Inductive and Coinductive Predicate Liftings for Effectful Programs

Niccolò Veltri * Niels F.W. Voorneveld *
Tallinn University of Technology
Tallinn, Estonia
niccolo@cs.ioc.ee niels@cs.ioc.ee

We formulate a framework for describing behaviour of effectful higher-order recursive programs. Examples of effects are implemented using effect operations, and include: execution cost, nondeterminism, global store and interaction with a user. The denotational semantics of a program is given by a coinductive tree in a monad, which combines potential return values of the program in terms of effect operations.

Using a simple test logic, we construct two sorts of predicate liftings, which lift predicates on a result type to predicates on computations that produce results of that type, each capturing a facet of program behaviour. Firstly, we study inductive predicate liftings which can be used to describe effectful notions of total correctness. Secondly, we study coinductive predicate liftings, which describe effectful notions of partial correctness. The two constructions are dual in the sense that one can be used to disprove the other.

The predicate liftings are used as a basis for an endogenous logic of behavioural properties for higher-order programs. The program logic has a derivable notion of negation, arising from the duality of the two sorts of predicate liftings, and it generates a program equivalence which subsumes a notion of bisimilarity. Appropriate definitions of inductive and coinductive predicate liftings are given for a multitude of effect examples.

The whole development has been fully formalized in the Agda proof assistant.

1 Introduction

Programs may exhibit different behaviour depending on various circumstances. The environment can induce an effect upon program evaluation in many ways, e.g. nondeterministic decision making, access to some global store, or evaluation timeouts. Such effects can be represented as *algebraic effect operations* [24], which trigger an effect and capture all possible continuations.

General recursive computations with effect triggering operations can be seen as generating evaluation traces in the form of *coinductive trees*. Such trees have as leaves the potential results a computation can return, and as nodes effect operations which branch into each of the possible continuations. Due to the coinductive nature of these trees, they can be of infinite height, denoting potentially diverging computations.

In practice, when a program is evaluated, each of the effect operations is handled by choices made by the environment. An external observer, e.g. a user of a program, cannot directly inspect the generated tree. However, this spectator may make certain *observations* capturing effectful behaviour [11]. Such observations include: possible termination, evaluation within a time limit, and how a program alters a global state.

*The authors were supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001) and the Estonian Research Council grant PSG659.

Ana Sokolova (Eds.): MFPS 2021
EPTCS 351, 2021, pp. 260–277, doi:10.4204/EPTCS.351.16

We capture such observations with a collection of predicate liftings [27, 18]. These consist of a set of tokens denoting each possible observation, and for each token a device that lifts a predicate on return values to a predicate on coinductive trees over these values. Using a simple test logic, these predicate liftings are described "locally" in terms of effect operations. The local definitions generate an inductive predicate lifting, which captures a notion of effectful termination, and is used to specify total correctness with respect to the appropriate observation. The predicate liftings are similar to the ones derived from ordered monads by Hasuo [9].

Dually, we also generate coinductive predicate liftings. These allow computations to satisfy observations either immediately, or by postponing the burden of proof, potentially indefinitely. As such, they capture a notion of effectful divergence, and can be used to specify partial correctness with respect to effect observations.

A main contribution of this paper is formulating the coinductive predicate liftings in such a way that they can be used as a constructive notion of negation for the inductive predicates. As a result, these coinductive predicate liftings can be used to disprove inductive properties, and vice versa. This complementing pair hence helps us recover in our constructive setting some of the expressive power available when reasoning classically as in [27].

From the inductive and coinductive predicate liftings, we generate a logic for higher-order programs, extending the logic from [27] with coinductive predicate liftings (modalities). This logic captures behavioural properties (observations) for program denotations, and is *endogenous* in the sense of Pnueli [26]. It is possible to build nested statements, mixing inductive and coinductive properties. The logic gives rise to a notion of program equivalence on program denotations, which subsumes a notion of applicative bisimilarity in the sense of [3, 12].

We use predicate liftings and the resulting logic to study programs exhibiting various different effects, including: evaluation cost, nondeterminism, global store, and user input. The adaptive nature of the generic formulations allows for many more examples to be implemented. As such, we believe it gives a foundation for the verification of effectful programs in many situations.

In particular, we study the preservation of the logical program equivalence over certain program transformations. One such transformation prevalent for higher-order programs is that of *sequencing*, as used e.g. by let-binding, program composition and the uncurrying operation. We show that, for our effect examples, a property for a sequenced program can be *decomposed* into (or pulled back to) a property for unsequenced programs. This adapts the notion of *decomposability* featured in [27], and gives us a variety of proof techniques for the verification of higher order programs.

The development in this paper has been fully formalized in the Agda proof assistant. The code is freely accessible at: `https://github.com/niccoloveltri/ind-coind-pred-lifts`. It uses Agda 2.6.1 with standard library 1.6-dev. One consequence of this formalization is that all the behavioural properties described by our predicate liftings and logic are fully *constructive*. Programming languages that can be deeply embedded in Agda can be studied using this logic too. For instance, we have an implementation of an effectful version of fine-grained call-by-value PCF [23, 16] with its own variation on the logic similar to the one used in [27].

Basic Type-Theoretic Definitions and Notation. Our work takes place in Martin-Löf type theory extended with inductive and coinductive types, and practically in the Agda proof assistant. We use Agda notation for dependent function types: $(x : A) \to B\,x$. We write 0 for the empty type and 1 for the unit type with unique inhabitant $\mathtt{tt} : 1$. We use 2 for the type of Booleans with two inhabitants $\mathsf{true}, \mathsf{false} : 2$. We write $\{x\}$ as a synonym for 1 when we want to use a specific name x instead of tt. Similarly, we write $\{x, y\}$ for the synonym of 2 where true and false are replaced by x and y. We use A^* for the type of finite lists of elements of type A. Propositional equality is $\equiv$ and judgemental equality is $=$ (as in Agda).

Types are stratified in a cumulative hierarchy of universes Set_k. The first universe is simply denoted Set and when we write statements like "X is a type", we mean $X : \mathsf{Set}_k$ for some universe level k. For readability reasons, in the paper all mentions of universe levels have been removed. We often employ the proposition-as-types perspective, and write statements like "X implies Y", which formally expresses the existence of a function from type X to type Y, or "X holds", meaning that there exists an element of type X.

2 Programs as Trees

Assume given a type K and a type family $I : K \to \mathsf{Set}$. The pair (K, I) is frequently called a *container* [1], and it is used as a *signature* [19] to specify branching of trees in type theory.

Definition 1. The type of *coinductive trees* is generated by two constructors:

$$\frac{x : A}{\mathsf{leaf}\ x : \mathsf{Tree}\ I\ A} \qquad \frac{\begin{array}{c}k : K \quad ts : I\ k \to \mathsf{Tree}\ I\ A\end{array}}{\mathsf{node}\ k\ ts : \mathsf{Tree}\ I\ A} \tag{1}$$

A coinductive tree $t : \mathsf{Tree}\ I\ A$ can either be of the form $\mathsf{leaf}\ x$, for x a value of type A, or $\mathsf{node}\ k\ ts$, where ts represents the immediate subtrees of t. The branching is specified by the type $I\ k$. Following Leroy and Grall's notation [14], we use the double line in the inference rule of node to indicate that this is a coinductive constructor, and as such it can be iterated an infinite amount of times. For example, assuming $\mathsf{sk} : K$ with $I\ \mathsf{sk} = 1$, the corecursive definition of the infinite tree $\mathsf{diverge} : \mathsf{Tree}\ I\ A$ with exactly one branch at each level goes as follows: $\mathsf{diverge} = \mathsf{node}\ \mathsf{sk}\ (\lambda x.\ \mathsf{diverge})$.

In Agda, coinductive types are encoded as *coinductive records* which can optionally be parametrized by a *size* to ease the productivity checking of corecursive definitions [2, 8]. For example, the type of trees given above is implemented in Agda as the following pair of mutually defined types:

```
data Tree (I : K → Set) (A : Set) (i : Size) : Set where
  leaf : (x : A) → Tree I A i
  node : (k : K) (ts : I k → Tree′ I A i) → Tree I A i
```

```
record Tree′ (I : K → Set) (A : Set) (i : Size) : Set where
  coinductive
  field
    force : {j : Size< i} → Tree I A j
```

(2)

The type $\mathsf{Tree}\ I\ A\ i$ is inductive, while $\mathsf{Tree}'\ I\ A\ i$ is a coinductive record type. The inductive constructors leaf and node are similar to the inference rules in (1), but now the subtrees ts in $\mathsf{node}\ k\ ts$ have return type $\mathsf{Tree}'\ I\ A\ i$. Elements of the latter type are like thunked computations that can be resumed by feeding a token, in the form of a size j smaller than i, to the destructor force. The resumed subtree inhabits the type $\mathsf{Tree}\ I\ A\ j$. The decrease in size can be explained by viewing sizes as abstract ordinals representing the number of unfoldings that a coinductive definitions can undergo. Sizes come with a top element $\infty : \mathsf{Size}$, and a fully formed coinductive tree t has type $\mathsf{Tree}\ I\ A\ \infty$. This means that a user has an infinite number of tokens that she can spend for accessing arbitrarily deep subtrees of t. The presence of sizes in the types of coinductive records is crucial for ensuring productivity of corecursively defined functions, which would otherwise need to pass Agda's strict syntactic productivity check.

In this paper, we opt to work with an informal treatment of coinductive types, specified by constructors as in (1). Accordingly, corecursive functions are given as usual in terms of recursively defined functions. In particular, all mentions to sizes in types have been dropped, but the interested reader can recover them in our Agda formalization.

The $\mathsf{Tree}\ I$ datatype is a functor, and we call mapTree its action on functions. It is also a monad, with unit $\eta = \mathsf{leaf}$ and multiplication $\mu : \mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ A) \to \mathsf{Tree}\ I\ A$ corecursively defined as:

$$\mu\ (\mathsf{leaf}\ x) = x, \quad \mu\ (\mathsf{node}\ k\ ts) = \mathsf{node}\ k\ (\lambda x.\ \mu\ (ts\ x)).$$

In our programs-as-trees perspective, elements in a leaf position represent return values of computations, while each node in a tree denotes the presence of an effect operation. K is the type of admissible operations and, for each $k : K$, $I\,k$ is a type corresponding to the arity of operation k. The existence of a *skip* operation $\mathsf{sk} : K$ with $I\,\mathsf{sk} = 1$ enables the encoding of possibly non-terminating computations, as done in [6], like the diverging program diverge introduced previously in this section. If K is equivalent to the unit type and sk is its unique inhabitant, then $\mathsf{Tree}\,I\,A$ corresponds to the type of deterministic possibly non-terminating programs returning values of type A after some number of steps.

