morphism $\beta : T \to R$ such that $\alpha_i = \beta\sigma_i$, $i = 1, 2$.

Intuitively, the sum of effects yields their free combination whereas the tensor is formed analogously but modulo commutation of effects over each other. Many well-known monad transformers turn out to be instances of either the sum or the tensor construction:

Example 4. [23, 22] Let $\mathbb{T}$ range over monads on $\mathbf{C}$. Then

- the correspondence $\mathbb{T} \mapsto \mathbb{T} \otimes \mathbb{I}_S^{st}$, where $\mathbb{I}_S^{st}$ is the state monad $I_S^{st}A = S \to (S \times A)$, yields the state monad transformer; in particular, the functorial part of $\mathbb{T} \otimes \mathbb{I}_S^{st}$ is $T_S^{st}A = S \to T(S \times A)$;
- the correspondence $\mathbb{T} \mapsto \mathbb{T} \otimes \mathbb{I}_S^{rd}$, where $\mathbb{I}_S^{rd}$ is the reader monad $I_S^{rd}A = S \to A$, yields the reader monad transformer; in particular, the functorial part of $\mathbb{T} \otimes \mathbb{I}_S^{rd}$ is $T_S^{rd}A = S \to TA$;
- given a free monad $\mathbb{I}_{\mathbb{F}}^{\mu}$ over an endofunctor $F : \mathbf{C} \to \mathbf{C}$ (whose functorial part is $I_F^{\mu}A = \mu\gamma.(A + F\gamma)$), the correspondence $\mathbb{T} \mapsto \mathbb{T} + \mathbb{I}_F^{\mu}$ defines the free monad transformer, whose functorial part is $T_F^{\mu}A = \mu\gamma.T(A + F\gamma)$. Notably, this captures exceptions ($FA = E$), input ($FA = U \to A$) and output ($FA = U \times A$).

One particular case of the last clause of Example 4 is by taking F to be the identity functor. The resulting monad transformer $T \mapsto \mu\gamma.T(\gamma + \_)$ is known as the *resumption monad transformer* [7] and can be used for modelling parallel programs [21]. Depending on the underlying category, in particular when $\mathbf{C} = \mathbf{Set}$, the resumption monad transformer may yield a model for finite processes only. As the main objective of our work is to provide a calculus for side-effecting reactive processes, we base our model on a greatest fixed point instead, the monad transformer for *ν-resumptions*

$$T \mapsto \nu\gamma.T(\gamma + \_). \tag{2}$$

We refer to the functor on the right-hand side as T^{ν}. Intuitively, the idea of T^{ν} is to capture atomic computation steps by T; each atomic step may either yield a remaining process, the *resumption*, or a final result. Use of the greatest fixed point operator ν (which formally denotes a final coalgebra) implies that processes may continue to yield resumptions indefinitely, i.e. run forever without terminating.

Proposition 5. *Given a monad $\mathbb{T}$ on $\mathbf{C}$, if the greatest fixed point $\nu\gamma.T(\gamma + A)$ exists for all objects A in $\mathbf{C}$ then T^{ν} extends to a monad $\mathbb{T}^{\nu}$ on $\mathbf{C}$.*

Proof. It is easy to see that the correspondence sending every object A of $\mathbf{C}$ to the monad $\mathbb{T} + \mathbb{I}_A^{ex}$, where $\mathbb{I}_A^{ex}$ is the exception monad with A being the object of exceptions, extends to a functor from $\mathbf{C}$ to the category of monads on $\mathbf{C}$. Equivalently, the bifunctor $T(\_ + \_)$ yields a *parametrized monad* [48], and if the greatest fixed point $T^{\nu}A$ of $\gamma \mapsto T(\gamma + A)$ exists for all A then T^{ν} carries the structure of a monad (Theorem 3.9 in *op. cit.*). □

In order to model nondeterministic side-effecting processes we need to impose additional structure on the base monad. A general notion of additional algebraic structure on monads is that of an *algebraic operation*.

Definition 6 (Algebraic operation [39]). Given a monad $\mathbb{T}$ over $\mathbf{C}$ and a natural number n, a natural transformation $\alpha_X : (TX)^n \to TX$ is an *(n-ary) algebraic operation* if

- for every $f \in \mathbf{C}(A, TB)$, $\alpha_B(f^\star)^n = f^\star \alpha_A$ and
- for every $A, B \in \mathrm{Ob}\,\mathbf{C}$, $\tau(\mathrm{id}_A \times \alpha_B) = \alpha_{A\times B}\tau^n\vartheta_n$

where $\vartheta_n : A\times(TB)^n \to (A\times TB)^n$ is the n-tupling morphism whose i-th component is $\mathrm{id}\times\pi_i : A\times(TB)^n \to A \times TB$.

Examples of algebraic operations include exception raising ($T^0 \to T$), reading a value ranging over n ($T^n \to T$) and writing some value ($T^1 \to T$) to a memory location, etc. The following result is shown in [26].

Proposition 7. *Let $\mathbb{T}$ and $\mathbb{S}$ be two monads over $\mathbf{C}$ and let $\sigma : \mathbb{T} \to \mathbb{S}$ be a monad morphism. Then any algebraic operation $\alpha : T_1^n \to T_1$ canonically lifts to $\mathbb{S}$ so that the diagram*

$$\begin{array}{ccc} T^n & \xrightarrow{\alpha} & T \\ {\scriptstyle \sigma^n}\downarrow & & \downarrow{\scriptstyle \sigma} \\ S^n & \xrightarrow{\hat{\alpha}} & S. \end{array}$$

commutes where $\hat{\alpha} : S^n \to S$ denotes the lifting.

Here, we are interested in monads supporting finitary nondeterminism suitably captured by algebraic operations.

Definition 8 (Semi-additive monads). A monad $\mathbb{T}$ is *semi-additive* if it supports two algebraic operations: deadlock $\delta : 1 \to T$ and choice $\varpi : T^2 \to T$ making every TA an internal bounded join-semilattice.

Note that the above definition requires the semilattice structure to distribute over binding from the left (but not necessarily from the right) as reflected in our calculus in Section 4. In terms of the meta-language of effects we capture semi-additive monads by the two additional rules in the bottom section of Fig. 1.

A prototypical example of a semi-additive monad over **Set** is the powerset monad and its variants $\mathcal{P}_\alpha$. Note that the former can be defined over any topos and is a partial case of a more general semi-additive *quantale monad* [24] $\lambda X.\, Q^X$ where Q is a quantale [40]. In the domain-theoretic setting, counterparts of the powerset monad are the various powerdomain constructions. Another stand-alone example of a semi-additive monad is the continuation monad with the answer object O carrying a bounded join-semilattice structure.

Further examples of semi-additive monads can be obtained from Example 4 by Proposition 7. E.g., the nondeterministic state monad $TX = S \to \mathcal{P}(S \times X)$ is semi-additive: being the tensor product of the state monad and the powerset monad, it inherits the choice and the deadlock from the latter. Another interesting corollary of Proposition 7 for our current purposes is

Corollary 9. *If $\mathbb{T}$ is a semi-additive monad, then $\mathbb{T}^\nu$ is also semi-additive.*

Proof. By applying Proposition 7 to the obvious monad morphism $\mathbb{T} \to \mathbb{T}^\nu$ given for every $A \in \mathrm{Ob}\,\mathbf{C}$ as $\alpha_A^{-1} \circ T\kappa_2 : TA \to T^\nu A$ where $\alpha_A : T^\nu A \to T(T^\nu A + A)$ is the final coalgebra structure (in particular an isomorphism by Lambek's lemma). □

4. A Calculus for Side-effecting Processes

The meta-language of effects presented in the previous section is essentially a generic sequential imperative programming language. Here we introduce an extension of it, the *corecursive meta-language*, based semantically on the resumption monad transformer (2); its distinctive feature is support for corecursion.

As in the case of the meta-language of effects, we consider a countable signature Σ including a set of atomic types W, from which the type system is generated by the grammar

$$A ::= W \mid 1 \mid A \times A \mid A + A \mid TA \mid T^\nu A$$

— base effects are represented by T, and ν-resumptions by T^ν; in other words, values of type TA are single computation steps possibly returning values of type A, and values of type $T^\nu A$ are processes taking steps with side-effects specified by T and returning values of type A upon possible termination. The term language is obtained by completing the formation rules in the top section of Fig. 1 by the two additional rules

$$\textbf{(out)}\ \frac{\Gamma \rhd p : T^\nu A}{\Gamma \rhd \mathsf{out}(p) : T(T^\nu A + A)} \qquad \textbf{(unf)}\ \frac{\Gamma \rhd t : A \quad \Gamma, x : A \rhd q : T(A + B)}{\Gamma \rhd \mathsf{init}\, x := t\ \mathsf{unfold}\, \{q\} : T^\nu B}.$$

Given a resumption $p : T^\nu A$, $\mathsf{out}(p) : T(T^\nu A + A)$ executes the first step of p, returning either the remaining resumption or a final value. Moreover, for $q : T(A + B)$ depending on $x : A$, $\mathsf{init}\, x := t\ \mathsf{unfold}\, \{q\} : T^\nu B$ iterates q indefinitely or until it outputs a value in B; values $x : A$ are fed through the loop, starting with the initial value $t : A$. Terms whose type is of the form $T^\nu A$ are called *processes*.

Convention 10. Technically, **(unf)** does not allow for creating terms of the form $\mathsf{init}\, x := t\ \mathsf{unfold}\, \{q\}$ with $x \in \mathrm{Vars}(t)$, for the notation $\Gamma, x : A$ assumes that x does not occur in Γ, but since $x \in \mathrm{Vars}(t)$ it must. Still, we agree to write $\mathsf{init}\, x := t\ \mathsf{unfold}\, \{q\}$ even if $x \in \mathrm{Vars}(t)$ as an abbreviation for $\mathsf{init}\, y := t\ \mathsf{unfold}\, \{q[y/x]\}$ with some fresh variable y and adopt the arising α-conversion henceforth.

The semantics of the corecursive meta-language is defined over $\mathsf{ME}_{\boldsymbol{\nu}}$*-models*, referred to just as *models* below. A model is based on a distributive category $\mathbf{C}$ equipped with a strong monad $\mathbb{T}$ on $\mathbf{C}$ such that for every $A \in \mathrm{Ob}\,\mathbf{C}$ the greatest fixed point $\nu\gamma.\, T(\gamma + A)$ exists. A model interprets base types as objects of $\mathbf{C}$ and terms in context as morphisms of $\mathbf{C}$ in the way prescribed in Section 2. It only remains to interpret $T^\nu A$ and the two new term constructors. We put $[\![T^\nu A]\!] = T^\nu [\![A]\!]$. Then, out is just the final coalgebra structure

$$\alpha_{[\![A]\!]} : T^\nu [\![A]\!] \to T(T^\nu [\![A]\!] + [\![A]\!]),$$

and the loop construct $\mathsf{init}\, x := p\ \mathsf{unfold}\, \{q\}$ is defined by coiteration, i.e. as the unique morphism from the coalgebra defined by q into the final coalgebra. Explicitly, for a coalgebra $f : X \to T(X + [\![A]\!])$, let $\digamma f : X \to T^\nu [\![A]\!]$ be the unique coalgebra morphism. Then we put

$$[\![\Gamma \rhd \mathsf{out}(p) : T(T^\nu A + A)]\!] = \alpha_{[\![A]\!]} \circ [\![\Gamma \rhd p : T^\nu A]\!]$$

$$[\![\Gamma \rhd \mathsf{init}\, x := t\ \mathsf{unfold}\, \{q\} : T^\nu B]\!] = T^\nu \pi_2 \circ \digamma f \circ \langle \mathrm{id}, [\![\Gamma \rhd t : A]\!] \rangle$$

where $f = T(dist) \circ \tau \circ \langle \pi_1, g \rangle$ is the extension of $g = [\![\Gamma, x : A \rhd q : T(A + B)]\!]$ with the context $[\![\Gamma]\!]$:

$$\begin{array}{ccc} [\![\Gamma]\!] \times [\![A]\!] & \xrightarrow{\langle \pi_1, g \rangle} & [\![\Gamma]\!] \times T([\![A]\!] + [\![B]\!]) \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle \tau} \\ T([\![\Gamma]\!] \times [\![A]\!] + [\![\Gamma]\!] \times [\![B]\!]) & \xleftarrow{T(dist)} & T([\![\Gamma]\!] \times ([\![A]\!] + [\![B]\!])). \end{array}$$

(case_inl) $\mathsf{case}\ \mathsf{inl}\, s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t;\ \mathsf{inr}\, y \mapsto u = t[s/x]$ **(fst)** $\mathsf{fst}\langle s, t\rangle = s$

(case_inr) $\mathsf{case}\ \mathsf{inr}\, s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t;\ \mathsf{inr}\, y \mapsto u = u[s/y]$ **(snd)** $\mathsf{snd}\langle s, t\rangle = t$

(case_id) $\mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \mathsf{inl}\, x;\ \mathsf{inr}\, y \mapsto \mathsf{inr}\, y = s$ **(pair)** $\langle \mathsf{fst}\, s, \mathsf{snd}\, s\rangle = s$

(case_sub) $\mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t[u/z];\ \mathsf{inr}\, y \mapsto t[w/z] = t[\mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto u;\ \mathsf{inr}\, y \mapsto w/z]$ $(x, y \notin \mathrm{Vars}(u))$

$(\star)$ $s : 1 = \star$ **$(\mathbf{unit_1})$** $\mathsf{do}\ x \leftarrow p; \mathsf{ret}\, x = p$ **$(\mathbf{unit_2})$** $\mathsf{do}\ x \leftarrow \mathsf{ret}\, a; p = p[a/x]$

(assoc) $\mathsf{do}\ x \leftarrow (\mathsf{do}\ y \leftarrow p; q); r = \mathsf{do}\ x \leftarrow p; y \leftarrow q; r$ $(y \notin \mathrm{Vars}(r))$

(coiter) $$\frac{\mathsf{out}(p) = \mathsf{next}\ q\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, p;\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y}{p = \mathsf{init}\, x := x\ \mathsf{unfold}\, \{q\}}$$

(nil) $p + \varnothing = p$ **(comm)** $p + q = q + p$ **(idem)** $p + p = p$

(assoc_plus) $p + (q + r) = (p + q) + r$ **(dist_nil)** $\mathsf{do}\ x \leftarrow \varnothing; r = \varnothing$

(dist_plus) $\mathsf{do}\ x \leftarrow (p + q); r = \mathsf{do}\ x \leftarrow p; r + \mathsf{do}\ x \leftarrow q; r$

Figure 2: Axiomatization of the corecursive (top) and concurrent corecursive meta-language (top and bottom).

Thus, F_f is uniquely determined by the commutative diagram.

$$\begin{array}{ccc} [\![\Gamma]\!] \times [\![A]\!] & \xrightarrow{\quad f \quad} & T([\![\Gamma]\!] \times [\![A]\!] + [\![\Gamma]\!] \times [\![B]\!]) \\ {\scriptstyle F_f}\downarrow & & \downarrow{\scriptstyle T(F_f + \mathrm{id})} \\ T^\nu([\![\Gamma]\!] \times [\![B]\!]) & \xrightarrow{\alpha_{[\![\Gamma]\!] \times [\![B]\!]}} & T(T^\nu([\![\Gamma]\!] \times [\![B]\!]) + [\![\Gamma]\!] \times [\![B]\!]). \end{array}$$

A model is said to *satisfy* a well-typed equation $\Gamma \rhd t = s$ if $[\![\Gamma \rhd t : A]\!] = [\![\Gamma \rhd s : A]\!]$.