3 Observations

As discussed in the last paragraph of the previous section, a user of a program represented by a tree in $\mathsf{Tree}\,I\,A$, with only one admissible skip operation $\mathsf{sk} : K$, is able to observe the terminating and non-terminating behaviour of the program and its computation time. A user of a program represented by a nondeterministic binary tree may observe that some of the possible return values in the tree satisfy a certain property, or that none of them do. In the case of computer programs exhibiting a variety of different computational effects, the user can observe other relevant behaviour and therefore make appropriate queries to the system.

The collection of admissible queries on effectful programs in $\mathsf{Tree}\,I\,A$, capturing observations of effectful behaviour, is given by a particular class of tree predicates, generated by a predicate tree lifting. We introduce an inductive grammar of logical statements, whose elements we call *tests*, to define these predicate liftings.

$$\frac{a : A}{\mathsf{atom}\,a : \mathsf{Test}\,A} \qquad \frac{}{\mathsf{True} : \mathsf{Test}\,A} \qquad \frac{}{\mathsf{False} : \mathsf{Test}\,A}$$

$$\frac{t,u : \mathsf{Test}\,A}{t \wedge u : \mathsf{Test}\,A} \qquad \frac{t,u : \mathsf{Test}\,A}{t \vee u : \mathsf{Test}\,A} \qquad \frac{t : \mathbb{N} \to \mathsf{Test}\,A}{\bigwedge t : \mathsf{Test}\,A} \qquad \frac{t : \mathbb{N} \to \mathsf{Test}\,A}{\bigvee t : \mathsf{Test}\,A}$$

Elements of $\mathsf{Test}\,A$ are interpreted as types via the lifting $\overline{\mathsf{Test}} : (A \to \mathsf{Set}) \to \mathsf{Test}\,A \to \mathsf{Set}$. Atoms, i.e. tests of the form $\mathsf{atom}\,a$, are modelled as $P\,a$ where $P : A \to \mathsf{Set}$ is the input predicate. The other constructors of $\mathsf{Test}\,A$ are modelled via Curry-Howard correspondence. E.g. conjunction of tests t and u is interpreted as the Cartesian product of the interpretations of t and u, and similarly for all the other logical operations. The type former Test is a functor, we call $\mathsf{mapTest}$ its action on functions.

Observations on coinductive trees are formulated using a threefold specification:

1. A type O of tokens, each denoting a particular observation or test on an effectful program.
2. A decidable subset $\pi_{\mathsf{leaf}} : O \to 2$ specifying which observations are satisfied by the production of a successful result. This is called the *leaf function* for O.
3. A function $\pi_{\mathsf{node}} : (k : K) \to O \to \mathsf{Test}\,(I\,k \times O)$ formulating when a tree with a root node satisfies an observation. This is called the *node function* for O. Note that the constructors of the grammar Test allow for both parallel and sequenced testing.

To each observation $o : O$ and tree $t : \mathsf{Tree}\,I\,\mathsf{Set}$, which we think of as obtained from a tree in $\mathsf{Tree}\,I\,A$ after an application of a predicate $P : A \to \mathsf{Set}$ to its leaves, we wish to associate a type $\alpha\;o\;t : \mathsf{Set}$ corresponding to the *handling* of t [25]. The result should be a particular logical combination of the truth values from t dictated by the input observation o. This is done in the following definition:

Definition 2. Given $\pi_{\mathsf{leaf}} : O \to 2$ and $\pi_{\mathsf{node}} : (l : K) \to O \to \mathsf{Test}\,I\,k \times O$, we define the O-indexed algebra $\alpha : O \to \mathsf{Tree}\,I\,\mathsf{Set} \to \mathsf{Set}$ inductively as follows:

$$\frac{b : \pi_{\mathsf{leaf}}\ o \equiv \mathsf{true} \qquad p : P}{\mathsf{leaf}_\alpha\ b\ p : \alpha\ o\ (\mathsf{leaf}\ P)} \qquad \frac{ps : \overline{\mathsf{Test}}\ (\lambda(x, o').\ \alpha\ o'\ (Ps\ x))\ (\pi_{\mathsf{node}}\ k\ o)}{\mathsf{node}_\alpha\ ps : \alpha\ o\ (\mathsf{node}\ k\ Ps)}$$

The satisfiability of the predicate $\alpha\ o$ by a tree $t : \mathsf{Tree}\ I\ \mathsf{Set}$ depends on the shape of t. If $t = \mathsf{leaf}\ P$, for some $P : \mathsf{Set}$, then it satisfies $\alpha\ o$ in case P holds and the token o is correct according to the leaf function π_{leaf}. If $t = \mathsf{node}\ k\ Ps$, for some $k : K$ and $Ps : I\ k \to \mathsf{Tree}\ I\ \mathsf{Set}$, then the test $\pi_{\mathsf{node}}\ k\ o$ specifies a combination of continuations x and new observations o'. If subtrees $Ps\ x$ satisfy $\alpha\ o'$, according to the logical formula corresponding to $\pi_{\mathsf{node}}\ k\ o$, then also t satisfies $\alpha\ o$. The use of the inductively-defined grammar of logical statements Test and its interpretation in types $\overline{\mathsf{Test}}$ in the premise of node_α guarantees that α is a well-defined inductive type family and passes Agda strict positivity check.

Definition 3. The *inductive O-indexed predicate lifting* of a tree given by α is defined as

$$(-)^\alpha\ :\ O \to (A \to \mathsf{Set}) \to \mathsf{Tree}\ I\ A \to \mathsf{Set}, \qquad o^\alpha\ P\ t = \alpha\ o\ (\mathsf{mapTree}\ P\ t).$$

Here we see the predicate P as some observable property on values of type A. For each observation o, o^α lifts this property to an extensionally observable property on computations of type A.

For each effect, an O-indexed predicate lifting is specified according to two design principles:

- All predicate liftings in the family capture a notion of observation which is extensionally observable by a user of the program.
- Together, the predicate liftings are powerful enough to express differences between programs that might be detectable after certain canonical program transformations.

Later in this paper, we will formulate a notion of behavioural equivalence specified by this family of observations. The second design principle expresses the desire that this behavioural equivalence is preserved by certain program transformations. One such transformation of particular interest to us is that of program sequencing.

Example 4. We start with purely deterministic programs, represented by trees in $\mathsf{Tree}\ I\ A$ where $K = \{\mathsf{sk}\}$ and $I\ \mathsf{sk} = 1$; that is, trees have only one node label sk which has only one continuation. The type $\mathsf{Tree}\ I\ A$ is therefore equivalent to Capretta's delay monad applied to A [6]. The label sk corresponds to a deterministic program step, which may or may not be observable by the user of the program. A family of observations is chosen depending on whether the skip is observable.

If skips are unobservable, we can only observe *termination*. We take $O = \{\downarrow\}$ and define $\pi_{\mathsf{leaf}}\ \downarrow = \mathsf{true}$, meaning that we observe $\downarrow$ when the computation terminates, and $\pi_{\mathsf{node}}\ \mathsf{sk}\ \downarrow = \mathsf{atom}\ (\mathsf{tt}, \downarrow)$, meaning that we keep observing $\downarrow$ on the continuation of a not-yet-terminated computation. In this case, $\downarrow^\alpha\ P\ t$ is provable when t eventually produces a leaf $a : A$ such that $P\ a$ holds.

Alternatively, we may consider sk to be observable, for instance as a measure of evaluation time. We use as observations $O = \mathbb{N}$ expressing time limits on termination. We take $\pi_{\mathsf{leaf}}\ n = \mathsf{true}$, $\pi_{\mathsf{node}}\ \mathsf{sk}\ 0 = \mathsf{false}$ and $\pi_{\mathsf{node}}\ \mathsf{sk}\ (n+1) = \mathsf{atom}\ (\mathsf{tt}, n)$, making n^α observe termination within at most n skips.

Example 5. Consider an unpredictable scheduler making binary decisions for a program. Such computations are denoted by *binary decision trees*, i.e. coinductive trees in $\mathsf{Tree}\ I\ A$ over signature $K = \{\mathsf{or}\}$, $I\ \mathsf{or} = \{\mathsf{left}, \mathsf{right}\}$, whose nodes display choices given by the or operation. Due to the unpredictability of the decisions made, we interpret these choices as being resolved *nondeterministically*. Nondeterminism in functional languages has been thoroughly studied by Ong [20] and Lassen [13] and many others.

Users of nondeterministic programs cannot observe decision paths. They may at most observe what *may* be possible, and what *must* happen. As such, we formulate two observations $O = \{\mathsf{may}, \mathsf{must}\}$, with accompanying functions $\pi_{\mathsf{leaf}}\ x = \mathsf{true}$, $\pi_{\mathsf{node}}\ \mathsf{or}\ \mathsf{may} = \mathsf{atom}\ (\mathsf{left}, \mathsf{may}) \vee \mathsf{atom}\ (\mathsf{right}, \mathsf{may})$ and

$\pi_{\mathsf{node}}\ \mathsf{or}\ \mathsf{must} = \mathsf{atom}\ (\mathsf{left}, \mathsf{must}) \wedge \mathsf{atom}\ (\mathsf{right}, \mathsf{must})$. In this case, $\mathsf{may}^{\alpha}\ P\ t$ holds if there is some sequence of resolutions of choices for which t produces a result satisfying P. On the other hand, $\mathsf{must}^{\alpha}\ P\ t$ holds if t is guaranteed to produce a result satisfying P, no matter the decisions made.

Example 6. Suppose programs can consult some global store containing a natural number, using a $\mathsf{lookup} : K$ and an update operation $\mathsf{update} : \mathbb{N} \to K$. The lookup operation is countably branching, with $I\ \mathsf{lookup} = \mathbb{N}$, and continues according to the current state. For any $n : \mathbb{N}$, $\mathsf{update}\ n$ stores n in the global store, and continues in a unique way, defining $I\ (\mathsf{update}\ n) = 1$. Computations in $\mathsf{Tree}\ I\ A$ express communication patterns between a program and a global store, potentially yielding a result of type A.

We suppose that both the state before the execution of a program and the state after the execution of a program are observable. As observations, we use pairs $O = \mathbb{N} \times \mathbb{N}$ of initial and final states. With $(n,m)^{\alpha}\ P\ t$ we aim to express that, when program t is evaluated with starting state n, it produces a result satisfying P with the final state equal to m. To this end, we define:

- $\pi_{\mathsf{leaf}}\ (n,n) = \mathsf{true}$, but $\pi_{\mathsf{leaf}}\ (n,m) = \mathsf{false}$ if $n \neq m$, since termination does not change the state,
- $\pi_{\mathsf{node}}\ (\mathsf{update}\ k)\ (n,m) = \mathsf{atom}\ (\mathsf{tt}, (k,m))$, and $\pi_{\mathsf{node}}\ \mathsf{lookup}\ (n,m) = \mathsf{atom}\ (n, (n,m))$.

Example 7. In this last example, we consider computations which repeatedly ask input bits from a user. We have one operation $\mathsf{input} : K$ of arity $I\ \mathsf{input} = \{\mathsf{left}, \mathsf{right}\}$. The user chooses inputs for the program, and we specify two sorts of observations. For any list of bits $l \in 2^*$ of length n, the user can check:

- whether the computation terminates after the sequence of bits l is entered;
- whether after entering the sequence of bits l, the computation requests another input.

Note that the user can only input the sequence one by one, and only if the computation keeps requesting more inputs until the bitlist is exhausted. Observations $O = 2 \times 2^*$ consist of a bit denoting which of the two tests are performed and the list of bits used in the test. We implement the observations as follows:

$$\pi_{\mathsf{leaf}}\ (b,l) = (b \equiv \mathsf{left}) \wedge (l \equiv \langle\rangle), \qquad \pi_{\mathsf{node}}\ \mathsf{input}\ (b,l) = \begin{cases} b \equiv \mathsf{right} & \text{if } l \equiv \langle\rangle \\ \mathsf{atom}\ (\mathsf{head}\ l, (b, \mathsf{tail}\ l)) & \text{otherwise.} \end{cases}$$

For any observation o in the examples above, the predicate $o^{\alpha}\ P$ expresses some effectful variation on termination checking, such as termination within some time limit, decision invariant termination, and termination with a certain final state. The particular relevant notion of termination is dependent on our interpretation of the effect. We say that $o^{\alpha}\ P\ t$ captures a notion of *effectful total correctness* according to observation o.

The inductive predicates generated by α are however unable to 'detect' the dual to termination: divergence. From the user's perspective, divergence is mainly the absence of termination, the indefinite continuation of a program. It is not possible to extensionally confirm whether a program diverges. For some programs however, like in the given example diverge, we can prove intensionally that a program will go on forever. Following this example's lead, we will express *provable effectful divergence* next, using coinductive predicate liftings.

4 Co-observations

To prove correctness of a program, we either show that it terminates and produces a correct result, or we show it requires the resolution of some (effect) operation, in which case we may postpone the burden of proof to after this operation has been resolved. In total correctness, we must verify that the program will eventually terminate. In partial correctness however, we can get away with postponing the burden

of proof indefinitely. In that case, if the program diverges, it is still considered correct, since it will never return an incorrect result.

We implement this notion of partial correctness via *coinductive predicate liftings*, a dual to the inductive predicate liftings from last section. This additionally provides a technique for disproving inductive properties. For example, to disprove that a program produces an even number, we can either show that the program produces an odd number, or that the program diverges. In other words, if it is partially incorrect that a program produces an even number, then we can conclude that it is not totally correct that it produces an even number. Using coinduction, we set up this notion of partial correctness in a constructive way, with the additional motivation to use it as a constructive notion of 'negation'. However, due to decidability issues, this will never truly be a perfect negation, only a generically large constructive complement.

Definition 8. Given $\pi_{\mathsf{leaf}} : O \to 2$ and $\pi_{\mathsf{node}} : (l : K) \to O \to \mathsf{Test}\ I\ k \times O$, we define the O-indexed algebra $\beta : O \to \mathsf{Tree}\ I\ \mathsf{Set} \to \mathsf{Set}$ *coinductively*, using the following judgments:

$$\frac{p : P}{\mathsf{leaf}_\beta\ p : \beta\ o\ (\mathsf{leaf}\ P)} \qquad \frac{b : \pi_{\mathsf{leaf}}\ o \equiv \mathsf{false}}{\mathsf{exep}_\beta\ b : \beta\ o\ (\mathsf{leaf}\ P)} \qquad \frac{\overline{ps : \overline{\mathsf{Test}}\ (\lambda(x,o').\ \beta\ o'\ (Ps\ x))\ (\pi_{\mathsf{node}}\ k\ o)}}{\mathsf{node}_\beta\ ps : \beta\ o\ (\mathsf{node}\ k\ Ps)}$$

When a program returning values of type Set terminates immediately, there are two possible ways of satisfying $\beta\ o$: either the program returns an inhabited type (as in the constructor leaf_β) or o is incorrect according to π_{leaf} (as in the constructor exep_β). Conceptually, exep_β states that we do not need to verify correctness of the result if we are not currently verifying an observation which we consider 'correct for termination'.

For instance, in Example 6 of global store, $\beta\ (0,1)$ expresses the following partial correctness property: if with starting state 0 the program terminates with final state 1, then it produces an inhabited type. This statement could be satisfied with an exception "exep_β", which entails that the program terminates with a final state we are not currently checking for. In that case, the property is satisfied since the condition we are testing for at termination is not met. Hence $\beta\ (0,1)\ (\mathsf{leaf}\ P)$ is inhabited.

The double line in the node_β constructor reflects the coinductive nature of the predicate lifting. In the Agda implementation, β is formulated using mutual induction-coinduction, similarly to the encoding of coinductive trees in (2). In particular, sizes appear as extra arguments to ensure productivity of corecursive definitions. From $\mathsf{node}_\beta\ ps$ we can extract a proof in terms of proofs on the continuation Ps. Since node_β is a coinductive constructor, it is possible for a proof of $\beta\ o\ t$ to contain an infinite amount of such extractions. Hence, proofs of β can be self-referential, and can refer to infinitely many nodes in the tree.

Definition 9. The *coinductive O-indexed predicate lifting* of a tree given by β is defined as:

$$(-)^\beta : O \to (A \to \mathsf{Set}) \to \mathsf{Tree}\ I\ A \to \mathsf{Set}, \qquad o^\beta\ P\ t = \beta\ o\ (\mathsf{mapTree}\ P\ t).$$

The coinductive O-indexed predicate lifting specify notions of partial correctness. In an effect-free language, this amounts to saying that, if the program terminates, it produces a correct result. For effects, it captures more general notions of partial correctness, checking correctness of a result only at termination and only under certain circumstances. These circumstances are given by observations $o : O$ for which $\pi_{\mathsf{leaf}}\ o \equiv \mathsf{true}$.

Partial correctness is useful both in cases in which termination can be checked independently, or when one is more interested in the safety of results at the expense of possible divergence. Considering non-terminating programs as at least not dangerous, gives more freedom when trying to design safe programs, and the coinductive predicate liftings give tools for the verification of such programs in many different effectful situations.

4.1 Coinduction as Negation

As mentioned before, another main use for coinduction is as a notion of negation for inductive properties. As such, it also has uses when studying and reasoning about inductive predicates. But coinduction cannot always directly be used this way. We need to modify its formulation slightly.

In this paper, we introduce effectful versions of termination and divergence, which result in effectful versions of total and partial correctness complementing each other. Considering that effect observations are formulated using tests, in order to properly function as a notion of constructive complement, we need to formulate the complement or *dual* of a test, called $\mathsf{dualTest} : \mathsf{Test}\, A \to \mathsf{Test}\, A$.

$$\begin{aligned}
\mathsf{dualTest}\ (\mathsf{atom}\ a) &= \mathsf{atom}\ a & \mathsf{dualTest}\ (t \vee u) &= \mathsf{dualTest}\ t \wedge \mathsf{dualTest}\ u \\
\mathsf{dualTest}\ \mathsf{True} &= \mathsf{False} & \mathsf{dualTest}\ (\bigwedge t) &= \bigvee(\lambda n.\mathsf{dualTest}\ (t\ n)) \\
\mathsf{dualTest}\ \mathsf{False} &= \mathsf{True} & \mathsf{dualTest}\ (\bigvee t) &= \bigwedge(\lambda n.\mathsf{dualTest}\ (t\ n)) \\
\mathsf{dualTest}\ (t \wedge u) &= \mathsf{dualTest}\ t \vee \mathsf{dualTest}\ u
\end{aligned}$$

Following well-established results in logic, this function can be used to give a constructive complement. This is formulated by showing that it lifts *disjoint* predicates on A to disjoint predicates on $\mathsf{Test}\, A$.

Definition 10. Two predicates $P, Q : A \to \mathsf{Set}$ on A are *disjoint* if there is a proof of $(a : A) \to P\ a \to Q\ a \to 0$. A predicate lifting over $F : \mathsf{Set} \to \mathsf{Set}$ is a function $f : (A \to \mathsf{Set}) \to (F\, A \to \mathsf{Set})$. Two predicate liftings $f, g : (A \to \mathsf{Set}) \to (F\, A \to \mathsf{Set})$ are *distinct* if for any two disjoint predicates P and Q on A, the pair of lifted predicates $f\ P$ and $g\ Q$ are disjoint.

Lemma 11. *The predicate liftings* $\overline{\mathsf{Test}}$ *and* $(\lambda P, t.\ \overline{\mathsf{Test}}\ P\ (\mathsf{dualTest}\ t))$ *on* Test *are distinct.*

We use this dualization of tests to motivate a particular specification for our formulation of $(-)^\beta$. If for $(-)^\alpha$ we use π_{leaf} and π_{node}, then we will use π_{leaf} and $\lambda k, o.\ \mathsf{dualTest}\ (\pi_{\mathsf{node}}\ k\ o)$ as specification for the formulation of $(-)^\beta$. In this case, the premise of node_β in Definition 8 unfolds to the type $\overline{\mathsf{Test}}\ (\lambda (x, o').\ \beta\ o'\ (Ps\ x))\ (\mathsf{dualTest}\ (\pi_{\mathsf{node}}\ k\ o))$.