As suggestive abbreviations for use in process definitions, we write cont for $\mathsf{ret}\,\mathsf{inl}$ and stop for $\mathsf{ret}\,\mathsf{inr}$ (so that for $p : T^\nu A$ and $t : A$, both $\mathsf{cont}(p)$ and $\mathsf{stop}(t)$ have type $T(T^\nu A + A)$). Moreover, for $p : T(B + A)$, we shorten $(\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto q;\ \mathsf{inr}\, y \mapsto r)$ down to $(\mathsf{next}\ p\ \mathsf{is}\ \mathsf{rest}\, x \mapsto q;\ \mathsf{done}\, y \mapsto r)$. We also define a converse $\mathsf{tuo} : T(T^\nu A + A) \to T^\nu A$ to out by

$$\mathsf{tuo}(p) \;=\; \mathsf{init}\, z := p\ \mathsf{unfold}\, \{\mathsf{next}\ z\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y\}.$$

— i.e. $\mathsf{tuo}(p)$ initializes $q : T(T^\nu A + A)$ to p, executes q, and then either stops and returns y if q returns $y : A$, or else continues with the process $x : T^\nu A$ returned by q inserted into $T(T^\nu A + A)$ via cont, thus effectively appending the steps of x after the first step p. An axiomatization ME_ν of the concurrent meta-language is given in the top section of Fig. 2 (where we omit the standard equational logic ingredients including the obvious congruence rules). Apart from the standard axioms for products and coproducts, ME_ν contains three well-known monad laws and a novel (bidirectional) rule **(coiter)** for effectful coiteration. The latter

rule combines two important principles, heavily used henceforth: (i) the unfolding equation

$$\mathsf{out}(p) = \mathsf{next}\ q\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, p;\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y$$

uniquely characterizes p and (ii) the solution is expressible in the language. Intuitively, the unfolding equation confirms that $\mathsf{init}\, x := r\ \mathsf{unfold}\, \{q\}$ indeed behaves as indicated at the time of the introduction of its syntax: it initializes $x : B$ to r, executes $q : T(A + B)$ (which may mention x), and then either stops and returns y if q returns $y : A$, or otherwise starts a further process step, entering a loop with the new value of $x : B$ returned by q.

One can now formally prove in $\mathsf{ME}_{\boldsymbol{\nu}}$ the outlined property of tuo and out (which is, in fact, nothing but an instance of Lambek's lemma) as a pair of equations:

$$\mathsf{tuo}(\mathsf{out}(z)) = z, \qquad \mathsf{out}(\mathsf{tuo}(z)) = z. \tag{3}$$

Indeed, note that by **(coiter)** tuo is uniquely defined by the equation

$$\mathsf{out}(\mathsf{tuo}(z)) = \mathsf{next}\ z\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, \mathsf{tuo}(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y, \tag{4}$$

which collapses to the right identity in (3) once we show the left one. In order to prove the left identity we replace z in (4) by $\mathsf{out}(z)$ and hence obtain

$$\mathsf{out}(\mathsf{tuo}(\mathsf{out}(z))) = \mathsf{next}\ \mathsf{out}(z)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, \mathsf{tuo}(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y.$$

By **(coiter)**, the resulting equation uniquely defines $\mathsf{tuo}(\mathsf{out}(z))$, however z obviously satisfies the same equation, hence $\mathsf{tuo}(\mathsf{out}(z)) = z$.

Soundness and completeness of this calculus are unsurprising as such. Soundness however is technically nontrivial because of the interplay of monadic strength with corecursion hidden in the definition of unfold.

Theorem 11. $\mathsf{ME}_{\boldsymbol{\nu}}$ *is sound and strongly complete over* $\mathsf{ME}_{\boldsymbol{\nu}}$*-models.*

(Recall that *strong completeness* refers to completeness for logical consequences of possibly infinite sets of axioms.)

Proof. *Soundness.* We only establish soundness of the rule **(coiter)** since the remainder is more or less standard. Let $g = [\![\Gamma, x : A \rhd q : T(A + B)]\!]$, $h = [\![\Gamma, x : A \rhd p : T^\nu B]\!]$ and $f = T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \circ \pi_1$. Then the bottom of **(coiter)** can be rewritten to

$$h = T^\nu \pi_2 \circ \boldsymbol{\digamma} f \circ \langle \mathrm{id}, \pi_2\rangle. \tag{5}$$

Let us show that the top of **(coiter)** can be rewritten to

$$\alpha_{[\![B]\!]} \circ h = T(h + \pi_2) \circ f \circ \langle \mathrm{id}, \pi_2\rangle. \tag{6}$$

Indeed, we have

$$[\![\mathsf{out}(p)]\!] = \alpha_{[\![B]\!]} \circ h$$

and

$$\begin{aligned}
&[\![\mathsf{next}\ q\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, p;\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y]\!] \\
&= [\![\mathsf{do}\ z \leftarrow q;\ \ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \mathsf{ret}\, \mathsf{inl}\, p;\ \mathsf{inr}\, y \mapsto \mathsf{ret}\, \mathsf{inr}\, y]\!] \\
&= [\![\mathsf{do}\ z \leftarrow q; \mathsf{ret}(\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \mathsf{inl}\, p;\ \mathsf{inr}\, y \mapsto \mathsf{inr}\, y)]\!] \\
&= \big(\eta \circ (h \circ (\pi_1 \times \mathrm{id}) + \pi_2) \circ \mathit{dist}\big) \diamond \tau\langle \mathrm{id}, g\rangle \\
&= T\big((h \circ (\pi_1 \times \mathrm{id}) + \pi_2) \circ \mathit{dist}\big) \circ \tau\langle \mathrm{id}, g\rangle \\
&= T\big((h + \pi_2) \circ (\pi_1 \times \mathrm{id} + \pi_1 \times \mathrm{id}) \circ \mathit{dist}\big) \circ \tau\langle \mathrm{id}, g\rangle
\end{aligned}$$

$$
\begin{aligned}
&= T\big((h + \pi_2) \circ \mathit{dist} \circ (\pi_1 \times \mathrm{id})\big) \circ \tau\langle \mathrm{id}, g\rangle \\
&= T(h + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(h + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \circ \pi_1 \circ \langle \mathrm{id}, \pi_2\rangle \\
&= T(h + \pi_2) \circ f \circ \langle \mathrm{id}, \pi_2\rangle.
\end{aligned}
$$

using in particular naturality of *dist*. In order to complete the proof, we are left to establish the equivalence of (5) and (6). The proof of the implication (5) $\Rightarrow$ (6) is as follows:

$$
\begin{aligned}
\alpha_{[\![B]\!]} \circ h &= \alpha_{[\![B]\!]} \circ T^\nu \pi_2 \circ \mathit{F}_f \circ \langle \mathrm{id}, \pi_2\rangle \\
&= T(T^\nu \pi_2 + \pi_2) \circ \alpha_{[\![\Gamma]\!]\times[\![B]\!]} \circ \mathit{F}_f \circ \langle \mathrm{id}, \pi_2\rangle \\
&= T(T^\nu \pi_2 + \pi_2) \circ T(\mathit{F}_f + \mathrm{id}) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \circ \pi_1 \circ \langle \mathrm{id}, \pi_2\rangle \\
&= T(T^\nu \pi_2 \circ \mathit{F}_f + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(h + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(h + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \circ \pi_1 \circ \langle \mathrm{id}, \pi_2\rangle \\
&= T(h + \pi_2) \circ f \circ \langle \mathrm{id}, \pi_2\rangle.
\end{aligned}
$$

using in particular (5), the defining property of F_f, the definition of f, and naturality of $\alpha : T^\nu \to T(T^\nu + Id)$. In order to prove (6) $\Rightarrow$ (5), first observe that we can equivalently present (6) as $\alpha_{[\![B]\!]} \circ h = T(h + \mathrm{id}) \circ w$ where $w = T(\mathrm{id} + \pi_2) \circ f \circ \langle \mathrm{id}, \pi_2\rangle$, i.e. h satisfies the equation characterizing F_w, and thus $h = \mathit{F}_w$. We are left to show that $T^\nu \pi_2 \circ \mathit{F}_f$ also satisfies this equation. Indeed:

$$
\begin{aligned}
&\alpha_{[\![B]\!]} \circ T^\nu \pi_2 \circ \mathit{F}_f \circ \langle \mathrm{id}, \pi_2\rangle \\
&\quad = T(T^\nu \pi_2 + \pi_2) \circ \alpha_{[\![\Gamma]\!]\times[\![B]\!]} \circ \mathit{F}_f \circ \langle \mathrm{id}, \pi_2\rangle \\
&\quad = T(T^\nu \pi_2 + \pi_2) \circ T(\mathit{F}_f + \mathrm{id}) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \circ \pi_1 \circ \langle \mathrm{id}, \pi_2\rangle \\
&\quad = T(T^\nu \pi_2 \circ \mathit{F}_f + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&\quad = T(T^\nu \pi_2 \circ \mathit{F}_f + \pi_2) \circ f \\
&\quad = T(T^\nu \pi_2 \circ \mathit{F}_f + \mathrm{id}) \circ T(\mathrm{id} + \pi_2) \circ f \\
&\quad = T(T^\nu \pi_2 \circ \mathit{F}_f + \mathrm{id}) \circ w
\end{aligned}
$$

again using naturality of α. We have thus $h = \mathit{F}_w = T^\nu \pi_2 \circ \mathit{F}_f \circ \langle \mathrm{id}, \pi_2\rangle$ and the proof is completed.

Completeness. By a standard term model construction (see, e.g., [10]). □

A core result on the corecursive meta-language is an expressive corecursion scheme supported by the given simple axiomatization. The scheme is formally defined as follows.

Definition 12 (Corecursion). Let $\hat{f}_i : A_i \to T^\nu B_i$, $i = 1, \ldots, k$ be a collection of fresh function symbols. A *guarded corecursive scheme* is a system of equations

$$
\mathsf{out}(\hat{f}_i(x)) = \mathsf{do}\ z \leftarrow p_i; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto p_1^i; \ldots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto p_{n_i}^i \qquad (*)
$$

for $i = 1, \ldots, k$ such that for every i, $p_i : T(C_{i1} + \cdots + C_{in_i})$ does not contain any $\hat{f}_j$, and for every i, j p_j^i either does not contain any $\hat{f}_m$ or is of the form $p_j^i \equiv \mathsf{cont}\, \hat{f}_m(x_j)$ for some m. In case $k = 1$ we just speak of a *guarded corecursive equation*, i.e. an equation of the form

$$
\mathsf{out}(\hat{f}(x)) = \mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto p_1; \ldots; \mathsf{inj}_n^n\, x_n \mapsto p_n. \qquad (**)
$$

Intuitively, a guarded corecursive equation states that $\hat{f}(x)$ has a first step that begins by executing $p : T(C_1 + \cdots + C_n)$ (which may mention x but not $\hat{f}$), and then proceeds by case distinction over the return value of p, in each case either entering a new step with a corecursive call to $\hat{f}(x)$ or else proceeding arbitrarily

(in particular either entering a new step directly or first continuing the first step) but without corecursive calls to $\hat{f}(x)$. A guarded corecursive scheme is understood similarly but defines several operations by mutual corecursion.

To start off, we prove that guarded corecursive equations have unique solutions.

Lemma 13 (Corecursion). *Given appropriately typed programs p and p_i such that (**) is guarded, there is a unique function f satisfying (**), and this function is defined by an effectively computable term in ME_ν.*

Proof. The idea is to start from a special case and successively extend generality.

(i) Suppose that $n = 2$, $p_1 \equiv \mathsf{cont}\,\hat{f}(x)$ and p_2 does not contain $\hat{f}$. Let $A \to T^\nu B$ be the type profile of $\hat{f}$ and let us define a function $F : A + T^\nu B \to T^\nu B$ by putting: $F(z) = \big(\mathsf{init}\, z := z\ \mathsf{unfold}\,\{H(z)\}\big)$ where

$$\begin{aligned} H(z) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \big(&\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of} \\ &\mathsf{inl}\, x_1 \mapsto \mathsf{cont}(\mathsf{inl}\, x_1); \\ &\mathsf{inr}\, x_2 \mapsto (\mathsf{next}\ p_2\ \mathsf{is}\ \ \mathsf{rest}\, x \mapsto \mathsf{cont}(\mathsf{inr}\, x); \\ &\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{stop}\, x)\big); \\ \mathsf{inr}\, r \mapsto \big(&\mathsf{next}\ \mathsf{out}(r)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(\mathsf{inr}\, x); \\ &\mathsf{done}\, x \mapsto \mathsf{stop}\, x\big). \end{aligned}$$

Let us show that $F(\mathsf{inr}\, x) = x$. By **(coiter)**,

$$\mathsf{out}(F(z)) = \mathsf{next}\ H(z)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, F(z);\ \mathsf{done} \mapsto \mathsf{stop}\, x. \tag{7}$$

Note that

$$H(\mathsf{inr}\, x) = \mathsf{next}\ \mathsf{out}(x)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(\mathsf{inr}\, x);\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x$$

from which we conclude that

$$\mathsf{out}(F(\mathsf{inr}\, x)) = \mathsf{next}\ \mathsf{out}(x)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, F(\mathsf{inr}\, x);\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x.$$

The latter means that $(F \circ \mathsf{inr})$ satisfies the same equation as the identity function. Hence, by **(coiter)** both these functions must be provably equal, i.e. for every x, $F(\mathsf{inr}\, x) = x$. Now, (7) can be simplified to

$$\begin{aligned} \mathsf{out}(F(z)) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \big(&\ \mathsf{next}\ p\ \mathsf{is}\ \mathsf{rest}\, x_1 \mapsto \mathsf{cont}\, F(\mathsf{inl}(x_1)); \\ &\qquad\qquad \mathsf{done}\, x_2 \mapsto p_2\big); \\ \mathsf{inr}\, r \mapsto r.& \end{aligned}$$

By **(coiter)** F is uniquely defined by this equation. It can be verified by routine calculation that $\hat{f}(x) = F(\mathsf{inl}\, x)$ is a solution of (**). In order to prove uniqueness, let us assume that g is some other solution of (**). Let

$$G(z) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto g(x);\ \mathsf{inr}\, r \mapsto \mathsf{tuo}(r).$$

Clearly, $g(x) = G(\mathsf{inl}\, x)$ and it can be shown that G satisfies the equation defining F. Therefore $g(x) = G(\mathsf{inl}\, x) = F(\mathsf{inl}\, x) = \hat{f}(x)$ and we are done.

(ii) Suppose that $n > 1$, $p_1 \equiv \mathsf{cont}\,\hat{f}(x_1)$, and for every $i > 1$, p_i does not contain $\hat{f}$. We reduce this case to the previous one as follows. Observe that, by assumption, the corecursive equation for $\hat{f}(x)$ is equivalent to

$$\mathsf{out}(\hat{f}(x)) = \mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x_1 \mapsto \mathsf{cont}\,\hat{f}(x_1);\ \mathsf{inr}\, z \mapsto q$$

where $q = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n-1}\, x_2 \mapsto p_2; \ldots; \mathsf{inj}_{n-1}^{n-1}\, x_n \mapsto p_n)$. According to (i), $\hat{f}$ is uniquely definable and thus we are done.

(iii) Suppose that for some k, $p_i \equiv \mathsf{cont}\,\hat{f}(x)$ for $i \leq k$ and p_i does not contain $\hat{f}$ for $i > k$. If $k = 1$ and $n > 1$ then we arrive precisely at the situation captured by the previous clause and hence we are done. If $k = n = 1$ then (∗∗) takes the form

$$\mathsf{out}(\hat{f}(x)) = \mathsf{do}\ x \leftarrow p; \mathsf{cont}\,\hat{f}(x),$$

which can be transformed to

$$\mathsf{out}(\hat{f}(x)) = \mathsf{next}\,(\mathsf{do}\ x \leftarrow p; \mathsf{rest}\,x)\ \mathsf{is}\ \mathsf{rest}\,x \mapsto \mathsf{cont}\,\hat{f}(x);\ \mathsf{done}\,x \mapsto \mathsf{stop}\,x$$

and hence we are done by **(coiter)**. Suppose that $k > 1$, $n > 1$ and proceed by induction over k. Let

$$q = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n-2}\,x_3 \mapsto p_3; \ldots; \mathsf{inj}_{n-2}^{n-2}\,x_{n-2} \mapsto p_{n-2}.$$

Then we have

$$\begin{aligned}
\mathsf{out}(\hat{f}(x)) = {} & \mathsf{do}\ z \leftarrow p;\ \ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x_1 \mapsto \mathsf{cont}\,\hat{f}(x_1); \\
& \qquad\qquad\qquad\qquad\quad \mathsf{inr}\,\mathsf{inl}\,x_2 \mapsto \mathsf{cont}\,\hat{f}(x_2); \\
& \qquad\qquad\qquad\qquad\quad \mathsf{inr}\,\mathsf{inr}\,z \mapsto q \\
= {} & \mathsf{next}\ \ \big(\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x_1 \mapsto \mathsf{cont}\,x_1; \\
& \qquad\qquad\qquad\qquad\quad \mathsf{inr}\,\mathsf{inl}\,x_2 \mapsto \mathsf{cont}\,x_2; \\
& \qquad\qquad\qquad\qquad\quad \mathsf{inr}\,\mathsf{inr}\,z \mapsto \mathsf{stop}\,z\big)\ \mathsf{is} \\
& \mathsf{rest}\,x \mapsto \mathsf{cont}\,\hat{f}(x);\ \mathsf{done}\,x \mapsto q
\end{aligned}$$

and thus we are done by induction hypothesis.