Definition 12. Given $\pi_{\mathsf{leaf}} : O \to 2$ and $\pi_{\mathsf{node}} : (l : K) \to O \to \mathsf{Test}\, I\, k \times O$, and suppose α is specified using $(O, \pi_{\mathsf{leaf}}, \pi_{\mathsf{node}})$ following Definition 3, and β is specified using $(O, \pi_{\mathsf{leaf}}, \lambda k, o.\ \mathsf{dualTest}\ (\pi_{\mathsf{node}}\ k\ o))$ following Definition 8, then we call $((-)^\alpha, (-)^\beta)$ a *complementing pair*.

Note that there are multiple complementing pairs for each set O, since the specifications in terms of π_{leaf} and π_{node} may vary. We do this in order to establish the following result.

Proposition 13. *Suppose* $((-)^\alpha, (-)^\beta)$ *is a complementing pair of O-indexed predicate liftings, then for any* $o : O$*, the predicate liftings* o^α *and* o^β *on* $\mathsf{Tree}\, I$ *are distinct.*

Proof. Let P and Q be two disjoint predicates on A, and let t be a coinductive tree. We need to show that it is absurd to assume both $o^\alpha\ P\ t$ and $o^\beta\ Q\ t$. The proof proceeds by pattern matching on t.

If $t = \mathsf{leaf}\ a$, then $o^\alpha\ P\ t$ must be proved by $\mathsf{leaf}_\alpha\ b\ p$ for some $b : \pi_{\mathsf{leaf}}\ o \equiv \mathsf{true}$ and $p : P\ a$. Similarly, $o^\beta\ P\ t$ must be proved by either 1) $\mathsf{leaf}_\beta\ q$ for some $q : Q\ a$, hence with $p : P\ a$ and disjointness of P and Q, we get a proof of absurdum; or 2) $\mathsf{exep}_\beta\ b'$ for some $b' : \pi_{\mathsf{leaf}}\ o \equiv \mathsf{false}$, hence $\mathsf{true} \equiv \mathsf{false}$ by transitivity and symmetry applied to b and b'.

If $t = \mathsf{node}\ k\ ts$, proofs of $o^\alpha\ P\ t$ and $o^\beta\ Q\ t$ are required to be of the form $\mathsf{node}_\alpha\ ps$ and $\mathsf{node}_\beta\ qs$ with $ps : \overline{\mathsf{Test}}\ (\lambda (x, o').\ \alpha\ o'\ (Ps\ x))\ (\pi_{\mathsf{node}}\ k\ o)$ and $qs : \overline{\mathsf{Test}}\ (\lambda (x, o').\ \beta\ o'\ (Ps\ x))\ (\mathsf{dualTest}\ (\pi_{\mathsf{node}}\ k\ o))$. By inductive hypothesis, the predicates $\lambda (i, o').\ o'^\alpha\ P\ (ts\ i)$ and $\lambda (i, o').\ o'^\beta\ Q\ (ts\ i)$ are disjoint. And this, together with the presence of both proof terms ps and qs, is absurd by Lemma 11. □

Notice that the above proof is finite, in the sense that the term of inductive type $o^\alpha\ P\ t$ gets smaller in each recursive call.

4.2 Examples

Let us now look at our running examples. Remember that we dualize the tests of our observations.

Example 14. Consider Example 4 concerning pure computations using the sk operation. In case sk is undetectable, and we have only one observation $\downarrow$ for termination, $\downarrow^{\beta}\ P\ t$ expresses partial correctness of t: either t terminates producing a result correct under P, or t diverges producing infinitely many skips. We get that $\downarrow^{\alpha}\ P\ t$ implies $\downarrow^{\beta}\ P\ t$. In the case that the sks can be counted and $O = \mathbb{N}$, $n^{\beta}\ P\ t$ expresses partial correctness of t under time limit n: either t terminates within n skips producing a result correct under P, or t takes at least $n+1$ skips. We get that $n^{\alpha}\ P\ t$ implies $n^{\beta}\ P\ t$.

Example 15. Consider Example 5 concerning nondeterministic computations using the binary choice operation or. Then $\mathsf{may}^{\beta}\ P\ t$ states that it is not possible to get a result which does not satisfy P, so any decision process for t must either lead to divergence or the production of a result correct under P. On the other hand, $\mathsf{must}^{\beta}\ P\ t$ says that we cannot guarantee that t produces a result which does not satisfy P, so either t may diverge or it may produce a result correct under P.

Note that dualTest swaps the $\vee$ and $\wedge$ definitions in π_{node} or o. As such, β may acts like a partial correctness version of must, and β must acts like a partial correctness version of may. As a result, $\mathsf{may}^{\alpha}\ P\ t$ implies $\mathsf{must}^{\beta}\ P\ t$, and $\mathsf{must}^{\alpha}\ P\ t$ implies $\mathsf{may}^{\beta}\ P\ t$.

Example 16. We already briefly discussed the coinductive predicate lifting generated by global store observations from Example 6. Then $(n,m)^{\beta}\ P\ t$ states that: if with starting state n the program t terminates with final state m, then it produces a result correct under P. We have that $(n,m)^{\alpha}\ P\ t$ implies $(n,m)^{\beta}\ P\ t$. Moreover, due to the exception base case exep_{β}, we have that for $m \neq k$, $(n,k)^{\alpha}\ (\lambda x.1)\ t$ implies $(n,m)^{\beta}\ P\ t$. These predicates vary from more traditional partial correctness properties for global store from the literature. However, with the logic in Section 5, these alternative formulations can be reconstructed.

Example 17. Lastly, we consider the input requesting computations from Example 7. These use as observations $O = 2 \times 2^{*}$, a pair consisting of a bit and a list of bit inputs. Then $(\mathsf{left}, l)^{\beta}\ P\ t$ expresses the partial property telling us that: if t can be given the input sequence l and if t terminates after it is given, then it returns a value satisfying predicate P. On the other hand, $(\mathsf{right}, l)^{\beta}\ P\ t$ says: if the user is able to input the sequence l, then the computation t will not ask for another input afterwards. Note that neither $(\mathsf{right}, l)^{\alpha}\ P\ t$ nor $(\mathsf{right}, l)^{\beta}\ P\ t$ concern themselves with the predicate P.

A Note on Computability. The coinductive counterpart $o^{\beta}\ P\ t$ to the inductive property $o^{\alpha}\ Q\ t$ does not offer a complete notion of negation, such as is offered by the more usual functions into the empty datatype $o^{\alpha}\ P\ t \to 0$. It is however a large constructive distinct property, which allows for double negation elimination. We briefly explore the difference between the two notions of negations.

Consider a possible enumeration of pairs of all Turing machines and their input arguments, and a function returning $\mathsf{tt} : 1$ on the ones that terminate on their input. In other words, we consider a function $\mathsf{TM} : \mathbb{N} \to \mathsf{Tree}\ I\ 1$, where I is the signature with skips of Example 4. Consider the always true and always false predicates $T, F : 1 \to \mathsf{Set}$. One can construct a function $\downarrow^{\alpha}\ T\ \mathsf{TM} : \mathbb{N} \to \mathsf{Set}$ which for $n : \mathbb{N}$ gives proofs that the n-th Turing Machine terminates on its input, and also a function $\downarrow^{\beta}\ F\ \mathsf{TM} : \mathbb{N} \to \mathsf{Set}$ which for $n : \mathbb{N}$ gives proofs that the n-th Turing Machine diverges. The two predicates on $\mathbb{N}$ are necessarily disjoint. Moreover, by the Halting problem, they cannot partition $\mathbb{N}$ either. Hence, they are not complete negations of each other.

This program can also be adapted to show differences between distinctions for other examples. In nondeterminism, $\lambda n.\ \mathsf{or}(\mathsf{TM}\ n, \mathsf{leaf}\ \mathsf{tt})$ lies in the gap between $\mathsf{must}^{\alpha}\ T$ and $\mathsf{must}^{\beta}\ F$. The program $\lambda n.\ \mathsf{or}(\mathsf{TM}\ n, \Omega)$, where Ω is a provably always diverging computation, lies in the gap between $\mathsf{may}^{\alpha}\ T$ and $\mathsf{may}^{\beta}\ F$.

5 Behavioural Logic

In this section we introduce our generic programming language of denotations and formulate a logic for expressing behavioural properties of programs in this language. These are meta-theoretic programs as can be given in Agda terms, rather than one specific programming language. As such, it is similar to other generic languages based on coinductive trees, like those formulated around interaction trees [30]. Their type is specified by the following small grammar, also appearing in Moggi's monadic metalanguage [19]:

$$\sigma,\rho ::= \mathsf{N} \mid \sigma \Rightarrow \rho \mid \sigma \otimes \rho \mid \mathsf{U}\ \sigma.$$

The collection of these syntactic types is called Ty, and it includes names for the type of natural numbers, function type, Cartesian product and a unary type former U for turning computations into values, often present in call-by-value languages. Well-typed terms of syntactic type σ are elements of type $\mathsf{Tm}\ b\ \sigma$, where $b : \{\mathsf{val}, \mathsf{cpt}\}$ is a Boolean used to distinguish value terms from computation terms. Notably, computation terms of syntactic type σ are coinductive trees returning σ-typed values in their leaves. Value terms of syntactic type $\mathsf{U}\ \sigma$ are exactly computation terms of syntactic type σ.