(iv) Finally, we reduce the general claim to the case captured by the previous clause as follows. First observe that if neither of the p_i contains $\hat{f}$ then the solution is given by the equation

$$\hat{f}(x) = \mathsf{tuo}(\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\,x_1 \mapsto p_1; \ldots; \mathsf{inj}_n^n\,x_n \mapsto p_n).$$

In the remaining case, there exists an index k and a permutation σ of $\{1, \ldots, n\}$ such that for every $i \leq k$, $p_{\sigma(i)} \equiv \mathsf{rest}\,\hat{f}(x)$ and for every $i > k$, $p_{\sigma(i)}$ does not contain $\hat{f}$. By a slight abuse of notation we also use σ as function $A_1 + \ldots + A_n \to A_{\sigma(1)} + \ldots + A_{\sigma(n)}$ rearranging the components of the coproduct in the obvious fashion. Then

$$\begin{aligned}
\mathsf{out}(\hat{f}(x)) = {} & \mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\,x_1 \mapsto p_1; \ldots; \mathsf{inj}_n^n\,x_n \mapsto p_n \\
= {} & \mathsf{do}\ z \leftarrow (\mathsf{do}\ z \leftarrow p; \mathsf{ret}\,\sigma(z)); \\
& \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_{\sigma(1)}^n\,x_{\sigma(1)} \mapsto p_{\sigma(1)}; \ldots; \mathsf{inj}_{\sigma(n)}^n\,x_{\sigma(n)} \mapsto p_{\sigma(n)}
\end{aligned}$$

and thus we are done by (iii). □

We now extend unique solvability to guarded corecursive schemes.

Theorem 14 (Mutual corecursion). *Let $\hat{f}_i : A_i \to T^\nu B_i$, $i = 1, \ldots, k$ be fresh function symbols. A guarded corecursive scheme (∗) uniquely defines $\hat{f}_1, \ldots, \hat{f}_k$ (as morphisms in the model), and the solutions $\hat{f}_i$ are expressible as effectively computable terms in ME_ν.*

In order to prove the theorem we will need the following slight generalization of Lemma 13.

Lemma 15. *Let $\hat{f}$ be a fresh functional symbol, i.e. $\hat{f} \notin \Sigma$. Given appropriately typed programs $p, p_1, \ldots, p_n, q_1, \ldots, q_n$ such that for every i, p_i either does not contain $\hat{f}$ or is of the form $\mathsf{cont}\,\hat{f}(q_i)$, there is a unique function $\hat{f}$ satisfying (∗∗).*

PROOF. W.l.o.g. $q_i \equiv x$ whenever p_i does not contain f. We can rewrite $(**)$ to

$$\mathsf{out}(\hat{f}(x)) = \mathsf{do}\ z \leftarrow q; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}_1^n\, x_1 \mapsto r_1; \ldots; \mathsf{inl}_1^n\, x_n \mapsto r_n$$

where $q = (\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto \mathsf{ret}\,\mathsf{inj}_1^n\, q_1; \ldots; \mathsf{inj}_1^n\, x_n \mapsto \mathsf{ret}\,\mathsf{inj}_1^n\, q_n)$, $r_i = \mathsf{cont}(\hat{f}(x))$ if $p_i \equiv \mathsf{cont}(\hat{f}(q_i))$ and $r_i = p_i$ otherwise. Now we are done by Lemma 13. □

PROOF (THEOREM 14). W.l.o.g. $n_1 = \ldots = n_k$: otherwise we replace every p_i by

$$\mathsf{do}\ z \leftarrow p_i; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto \mathsf{inj}_1^n\, x_1; \ldots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto \mathsf{inj}_{n_i}^n\, x_n$$

where $n = \max_i n_i$, and complete the case statement in $(*)$ arbitrarily to match the typing. Observe that if the result types of the $\hat{f}_i$ do not coincide, $(*)$ falls into mutually independent parts, as $\hat{f}_k$ can appear in the definition of $\hat{f}_i$ only if $\hat{f}_k$ and $\hat{f}_i$ have the same type. Therefore, in the remainder we assume w.l.o.g. that all $\hat{f}_i$ have the same type. Consider the corecursive definition

$$\begin{aligned}\mathsf{out}(F(y)) = \mathsf{do}\ v \leftarrow q; \mathsf{case}\ v\ \mathsf{of}\\ \mathsf{inj}_1^k\, z \mapsto (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto q_1^1; \ldots; \mathsf{inj}_n^n\, x_n \mapsto q_n^1);\\ \vdots\\ \mathsf{inj}_k^k\, z \mapsto (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto q_1^k; \ldots; \mathsf{inj}_n^n\, x_n \mapsto q_n^k)\end{aligned}$$

where

$$q = \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inj}_1^k\, x_1 \mapsto (\mathsf{do}\ z \leftarrow p_i; \mathsf{ret}\,\mathsf{inj}_1^k\, z); \ldots; \mathsf{inj}_k^k\, x_k \mapsto (\mathsf{do}\ z \leftarrow p_k; \mathsf{ret}\,\mathsf{inj}_k^k\, z)$$

and the q_j^i are defined by $q_j^i = \mathsf{cont}\, F(\mathsf{inj}_m^k\, x_j)$ if $p_j^i \equiv \mathsf{cont}\, \hat{f}_m(x_j)$ and $q_j^i = p_j^i$ otherwise. By Lemma 15, it uniquely defines F. It is easy to calculate that by taking $\hat{f}_i(x) = F(\mathsf{inj}_i^k(x))$ we obtain a solution of $(*)$. Let us show it is also unique. Suppose that $\hat{g}_i$ is another solution. Then G defined by the equation

$$G(y) = \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inj}_1^k\, x \mapsto \hat{g}_1(x); \ldots; \mathsf{inj}_k^k\, x \mapsto \hat{g}_k(x)$$

is easily seen to satisfy the same corecursive scheme as F and thus $F = G$. Therefore, for every i, $\hat{g}_i(x) = G(\mathsf{inj}_i^k\, x) = F(\mathsf{inj}_i^k\, x) = \hat{f}_i(x)$ and we are done. □

As a first application of guarded corecursive schemes, we show how to explicitly introduce monadic structure for the type constructor T^ν via the meta-language. Let $q : TA$ be a program, and let x be a variable of type A. Then we define a function $B_{x,q}$ corecursively by

$$\mathsf{out}(B_{x,q}(p)) = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}\, B_{x,q}(r); \mathsf{done}\, x \mapsto \mathsf{out}(q).$$

Note how this definition exploits that the format of guarded corecursive schemes allows for extensions of steps in the non-corecursive cases: $B_{x,q}(p)$ will first copy all steps of p, but when the last step of p is reached (case $\mathsf{done}\, x$), the first step of q will be executed as part of this step before the next step is generated — in other words, $B_{x,q}(p)$ merges the last step of p with the first of q.

We can now define new term constructors

$$(\mathbf{ret_\nu})\ \frac{\Gamma \rhd p : A}{\Gamma \rhd \mathsf{ret}_\nu\, p : T_\nu A} \qquad (\mathbf{bind_\nu})\ \frac{\Gamma \rhd p : T_\nu A \quad \Gamma, x : A \rhd q : T_\nu B}{\Gamma \rhd \mathsf{do}_\nu\ x \leftarrow p; q : T_\nu B}$$

by putting $\mathsf{ret}_\nu\, p = \mathsf{tuo}(\mathsf{stop}\, p)$ and $\mathsf{do}_\nu\ x \leftarrow p; q = B_{x,q}(p)$. The following proposition is an easy consequence of Theorem 14.

Proposition 16. *The monad laws* $(\mathbf{unit_1})$, $(\mathbf{unit_2})$ *and* $(\mathbf{assoc})$ *are satisfied with* ret *and* do *replaced by* ret_ν *and* do_ν *correspondingly.*

5. Nondeterministic Processes

The calculus $\mathsf{ME}_{\boldsymbol{\nu}}$ presented in the previous section is meant to capture deterministic side-effecting processes. It does of course remain adequate if the underlying monad $\mathbb{T}$ comes with support for nondeterminism. Unsurprisingly, however, in order to define process interleaving we need to provide explicit support for nondeterminism. To that end we rebase our models on semi-additive monads as introduced in Section 3. Let us revisit the results of the previous section in the new setting.

A *concurrent corecursive meta-language* is obtained by completing the corecursive meta-language with the two additional primitives

$$\frac{}{\Gamma \rhd \varnothing : TA} \qquad\qquad \frac{\Gamma \rhd p : TA \quad \Gamma \rhd q : TA}{\Gamma \rhd p + q : TA}$$

where $\varnothing$ represents deadlock, and $+$ nondeterministic choice. The semantics of these operators is given by the assignments

- $[\![\Gamma \rhd \varnothing : TA]\!] = \delta_{[\![A]\!]} \circ !_{[\![\Gamma]\!]}$,
- $[\![\Gamma \rhd p + q : TA]\!] = \varpi_{[\![A]\!]} \circ \langle [\![\Gamma \rhd p : TA]\!], [\![\Gamma \rhd q : TA]\!] \rangle$

where we recall that $\delta : T^0 \to T$ and $\varpi : T^2 \to T$ are the algebraic operations presenting deadlock and choice. An suitable equational calculus $\mathsf{ME}^+_{\boldsymbol{\nu}}$ is given in Figure 2. An $\mathsf{ME}^+_{\boldsymbol{\nu}}$*-model* is an $\mathsf{ME}_{\boldsymbol{\nu}}$-model over a semi-additive monad.

Theorem 17 (Soundness and completeness of $\mathsf{ME}^+_{\boldsymbol{\nu}}$). *The calculus $\mathsf{ME}^+_{\boldsymbol{\nu}}$ is sound and strongly complete w.r.t. $\mathsf{ME}^+_{\boldsymbol{\nu}}$-models.*

Proof. Soundness is easy to see. Completeness is by adaptation of the term model construction for $\mathsf{ME}_{\boldsymbol{\nu}}$, noting that semi-additivity is just a form of algebraic structure. □

Obviously, the main result of the previous section, Theorem 14, remains true w.r.t. $\mathsf{ME}^+_{\boldsymbol{\nu}}$ for the former is not sensitive to extending the language with new operations and equations. However, we can reformulate it so as to make use of the semi-additive structure of $\mathbb{T}$. In this formulation our corecursive scheme is reminiscent of the guarded recursive equation systems of ACP [4].

Theorem 18. *A system of equations of the form*

$$\mathsf{out}(\hat{f}_i(x)) = \mathsf{do}\ z \leftarrow p_i; (\mathsf{do}\ x_1 \leftarrow q^i_1; p^i_1 + \ldots + \mathsf{do}\ x_{n_i} \leftarrow q^i_{n_i}; p^i_{n_i}) \tag{8}$$

where the p_i are as in Theorem 14 uniquely defines $\hat{f}_1, \ldots, \hat{f}_k$ (as morphisms in the model). Moreover, systems of equations (8) are exactly as expressive as (∗), that is: every family of operations definable by (∗) can be defined by (8) and vice versa.

Proof. Transforming the right-hand side of (8), we have

$$\begin{aligned}
&\mathsf{do}\ z \leftarrow p_i; (\mathsf{do}\ x_1 \leftarrow q^i_1; p^i_1 + \ldots + \mathsf{do}\ x_{n_i} \leftarrow q^i_{n_i}; p^i_{n_i}) \\
=\ &\mathsf{do}\ z \leftarrow p_i; c \leftarrow (\mathsf{do}\ x_1 \leftarrow q^i_1; \mathsf{ret}\,\mathsf{inj}^{n_i}_1\, x_1 + \ldots + \mathsf{do}\ x_{n_i} \leftarrow q^i_{n_i}; \mathsf{ret}\,\mathsf{inj}^{n_i}_{n_i}\, x_{n_i}); \\
&\mathsf{case}\ c\ \mathsf{of}\ \mathsf{inj}^{n_1}_1\, x_1 \mapsto p^i_1; \ldots; \mathsf{inj}^{n_i}_{n_i} \mapsto p^i_{n_i} \\
=\ &\mathsf{do}\ \langle z, c\rangle \leftarrow (\mathsf{do}\ z \leftarrow p_i; c \leftarrow (\mathsf{do}\ x_1 \leftarrow q^i_1; \mathsf{ret}\,\mathsf{inj}^{n_i}_1\, x_1 + \ldots\ + \\
&\mathsf{do}\ x_{n_i} \leftarrow q^i_{n_i}; \mathsf{ret}\,\mathsf{inj}^{n_i}_{n_i}\, x_{n_i}); \mathsf{ret}\langle z, c\rangle); \\
&\mathsf{case}\ c\ \mathsf{of}\ \mathsf{inj}^{n_1}_1\, x_1 \mapsto p^i_1; \ldots; \mathsf{inj}^{n_i}_{n_i} \mapsto p^i_{n_i}.
\end{aligned}$$

Therefore, by Theorem 14, the system (8) has a unique solution, which is definable by a corecursive scheme.

In order to complete the proof we are left to show that any solution of $(*)$ satisfies a system of equations of the form (8). Let $\hat{f}_1, \dots, \hat{f}_k$ be the solution of $(*)$. Then for every i,

$$\begin{aligned}\mathsf{out}(\hat{f}_i(x)) &= \mathsf{do}\ z \leftarrow p_i; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto p_1^i; \dots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto p_{n_i}^i \\ &= \mathsf{do}\ z \leftarrow p_i; (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto p_1^i; \dots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto \varnothing\ + \\ &\qquad\qquad \vdots \\ &\qquad\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto \varnothing; \dots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto p_{n_i}^i) \\ &= \mathsf{do}\ z \leftarrow p_i; \big(\mathsf{do}\ x_1 \leftarrow q_1^i; p_1^i + \cdots + \mathsf{do}\ x_n \leftarrow q_{n_i}^i; p_{n_i}^i\big)\end{aligned}$$

where for every j, $q_j^i = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto \varnothing; \dots; \mathsf{inj}_j^{n_i}\, x_j \mapsto \mathsf{ret}\, x_j; \dots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto \varnothing$. □

By Corollary 9 we know that the semi-additive structure of $\mathbb{T}$ lifts to $\mathbb{T}^\nu$. We can reflect this result on the meta-language level as follows. Let us introduce term constructors

$$\textbf{(nil}_\nu\textbf{)}\ \frac{}{\Gamma \rhd \varnothing_\nu : T_\nu A} \qquad \textbf{(plus}_\nu\textbf{)}\ \frac{\Gamma \rhd p : T_\nu A \quad \Gamma \rhd q : T_\nu A}{\Gamma \rhd p +_\nu q : T_\nu A}$$

as abbreviations defined by

$$\varnothing_\nu = \mathsf{tuo}(\varnothing), \qquad\qquad p +_\nu q = \mathsf{tuo}(\mathsf{out}(p) + \mathsf{out}(q)).$$

The operations we have defined justify the use of types $T_\nu A$ as domains for nondeterministic processes: $\varnothing_\nu$ is a deadlocked process, $+_\nu$ is a nondeterministic choice of two processes.

Proposition 19. *All axioms listed in the bottom section of Fig. 2, with* do *replaced by* do_ν, $\varnothing$ *replaced by* $\varnothing_\nu$, *and* $+$ *replaced by* $+_\nu$, *are theorems of* ME_ν^+. □

6. Programming with Side-effecting Processes

We proceed to extend the set of process constructs, including more complex operations defined by corecursion, such as parallel composition.

Atomic processes. We can convert $p : TA$ into a one-step process $\mathsf{tuo}(\mathsf{do}\ x \leftarrow p; \mathsf{stop}\, x)$, shortly denoted $[p]$.

Sequential composition. Although T^ν is a monad, its binding operator is not quite what one would want as sequential composition of processes, as it merges the last step of the first process with the first step of the second process. We can, however, capture sequential composition (with the same typing) in the intended way by putting

$$\mathsf{seq}\ x \leftarrow p; q = \mathsf{do}_\nu\ x \leftarrow p; \mathsf{tuo}(\mathsf{cont}\, q).$$

Note that the term $\mathsf{tuo}(\mathsf{cont}\, q)$ represents a process whose first step just returns q as the resumption without causing any side-effects. Explicitly, seq has the corecursive definition

$$\mathsf{out}(\mathsf{seq}\ x \leftarrow p; q) = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r\ \mapsto \mathsf{cont}(\mathsf{seq}\ x \leftarrow r; q);\ \mathsf{done}\, x \mapsto \mathsf{cont}(q)$$

(which differs from the definition of $\mathsf{do}_\nu\ x \leftarrow p; q$ in having $\mathsf{cont}(q)$ in place of $\mathsf{out}(q)$ in the second alternative, so that the last step of p and the first step of q remain separate).