$\mathsf{Val} : \mathsf{Ty} \to \mathsf{Set}$		$\mathsf{Cpt} : \mathsf{Ty} \to \mathsf{Set}$	$\mathsf{Tm} : 2 \to \mathsf{Ty} \to \mathsf{Set}$
$\mathsf{Val}\ \mathsf{N} = \mathbb{N}$	$\mathsf{Val}\ (\sigma \otimes \rho) = \mathsf{Val}\ \sigma \times \mathsf{Val}\ \rho$	$\mathsf{Cpt}\ \sigma = \mathsf{Tree}\ I\ (\mathsf{Val}\ \sigma)$	$\mathsf{Tm}\ \mathsf{val}\ \sigma = \mathsf{Val}\ \sigma$
$\mathsf{Val}\ (\mathsf{U}\ \sigma) = \mathsf{Cpt}\ \sigma$	$\mathsf{Val}\ (\sigma \Rightarrow \rho) = \mathsf{Val}\ \sigma \to \mathsf{Cpt}\ \rho$		$\mathsf{Tm}\ \mathsf{cpt}\ \sigma = \mathsf{Cpt}\ \sigma$

Our behavioural logic is composed of value formulae and computation formulae. Formulae are elements of type $\mathsf{Fma}\ b\ \sigma$, where b is either val (value) or cpt (computation) and σ is a syntactic type, which are inductively generated by the following inference rules:

$$\frac{n : \mathbb{N}}{\mathsf{eq}\ n : \mathsf{Fma}\ \mathsf{val}\ \mathsf{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{neq}\ n : \mathsf{Fma}\ \mathsf{val}\ \mathsf{N}} \qquad \frac{V : \mathsf{Val}\ \sigma \quad \phi : \mathsf{Fma}\ \mathsf{cpt}\ \rho}{V \mapsto \phi : \mathsf{Fma}\ \mathsf{val}\ (\sigma \Rightarrow \rho)}$$

$$\frac{\phi : \mathsf{Fma}\ \mathsf{val}\ \sigma}{\mathsf{fst}\ \phi : \mathsf{Fma}\ \mathsf{val}\ (\sigma \otimes \rho)} \qquad \frac{\phi : \mathsf{Fma}\ \mathsf{val}\ \rho}{\mathsf{snd}\ \phi : \mathsf{Fma}\ \mathsf{val}\ (\sigma \otimes \rho)} \qquad \frac{\phi : \mathsf{Fma}\ \mathsf{cpt}\ \sigma}{\mathsf{thunk}\ \phi : \mathsf{Fma}\ \mathsf{val}\ (\mathsf{U}\ \sigma)} \tag{3}$$

$$\frac{\phi s : \mathsf{Test}\ (\mathsf{Fma}\ b\ \sigma)}{\mathsf{test}\ \phi s : \mathsf{Fma}\ b\ \sigma} \qquad \frac{o : O \quad \phi : \mathsf{Fma}\ \mathsf{val}\ \sigma}{\mathsf{obs}_\alpha\ o\ \phi : \mathsf{Fma}\ \mathsf{cpt}\ \sigma} \qquad \frac{o : O \quad \phi : \mathsf{Fma}\ \mathsf{val}\ \sigma}{\mathsf{obs}_\beta\ o\ \phi : \mathsf{Fma}\ \mathsf{cpt}\ \sigma}$$

We assume $((-)^\alpha, (-)^\beta)$ is a complementing pair of O-indexed predicate liftings. We define satisfiability as a relation between syntactic terms in $\mathsf{Tm}\ b\ \sigma$ and formulae in $\mathsf{Fma}\ b\ \sigma$.

$$\begin{array}{lll}
m \models \mathsf{eq}\ n = m \equiv n & (V,W) \models \mathsf{fst}\ \phi = V \models \phi & P \models \mathsf{test}\ \phi s = \overline{\mathsf{Test}}\ (\lambda \phi.\ P \models \phi)\ \phi s \\
m \models \mathsf{neq}\ n = m \not\equiv n & (V,W) \models \mathsf{snd}\ \phi = W \models \phi & M \models \mathsf{obs}_\alpha\ o\ \phi = o^\alpha\ (\lambda V.\ V \models \phi)\ M \\
W \models V \mapsto \phi = W\ V \models \phi & V \models \mathsf{thunk}\ \phi = V \models \phi & M \models \mathsf{obs}_\beta\ o\ \phi = o^\beta\ (\lambda V.\ V \models \phi)\ M
\end{array}$$

The formulation of the logic and the satisfiability relation is quite standard [27], with a few exceptions. Formula formers $\mathsf{obs}_\alpha\ o$ and $\mathsf{obs}_\beta\ o$ play the role of modalities. E.g. in the case of Example 5, obs_α may and obs_α must correspond to modalities $\Diamond$ and $\Box$, and obs_β may and obs_β must are coinductive variants taking into account possible non-termination. The formula former test allows to construct formulae via the simple test logic Test . For function testing, we opted for a concrete argument passing test as used in [27]. This follows traditional testing approaches as used in applicative bisimilarity [3] and corresponding testing logics. Alternatively, Hoare-style predicates based on preconditions could be utilized. However, it seems in practice these are both cumbersome and difficult to work with.

The logic can be used for testing a variety of useful properties. Particularly interesting is the nesting of inductive and coinductive predicates, obtained by putting the predicates in a sequence. Using combinations of predicate liftings, we can easily swap between inductive and coinductive at different type

levels, as needed. As an example, using countable skips from Examples 4 and 14, we can formulate a formula capturing the following property of programs of type $\mathsf{U}\,\sigma$: if we evaluate both the program and the result it produces, it will not take longer to evaluate the result then it took to evaluate the initial program. This can for instance be constructed as follows: $\mathsf{test}(\bigwedge \lambda n.\ \mathsf{atom}\ (\mathsf{obs}_\beta\ n\ (\mathsf{thunk}\ (\mathsf{obs}_\alpha\ n\ (\mathsf{test}\ \mathsf{True})))))$.

Satisfiability of formulas is employed in the specification of an extensional ordering on syntactic terms. Given $P, Q : \mathsf{Tm}\ b\ \sigma$, we define the *logical approximation* $P \leq_{\mathsf{Tm}} Q = (\phi : \mathsf{Fma}\ b\ \sigma) \to P \models \phi \to Q \models \phi$. Program P is below program Q in this ordering if and only if every formula satisfied by P is also satisfied by Q. An extensional *logical equivalence* of syntactic terms $P \equiv_{\mathsf{Tm}} Q$ is given by $(P \leq_{\mathsf{Tm}} Q) \times (Q \leq_{\mathsf{Tm}} P)$.

The negation of a formula ϕ is not among the generating connectives of the behavioural logic in (3). Nevertheless, an emergent notion of negation is admissible.

Definition 18. We define negation as a function $\mathsf{negFma} : \mathsf{Fma}\ b\ \sigma \to \mathsf{Fma}\ b\ \sigma$ using the following rules:

$$
\begin{array}{ll}
\mathsf{negFma}\ (\mathsf{eq}\ n) & = \mathsf{neq}\ n \\
\mathsf{negFma}\ (\mathsf{fst}\ \phi) & = \mathsf{fst}\ (\mathsf{negFma}\ \phi) \\
\mathsf{negFma}\ (V \mapsto \phi) & = V \mapsto \mathsf{negFma}\ \phi \\
\mathsf{negFma}\ (\mathsf{obs}_\alpha\ o\ \phi) & = \mathsf{obs}_\beta\ o\ (\mathsf{negFma}\ \phi) \\
\mathsf{negFma}\ (\mathsf{test}\ \phi s) & = \mathsf{test}\ (\mathsf{mapTest}\ \mathsf{negFma}\ \phi s)
\end{array}
\qquad
\begin{array}{ll}
\mathsf{negFma}\ (\mathsf{neq}\ n) & = \mathsf{eq}\ n \\
\mathsf{negFma}\ (\mathsf{snd}\ \phi) & = \mathsf{snd}\ (\mathsf{negFma}\ \phi) \\
\mathsf{negFma}\ (\mathsf{thunk}\ \phi) & = \mathsf{thunk}\ (\mathsf{negFma}\ \phi) \\
\mathsf{negFma}\ (\mathsf{obs}_\beta\ o\ \phi) & = \mathsf{obs}_\alpha\ o\ (\mathsf{negFma}\ \phi)
\end{array}
$$

The base cases eq and neq are each others complements, and a similar relationship exists between obs_α and obs_β. Negation negFma is involutive in the following sense: for all formulae ϕ, we have the equivalence $\mathsf{negFma}\ (\mathsf{negFma}\ \phi) \equiv \phi$. Notice in particular that the double negation of a formula ϕ is syntactically equal to ϕ, not merely related by some notion of logical equivalence.

The use of the word "negation" for the formula operation negFma is justified by the fact that no syntactic program P satisfies simultaneously a formula ϕ and its complement $\mathsf{negFma}\ \phi$.

Proposition 19. *For all $\phi : \mathsf{Fma}\ b\ \sigma$, the predicates $(-) \models \phi$ and $(-) \models \mathsf{negFma}\ \phi$ are disjoint.*

Proof. Given a term P, we need to show that assuming both $P \models \phi$ and $P \models \mathsf{negFma}\ \phi$ is absurd. The interesting case is $\phi = \mathsf{obs}_\alpha\ o\ \psi$ (and dually $\phi = \mathsf{obs}_\beta\ o\ \psi$, which is proved in a similar way). In this case, the two assumptions rewrite to $o^\alpha\ (\lambda V.\ V \models \psi)\ P$ and $o^\beta\ (\lambda V.\ V \models \mathsf{negFma}\ \psi)\ P$ respectively. By inductive hypothesis, $(-) \models \psi$ and $(-) \models \mathsf{negFma}\ \psi$ are disjoint, therefore invoking Proposition 13 generates a contradiction. □

The logical approximation $\leq_{\mathsf{Tm}}$ is not symmetric, since $P \leq_{\mathsf{Tm}} Q$ does not generally entail $Q \leq_{\mathsf{Tm}} P$. The validity of $P \leq_{\mathsf{Tm}} Q$ seems also not sufficient for deriving $(\phi : \mathsf{Fma}\ b\ \sigma) \to Q \models \phi \to ((P \models \phi) \to 0) \to 0$, i.e. a doubly-negated variant of $Q \leq_{\mathsf{Tm}} P$. What can be proved from $P \leq_{\mathsf{Tm}} Q$ instead is the impossibility of P and Q to satisfy dual formulae: for any formula ϕ, it is not the case that both $Q \models \phi$ and $P \models \mathsf{negFma}\ \phi$ are derivable. This suggests the introduction of a weak notion of logical approximation $P \simeq_{\mathsf{Tm}} Q = (\phi : \mathsf{Fma}\ b\ \sigma) \to (P \models \phi) \to (Q \models \mathsf{negFma}\ \phi) \to 0$, so that $P \leq_{\mathsf{Tm}} Q$ implies $P \simeq_{\mathsf{Tm}} Q$. Unlike the logical approximation $\leq_{\mathsf{Tm}}$, the relation $\simeq_{\mathsf{Tm}}$ is symmetric, which is easily provable invoking the involutive property of negation negFma.

Alternatively, we could consider a logic with only inductive predicate liftings like in [27], and a logic with only coinductive predicate liftings. The negation operation is a function between the two logics, which sends a formula of one logic to its constructive complement. Even if one only considers behavioural equivalence under one of these logics, statements like $M \models \phi \to N \models \phi$ may be verified or falsified using the constructive negation of ϕ from the other logic. As such, coinductive statements enrich our toolkit for reasoning about pre-existing notions of equivalence based on induction.

6 Decomposability

Higher-order computations can return values which are computations themselves. Using the logic of the previous section, we can formulate behavioural properties of such computations. Properties of higher-order programs may contain multiple inductive and/or coinductive predicate liftings in succession, allowing for nuanced statements.

One thing we can do with a higher-order program is *sequence* it. This evaluates the program, and then continues by evaluating its result, thereby putting the two evaluations in sequence. The question is, can we prove whether a sequenced program satisfies some behavioural property, using behavioural properties of the unsequenced higher-order program? Answering this question is fundamental for establishing a plethora of compositionality results.

To simplify the question, we consider *double trees* of type $\mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ \mathsf{Set})$. These trees occur naturally as a result of, for instance, the study of computations of type $\mathsf{U}\ \tau$ using a value formula on τ. The sequencing of programs corresponds to an application of the monad multiplication map $\mu\ :\ \mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ \mathsf{Set}) \to \mathsf{Tree}\ I\ \mathsf{Set}$. Both single trees in $\mathsf{Tree}\ I\ \mathsf{Set}$ and double trees have natural relations of extensional ordering:

- For $t_0, t_1 : \mathsf{Tree}\ I\ \mathsf{Set}$ we say $t_0 \sqsubseteq t_1$ if for any $o : O$, $\alpha\ o\ t_0$ implies $\alpha\ o\ t_1$, and $\beta\ o\ t_0$ implies $\beta\ o\ t_1$.
- For $d_0, d_1 : \mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ \mathsf{Set})$ we say $d_0 \sqsubseteq d_1$ if for any $o_0, o_1 : O$, ${o_0}^{\alpha}\ (\alpha\ o_1)\ d_0$ implies ${o_0}^{\alpha}\ (\alpha\ o_1)\ d_1$, and ${o_0}^{\beta}\ (\beta\ o_1)\ d_0$ implies ${o_0}^{\beta}\ (\beta\ o_1)\ d_1$.

Statements of the form ${o_0}^{\alpha}\ (\alpha\ o_1)\ d_0$ and ${o_0}^{\beta}\ (\beta\ o_1)\ d_0$ are called *observational towers*, hinting at the fact that these can be extended to include even more layers of predicate liftings. We express preservation of observations over program sequencing as follows.

Definition 20. We call a triple $(O, \pi_{\mathsf{leaf}}, \pi_{\mathsf{node}})$ *strongly decomposable* if for any two double trees $d_0, d_1 : \mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ \mathsf{Set})$, $d_0 \sqsubseteq d_1$ implies $\mu\ d_0 \sqsubseteq \mu\ d_1$.

One way of establishing strong decomposability is by showing that any observations $\alpha\ o$ and $\beta\ o$ on sequenced programs can be *decomposed* into a test $\pi_{\mathsf{d}}\ o : \mathsf{Test}\ (O \times O)$ whose atoms are given by pairs $(o_0, o_1) \in O \times O$ representing observational tower statements.

Definition 21. We say that $\pi_{\mathsf{d}} : O \to \mathsf{Test}\ (O \times O)$ is an α*-decomposition* if, for any $d : \mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ \mathsf{Set})$, $\alpha\ o\ (\mu\ d)$ iff $\overline{\mathsf{Test}}\ (\lambda(o_0, o_1).\ {o_0}^{\alpha}\ (\alpha\ o_1)\ d)\ (\pi_{\mathsf{d}}\ o)$. We say that $\pi_{\mathsf{d}} : O \to \mathsf{Test}\ (O \times O)$ is a β*-decomposition* if, for any $d : \mathsf{Tree}\ I\ (\mathsf{Tree}\ I\ \mathsf{Set})$, $\beta\ o\ (\mu\ d)$ if and only if $\overline{\mathsf{Test}}\ (\lambda(o_0, o_1).\ {o_0}^{\beta}\ (\beta\ o_1)\ d)\ (\pi_{\mathsf{d}}\ o)$.

Lemma 22. *$(O, \pi_{\mathsf{leaf}}, \pi_{\mathsf{node}})$ is strongly decomposable if there exist both an α-decomposition and a β-decomposition.*

Note that in the lemma, we do not require the two decompositions to be the same. In fact, in our examples described later on, it is possible to find an α-decomposition π_{d} such that its dual $\lambda o.\ \mathsf{dualTest}\ (\pi_{\mathsf{d}}\ o)$ is a β-decomposition.

The word 'strong' in 'strongly decomposable' reflects the fact that the relation $\sqsubseteq$ between double trees is weaker than the relation induced by the logic on higher-order programs. As a result, the property is stronger than necessary for establishing most compositionality results. To establish the weaker notion of sequence preservation, *decomposability*, it is sufficient to show the existence of α and β-decompositions of the form $O \to \mathsf{Test}\ (O \times \mathsf{Test}\ O)$. Luckily, all examples we consider are strongly decomposable following Lemma 22. Hence we need not consider this weaker notion.

The decompositions can be generalized to any computation of type $\mathsf{U}\ \tau$, for any syntactic type $\tau : \mathsf{Ty}$. Given a formula $\phi : \mathsf{Fma}\ \mathsf{Val}\ \tau$ and a test $t : \mathsf{Test}\ (O \times O)$, we define a formula $t_\alpha[\phi] : \mathsf{Fma}\ \mathsf{Cpt}\ (\mathsf{U}\ \tau)$ as $\mathsf{test}\ (\mathsf{mapTest}\ (\lambda(o_1, o_2).\ \mathsf{obs}_\alpha\ o_1\ (\mathsf{thunk}\ \mathsf{obs}_\alpha\ o_2\ \phi))\ t)$. We define a formula $t_\beta[\phi] : \mathsf{Fma}\ \mathsf{Cpt}\ (\mathsf{U}\ \tau)$ in the same way but replacing each occurrence of α by β.

Lemma 23. *Suppose given* $\pi_{\mathsf{d}} : O \to \mathsf{Test}\ (O \times O)$*,*

- *if* π_{d} *is an* α*-decomposition, then, for any* $\phi : \mathsf{Fma\ val}\ \tau$*,* $M : \mathsf{Cpt}\ \tau$ *and* $o : O$*, we have* $\mu\ M \models \mathsf{obs}_\alpha\ o\ \phi$ *if and only if* $M \models (\pi_{\mathsf{d}}\ o)_\alpha[\phi]$*;*
- *if* π_{d} *is a* β*-decomposition, then, for any* $\phi : \mathsf{Fma\ val}\ \tau$*,* $M : \mathsf{Cpt}\ \tau$ *and* $o : O$*, we have* $\mu\ M \models \mathsf{obs}_\beta\ o\ \phi$ *if and only if* $M \models (\pi d\ o)_\beta[\phi]$*.*

We present instances of α and β-decompositions for our examples, proving they are strongly decomposable. In each instance, the β-decomposition is given by the test dual to the α-decomposition.

Example 24. Consider Example 4 concerning pure computations using an undetectable sk operation. Both the α- and β-decomposition can be given by $\pi_{\mathsf{d}}\downarrow = \mathsf{atom}\ (\downarrow,\downarrow)$ signifying that a sequenced program terminates if and only if the unsequenced program terminates and returns a terminating program.

In the case that the number of sk's can be measured, we give the α-decomposition as

$$\pi_{\mathsf{d}}\ n = \bigvee(\lambda m.\ \bigvee(\lambda k.\ \text{if}\ (m+k \leq n)\ \text{then}\ \mathsf{atom}\ (m,k)\ \text{else}\ \mathsf{False})).$$

This states that a sequenced program terminates within n steps, if the unsequenced program returns within m steps a program which itself terminates within k steps, such that the total number of steps $m+k$ is at most n. The β-decomposition π'_{d} is given by the dual:

$$\pi'_{\mathsf{d}}\ n = \bigwedge(\lambda m.\ \bigwedge(\lambda k.\ \text{if}\ (m+k \leq n)\ \text{then}\ \mathsf{atom}\ (m,k)\ \text{else}\ \mathsf{True})).$$

Example 25. For nondeterministic computations following Example 5, both α- and β-decompositions are given by $\lambda o.\ \mathsf{atom}\ (o,o)$. E.g., a sequenced program may terminate if the unsequenced program may return a program which may terminate.

Example 26. For global store computations from Example 6, the decompositions check for the existence of some intermediate value which specifies the state after the first evaluation and before the second evaluation of a double tree. The α-decomposition is given by $\pi_{\mathsf{d}}\ (n,m) = \bigvee(\lambda k.\ \mathsf{atom}\ ((n,k),(k,m)))$. Intuitively, this states that a sequenced program goes from state n to terminating with state m, if and only if an unsequenced program goes from n to some intermediate state k, and evaluating the returned program with state k yields termination with state m. The β-decomposition is given by the dual $\pi'_{\mathsf{d}}\ (n,m) = \bigwedge(\lambda k.\ \mathsf{atom}\ ((n,k),(k,m)))$.

Example 27. The decomposition of input observations from Example 7 relies on the enumerability of the set of bitlists 2^* and the decidability of its propositional equality. Notice that countable conjunction $\bigwedge$ and disjunction $\bigvee$ in Test are both indexed by natural numbers. But, due to to the enumerability of bitlists, they can also be indexed by 2^*. We define an α-decomposition $\pi_{\mathsf{d}} : O \to \mathsf{Test}\ (O \times O)$ for the input example, whose dual is a β-decomposition. This function splits the bitlist l into a bitlist x before the first termination, and a bitlist y for testing the program after the first termination.

$$\begin{aligned}\pi_{\mathsf{d}}\ (\mathsf{left},l) &= \bigvee(\lambda x,y : 2^*.\ \text{if}\ (x{+\!\!+}y \equiv l)\ \text{then}\ \mathsf{atom}\ ((\mathsf{left},x),(\mathsf{left},y))\ \text{else}\ \mathsf{False})\\ \pi_{\mathsf{d}}\ (\mathsf{right},l) &= \mathsf{atom}\ ((\mathsf{right},l),(\mathsf{right},l)) \vee\\ &\qquad \bigvee(\lambda x,y : 2^*.\ \text{if}\ (x{+\!\!+}y \equiv l)\ \text{then}\ \mathsf{atom}\ ((\mathsf{left},x),(\mathsf{right},y))\ \text{else}\ \mathsf{False})\end{aligned}$$

Higher-order Nondeterminism. We study some example of nondeterministic programs. Firstly, for any type τ, we have the diverging computation $\Omega_\tau : \mathsf{Cpt}\ \tau$ defined corecursively as the program satisfying the following equation: $\Omega_\tau = \mathsf{node\ or}\ (\lambda\,\mathsf{left}.\ \Omega_\tau, \lambda\,\mathsf{right}.\ \Omega_\tau)$. For readability's sake, we simply write or in place of node or, and we avoid writing $\lambda\,\mathsf{left}$ and $\lambda\,\mathsf{right}$, since it is clear that the left argument corresponds to the left branch of the tree, similarly for the right case. It can be shown that the Ω_τ computation *must diverge*, satisfying the formula $\mathsf{mustdiv} = \mathsf{negFma}\ (\mathsf{obs}_\alpha\ \mathsf{may}\ (\mathsf{test\ True}))$, which is

equal to obs_β may (test False). Note the use of the formula test False which cannot be satisfied, hence mustdiv cannot be satisfied by a computation that can produce a result.