Branching. Using the effect-free if operator defined in Section 2, we can define a conditional branching operator for processes $p, q : T^\nu A$ and a condition $b : T2$ (recall that the object $2 = 1 + 1$ comes with the structure of a Boolean algebra, hence $b : T2$ is effectively a computation returning a truth value) by

$$\mathsf{if}_\nu\, b\, \mathsf{then}\, p\, \mathsf{else}\, q = \mathsf{tuo}\big(\mathsf{do}\ z \leftarrow b; \mathsf{if}\ z\ \mathsf{then}\ (\mathsf{cont}\, p)\ \mathsf{else}\ (\mathsf{cont}\, q)\big).$$

Notice that the conditional term on the right hand side equals $\mathsf{cont}(\mathsf{if}\ z\ \mathsf{then}\ p\ \mathsf{else}\ q)$. The chosen granularity of the if statement allows interruption of the process after evaluation of the condition.

Looping. For terms $\Gamma \rhd t : A$; $\Gamma, x : A \rhd b : T2$; and $\Gamma, x : A \rhd q : T^\nu A$, we define a loop construct

$$\Gamma \rhd \mathsf{init}\, x := p\ \mathsf{while}\ b\ \mathsf{do}\ q : T^\nu A$$

as follows. The intended definition is via nested corecursion,

$$\begin{aligned}\mathsf{out}(\mathsf{init}\, x := p\ \mathsf{while}\ b\ \mathsf{do}\ q) = \\ \mathsf{do}\ v \leftarrow b[p/x]; \mathsf{if}\ v\ \mathsf{then}\ \mathsf{cont}(\mathsf{seq}\ y \leftarrow q[p/x]; \mathsf{init}\, x := y\ \mathsf{while}\ b\ \mathsf{do}\ q)\ \mathsf{else}\ \mathsf{stop}(p).\end{aligned}$$

This definition is not directly covered by our corecursion schema, but as usual, we can resolve nesting into mutual corecursion. Explicitly, we abbreviate $W^b_{x,q}(p) = (\mathsf{init}\, x := p\ \mathsf{while}\ b\ \mathsf{do}\ q)$, and introduce a further operator $V^b_{x,q}(r)$ for $\Gamma \rhd r : T^\nu A$ intended to represent $\mathsf{seq}\ y \leftarrow r; \mathsf{init}\, x := y\ \mathsf{while}\ b\ \mathsf{do}\ q$. We then define $W^b_{x,q}$ and $V^b_{x,q}$ by the guarded corecursive scheme

$$\begin{aligned}\mathsf{out}(W^b_{x,q}(p)) &= \mathsf{do}\ v \leftarrow b[p/x]; \mathsf{if}\ v\ \mathsf{then}\ \mathsf{cont}(V^b_{x,q}(q[p/x]))\ \mathsf{else}\ \mathsf{stop}(p), \\ \mathsf{out}(V^b_{x,q}(r)) &= \mathsf{next}\ \mathsf{out}(r)\ \mathsf{is}\ \mathsf{rest}\, z \mapsto \mathsf{cont}(V^b_{x,q}(z)); \mathsf{done}\, y \mapsto \mathsf{cont}(W^b_{x,q}(y))\end{aligned}$$

which is easily seen to be equivalent to the intended corecursive definition of while, together with the indicated definition of $V^b_{x,q}$ in terms of while. As in the case of the conditional construct, we allow interruption after evaluation of the condition. Note moreover that the last step of a while process is always the evaluation of the condition, and the result value is either that of the last execution of the loop body, or the initial value in case the loop body is never executed.

Exceptions. As the corecursive meta-language includes coproducts, the exception monad transformer ($T^E A = T(A + E)$) [7] and the corresponding operations for raising and handling exceptions are directly expressible in ME_ν.

Interleaving. We introduce process interleaving $\| : T^\nu A \times T^\nu B \to T^\nu(A \times B)$ by a CCS-style expansion law [32],using an auxiliary left merge $\between$:

$$\mathsf{out}(p \parallel q) = \mathsf{out}(p \between q) + \mathsf{out}(\mathsf{do}_\nu\, \langle x, y\rangle \leftarrow q \between p; \mathsf{ret}_\nu \langle y, x\rangle), \tag{9}$$

$$\begin{aligned}\mathsf{out}(p \between q) = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(r \parallel q); \\ \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\, y \leftarrow q; \mathsf{ret}_\nu \langle x, y\rangle).\end{aligned} \tag{10}$$

Thus, the left merge $p \between q$ executes the first step of p and then proceeds by interleaving p and q if p continues, and else transfers control to q. The interleaving $p \parallel q$ nondetermistically either performs a left merge $p \between q$ or a left merge $q \between p$, in the latter case with the obvious rearrangement of result values. While this equation system itself is not a guarded corecursive scheme, we can equivalently transform it into one, thus showing that $\|$ is uniquely defined by (9)–(10). Explicitly,

Lemma 20. *The system (9)–(10) is equivalent to a guarded corecursive scheme defining $\|$.*

Proof. First, let us show that the system (9)–(10) is equivalent to the system

$$\mathsf{out}(p \parallel q) = \mathsf{out}(p \between q) + \mathsf{out}(q \between^\star p), \tag{11}$$

$$\begin{aligned}\mathsf{out}(p \between q) = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(r \parallel q); \\ \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\, y \leftarrow q; \mathsf{ret}_\nu \langle x, y\rangle),\end{aligned} \tag{12}$$

$$\begin{aligned}\mathsf{out}(q \between^\star p) = \mathsf{next}\ \mathsf{out}(q)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(p \parallel r); \\ \mathsf{done}\, y \mapsto \mathsf{cont}(\mathsf{do}_\nu\, x \leftarrow p; \mathsf{ret}_\nu \langle x, y\rangle)\end{aligned} \tag{13}$$

(note that (12) is identical to (10)) in sense that any solution of the first system is a solution of the second one and vice versa — in order to convert a solution of the first system into a solution of the second one we use (13) as the definition.

First, let $\parallel$ and $\lfloor\!\lfloor$ be defined by (9)–(10). Observe that

$$p \parallel q = \mathsf{do}_\nu \langle x, y\rangle \leftarrow q \parallel p; \mathsf{ret}_\nu\langle y, x\rangle. \tag{14}$$

Indeed,

$$\begin{aligned}
&\mathsf{out}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow q \parallel p; \mathsf{ret}_\nu\langle y, x\rangle)\\
&\quad= \mathsf{next}\ \mathsf{out}(q \parallel p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle)\\
&\quad= \mathsf{next}\ \mathsf{out}(q \mathbin{\lfloor\!\lfloor} p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle) +\\
&\qquad \mathsf{next}\ \mathsf{out}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow p \mathbin{\lfloor\!\lfloor} q; \mathsf{ret}_\nu\langle y, x\rangle)\ \mathsf{is}\\
&\qquad\qquad\qquad\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad\quad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle)\\
&\quad= \mathsf{next}\ \mathsf{out}(q \mathbin{\lfloor\!\lfloor} p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle) +\\
&\qquad \mathsf{next}\ \mathsf{out}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow p \mathbin{\lfloor\!\lfloor} q; \mathsf{ret}_\nu\langle y, x\rangle)\ \mathsf{is}\\
&\qquad\qquad\qquad\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad\quad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle)\\
&\quad= \mathsf{next}\ \mathsf{out}(q \mathbin{\lfloor\!\lfloor} p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle) +\\
&\qquad \mathsf{next}\ \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}\, r;\ \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle)\\
&\quad= \mathsf{out}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow q \mathbin{\lfloor\!\lfloor} p; \mathsf{ret}_\nu\langle y, x\rangle) + \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q)
\end{aligned}$$

and we obtain (14) by application of tuo.

Now, let us show that $p \parallel q$ satisfies (11). We have

$$\begin{aligned}
\mathsf{out}(p \parallel q) &= \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q) + \mathsf{out}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow q \mathbin{\lfloor\!\lfloor} p; \mathsf{ret}_\nu\langle y, x\rangle)\\
&= \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q) + \mathsf{next}\ \mathsf{out}(q \mathbin{\lfloor\!\lfloor} p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad\qquad \mathsf{done}\langle x, y\rangle \mapsto \mathsf{out}(\mathsf{ret}_\nu\langle y, x\rangle)\\
&= \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q) + \mathsf{next}\ \mathsf{out}(q)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu \langle x, y\rangle \leftarrow r \parallel p; \mathsf{ret}_\nu\langle y, x\rangle);\\
&\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow p; \mathsf{ret}_\nu\langle y, x\rangle)\\
&= \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q) + \mathsf{next}\ \mathsf{out}(q)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(p \parallel r);\\
&\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow p; \mathsf{ret}_\nu\langle y, x\rangle)\\
&= \mathsf{out}(p \mathbin{\lfloor\!\lfloor} q) + \mathsf{out}(q \mathbin{\lfloor\!\lfloor}^\star p).
\end{aligned}$$

Conversely, given a solution of (11)–(13) one can show that it also yields a solution of (9)–(10) by applying the whole argument backwards; equation (14) in this case follows trivially.

Finally, we convert (11)–(13) into a guarded corecursive equation by substitution:

$$\begin{aligned}
\mathsf{out}(p \parallel q) = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r &\mapsto \mathsf{cont}(r \parallel q);\\
\mathsf{done}\, x &\mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle) +
\end{aligned}$$

$$\begin{aligned} &\mathsf{next\ out}(q)\ \mathsf{is\ rest}\, r \mapsto \mathsf{cont}(q \parallel r);\\ &\qquad\qquad \mathsf{done}\, y \mapsto \mathsf{cont}(\mathsf{do}_\nu\ x \leftarrow p; \mathsf{ret}_\nu\langle x, y\rangle)\\ =\ &\mathsf{do}\ u \leftarrow (p \mathbin{\lfloor} q + p \mathbin{\rfloor} q);\\ &\mathsf{case}\ u\ \mathsf{of}\ \mathsf{inl}\langle s, t\rangle \mapsto \mathsf{cont}(s \parallel t);\ \mathsf{inr}\, r \mapsto \mathsf{cont}\, r \end{aligned}$$

where $p \mathbin{\lfloor} q : T(T^\nu A \times T^\nu B + T^\nu(A \times B))$ is defined as

$$p \mathbin{\lfloor} q = \mathsf{next\ out}(p)\ \mathsf{is\ rest}\, r \mapsto \mathsf{ret\,inl}\langle r, q\rangle;\ \mathsf{done}\, x \mapsto \mathsf{ret\,inr}(\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle)$$

and $p \mathbin{\rfloor} q : T(T^\nu A \times T^\nu B + T^\nu(A \times B))$ is the evident dual of $p \mathbin{\lfloor} q$. □

7. Process Verification

We now start exploring the potential of ME_ν as a *verification* framework, extending existing monad-based program logics [43, 44] to concurrent processes. We first recall some aspects of generic sequential verification over the base monad $\mathbb{T}$, and then introduce new formalisms for $\mathbb{T}^\nu$. The latter revolve around the notion of invariant, and are currently geared mainly towards the proof of safety properties (i.e. non-violation of safety constraints throughout the execution of the process). It should be noted that while the satisfaction of safety constraints can be detected from the set of finite traces of a process, it is still a property of the whole trace set, i.e. of a possibly infinite object, thus justifying the use of ν-resumptions in place of μ-resumptions. E.g. one might conceivably impose the safety property that the process never terminates, a property that is unsatisfiable for μ-resumptions but easy to implement using an infinite ν-resumption.

A cornerstone of monad-based program logic is a notion of *pure* program:

Definition 21 (Pure programs). A program $p : TA$ is *pure* if

- p is *discardable*, i.e., $\mathsf{do}\ y \leftarrow p; \mathsf{ret} \star = \mathsf{ret} \star$;
- p is *copyable*, i.e. $\mathsf{do}\ x \leftarrow p; y \leftarrow p; \mathsf{ret}\langle x, y\rangle = \mathsf{do}\ x \leftarrow p; \mathsf{ret}\langle x, x\rangle$; and
- p commutes with any other discardable and copyable program q, i.e. $(\mathsf{do}\ x \leftarrow p; y \leftarrow q; \mathsf{ret}\langle x, y\rangle) = \mathsf{do}\ y \leftarrow q; x \leftarrow p; \mathsf{ret}\langle x, y\rangle$.

Intuitively, pure programs are those that can access internal data behind the computation, such as state or pending exception, but cannot affect it. A typical example of a pure program is a getter method. Pure programs form a submonad P of T.

Definition 22. A *test* is a program of type $P2$ (recall that 2 denotes the type of Booleans). All standard logical connectives extend to tests; e.g. $\neg\phi = (\mathsf{do}\ x \leftarrow \phi; \mathsf{ret} \neg x)$ for $\phi : P2$ [43]. A test ϕ is *valid* if $\phi = \mathsf{ret} \top$.

For a Boolean term $a : 2$, we define a value-preserving version of the test $a?$ in the spirit of Kleene algebra with tests as follows. Given a program $p : TB$, the term $a?p : TB$ is defined as

$$a?p = \mathsf{if}\ a\ \mathsf{then}\ p\ \mathsf{else}\ \varnothing.$$

Using this notation, we define, for a program $p : TA$ and tests $\phi, \psi : P2$ (all three being terms possibly containing free variables), the program $\mathsf{F}(\phi, y \leftarrow p, \psi) : TA$ as

$$\mathsf{F}(\phi, y \leftarrow p, \psi) = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; (x \Rightarrow z)?(\mathsf{ret}\, y).$$

Intuitively, $\mathsf{F}(\phi, y \leftarrow p, \psi)$ modifies (*filters*) the given program p by removing those possible executions that satisfy the precondition ϕ but fail the postcondition ψ, where y denotes the result of the computation p for use in ψ. This enables us to encode a Hoare triple by the equivalence

$$\{\phi\}\ y \leftarrow p\ \{\psi\} \iff \mathsf{F}(\phi, y \leftarrow p, \psi) = p$$

— i.e. the Hoare triple $\{\phi\}\ y \leftarrow p\ \{\psi\}$ is satisfied iff $\mathsf{F}(\phi, y \leftarrow p, \psi)$ does not remove any possible executions from p.

We note some basic properties of the above concepts:

Lemma 23. *For a program $p : TA$ and a Boolean term $a : 2$, we have*

$$\mathsf{do}\ x \leftarrow a?(\mathsf{ret}\, x); p[a/z] = \mathsf{do}\ x \leftarrow a?(\mathsf{ret}\, x); p[\top/z].$$

Proof. Case distinction over a. □

Lemma 24. *Let $\phi[\psi_1/a_1, \dots, \psi_n/a_n]$ be a valid test, where ϕ is a propositional formula over $a_1, \dots, a_n$, and the ψ_i are tests. Then*

$$\mathsf{do}\ a_1 \leftarrow \psi_1; \dots; a_n \leftarrow \psi_n; \mathsf{ret}(a_1, \dots, a_n, \phi) = \mathsf{do}\ a_1 \leftarrow \psi_1; \dots; a_n \leftarrow \psi_n; \mathsf{ret}(a_1, \dots, a_n, \top),$$

i.e. ϕ can be replaced by $\top$ in all subsequent computations.

Proof. This is a special case of [43, Lemma 4.32]. □

It turns out that the above definition of Hoare triple is equivalent to a previous generic definition, which in particular enables use of existing sequential monad-based Hoare calculi [43, 44]:

Lemma 25. *For every program p and tests ϕ, ψ, $\{\phi\}\ y \leftarrow p\ \{\psi\}$ is equivalent to the equation*

$$\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle y, x \Rightarrow z\rangle = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle y, \top\rangle. \tag{15}$$

Proof. Suppose that $\{\phi\}\ y \leftarrow p\ \{\psi\}$ holds, i.e. $p = \mathsf{F}(\phi, y \leftarrow p, \psi)$. Then we have

$$\begin{aligned}
&\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle y, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow \mathsf{F}(\phi, p, \psi); z \leftarrow \psi; \mathsf{ret}\langle y, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; x' \leftarrow \phi; y' \leftarrow p; z' \leftarrow \psi; y \leftarrow (x' \Rightarrow z')?(\mathsf{ret}\, y'); z \leftarrow \psi; \mathsf{ret}\langle y, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; x' \leftarrow \phi; y' \leftarrow p; z' \leftarrow \psi; y \leftarrow (x' \Rightarrow z')?(\mathsf{ret}\, y'); z \leftarrow \psi; \mathsf{ret}\langle y, \top\rangle \\
&\quad = \dots \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle y, \top\rangle.
\end{aligned}$$

using the assumption $\{\phi\}\ y \leftarrow p\ \{\psi\}$ and Lemma 23.

Conversely, if (15) holds, we have

$$\begin{aligned}
&\mathsf{F}(\phi, y \leftarrow p, \psi) \\
&= \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; (x \Rightarrow z)?(\mathsf{ret}\, y) \\
&= \mathsf{do}\ \langle y, v\rangle \leftarrow (\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}(y, x \Rightarrow z)); (v)?(\mathsf{ret}\, y) \\
&= \mathsf{do}\ \langle y, v\rangle \leftarrow (\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}(y, \top)); (v)?(\mathsf{ret}\, y) \\
&= \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; (\top)?(\mathsf{ret}\, y) \\
&= p,
\end{aligned}$$

thus establishing $\{\phi\}\ y \leftarrow p\ \{\psi\}$. □

Lemma 25 enables us to just import a generic sound Hoare calculus [44]; we refrain from spelling out the details of the calculus here, but emphasize that it does support the standard Hoare rules in a generic monad-based environment. In order to prove properties concerning a process as a whole, such as safety, we need to introduce suitable correctness judgements for processes.

7.1. Resumptive Hoare triples

We start from the observation that filtering by pre- and post-condition can be extended to processes by putting

$$\mathsf{F}_\nu(\phi, z \leftarrow p, \psi) = \mathsf{init}\, y := p\ \mathsf{unfold}\, \{\mathsf{F}(\phi, z \leftarrow \mathsf{out}(y), \psi)\}.$$

Thus, $\mathsf{F}_\nu(\phi, z \leftarrow p, \psi)$ removes executions satisfying the precondition ϕ but not the postcondition ψ from every step of the process. We note the following expected property.

Lemma 26. *For $p : T^\nu(A)$, we have $\{\phi\}\ z \leftarrow \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p, \psi))\ \{\psi\}$.*

Proof. We have

$$\begin{aligned}
&\mathsf{F}(\phi, \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p, \psi)), \psi)\\
&= \mathsf{F}(\phi, z \leftarrow \mathsf{next}\ \mathsf{F}(\ \phi, z \leftarrow \mathsf{out}(p), \psi)\ \mathsf{is}\\
&\qquad\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi));\\
&\qquad\quad \mathsf{done}\, y \mapsto \mathsf{stop}\, y, \psi)\\
&= \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p), \psi)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi));\\
&\qquad\qquad\qquad\qquad \mathsf{done}\, y \mapsto \mathsf{stop}\, y\\
&= \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p, \psi))
\end{aligned}$$

and thus we are done by the definition of Hoare triples. □

We then introduce *resumptive Hoare triples* $\{\{\phi\}\}\ y \leftarrow p\ \{\{\psi \mid \xi\}\}$ simply as abbreviations for the equation

$$\mathsf{F}_\nu(\phi, z \leftarrow p, \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \xi; \mathsf{inr}\, y \mapsto \psi) = p$$

where $x \notin FV(\xi)$. We use the notation $\psi \mid \xi$ in the same sense in standard Hoare triples $\{\phi\}\ z \leftarrow q\ \{\psi \mid \xi\}$ where $q : T(T^\nu(A) + A)$, as well as directly in filter expressions $\mathsf{F}(\phi, z \leftarrow p, \psi \mid \xi)$ and $\mathsf{F}_\nu(\phi, z \leftarrow p, \psi \mid \xi)$. Intuitively, a resumptive Hoare triple $\{\{\phi\}\}\ y \leftarrow p\ \{\{\psi \mid \xi\}\}$ is satisfied if every terminal step satisfying the precondition ϕ satisfies the postcondition ψ when the outcome of the computation is bound to y; and every non-terminal step satisfying the precondition ϕ satisfies the postcondition ξ, where we call a step of a process *terminal* if it terminates, returning a value, and *non-terminal* otherwise, i.e. if it returns a resumption. In the current version of the calculus, the postcondition ξ cannot mention the resumption in the non-terminal case; it is an interesting point for future research to study higher-order partial correctness statements where this restriction is lifted. A calculus for resumptive Hoare triples is given in Fig. 3. Moreover, by Lemma 26, $\{\{\phi\}\}\ y \leftarrow p\ \{\{\psi \mid \xi\}\}$ implies $\{\phi\}\ y \leftarrow \mathsf{out}(p)\ \{\psi \mid \xi\}$. Some comments on the individual rules:

- **(atom)**: The rule essentially states that $[p]$ terminates after the first step.
- **(seq)**: A terminal step of p becomes a non-terminal step of $\mathsf{seq}\ x \leftarrow p; q$; hence the postcondition in the resumptive Hoare triple for p in the premise is $\xi \mid \xi$.
- **(par)**: This rule just captures the fact that every terminal step of $p \parallel q$ is a terminal step of either p or q, and correspondingly for non-terminal steps. There is, due to the weakening rule **(wk)**, no great difference between a disjunctive formulation of the postcondition as in the terminal case and a uniform formulation as in the non-terminal case; the main reason to prefer a disjunctive formulation in the terminal case is that the postconditions ψ and χ concern different variables.
- **(if)**: Observe that b is a non-terminal step in $\mathsf{if}_\nu\, b\, \mathsf{then}\, p\, \mathsf{else}\, q$.
- **(while)**: For the terminal case, the postcondition is split into two parts here: the stateless condition $\mathsf{ret}(\chi)$ is an invariant property of the iteration parameter x, i.e. required to hold of the initial value p and to be preserved in every iteration of q. On the other hand, the stateful condition ψ is required to be satisfied after the only possible type of terminal step, namely evaluation of the condition b with negative result. The fact that all other steps (including evaluation of b with positive result) are non-terminal is reflected in the pervasive appearance of the postcondition ξ in the premises.

Lemma 27. *The calculus presented in Fig. 3 is sound.*

$$\textbf{(atom)}\ \frac{\{\phi\}\ x \leftarrow p\ \{\psi\}}{\{\{\phi\}\}\ x \leftarrow [p]\ \{\{\psi \mid \xi\}\}} \qquad \textbf{(seq)}\ \frac{\{\{\phi\}\}\ x \leftarrow p\ \{\{\xi \mid \xi\}\} \quad \{\{\phi\}\}\ y \leftarrow q\ \{\{\psi \mid \xi\}\}}{\{\{\phi\}\}\ y \leftarrow (\mathsf{seq}\ x \leftarrow p; q)\ \{\{\psi \mid \xi\}\}}$$

$$\textbf{(par)}\ \frac{\{\{\phi\}\}\ x \leftarrow p\ \{\{\psi \mid \xi\}\} \quad \{\{\phi\}\}\ y \leftarrow q\ \{\{\chi \mid \xi\}\}}{\{\{\phi\}\}\ \langle x, y\rangle \leftarrow p \parallel q\ \{\{\psi \vee \chi \mid \xi\}\}}$$

$$\textbf{(if)}\ \frac{\{\phi\}\ b\ \{\xi\} \quad \{\{\phi\}\}\ x \leftarrow p\ \{\{\psi \mid \xi\}\} \quad \{\{\phi\}\}\ x \leftarrow q\ \{\{\psi \mid \xi\}\}}{\{\{\phi\}\}\ x \leftarrow (\mathsf{if}_\nu\ b\ \mathsf{then}\ p\ \mathsf{else}\ q)\ \{\{\psi \mid \xi\}\}}$$

$$\textbf{(while)}\ \frac{\{\{\mathsf{ret}(\chi)\}\}\ x \leftarrow q\ \{\{\mathsf{ret}(\chi) \mid \top\}\} \quad \mathsf{ret}(\chi[p/x]) \quad \{\{\phi\}\}\ x \leftarrow q\ \{\{\xi \mid \xi\}\} \quad \{\phi \wedge \mathsf{ret}(\chi)\}\ v \leftarrow b\ \{(v \wedge \xi) \vee (\neg v \wedge \psi)\}}{\{\{\phi\}\}\ x \leftarrow (\mathsf{init}\ x := p\ \mathsf{while}\ b\ \mathsf{do}\ q)\ \{\{\psi \wedge \mathsf{ret}(\chi) \mid \xi\}\}}$$

$$\textbf{(wk)}\ \frac{\phi' \Rightarrow \phi \quad \psi \Rightarrow \psi' \quad \xi \Rightarrow \xi' \quad \{\{\phi\}\}\ x \leftarrow p\ \{\{\psi \mid \xi\}\}}{\{\{\phi'\}\}\ x \leftarrow p\ \{\{\psi' \mid \xi'\}\}}$$

Figure 3: A calculus of resumptive Hoare triples.

Proof. Rule **(atom)**. Let $\zeta = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, y \mapsto \xi;\ \mathsf{inr}\, x \mapsto \psi)$. By definition,

$$\begin{aligned}
&\{\{\phi\}\}\ x \leftarrow [p]\ \{\{\psi \mid \xi\}\}\\
&\quad\iff \mathsf{F}_\nu(\phi, z \leftarrow [p], \zeta) = [p]\\
&\quad\iff \mathsf{init}\, v := [p]\ \mathsf{unfold}\, \{\mathsf{F}(\phi, z \leftarrow \mathsf{out}(v), \zeta)\} = [p]\\
&\quad\iff \mathsf{out}(\mathsf{init}\, v := [p]\ \mathsf{unfold}\, \{\mathsf{F}(\phi, z \leftarrow \mathsf{out}(v), \zeta)\}) = \mathsf{out}[p]\\
&\quad\iff \mathsf{next}\ \mathsf{F}(\phi,\ z \leftarrow \mathsf{out}[p], \zeta)\ \mathsf{is}\\
&\qquad\qquad \mathsf{rest}\, x \mapsto \mathsf{cont}(\mathsf{init}\, y := x\ \mathsf{unfold}\, \{\mathsf{F}(\phi, z \leftarrow \mathsf{out}(y), \zeta)\});\\
&\qquad\quad \mathsf{done}\, r \mapsto \mathsf{stop}\, r\\
&\qquad = \mathsf{out}[p]\\
&\quad\iff \mathsf{do}\ v \leftarrow \mathsf{F}(\phi, x \leftarrow p, \psi); \mathsf{stop}(v) = \mathsf{do}\ v \leftarrow p; \mathsf{stop}(v).
\end{aligned}$$

The latter equation is an obvious consequence of the premise, i.e. of $\{\phi\}\ x \leftarrow p\ \{\psi\}$, and hence we are done.

Rule **(seq)**. Let $\zeta = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, v \mapsto \xi;\ \mathsf{inr}\, y \mapsto \psi)$. Then the premises of the rule can be written as

$$p = \mathsf{F}_\nu(\phi, z \leftarrow p, \xi), \tag{16}$$
$$q = \mathsf{F}_\nu(\phi, z \leftarrow q, \zeta). \tag{17}$$

Note that by (16) and by Lemma 26, $\mathsf{F}(\phi, \mathsf{out}(p), \xi) = \mathsf{out}(p)$. Now we prove the conclusion by coinduction, i.e. by showing that $\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow p; q), \zeta)$ satisfies the same corecursive equation as $(\mathsf{seq}\ x \leftarrow p; q)$. Namely, we show

$$\begin{aligned}
&\mathsf{out}(\,\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow p; q), \zeta))\\
&\quad=\ \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow r; q), \zeta));\\
&\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}\, q.
\end{aligned}$$

Indeed, we have

$$\mathsf{out}(\,\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow p; q), \zeta))$$

$$
\begin{aligned}
&= \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(\mathsf{seq}\ x \leftarrow p; q), \zeta)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \zeta));\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{done}\, r \mapsto \mathsf{stop}\, r\\
&= \mathsf{next}\ \mathsf{F}(\phi, \mathsf{out}(p), \xi)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow r; q), \zeta));\\
&\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow q, \zeta))\\
&= \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow r; q), \zeta));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}\, q
\end{aligned}
$$

and we are done.

Rule **(par)**. Given the equations

$$
\mathsf{F}_\nu(\phi, z \leftarrow p, \psi') = p, \qquad\qquad \mathsf{F}_\nu(\phi, z \leftarrow q, \chi') = q,
$$

where $\psi' = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, v \mapsto \xi; \mathsf{inr}\langle x, y\rangle \mapsto \psi)$, $\chi' = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, v \mapsto \xi; \mathsf{inr}\langle x, y\rangle \mapsto \chi)$, we show by coinduction the identity

$$
\mathsf{F}_\nu(\phi, z \leftarrow p \parallel q, \psi' \vee \chi') = p \parallel q.
$$

Soundness of **(par)** will then follow by the definition of a resumptive Hoare triple. We prove the identity in question by showing that

$$
\begin{aligned}
&\mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p \parallel q, \psi' \vee \chi'))\\
&\qquad = \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p \between q, \psi' \vee \chi')) +\\
&\qquad\quad \mathsf{out}(\mathsf{do}_\nu\, \langle y, x\rangle \leftarrow \mathsf{F}_\nu(\phi, z \leftarrow q \between p, \psi' \vee \chi'); \mathsf{ret}_\nu\langle x, y\rangle),
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p \between q, \psi' \vee \chi'))\\
&\qquad = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r \parallel q, \psi' \vee \chi'));\\
&\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle).
\end{aligned}
$$

Since the same system of equations uniquely defines $p \parallel q$ and $p \between q$ we would indeed arrive at the equality in question. We have

$$
\begin{aligned}
&\mathsf{out}(\ \mathsf{F}_\nu(\phi, z \leftarrow p \parallel q, \psi' \vee \chi'))\\
&\quad = \mathsf{out}(\mathsf{init}\, v := p \parallel q\ \mathsf{unfold}\, \{\mathsf{F}(\phi, z \leftarrow \mathsf{out}(v), \psi' \vee \chi')\})\\
&\quad = \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p \parallel q), \psi' \vee \chi')\ \mathsf{is}\\
&\qquad\quad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi' \vee \chi')); \mathsf{done}\langle x, y\rangle \mapsto \mathsf{stop}\langle x, y\rangle\\
&\quad = \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p \between q), \psi' \vee \chi')\ \mathsf{is}\\
&\qquad\quad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi' \vee \chi')); \mathsf{done}\langle x, y\rangle \mapsto \mathsf{stop}\langle x, y\rangle\ +\\
&\qquad \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(\mathsf{do}_\nu\, \langle y, x\rangle \leftarrow q \between p; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')\ \mathsf{is}\\
&\qquad\quad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi' \vee \chi')); \mathsf{done}\langle x, y\rangle \mapsto \mathsf{stop}\langle x, y\rangle\\
&\quad = \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p \between q, \psi' \vee \chi')) +\\
&\qquad \mathsf{next}\ \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow q \between p, \psi' \vee \chi'))\ \mathsf{is}\\
&\qquad\quad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{do}_\nu\, \langle y, x\rangle \leftarrow r; \mathsf{ret}_\nu\langle x, y\rangle); \mathsf{done}\langle y, x\rangle \mapsto \mathsf{stop}\langle x, y\rangle\\
&\quad = \mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow p \between q, \psi' \vee \chi')) +\\
&\qquad \mathsf{out}(\mathsf{do}_\nu\, \langle y, x\rangle \leftarrow \mathsf{F}_\nu(\phi, z \leftarrow q \between p, \psi' \vee \chi'); \mathsf{ret}_\nu\langle x, y\rangle).
\end{aligned}
$$

Analogously,

$$
\begin{aligned}
&\mathsf{out}(\ \mathsf{F}_\nu(\phi, z \leftarrow p \between q, \psi' \vee \chi'))\\
&\quad = \mathsf{out}(\mathsf{init}\, v := p \between q\ \mathsf{unfold}\, \{\mathsf{F}(\phi, z \leftarrow \mathsf{out}(v), \psi' \vee \chi')\})
\end{aligned}
$$

$$
\begin{aligned}
&= \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p \,[\!]\, q), \psi' \vee \chi')\ \mathsf{is} \\
&\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi' \vee \chi')); \ \mathsf{done}\langle x, y\rangle \mapsto \mathsf{stop}\langle x, y\rangle \\
&= \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p), \psi')\ \mathsf{is} \\
&\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r \parallel q, \psi' \vee \chi')); \\
&\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')) \\
&= \ \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r \parallel q, \psi' \vee \chi')); \\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')).
\end{aligned}
$$

To match the goal it suffices to show that

$$
\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi') = \mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle.
$$

Since

$$
\begin{aligned}
&\mathsf{out}(\ \mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')) \\
&\quad = \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')\ \mathsf{is} \\
&\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \psi' \vee \chi')); \ \mathsf{done}\langle x, y\rangle \mapsto \mathsf{stop}\langle x, y\rangle \\
&\quad = \ \mathsf{next}\ \mathsf{F}(\phi,\ z \leftarrow \mathsf{out}(q), \psi' \vee \chi')\ \mathsf{is} \\
&\qquad \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow r; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')); \\
&\qquad \mathsf{done}\, y \mapsto \mathsf{stop}\langle x, y\rangle \\
&\quad = \ \mathsf{next}\ \mathsf{out}(q)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow r; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')); \\
&\qquad\qquad \mathsf{done}\, y \mapsto \mathsf{stop}\langle x, y\rangle,
\end{aligned}
$$

$\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle), \psi' \vee \chi')$ satisfies the same corecursive equation as $\mathsf{do}_\nu\ y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle$ and therefore both must be equal.