As a case study of properties of higher-order programs, we consider the following two computations of type U (U τ) for some type τ, a variation on an example from [13, 20]: Take the program $P =$ or (leaf ($\Omega_{\mathsf{U}\tau}$), leaf (leaf Ω_τ)) and $Q =$ leaf (or ($\Omega_{\mathsf{U}\tau}$, leaf Ω_τ)). We can show that P may produce a result that must diverge, and hence it satisfies $\phi = \mathsf{obs}_\alpha$ may (thunk mustdiv). On the other hand, all programs which Q returns may terminate, hence Q satisfies negFma $\phi = \mathsf{obs}_\beta$ may (thunk (obs_α may (test True))). Hence, P and Q are not related via the logical equivalence. Note moreover that ϕ only uses the may observation. Via the connection to applicative bisimilarity shown in the next section, we are able to reproduce Ong's result that these two terms are not applicatively bisimilar [20]. See [29] for more classical examples of properties for higher-order effectful programs.

7 Simulations and Equality

The logic from section 5 is capable of expressing a behavioural or observational difference between two programs. For two terms $P, Q : \mathsf{Tm}\ b\ \tau$, such a difference is a formula $\phi : \mathsf{Fma}\ b\ \tau$ such that $P \models \phi$ and $Q \models \mathsf{negFma}\ \phi$. This shows that $P \not\equiv_{\mathsf{Tm}} Q$. Showing that two programs have no behavioural differences is not trivial. One strategy for proving behavioural equivalence is using *applicative simulations* [3].

Applicative simulations, or more generally just simulations, are relations on program terms which prove *similarity* between them. If two programs are related by a simulation they are *similar*, and if they are related by a symmetric simulation, also called a *bisimulation*, then they are *bisimilar*. In this section we define these notions, and show how they relate to our logically induced behavioural equivalence.

In the context of algebraic effects, we need to specify how to compare the effect operations of two programs. This is done in [12] using *relators* [15], which lift relations between types X and Y to relations between the monadic liftings Tree $I\ X$ and Tree $I\ Y$. In this paper, we use a simpler variation of the notion of relator considering only homogeneous relations. Given some relation $\mathscr{R} : A \to A \to \mathsf{Set}$, a predicate $f : A \to \mathsf{Set}$ is considered $\mathscr{R}$*-correct* if there is a function of type $(a\ b : A) \to f\ a \to \mathscr{R}\ a\ b \to f\ b$.

Definition 28. Given $(O, \pi_{\mathsf{leaf}}, \pi_{\mathsf{node}})$ and $\mathscr{R} : A \to A \to \mathsf{Set}$, we define a new relation $\Gamma(\mathscr{R}) : \mathsf{Tree}\ I\ A \to \mathsf{Tree}\ I\ A \to \mathsf{Set}$ such that $\Gamma(\mathscr{R})\ t_0\ t_1$ holds if and only if, for any $\mathscr{R}$-correct predicate f and observation $o : O$, we have the following two implications: $o^\alpha\ f\ t_0 \to o^\alpha\ f\ t_1$ and $o^\beta\ f\ t_0 \to o^\beta\ f\ t_1$.

The original definition of a relator using modalities involves a feedback loop between predicates and relations, which necessarily increases the universe level when describing higher order types. This is problematic when trying to show preservation over sequencing. The above variation avoids this problem, at the cost of compositionality, but without sacrificing transitivity of the resulting notions of program equivalence.

The relation lifting satisfies some relator properties, such as monotonicity with respect to the relation order $\mathscr{R} \subseteq \mathscr{S}$. Moreover, the resulting relation behaves well with respect to sequencing, as seen below:

Proposition 29. *If $(O, \pi_{\mathsf{leaf}}, \pi_{\mathsf{node}})$ is strongly decomposable, then for any relation $\mathscr{R} : A \to A \to \mathsf{Set}$ and any two double trees $d_0, d_1 :$ Tree I (Tree $I\ A$), if $\Gamma(\Gamma(\mathscr{R}))\ d_0\ d_1$ then $\Gamma(\mathscr{R})\ (\mu\ d_0)\ (\mu\ d_1)$.*

The relation lifting Γ is unfortunately not *compositional*, since it does not preserve relation composition. Despite this fact though, we are still able to show transitivity of the resulting notions of program relation, as we will see in Proposition 33.

Using the relation lifting Γ, we define a notion of similarity between programs. A *well-typed* relation $\overline{\mathscr{R}}$ is a collection of relations on syntactic terms indexed by sorts and types: $\overline{\mathscr{R}} : (b : \{\mathsf{val}, \mathsf{cpt}\})\ (\tau : \mathsf{Ty}) \to \mathsf{Tm}\ b\ \tau \to \mathsf{Tm}\ b\ \tau \to \mathsf{Set}$.

Definition 30. A well-typed relation $\overline{\mathscr{R}}$ is an *applicative* Γ-*simulation* when:

- If $\overline{\mathscr{R}}\ \mathsf{val}\ \mathbb{N}\ n\ m$, then $n \equiv m$.
- If $\overline{\mathscr{R}}\ \mathsf{val}\ (\sigma \Rightarrow \tau)\ V\ W$, then, for any $U : \mathsf{Val}\ \sigma$, $\overline{\mathscr{R}}\ \mathsf{cpt}\ \tau\ (V\,U)\ (W\,U)$.
- If $\overline{\mathscr{R}}\ \mathsf{val}\ (\sigma \otimes \tau)\ (V_0, V_1)\ (W_0, W_1)$, then $\overline{\mathscr{R}}\ \mathsf{val}\ \sigma\ V_0\ W_0$ and $\overline{\mathscr{R}}\ \mathsf{val}\ \tau\ V_1\ W_1$.
- If $\overline{\mathscr{R}}\ \mathsf{val}\ (\mathsf{U}\ \tau)\ V\ W$, then $\overline{\mathscr{R}}\ \mathsf{cpt}\ \tau\ V\ W$.
- If $\overline{\mathscr{R}}\ \mathsf{cpt}\ \tau\ M\ N$, then $\Gamma(\overline{\mathscr{R}}\ \mathsf{val}\ \tau)\ M\ N$.

Definition 31. For any sort $b : \{\mathsf{val}, \mathsf{cpt}\}$ and type $\tau : \mathsf{Ty}$, we call two terms $P, Q : \mathsf{Tm}\ b\ \tau$ *applicatively* Γ-*similar* if there is a Γ-simulation $\overline{\mathscr{R}}$ such that $\overline{\mathscr{R}}\ b\ \tau\ P\ Q$ holds. Two terms $P, Q : \mathsf{Tm}\ b\ \tau$ are called *applicatively* Γ-*bisimilar* if there is a symmetric Γ-simulation $\overline{\mathscr{R}}$ such that $\overline{\mathscr{R}}\ b\ \tau\ P\ Q$ holds.

Importantly, the notions of similarity and bisimilarity give us proof techniques for showing that there are no observable differences between programs.

Proposition 32. *Given two programs P and Q of the same type, then:*

- *If P and Q are applicatively* Γ-*similar, then* $P \leq_{\mathsf{Tm}} Q$.
- *If P and Q are applicatively* Γ-*bisimilar, then* $P \equiv_{\mathsf{Tm}} Q$.

Proof. Given an applicative Γ-simulation $\overline{\mathscr{R}}$, we show by induction on formulae ϕ, that for any P and Q related by $\overline{R}$, $P \models \phi$ implies $Q \models \phi$. If this holds, we say $\overline{\mathscr{R}}$ preserves ϕ.

Most cases are straightforward. Most difficult are the instances where $\phi = \mathsf{obs}_\alpha\ o\ \psi$ and $\phi = \mathsf{obs}_\beta\ o\ \psi$ of some type τ. In these cases, we use the induction hypothesis on ψ to show $\overline{\mathscr{R}}$ preserves ψ. Hence $\lambda V.(V \models \psi)$ is $(\overline{\mathscr{R}}\ \mathsf{val}\ \tau)$-correct, and the result can be derived from the appropriate simulation condition. □

It seems however impossible to derive implications in the opposite direction. It appears that bisimilarity and logical equivalence as stated in this paper are not the same, as opposed to the classical variants defined by Simpson and Voorneveld [27]. This has to do with the limitations of explicit enumeration.

The variant formulation of relator in Definition 28 was chosen such that it yields program relations classically similar to the original ones, yet allow us to constructively prove Propositions 33 and 29.

Proposition 33. *If P is similar to Q, and Q is similar to R, then P is similar to R.*

Proof. The proof follows ideas from [22], showing first that the union of two simulations is again a simulation. We then show that the reflexive-transitive closure $\overline{\mathscr{R}}^*$ of a simulation is a simulation too. Then, if $\overline{\mathscr{R}}\ P\ Q$ and $\overline{\mathscr{S}}\ Q\ T$, then $(\overline{\mathscr{R}} \cup \overline{\mathscr{S}})^*\ P\ T$, hence they are similar. □

For particular programming languages, such as Plotkin's PCF [23], it can be classically proven that the resulting notion of Γ-similarity and Γ-bisimilarity is a *precongruence* and *congruence* respectively. These kinds of proofs typically follow *Howe's method* [10]. This is a desirable property, since any two programs related by a congruence are *contextually equivalent*. Moreover, in [17, 18] it is shown that for a continuation-passing style language, such a program equivalence coincides with contextual equivalence.