Rule **(if)**. We abbreviate $f_\nu(r) = \mathsf{F}_\nu(\phi, x \leftarrow r, \psi \mid \xi)$ and $f(s) = \mathsf{F}(\phi, x \leftarrow s, \psi \mid \xi)$. Under the premises of the rule, we have

$$
\begin{aligned}
&\mathsf{out}(f_\nu(\mathsf{if}_\nu\, b\, \mathsf{then}\, p\, \mathsf{else}\, q)) \\
&\quad = \mathsf{next}\ f(\mathsf{out}(\mathsf{if}_\nu\, b\, \mathsf{then}\, p\, \mathsf{else}\, q))\ \mathsf{is}\ \mathsf{rest}\, z \mapsto \mathsf{cont}\, f_\nu(z); \ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&\quad = \mathsf{next}\ f(\mathsf{do}\ z \leftarrow b; \mathsf{cont}(\mathsf{if}\ z\ \mathsf{then}\ p\ \mathsf{else}\ q))\ \mathsf{is}\ \mathsf{rest}\, z \mapsto \mathsf{cont}\, f_\nu(z); \ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&\quad = \mathsf{next}\,(\mathsf{do}\ z \leftarrow \mathsf{F}(\phi, b, \xi); \mathsf{cont}(\mathsf{if}\ z\ \mathsf{then}\ p\ \mathsf{else}\ q))\ \mathsf{is} \\
&\qquad\quad \mathsf{rest}\, z \mapsto \mathsf{cont}\, f_\nu(z); \ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&\quad = \mathsf{do}\ z \leftarrow \mathsf{F}(\phi, b, \xi); \mathsf{cont}(\mathsf{if}\ z\ \mathsf{then}\ f_\nu(p)\ \mathsf{else}\ f_\nu(q)) \\
&\quad = \mathsf{do}\ z \leftarrow b; \mathsf{cont}(\mathsf{if}\ z\ \mathsf{then}\ p\ \mathsf{else}\ q) \\
&\quad = \mathsf{out}(\mathsf{if}_\nu\, b\, \mathsf{then}\, p\, \mathsf{else}\, q)
\end{aligned}
$$

where we have used the premises in the second last step. Application of tuo on both sides yields the conclusion of the rule.

Rule **(while)**. Let $\zeta = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, v \mapsto \xi;\ \mathsf{inr}\, x \mapsto \psi \wedge \mathsf{ret}\, \chi)$. We assume the premises of the rule and show that $\{\{\phi\}\}\ x \leftarrow \mathsf{tuo}(\chi?\, \mathsf{out}(\mathsf{init}\, x := x\ \mathsf{while}\ b\ \mathsf{do}\ q))\ \{\{\psi \wedge \mathsf{ret}\, \chi \mid \xi\}\}$. This will easily imply the goal after substituting x for p and applying the assumption $\chi[p/x]$. First, observe that the equation

$$
\begin{aligned}
&\chi?\, \mathsf{out}(\mathsf{init}\, x := x\ \mathsf{while}\ b\ \mathsf{do}\ q) = \\
&\quad \mathsf{do}\ v \leftarrow b; \mathsf{if}\ v\ \ \mathsf{then}\ \chi?\, \mathsf{cont}(\mathsf{seq}\ x \leftarrow q; \mathsf{tuo}(\chi?\, \mathsf{out}(\mathsf{init}\, x := x\ \mathsf{while}\ b\ \mathsf{do}\ q))) \\
&\qquad\qquad\quad \mathsf{else}\ \chi?\, \mathsf{stop}(x)
\end{aligned}
$$

uniquely defines $\chi?\,\mathsf{out}(\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q)$ — the underlying argument is the same as the one behind the analogous definition of the while-loop $(\mathsf{init}\,x := p\ \mathsf{while}\ b\ \mathsf{do}\ q)$. We will be done once we show that $\mathsf{F}_\nu(\phi, z \leftarrow \mathsf{tuo}(\chi?\,\mathsf{out}(\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q)), \zeta)$ satisfies the same equation. We have

$$\begin{aligned}
&\mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow \mathsf{tuo}(\chi?\,\mathsf{out}(\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q)), \zeta)) \\
&\quad = \mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \chi?\,\mathsf{out}(\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta)\ \mathsf{is} \\
&\qquad \mathsf{rest}\,r \mapsto \mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow r, \zeta));\ \mathsf{done}\,r \mapsto \mathsf{stop}\,r \\
&\quad = \mathsf{do}\ v \leftarrow \mathsf{F}(\phi, v \leftarrow b, v \wedge \xi \vee \neg v \wedge \psi);\ \mathsf{if}\ v \\
&\qquad \mathsf{then}\ \chi?\,\mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow q; \mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta)) \\
&\qquad\ \mathsf{else}\ \chi?\,\mathsf{stop}(x) \\
&\quad = \mathsf{do}\ v \leftarrow b; \mathsf{if}\ v \\
&\qquad \mathsf{then}\ \chi?\,\mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow q; \mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta)) \\
&\qquad\ \mathsf{else}\ \chi?\,\mathsf{stop}(x) \\
&\quad = \mathsf{do}\ v \leftarrow b; \mathsf{if}\ v \\
&\qquad \mathsf{then}\ \chi?\,\mathsf{cont}(\mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{seq}\ x \leftarrow q; \\
&\qquad\qquad\qquad \mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta)), \zeta)) \\
&\qquad\ \mathsf{else}\ \chi?\,\mathsf{stop}(x)
\end{aligned}$$

where the last step is by Lemma 28 proved next. By assumption, $\{\{\phi\}\}\ x \leftarrow q\ \{\{\xi \mid \xi\}\}$ and also, obviously, $\{\{\phi\}\}\ x \leftarrow \mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta)\ \{\{\psi \wedge \mathsf{ret}\,\chi \mid \xi\}\}$. Therefore, by **(seq)**, $\{\{\phi\}\}\ x \leftarrow (\mathsf{seq}\ x \leftarrow q; \mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta))\ \{\{\psi \wedge \mathsf{ret}\,\chi \mid \xi\}\}$, or, equivalently,

$$\begin{aligned}
&\mathsf{F}_\nu(\phi, z \leftarrow (\,\mathsf{seq}\ x \leftarrow q; \mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta)), \zeta) \\
&\qquad = \mathsf{seq}\ x \leftarrow q; \mathsf{F}_\nu(\phi, z \leftarrow (\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q), \zeta).
\end{aligned}$$

Now the calculation can be continued so as to obtain

$$\begin{aligned}
&\mathsf{out}(\mathsf{F}_\nu(\phi, z \leftarrow \mathsf{tuo}(\chi?\,\mathsf{out}(\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q)), \zeta)) \\
&\quad = \mathsf{do}\ v \leftarrow b; \mathsf{if}\ v \\
&\qquad \mathsf{then}\ \chi?\,\mathsf{cont}(\mathsf{seq}\ x \leftarrow q; \mathsf{F}_\nu(\phi, z \leftarrow \mathsf{tuo}(\chi?\,\mathsf{out}(\mathsf{init}\,x := x\ \mathsf{while}\ b\ \mathsf{do}\ q))), \zeta)) \\
&\qquad\ \mathsf{else}\ \chi?\,\mathsf{stop}(x).
\end{aligned}$$

Rule **(wk)**: by straightforward coinduction. □

It remains to pay the debts incurred during the proof:

Lemma 28. *For all appropriately typed ϕ, ψ, ξ, p, q, we have*

$$\mathsf{F}_\nu(\phi, x \leftarrow (\mathsf{seq}\ x \leftarrow p; \mathsf{F}_\nu(\phi, x \leftarrow q, \psi \mid \xi)), \psi \mid \xi) = \mathsf{F}_\nu(\phi, x \leftarrow (\mathsf{seq}\ x \leftarrow p; q), \psi \mid \xi).$$

Proof. We abbreviate $\mathsf{F}_\nu(\phi, x \leftarrow \_, \psi \mid \xi)$ as f_ν, and $\mathsf{F}(\phi, x \leftarrow \_, \psi \mid \xi)$ as f. The function $g(p) = f_\nu(\mathsf{seq}\ x \leftarrow p; q)$ satisfies, and hence is uniquely defined by, the corecursive equation

$$\mathsf{out}(g(p)) = \mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\,z \mapsto \mathsf{cont}(g(z));\ \mathsf{done}\,x \mapsto \mathsf{cont}(f_\nu(q)).$$

To prove the claim, it hence suffices to show that $g'(p) = f_\nu(\mathsf{seq}\ x \leftarrow p; f_\nu(q))$ satisfies the same equation:

$$\begin{aligned}
&\mathsf{out}(g'(p)) = \\
&\qquad \mathsf{next}\ f(\mathsf{out}(\mathsf{seq}\ x \leftarrow p; f_\nu(q)))\ \mathsf{is}\ \mathsf{rest}\,x \mapsto \mathsf{cont}(f_\nu(x));\ \mathsf{done}\,y \mapsto \mathsf{stop}\,y =
\end{aligned}$$

$$\begin{aligned}
&\mathsf{next}\ f(\mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, z \mapsto (\mathsf{seq}\ x \leftarrow z; f_\nu(q)); \mathsf{done}\, x \mapsto \mathsf{cont}(f_\nu(q)))\ \mathsf{is}\\
&\qquad \mathsf{rest}\, x \mapsto \mathsf{cont}(f_\nu(x));\\
&\qquad \mathsf{done}\, y \mapsto \mathsf{stop}\, y =\\
&\mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\, z \mapsto \mathsf{cont}(f_\nu(\mathsf{seq}\ x \leftarrow z; f_\nu(q))); \mathsf{done}\, x \mapsto \mathsf{cont}(f_\nu(f_\nu(q))) =\\
&\mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\, z \mapsto \mathsf{cont}(g'(z)); \mathsf{done}\, x \mapsto \mathsf{cont}(f_\nu(q)),
\end{aligned}$$

where the last step uses the fact, easily proved by coinduction, that f_ν is idempotent. □

7.2. Safety

Next, we show that safety can be proved by means of step-wise invariants, formulated as resumptive Hoare triples (Lemma 31). We then apply this calculus to verify a toy example. It turns out that even simple examples require another ingredient, namely a systematic way of annotating programs with labels for use in invariants in the same style as advocated in [30].

Given a process $p : T^\nu A$, we define the *partial execution* $\mathsf{exec}(p) : T^\nu A$ of p by

$$\mathsf{exec}(p) = \mathsf{tuo}(\mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{out}(x); \mathsf{done}\, y \mapsto \mathsf{stop}\, y).$$

Note that this will execute first $\mathsf{out}(p)$ and then $\mathsf{out}(x)$ if the left alternative is taken; thus, $\mathsf{exec}(p)$ is the process obtained by collapsing the first two steps of p into one. The following lemma plays the role of a sequencing rule for exec.

Lemma 29. *If ϕ, ψ, ψ', and χ are tests such that $\psi \Rightarrow \psi' \wedge \chi$ and*

$$\mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p), \psi)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{F}(\psi', z \leftarrow \mathsf{out}(x), \chi); \mathsf{done}\, y \mapsto \mathsf{stop}\, y = \mathsf{out}(\mathsf{exec}(p)),$$

then

$$\{\phi\}\ z \leftarrow \mathsf{out}(\mathsf{exec}(p))\ \{\chi\}.$$

Proof. We have to show that

$$\mathsf{F}(\phi, z \leftarrow \mathsf{out}(\mathsf{exec}(p), \chi)) = \mathsf{out}(\mathsf{exec}(p)).$$

We proceed to analyse the left hand side. By the assumption of the lemma and the definition of F, it equals, after some simplification,

$$\mathsf{do}\ a \leftarrow \phi; y \leftarrow \mathsf{out}(p); c \leftarrow \psi; z \leftarrow q; b \leftarrow \chi; (a \Rightarrow b)?(\mathsf{ret}\, z) \tag{18}$$

where

$$\begin{aligned}
q = \mathsf{next}\,&(a \Rightarrow c)?(\mathsf{ret}\, y)\ \mathsf{is}\\
&\mathsf{rest}\, x \mapsto \mathsf{do}\ d \leftarrow \psi'; z \leftarrow \mathsf{out}(x); e \leftarrow \chi; (d \Rightarrow e)?(\mathsf{ret}\, z);\\
&\mathsf{done}\, y \mapsto \mathsf{stop}\, y.
\end{aligned}$$

We are done once we show that the term (18) equals $\mathsf{do}\ a \leftarrow \phi; y \leftarrow \mathsf{out}(p); c \leftarrow \psi; q$, as this term equals $\mathsf{out}(\mathsf{exec}(p))$ by assumption; to this end, it suffices to show that $a \Rightarrow b$ can be replaced with $\top$ in the last test. We analyse the two branches of q; in both branches, we can assume $a \Rightarrow c$ by Lemma 23. After executing the first branch of q, we can moreover assume $c \Rightarrow d$ by Lemma 24, and $d \Rightarrow e$ by Lemma 23. Continuing in the execution of (18), we can assume $e \Rightarrow b$ by Lemma 24, so that we can replace $a \Rightarrow b$ with $\top$ in the last test, thus proving the claim for this branch. For the second branch, we can assume $c \Rightarrow b$ by Lemma 24 (as nothing is executed in this case between the evaluation of $c \leftarrow \psi$ and $b \leftarrow \chi$), so that again we may replace $a \Rightarrow b$ with $\top$. □

We denote by $\mathsf{exec}^n(p)$ the n-fold application of exec to p, i.e. the process in which the first $n+1$ steps of p are collapsed into one. This allows us to formalize satisfaction of a safety property ϕ by a process p:

Definition 30 (Safety). We say that $x \leftarrow p$ is *safe w.r.t.* $\psi \mid \chi$ *at* ϕ and write $\mathsf{safe}(\phi; x \leftarrow p; \psi \mid \chi)$ if for every n,

$$\{\phi\}\ x \leftarrow \mathsf{out}(\mathsf{exec}^n(p))\ \{\psi \mid \chi\}.$$

Thus, $\mathsf{safe}(\phi; x \leftarrow p; \psi \mid \chi)$ holds if $\psi \mid \chi$ holds throughout the execution of p provided that the initial state satisfies ϕ and there is no interference from the environment (in sensible applications, the environment will already be incorporated in p).

Lemma 31. *Let* ϕ, ψ, *and* χ *be tests, and let* p *be a process. If*

$$\{\{\phi\}\}\ x \leftarrow p\ \{\{\psi \mid \chi \wedge \phi\}\}$$

then $\mathsf{safe}(\phi, x \leftarrow p, \psi \mid \chi)$.

Here, ϕ plays the role of an invariant to be preserved by all non-terminal steps. To prove the above lemma, we begin with the following observation, which extends Lemma 26 to cover also the second step of a process.

Lemma 32. *If* $\{\{\phi\}\}\ z \leftarrow p\ \{\{\psi \mid \xi\}\}$, *then*

$$\mathsf{next}\ \mathsf{F}(\phi, z \leftarrow \mathsf{out}(p), \psi \mid \xi)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{F}(\phi, z \leftarrow \mathsf{out}(x), \psi \mid \xi);\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y = \mathsf{out}(\mathsf{exec}(p)).$$

Proof. We abbreviate $\mathsf{F}(\phi, z \leftarrow \_, \psi \mid \xi)$ to f, and $\mathsf{F}_\nu(\phi, z \leftarrow \_, \psi \mid \xi)$ to f_ν. Then

$$\begin{aligned}
&\mathsf{next}\ f(\mathsf{out}(p)))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto f(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&= \mathsf{next}\ f(\mathsf{out}(f_\nu(p)))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto f(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&= \mathsf{next}\ f(\mathsf{out}(\mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(f_\nu(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y))\ \mathsf{is} \\
&\qquad \mathsf{rest}\, x \mapsto f(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&= \mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto f(\mathsf{out}(f_\nu(x)));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&= \mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto f(\mathsf{out}(f_\nu(x)));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y \\
&= \mathsf{next}\ f(\mathsf{out}(p))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{out}(f_\nu(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y
\end{aligned}$$

where we use Lemma 26 in the last step. The last term is easily seen to equal $\mathsf{out}(\mathsf{exec}(f_\nu(p)))$ by unfolding the definition of exec and the corecursive definition of $f_\nu(p)$. □

Proof (Lemma 31). We show by induction on n that for all n,

$$\{\{\phi\}\}\ x \leftarrow \mathsf{exec}^n(p)\ \{\{\psi \mid \chi \wedge \phi\}\}; \tag{19}$$

this implies the claim by Lemma 26.

The base case $n = 0$ is just the assumption. For the inductive step $n - 1 \to n$, we have

$$\mathsf{out}(\mathsf{exec}^n(p)) = \mathsf{next}\ \mathsf{out}(\mathsf{exec}^{n-1}(p)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, x;\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y. \tag{20}$$

We put $\rho = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \psi;\ \mathsf{inr}\, y \mapsto \phi \wedge \xi$ and then abbreviate $\mathsf{F}(\phi, x \leftarrow \_, \rho)$ by f and $\mathsf{F}_\nu(\phi, x \leftarrow \_, \rho)$ by f_ν. In this notation, (19) becomes $f_\nu(\mathsf{exec}^n(p)) = \mathsf{exec}^n(p)$, of which $f(\mathsf{out}(\mathsf{exec}^n(p))) = \mathsf{out}(\mathsf{exec}^n(p))$ is a consequence by Lemma 26. By the inductive assumption,

$$\begin{aligned}
\mathsf{out}(\mathsf{exec}^{n-1}(p) &= \mathsf{out}(f_\nu(\mathsf{exec}^{n-1}(p))) \\
&= \mathsf{next}\ f(\mathsf{out}(\mathsf{exec}^{n-1}(p)))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(f_\nu(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y.
\end{aligned}$$

Substituting this into the right hand side of (21), we obtain after some simplification

$$\mathsf{out}(\mathsf{exec}^n(p)) = \mathsf{next}\ f(\mathsf{out}(\mathsf{exec}^{n-1}(p)))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{out}(f_\nu(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y. \tag{21}$$

By unfolding the corecursive definition of f_ν, we obtain that the right hand side equals

$$\begin{aligned}&\mathsf{next}\, f(\mathsf{out}(\mathsf{exec}^{n-1}(p)))\ \mathsf{is}\\ &\quad \mathsf{rest}\, x \mapsto \mathsf{next}\ f(\mathsf{out}(x))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(f_\nu(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y;\\ &\quad \mathsf{done}\, y \mapsto \mathsf{stop}\, y.\end{aligned}$$

By rearranging case distinctions, we transform this term into

$$\begin{aligned}&\mathsf{next}\,(\mathsf{next}\ f(\mathsf{out}(\mathsf{exec}^{n-1}(p)))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto f(\mathsf{out}(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y)\ \mathsf{is}\\ &\quad \mathsf{rest}\, x \mapsto \mathsf{cont}(f_\nu(x));\\ &\quad \mathsf{done}\, y \mapsto \mathsf{stop}\, y.\end{aligned}$$

By the inductive assumption, we can apply Lemmas 32 to the inner next term and then obtain by Lemma 29

$$\begin{aligned}\mathsf{out}(\mathsf{exec}^n(p)) &= \mathsf{next}\ f(\mathsf{out}(\mathsf{exec}^n(p)))\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(f_\nu(x));\ \mathsf{done}\, y \mapsto \mathsf{stop}\, y\\ &= \mathsf{out}(f_\nu(\mathsf{exec}^n(p))).\end{aligned}$$

By application of tuo on both sides, this proves the claim. □

7.3. Labels

As indicated earlier, we need a further ingredient in our verification method that will allow us to extend programs with systematically managed labels. We proceed to develop the notion of labelling; its necessity will become apparent in the example that follows.

Definition 33 (Labels). A *label* is a pair of operations of the form $c : 1 \to Pn$ (recall notation for numerals introduced in Section 2) and $c{\downarrow}\_ : n \to T1$ satisfying the axioms

$$\begin{aligned}\mathsf{do}\ c{\downarrow}v; x \leftarrow p; y \leftarrow c; \mathsf{ret}\langle x, y\rangle &= \mathsf{do}\ c{\downarrow}v; x \leftarrow p; \mathsf{ret}\langle x, v\rangle,\\ \mathsf{do}\ c{\downarrow}w; c{\downarrow}v &= c{\downarrow}v\end{aligned}$$

where p is any program not mentioning c.

In other words, labels have the essential properties of global program variables, although they will be used in a different spirit. Using the standard state monad transformer, one can therefore extend a given monad by any number of labels. Given a label $c : 1 \to Pn$ and a variable v of type n, we define $c_v : 1 \to P2$ as

$$c_v = \mathsf{do}\ z \leftarrow c; \mathsf{ret}(v = z)$$

and let $c_{v_1,\dots,v_k}$ abbreviate the disjunction $c_{v_1} \vee \ldots \vee c_{v_k}$; thus, $c_{v_1,\dots,v_k}$ is a test that checks whether the value of c is in $\{v_1, \dots, v_k\}$. Let $\mathsf{inv}_c(p)$ stand for the equation

$$p = (\mathsf{init}\, r := p\ \mathsf{unfold}\,\{\mathsf{do}\ v \leftarrow c; z \leftarrow \mathsf{out}(r); c{\downarrow}v; \mathsf{ret}\, z\}),$$

i.e. $\mathsf{inv}_c(p)$ holds iff all the atomic steps of p maintain the value of c. Given a label $c : 1 \to Pn$, we define $\mathsf{ppg}_c : n \times T^\nu A \to T^\nu A$ by the corecursive equation

$$\begin{aligned}\mathsf{out}(\mathsf{ppg}_c(v, p)) = \mathsf{do}\ &c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c;\\ &\mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, r));\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x.\end{aligned}$$

The idea of $\mathsf{ppg}_c(p, v)$ is to propagate the value of c taken at the end of any step of p to the succeeding step, with v being the initial value. We summarize the basic properties of ppg_c:

Lemma 34. *For any label c and any processes p and q,*

1. $\mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p)) = \mathsf{ppg}_c(v,p)$;

2. $\mathsf{exec}(\mathsf{ppg}_c(v,p)) = \mathsf{ppg}_c(v, \mathsf{exec}(p))$;

3. $\mathsf{inv}_c(q)$ *implies* $\mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p) \parallel q) = \mathsf{ppg}_c(v, p \parallel q)$.

Proof. Since, by definition,

$$\begin{aligned}
&\mathsf{out}(\mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p))) \\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(\mathsf{ppg}_c(v,r)); w \leftarrow c; \\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w,p));\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c; \\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{ppg}_c(w,r)));\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x
\end{aligned}$$

$\mathsf{ppg}_c(v,p)$ satisfies the same corecursive definitions as $\mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p))$ and hence (1).

Furthermore, we obtain (2) by a routine calculation:

$$\begin{aligned}
&\mathsf{out}(\mathsf{exec}(\,\mathsf{ppg}_c(v,p))) \\
&\quad = \mathsf{next}\ \mathsf{out}(\mathsf{ppg}_c(v,p))\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{out}(r);\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(r); w \leftarrow c; \\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{out}(\mathsf{ppg}_c(w,p));\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto (\mathsf{do}\ w \leftarrow c; \mathsf{out}(\mathsf{ppg}_c(w,r)));\\
&\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto (\mathsf{do}\ w \leftarrow c; c{\downarrow}w; z \leftarrow \mathsf{out}(r); u \leftarrow c;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, s \mapsto \mathsf{cont}(\mathsf{ppg}_c(u,s));\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{stop}\, x);\\
&\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; \mathsf{next}\ \ \mathsf{out}(p)\ \mathsf{is} \\
&\qquad\quad \mathsf{rest}\, r \mapsto (\ \mathsf{next}\ \mathsf{out}(r)\ \mathsf{is}\ \mathsf{rest}\, s \mapsto (\mathsf{do}\ u \leftarrow c; \mathsf{cont}(\mathsf{ppg}_c(u,s)));\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{stop}\, x);\\
&\qquad\quad \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; \mathsf{next}\ \mathsf{out}(\mathsf{exec}(p))\ \mathsf{is}\ \mathsf{rest}\, r \mapsto (\mathsf{do}\ w \leftarrow c; \mathsf{cont}(\mathsf{ppg}_c(w,r)));\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(\mathsf{exec}(p)); w \leftarrow c; \\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w,r));\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{out}(\mathsf{ppg}_c(v, \mathsf{exec}(p))).
\end{aligned}$$

In order to show (3), suppose that $\mathsf{inv}_c(q)$ or, equivalently, $q = f(q)$ where f is uniquely defined by the corecursive scheme

$$f(q) = \mathsf{next}\,(\mathsf{do}\ v \leftarrow c; z \leftarrow \mathsf{out}(q); c{\downarrow}v; \mathsf{ret}\, z)\ \mathsf{is}\ \mathsf{rest}\, r \mapsto \mathsf{cont}\, f(r);\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x.$$

First observe that

$$\begin{aligned}
&\mathsf{out}(\mathsf{ppg}_c(v, p \parallel f(q))) \\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p \parallel f(q)); w \leftarrow c; \\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w,r));\ \mathsf{done}\, x \mapsto \mathsf{stop}\, x \\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c;
\end{aligned}$$

$$
\begin{aligned}
&\quad\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, r \parallel f(q)));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{do}_\nu\ y \leftarrow f(q); \mathsf{ret}_\nu\langle x, y\rangle)) +\\
&\quad\ \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(f(q)); w \leftarrow c;\\
&\quad\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, p \parallel r));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{do}_\nu\ x \leftarrow p; \mathsf{ret}_\nu\langle x, y\rangle))\\
&= \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c;\\
&\quad\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, r \parallel f(q)));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow \mathsf{ppg}_c(w, f(q)); \mathsf{ret}_\nu\langle x, y\rangle) +\\
&\quad\ \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(q); c{\downarrow}v; w \leftarrow c;\\
&\quad\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, p \parallel f(r)));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ x \leftarrow \mathsf{ppg}_c(w, p); \mathsf{ret}_\nu\langle x, y\rangle).
\end{aligned}
$$

By Theorem 14 the resulting definition uniquely determines $\lambda v.\, \lambda p.\, \lambda q.\ \mathsf{ppg}(v, p \parallel f(q))$ for the right-hand side if it can be equivalently written as

$$
\begin{aligned}
&\mathsf{do}\ w \leftarrow (\mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c; \mathsf{ret}\,\mathsf{inl}\, z +\\
&\qquad\qquad \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(q); c{\downarrow}v; w \leftarrow c; \mathsf{ret}\,\mathsf{inr}\, z);\\
&\mathsf{case}\ w\ \mathsf{of}\ \mathsf{inl}\,\mathsf{inl}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, r \parallel f(q)));\\
&\qquad\qquad \mathsf{inl}\,\mathsf{inr}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow \mathsf{ppg}_c(w, f(q)); \mathsf{ret}_\nu\langle x, y\rangle);\\
&\qquad\qquad \mathsf{inr}\,\mathsf{inl}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, p \parallel f(r)));\\
&\qquad\qquad \mathsf{inr}\,\mathsf{inr}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ x \leftarrow \mathsf{ppg}_c(w, p); \mathsf{ret}_\nu\langle x, y\rangle).
\end{aligned}
$$

On the other hand

$$
\begin{aligned}
&\mathsf{out}(\mathsf{ppg}_c(v, \mathsf{ppg}_c(v, p) \parallel f(q)))\\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(\mathsf{ppg}_c(v, p) \parallel f(q)); w \leftarrow c;\\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, r)); \mathsf{done}\, x \mapsto \mathsf{stop}\, x\\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(\mathsf{ppg}_c(v, p)); w \leftarrow c;\\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, r \parallel f(q)));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{do}_\nu\ y \leftarrow f(q); \mathsf{ret}_\nu\langle x, y\rangle)) +\\
&\qquad \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(f(q)); w \leftarrow c;\\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{ppg}_c(v, p) \parallel r));\\
&\qquad\qquad \mathsf{done}\, y \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{do}_\nu\ x \leftarrow \mathsf{ppg}_c(v, p); \mathsf{ret}_\nu\langle x, y\rangle))\\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c;\\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{ppg}_c(w, r) \parallel f(q)));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow \mathsf{ppg}_c(w, f(q)); \mathsf{ret}_\nu\langle x, y\rangle) +\\
&\qquad \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(f(q)); c{\downarrow}v; w \leftarrow c;\\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{ppg}_c(w, p) \parallel f(r)));\\
&\qquad\qquad \mathsf{done}\, y \mapsto \mathsf{cont}(\mathsf{do}_\nu\ x \leftarrow \mathsf{ppg}_c(w, \mathsf{ppg}_c(w, p)); \mathsf{ret}_\nu\langle x, y\rangle)\\
&\quad = \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(p); w \leftarrow c;\\
&\qquad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r \mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{ppg}_c(w, r) \parallel f(q)));\\
&\qquad\qquad \mathsf{done}\, x \mapsto \mathsf{cont}(\mathsf{do}_\nu\ y \leftarrow \mathsf{ppg}_c(w, f(q)); \mathsf{ret}_\nu\langle x, y\rangle) +\\
&\qquad \mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(f(q)); c{\downarrow}v; w \leftarrow c;
\end{aligned}
$$

$$\begin{aligned}\mathsf{case}\ z\ \mathsf{of}\ \mathsf{rest}\, r &\mapsto \mathsf{cont}(\mathsf{ppg}_c(w, \mathsf{ppg}_c(w,p) \parallel f(r)));\\ \mathsf{done}\, y &\mapsto \mathsf{cont}(\mathsf{do}_\nu\ x \leftarrow \mathsf{ppg}_c(w,p); \mathsf{ret}_\nu\langle x, y\rangle),\end{aligned}$$

which means that $\lambda v.\, \lambda p.\, \lambda q.\ \mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p) \parallel f(q))$ satisfies the same definition. Hence $\mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p) \parallel q) = \mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p) \parallel f(q)) = \mathsf{ppg}_c(v, p \parallel f(q)) = \mathsf{ppg}_c(v, p \parallel f(q))$ and we are done. □

The core results allowing us to assume labels in programs for verification purposes are, then, the following.

Lemma 35. *For all appropriate p, ϕ, ψ, ξ and v, $\mathsf{safe}(\phi \wedge c_v, z \leftarrow p, \psi \mid \xi)$ is equivalent to $\mathsf{safe}(\phi \wedge c_v, z \leftarrow \mathsf{ppg}_c(v,p), \psi \mid \xi)$.*

Proof. By Lemma 34(2) (applied n times in the second step),

$$\begin{aligned}&\{\phi \wedge c_v\}\ x \leftarrow \mathsf{out}(\mathsf{exec}^n(\mathsf{ppg}_c(v,p)))\ \{\psi \mid \xi\}\\ \iff &\{\phi \wedge c_v\}\ x \leftarrow \mathsf{out}(\mathsf{ppg}_c(v, \mathsf{exec}^n(v,p)))\ \{\psi \mid \xi\}\\ \iff &\{\phi \wedge c_v\}\ x \leftarrow (\mathsf{do}\ c{\downarrow}v; z \leftarrow \mathsf{out}(\mathsf{exec}^n(p)); u \leftarrow c; \mathsf{ret}\, z)\ \{\psi \mid \xi\}\\ \iff &\{\phi \wedge c_v\}\ x \leftarrow \mathsf{out}(\mathsf{exec}^n(p))\ \{\psi \mid \xi\}\end{aligned}$$

and thus we are done by the definition of safe. □

Lemma 36. *Suppose that $\mathsf{inv}_c(q)$ for a process q and a counter c. Then for any appropriate ϕ, ψ, ξ, p and v, $\mathsf{safe}(\phi \wedge c_v, z \leftarrow p \parallel q, \psi \mid \xi)$ iff $\mathsf{safe}(\phi \wedge c_v, z \leftarrow \mathsf{ppg}_c(p) \parallel q, \psi \mid \xi)$.*

Proof. By Lemmas 34, 35,

$$\begin{aligned}&\mathsf{safe}(\phi \wedge c_v, z \leftarrow p \parallel q, \psi \mid \xi)\\ &\quad\iff \mathsf{safe}(\phi \wedge c_v, z \leftarrow \mathsf{ppg}_c(v, p \parallel q), \psi \mid \xi)\\ &\quad\iff \mathsf{safe}(\phi \wedge c_v, z \leftarrow \mathsf{ppg}_c(v, \mathsf{ppg}_c(v,p) \parallel q), \psi \mid \xi)\\ &\quad\iff \mathsf{safe}(\phi \wedge c_v, z \leftarrow \mathsf{ppg}_c(v,p) \parallel q, \psi \mid \xi).\end{aligned}$$

and we are done. □

The crucial problem in the verification of concurrent programs is that their semantics presupposes the possibility of running them in an arbitrary environment, whose transitions may enter in between of any two atomic steps of the process. However, an environment itself can often be thought of as a process, which can be specified and run in parallel with the original one. The resulting specification becomes complete in sense that no external steps can enter any more, which is crucial for verification purposes. We illustrate this by giving a simple example. Let P_1 and P_2 be two processes as follows

$$\begin{aligned}P_1 &= \mathsf{if}_\nu\, b\ \mathsf{then}\ [x := x+1]\ \mathsf{else}\ [x := x+2]\\ P_2 &= \mathsf{if}_\nu\, b\ \mathsf{then}\ [x := x+2]\ \mathsf{else}\ [x := x+1]\end{aligned}$$

where $b : P2$. We would like to prove that the parallel composition $P_1 \parallel P_2$ is safe w.r.t. $(x \leq 3)$ at $x = 0$. However, Lemma 31 cannot be applied directly here, as it does not allow identifying the dead trace containing the assignment $x = x + 2$ in both parallel components. (The presence of this trace is a feature, not a bug: any compositional semantics of $P_1 \parallel P_2$ needs to include it, as it might in fact appear when $P_1 \parallel P_2$ is run in parallel with an environment that changes b.)