We can use the predicate liftings and constructive logic to study programming languages embedded into Agda. Whether or not the resulting program equivalence is a congruence is dependent on the operations of the programming language. We try to approach such congruence results by studying common operations, such as sequencing and function application, and prove preservation over such operations in general. We leave to future work the question of whether the traditional methods such as Howe's method

can be formalized and extended to work for embeddings of languages such as PCF into our generic programming language of denotations.

Program equivalences like applicative bisimilarity and logical equivalences are useful in regards to both applicability and formalization. They reduce the burden of proof for showing equivalence of programs to either finding a bisimulation that relates the programs, or testing for behavioural properties at the appropriate type.

8 Conclusion and Future Work

In our development we have been working with a generic notion of program given by elements of type $\mathsf{Tm}\ b\ \sigma$, for some label $b : \{\mathsf{val}, \mathsf{cpt}\}$ and syntactic type $\sigma : \mathsf{Ty}$. These kind of objects typically arise from the interpretation of effectful recursive programs in denotational domain-theoretic models, or, alternatively, from program evaluation w.r.t. some particular evaluation strategy. Our Agda formalization includes the implementation of two different object programming languages: fine-grained call-by-value PCF [23, 16] and an automaton-like state machine. Programs in these languages can potentially be modelled as terms of our generic language, and their behaviour analysed using the techniques developed in this paper. Our Agda code includes the interpretation of automata and PCF programs as coinductive trees. The full embedding of such computational constructs into our generic programming language of denotations is left for future work. For the latter, we should take inspiration from Benton et al.'s formalization of domain theory in Coq [5] and Paviotti et al.'s full abstraction result for PCF in guarded type theory [21].

The type of coinductive trees we employ naturally captures an "intensional" view of possibly nonterminating computations, in the sense that the user of a program may be allowed to observe the number of computation steps. If this is the case, programs returning the same values but with different computation time are considered different. One might wish to change the encoding of programs and move to an "extensional" view of termination, so that intensional aspects of computations such as computation time become unobservable. We foresee a possible modification of the type of trees involving the presence of the partiality monad [4, 7]. This modification in turn requires the switch to proof assistants with support for higher inductive types such as Cubical Agda [28].

Many other effects besides the given examples can be described using this formalization too. One such example is that of probability. It is possible to implement binary probabilistic choice, and define observations with the rational interval $O = [0,1)$ following [27]. The inductive predicate lifting generated by an observation $p \in O$ checks whether the probability of satisfying a predicate is higher than p. The coinductive counterpart instead checks whether satisfaction of the lifted property or divergence together has at least a probability of $(1-p)$. This example has been implemented, but has not been formally verified to be decomposable, due to the complexity of the necessary mathematical tools.

References

[1] Michael Gordon Abbott, Thorsten Altenkirch & Neil Ghani (2005): *Containers: Constructing strictly positive types*. *Theoretical Computer Science* 342(1), pp. 3–27, doi:10.1016/j.tcs.2005.06.002.

[2] Andreas Abel, Brigitte Pientka, David Thibodeau & Anton Setzer (2013): *Copatterns: programming infinite structures by observations*. In Roberto Giacobazzi & Radhia Cousot, editors: *Proc. of 40th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'13*, ACM, pp. 27–38, doi:10.1145/2429069.2429075.

[3] Samson Abramsky (1990): *The lazy lambda calculus*. In: *Research topics in functional programming*, Addison-Wesley Longman Publishing Co., Inc., pp. 65–116.

[4] Thorsten Altenkirch, Nils Anders Danielsson & Nicolai Kraus (2017): *Partiality, Revisited - The Partiality Monad as a Quotient Inductive-Inductive Type*. In Javier Esparza & Andrzej S. Murawski, editors: *Proc. of 20th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS'17*, *Lecture Notes in Computer Science* 10203, pp. 534–549, doi:10.1007/978-3-662-54458-7_31.

[5] Nick Benton, Andrew Kennedy & Carsten Varming (2009): *Some Domain Theory and Denotational Semantics in Coq*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Proc. of 22nd Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs'09*, *Lecture Notes in Computer Science* 5674, Springer, pp. 115–130, doi:10.1007/978-3-642-03359-9_10.

[6] Venanzio Capretta (2005): *General recursion via coinductive types*. *Logical Methods in Computer Science* 1(2), doi:10.2168/LMCS-1(2:1)2005.

[7] James Chapman, Tarmo Uustalu & Niccolò Veltri (2019): *Quotienting the delay monad by weak bisimilarity*. *Mathematical Structures in Computer Science* 29(1), pp. 67–92, doi:10.1017/S0960129517000184.

[8] Nils Anders Danielsson (2018): *Up-to techniques using sized types*. *Proc. of ACM on Programming Languages* 2(POPL), pp. 43:1–43:28, doi:10.1145/3158131.

[9] Ichiro Hasuo (2015): *Generic Weakest Precondition Semantics from Monads Enriched with Order*. *Theoretical Computer Science* 604(C), pp. 2–29, doi:10.1016/j.tcs.2015.03.047.

[10] Douglas J. Howe (1996): *Proving Congruence of Bisimulation in Functional Programming Languages*. *Information and Computation* 124(2), p. 103–112, doi:10.1006/inco.1996.0008.

[11] P. Johann, A. Simpson & J. Voigtländer (2010): *A Generic Operational Metatheory for Algebraic Effects*. In: *Proc. of 25th Annual IEEE Symp. on Logic in Computer Science, LICS'10*, IEEE Computer Society, pp. 209–218, doi:10.1109/LICS.2010.29.

[12] Ugo Dal Lago, Francesco Gavazzo & Paul Blain Levy (2017): *Effectful applicative bisimilarity: Monads, relators, and Howe's method*. In: *Proc. of 32nd Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS'17*, IEEE Computer Society, pp. 1–12, doi:10.1109/LICS.2017.8005117.

[13] Søren B. Lassen (1998): *Relational Reasoning about Functions and Nondeterminism*. Ph.D. thesis, University of Aarhus.

[14] Xavier Leroy & Hervé Grall (2009): *Coinductive big-step operational semantics*. *Inf. Comput.* 207(2), pp. 284–304, doi:10.1016/j.ic.2007.12.004.

[15] Paul Blain Levy (2011): *Similarity Quotients as Final Coalgebras*. In Martin Hofmann, editor: *Proc. of 14th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS'11*, *Lecture Notes in Computer Science* 6604, Springer, pp. 27–41, doi:10.1007/978-3-642-19805-2_3.

[16] Paul Blain Levy, John Power & Hayo Thielecke (2003): *Modelling environments in call-by-value programming languages*. *Information and Computation* 185(2), pp. 182–210, doi:10.1016/S0890-5401(03)00088-9.

[17] Cristina Matache (2019): *Program Equivalence for Algebraic Effects via Modalities*. Master's thesis, University of Oxford. arXiv:1902.04645.

[18] Cristina Matache & Sam Staton (2019): *A Sound and Complete Logic for Algebraic Effects*. In Mikolaj Bojanczyk & Alex Simpson, editors: *Proc. of 22nd Int. Conf. on Foundations of Software Science and Computation Structures, FoSSsCS'19*, *Lecture Notes in Computer Science* 11425, Springer, pp. 382–399, doi:10.1007/978-3-030-17127-8_22.

[19] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Information and Computation* 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[20] C.-H. Luke Ong (1993): *Non-Determinism in a Functional Setting*. In: *Proc. of 8th Ann. Symp. on Logic in Computer Science, LICS'93*, IEEE Computer Society, pp. 275–286, doi:10.1109/LICS.1993.287580.

[21] Marco Paviotti, Rasmus Ejlers Møgelberg & Lars Birkedal (2015): *A Model of PCF in Guarded Type Theory*. In Dan R. Ghica, editor: *Proc. of 31st Conf. on Mathematical Foundations of Programming Semantics, MFPS'15*, *Electronic Notes in Theoretical Computer Science* 319, Elsevier, pp. 333–349, doi:10.1016/j.entcs.2015.12.020.

[22] Andrew M. Pitts (2000): *Operational Semantics and Program Equivalence*. In Gilles Barthe, Peter Dybjer, Luís Pinto & João Saraiva, editors: *Adv. Lect. from Int. Summer School on Applied Semantics, APPSEM'00*, *Lecture Notes in Computer Science* 2395, Springer, pp. 378–412, doi:10.1007/3-540-45699-6_8.

[23] Gordon D. Plotkin (1977): *LCF considered as a programming language*. *Theoretical Computer Science* 5(3), pp. 223–255, doi:10.1016/0304-3975(77)90044-5.

[24] Gordon D. Plotkin & John Power (2001): *Adequacy for Algebraic Effects*. In Furio Honsell & Marino Miculan, editors: *Proc. of 4th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS'01*, *Lecture Notes in Computer Science* 2030, pp. 1–24, doi:10.1007/3-540-45315-6_1.

[25] Gordon D. Plotkin & Matija Pretnar (2013): *Handling Algebraic Effects*. *Log. Methods Comput. Sci.* 9(4, article 23), pp. 1–36, doi:10.2168/lmcs-9(4:23)2013.

[26] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: *Proc. of 18th Ann. Symp. on Foundations of Computer Science, FOCS'77*, IEEE Computer Society, pp. 46–57, doi:10.1109/SFCS.1977.32.

[27] Alex Simpson & Niels Voorneveld (2020): *Behavioural Equivalence via Modalities for Algebraic Effects*. *ACM Transactions on Programming Languages and Systems* 42(1), pp. 4:1–4:45, doi:10.1145/3363518.

[28] Andrea Vezzosi, Anders Mörtberg & Andreas Abel (2019): *Cubical Agda: a dependently typed programming language with univalence and higher inductive types*. *Proc. of ACM on Programming Languages* 3(ICFP), pp. 87:1–87:29, doi:10.1145/3341691.

[29] Niels Voorneveld (2020): *Equality between programs with effects*. Ph.D. thesis, University of Ljubljana.

[30] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce & Steve Zdancewic (2020): *Interaction trees: representing recursive and impure programs in Coq*. *Proc. ACM Program. Lang.* 4(POPL), pp. 51:1–51:32, doi:10.1145/3371119.