In order to enable use of Lemma 31 we introduce labels c and d for the two programs, and use Lemma 36 to show that this does not alter the safety of the overall program. That is, we proceed in two steps: (i) first we modify the processes P_1, P_2 to obtain

$$\begin{aligned}P_1' = \mathsf{if}_\nu(&\mathsf{do}\ z \leftarrow b; \mathsf{if}(z \wedge c_0)\ \mathsf{then}\ c{\downarrow}\bar{1}; \mathsf{if}(\neg z \wedge c_0)\ \mathsf{then}\ c{\downarrow}\bar{2}; \mathsf{ret}\, z)\\ &\mathsf{then}\ [\mathsf{if}\, c_1\ \mathsf{then}\ (\mathsf{do}\ x := x+1; c{\downarrow}\bar{3})]\end{aligned}$$

$$\mathsf{else}\ [\mathsf{if}\ c_2\ \mathsf{then}\ (\mathsf{do}\ x := x+2; c{\downarrow}\bar{3})]$$

$$\begin{aligned} P_2' = \mathsf{if}_\nu(&\mathsf{do}\ z \leftarrow b; \mathsf{if}(z \wedge d_0)\ \mathsf{then}\ d{\downarrow}\bar{1}; \mathsf{if}(\neg z \wedge d_0)\ \mathsf{then}\ d{\downarrow}\bar{2}; \mathsf{ret}\ z) \\ &\mathsf{then}\ [\mathsf{if}\ d_1\ \mathsf{then}\ (\mathsf{do}\ x := x+2; d{\downarrow}\bar{3})] \\ &\mathsf{else}\ [\mathsf{if}\ d_2\ \mathsf{then}\ (\mathsf{do}\ x := x+1; d{\downarrow}\bar{3})] \end{aligned}$$

and prove that $\mathsf{safe}(c_0 \wedge d_0 \wedge x = 0, P_1' \parallel P_2', x \leq 3)$ implies $\mathsf{safe}(x = 0, P_1 \parallel P_2, x \leq 3)$ hence reducing the latter claim to the former; and (ii) we prove that $P_1' \parallel P_2'$ satisfies the *invariant*

$$\begin{aligned} \xi = &b \wedge \big((x = 0 \wedge c_{0,1} \wedge d_{0,1}) \vee (x = 1 \wedge d_{0,1} \wedge c_3) \vee (x = 2 \wedge c_{0,1} \wedge d_3)\big) \vee \\ &\neg b \wedge \big((x = 0 \wedge c_{0,2} \wedge d_{0,2}) \vee (x = 2 \wedge d_{0,2} \wedge c_3) \vee (x = 1 \wedge c_{0,2} \wedge d_3)\big) \vee \\ &(x = 3 \wedge c_3 \wedge d_3), \end{aligned}$$

i.e. we have $\{\{\xi\}\}\ P_1' \parallel P_2'\ \{\{\xi \mid \xi\}\}$. Since, obviously, $(c_0 \wedge d_0 \wedge x = 0) \implies \xi \implies (x \leq 3)$, by Lemma 31 this will complete the proof. In order to prove (i) we note some easily provable identities:

$$\begin{aligned} \mathsf{ppg}_c(\bar{0}, P_1') = \mathsf{if}_\nu(&\mathsf{do}\ z \leftarrow b; \mathsf{if}(z \wedge c_0)\ \mathsf{then}\ c{\downarrow}\bar{1}; \mathsf{if}(\neg z \wedge c_0)\ \mathsf{then}\ c{\downarrow}\bar{2}; \mathsf{ret}\ z) \\ &\mathsf{then}\ [\mathsf{do}\ c{\downarrow}\bar{1}; x := x+1; c{\downarrow}\bar{3}]\ \mathsf{else}\ [\mathsf{do}\ c{\downarrow}\bar{2}; x := x+2; c{\downarrow}\bar{3}], \end{aligned}$$

$$\begin{aligned} \mathsf{ppg}_d(\bar{0}, P_2') = \mathsf{if}_\nu(&\mathsf{do}\ z \leftarrow b; \mathsf{if}(z \wedge d_0)\ \mathsf{then}\ d{\downarrow}\bar{1}; \mathsf{if}(\neg z \wedge d_0)\ \mathsf{then}\ d{\downarrow}\bar{2}; \mathsf{ret}\ z) \\ &\mathsf{then}\ [\mathsf{do}\ d{\downarrow}\bar{1}; x := x+2; d{\downarrow}\bar{3}]\ \mathsf{else}\ [\mathsf{do}\ d{\downarrow}\bar{2}; x := x+1; d{\downarrow}\bar{3}]. \end{aligned}$$

Note also that $\mathsf{inv}_c(P_2')$ and $\mathsf{inv}_d(\mathsf{ppg}_c(\bar{0}, P_1'))$. Therefore, by two applications of Lemma 36 we obtain

$$\mathsf{safe}(c_0 \wedge d_0 \wedge x = 0, P_1' \parallel P_2', x \leq 3) \iff \mathsf{safe}(c_0 \wedge d_0 \wedge x = 0, \mathsf{ppg}_c(P_1') \parallel \mathsf{ppg}_d(P_2'), x \leq 3).$$

Note that $\mathsf{ppg}_c(\bar{0}, P_1')$ and $\mathsf{ppg}_d(\bar{0}, P_1')$ treat c and d in the style of auxiliary variables as already in [35]: the only statements that depend on the values of c and d are write operations to c and d. It is thus easy to see that all the commands writing to c and d can be removed from $\mathsf{ppg}_c(\bar{0}, P_1')$ and $\mathsf{ppg}_d(\bar{0}, P_1')$ without affecting the safety properties (as these do not mention c and d). Hence

$$\begin{aligned} \mathsf{safe}(c_0 \wedge d_0 \wedge x = 0, P_1' \parallel P_2', x \leq 3) &\iff \mathsf{safe}(c_0 \wedge d_0 \wedge x = 0, P_1 \parallel P_2, x \leq 3) \\ &\implies \mathsf{safe}(x = 0, P_1 \parallel P_2, x \leq 3). \end{aligned}$$

Let us proceed with the proof of (ii). By rule **(par)** it suffices to show $\{\{\xi\}\}\ P_1\ \{\{\xi \mid \xi\}\}$ and $\{\{\xi\}\}\ P_2\ \{\{\xi \mid \xi\}\}$. We prove the former claim; the latter is completely analogous. By **(if)** and **(atom)**, we arrive at three Hoare triples

$$\begin{aligned} &\{\xi\}\ \ \mathsf{do}\ z \leftarrow b; \mathsf{if}(z \wedge c_0)\ \mathsf{then}\ c{\downarrow}\bar{1}; \mathsf{if}(\neg z \wedge c_0)\ \mathsf{then}\ c{\downarrow}\bar{2}; \mathsf{ret}\ z\ \ \{\xi\} \\ &\{\xi\}\ \ \mathsf{if}\ c_1\ \mathsf{then}\ (\mathsf{do}\ x := x+1; c{\downarrow}\bar{3})\ \ \{\xi\} \\ &\{\xi\}\ \ \mathsf{if}\ c_2\ \mathsf{then}\ (\mathsf{do}\ x := x+2; c{\downarrow}\bar{3})\ \ \{\xi\}, \end{aligned}$$

which can be verified routinely.

In [15] we indicated how Dekker's mutual exclusion algorithm can be verified in our framework. The invariant given in the conference version is not entirely correct, as it fails to mention labels; using labels, the safety proof in our framework essentially follows the proof in [30].

8. Conclusions and further work

We have studied asynchronous concurrency in a framework of generic effects. To this end, we have combined the theories of computational monads and final coalgebras to obtain a framework for processes with generic side-effecting steps, the *corecursive meta-language* ME_ν. We have presented a sound and complete equational

calculus for ME_ν, and we have obtained a syntactic corecursion scheme in which corecursive functions are syntactically reducible to a basic loop construct. Within this calculus, we have given generic definitions for standard imperative constructs and a number of standard process operators, most notably parallel composition. We have moreover presented initial results on building verification frameworks on top of this equational calculus, specifically an invariant-based method for safety proofs whose application moreover relies on systematically equipping programs with labels.

In future research, we intend to develop more expressive verification logics for side-effecting processes, detached from equational reasoning. An interesting perspective in this direction is to identify a variant of the rely/guarantee principle for side-effecting processes (cf. e.g. [50]). A further topic of investigation is to develop weak notions of process equivalence in our framework, such as testing equivalence [34]. Moreover, we want to extend the verification framework to cover also *liveness* properties, having concentrated on safety in the present work.

Another direction for future work is to extend the expressivity of the underlying programming language, with notable features to be covered including synchronization, local store, and fresh name creation. We envisage that the former may be modelled by means of extra structure on the monad as hinted at in the introduction, while the latter features would typically be handled by a judicious choice of base category (such as presheaves [38] and nominal sets [45])

Finally, the decidability status of ME_ν remains open. Note that in case of a positive answer, all equations between functions defined by corecursion schemes, e.g. process algebra identities, become decidable. While experience suggests that even very simple calculi that combine loop constructs with monadic effects tend to be undecidable, the corecursion axiom as a potential source of trouble seems rather modest, and no evident encoding of an undecidable problem appears to be directly applicable.

References

[1] M. Barr and C. Wells. *Toposes, Triples and Theories*, vol. 278 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1985.

[2] F. Bartels. Generalised coinduction. *Math. Struct. Comput. Sci.*, 13:321–348, 2003.

[3] N. Benton and M. Hyland. Traced premonoidal categories. *ITA*, 37:273–299, 2003.

[4] J. A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In *Automata, Languages and Programming, ICALP 1984*, vol. 172 of *LNCS*, pp. 82–94. Springer, 1984.

[5] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1, 2005.

[6] P. Cenciarelli. *Computational applications of calculi based on monads*. PhD thesis, University of Edinburgh, 1996.

[7] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. Technical report, Category Theory and Computer Science, 1993.

[8] K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9:313–323, 1999.

[9] J. R. B. Cockett. Introduction to distributive categories. *Mathematical Structures in Computer Science*, 3:277–307, 1993.

[10] R. L. Crole. *Categories for Types*. Cambridge Mathematical Textbooks. Cambridge, 1994.

[11] R. L. Crole and A. M. Pitts. New foundations for fixpoint computations. In *Logic in Computer Science, LICS 1990*, pp. 489–497. IEEE, 1990.

[12] L. Erkök and J. Launchbury. Recursive monadic bindings. In *ICFP'00*, pp. 174–185. ACM, 2000.

[13] A. Filinski. On the relations between monadic semantics. *Theor. Comp. Sci.*, 375:41–75, 2007.

[14] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. *Inf. Comput.*, 179:76–117, 2002.

[15] S. Goncharov and L. Schröder. A coinductive calculus for asynchronous side-effecting processes. In O. Owe, M. Steffen, and J. A. Telle, editors, *Fundamentals of Computation Theory, FCT 2011*, vol. 6914 of *LNCS*. Springer, 2011.

[16] S. Goncharov, L. Schröder, and T. Mossakowski. Kleene monads: handling iteration in a framework of generic effects. In *CALCO'09*, pp. 18–33, 2009.

[17] P. Hancock and A. Setzer. Guarded induction and weakly final coalgebras in dependent type theory. In *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pp. 115 – 134, 2005.

[18] W. Harrison, A. Procter, J. Agron, G. Kimmell, and G. Allwein. Model-driven engineering from modular monadic semantics: Implementation techniques targeting hardware and software. In W. M. Taha, editor, *Domain-Specific Languages, DSL 2009*, vol. 5658 of *LNCS*, pp. 20–44. Springer, 2009.

[19] W. L. Harrison. The essence of multitasking. In *AMAST'06*, vol. 4019 of *LNCS*, pp. 158–172. Springer, 2006.

[20] W. L. Harrison and J. Hook. Achieving information flow security through monadic control of effects. *J. Computer Security*, 17:599–653, 2009.

[21] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *MFCS*, pp. 108–120, 1979.

[22] M. Hyland, P. B. Levy, G. Plotkin, and J. Power. Combining algebraic effects with continuations. *Theoretical Computer Science*, 375:20 – 40, 2007. Festschrift for John C. Reynolds's 70th birthday.

[23] M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 2003.
[24] B. Jacobs. From coalgebraic to monoidal traces. *Electron. Notes Theor. Comput. Sci.*, 264:125–140, 2010.
[25] B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoret. Comput. Sci.*, 291:329–349, 2003.
[26] M. Jaskelioff. Modular monad transformers. In G. Castagna, editor, *European Symposium on Programming Languages and Systems, ESOP 2009*, vol. 5502 of *LNCS*, pp. 64–79. Springer, 2009.
[27] A. Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23:113–120, 1972.
[28] S. Krstic, J. Launchbury, and D. Pavlovic. Categories of processes enriched in final coalgebras. In *FOSSACS'01*, vol. 2030 of *LNCS*, pp. 303–317. Springer, 2001.
[29] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995.
[30] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995.
[31] S. Milius, T. Palm, and D. Schwencke. Complete iterativity for algebras with effects. In *CALCO'09*, pp. 34–48. Springer-Verlag, 2009.
[32] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.
[33] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, 1991.
[34] R. D. Nicola and M. Hennessy. Testing equivalence for processes. In *International Colloquium on Automata, Languages and Programming, ICALP 1983*, vol. 154 of *LNCS*, pp. 548–560. Springer, 1983.
[35] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. 10.1007/BF00268134.
[36] N. Papaspyrou and D. Macos. A study of evaluation order semantics in expressions with side effects. *J. Funct. Program.*, 10:227–244, 2000.
[37] S. Peyton Jones, editor. *Haskell 98 Language and Libraries — The Revised Report.* Cambridge, 2003. also: *J. Funct. Programming* 13, 2003.
[38] G. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS 2002, Lecture Notes in Computer Science 2303*, pp. 342–356. Springer, 2002.
[39] G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:2003, 2003.
[40] K. I. Rosenthal. *Quantales and their applications.* Pitman Research Notes in Mathematics Series. Longman Scientific & Technical, 1990.
[41] J. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.*, 249:3–80, 2000.
[42] L. Schröder and T. Mossakowski. Generic exception handling and the Java monad. In *Algebraic Methodology and Software Technology*, vol. 3116 of *Lecture Notes in Computer Science*, pp. 443–459. Springer Berlin, 2004.
[43] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HasCASL. *J. Logic Comput.*, 14:571–619, 2004.
[44] L. Schröder and T. Mossakowski. HasCASL: Integrated higher-order specification and program development. *Theor. Comput. Sci.*, 410:1217–1260, 2009.
[45] I. Stark. Free-algebra models for the π-calculus. *Theoretical Computer Science*, 390(2-3):248–270, 2008.
[46] D. Syme, A. Granicz, and A. Cisternino. *Expert F#.* Apress, 2007.
[47] A. P. Tolmach and S. Antoy. A monadic semantics for core Curry. In *WFLP'03*, vol. 86(3) of *ENTCS*, pp. 16–34, 2003.
[48] T. Uustalu. Generalizing substitution. *ITA*, 37:315–336, 2003.
[49] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29:240–263, 1997.
[50] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9:149–174, 1997.