*JFP*, 59 pages, 2025. © Cambridge University Press 2025
doi:10.1017/S0956796825100026


# *A Contextual Formalization of Structural Coinduction*

PAUL DOWNEN
*University of Massachusetts, Lowell (e-mail: paul_downen@uml.edu)*
ZENA M. ARIOLA
*University of Oregon (e-mail: ariola@cs.uoregon.edu)*

## Abstract

Structural induction is pervasively used by functional programmers and researchers for both informal reasoning as well as formal methods for program verification and semantics. In this paper, we promote its dual — structural coinduction — as a technique for understanding corecursive programs only in terms of the logical structure of their context. We illustrate this technique as an informal method of proofs which closely match the style of informal inductive proofs, where it is straightforward to check that all cases are covered and the coinductive hypotheses are used correctly. This intuitive idea is then formalized through a syntactic theory for deriving program equalities, which is justified purely in terms of the computational behavior of abstract machines and proved sound with respect to observational equivalence.

## 1 Introduction

Every day, a large community of computer scientists — working on applications and theory of functional programming, verification, type systems, and semantics — employ induction to effectively reason about software and its behavior. Whether mechanically checked by a computer or informally written with pen and paper, various forms of inductive techniques are applied with confidence that the result is well-founded. What is the secret to this confidence? The inductive principle itself limits recursive reasoning to only pieces of the original example which are *structurally smaller* than it (Burstall, 1969).

Coinduction, the dual to induction, is not understood or used with the same level of familiarity or frequency. It is usually relegated to coalgebras (Rutten, 2019), since traditionally only the categorical setting speaks clearly about the duality that relates induction and coinduction. Despite its difficulty, coinduction remains an essential principle for dealing with important software systems like concurrent processes, web servers, and operating systems (Barwise & Moss, 1997), which endlessly run while interacting with their environment.

Why, then, does coinduction see less use in both informally written and mechanically verified proofs of programming language theory? One major obstacle is that coinduction is easy to formulate in a dangerous way, where the recursive nature of coinduction seems too powerful on the surface and can lead to nonsensical, viciously circular proofs. To tamper

down on this unreasonable power, we must externally check that the coinductive hypothesis is only applied in certain special contexts, which fundamentally breaks compositional reasoning; certain proofs may seem valid until they are embedded into a larger context.

In this paper, we aim to alleviate the non-compositional difficulty of coinduction by reformulating it to more closely resemble the familiar forms of induction used in practice, with the hope that this presentation will make coinduction more suitable for widespread use in programming environments (Gordon, 2017). Our methodology is to work in a setting based on (Curien & Herbelin, 2000) where the important contexts are reified into first-class objects that can be labeled and have a predictable structure — similar to inductive objects like numbers and trees which can be named and analyzed structurally. The key idea here is that the coinductive principle limits recursion to only contexts which are *structurally smaller* than the starting point of coinduction, and that this requirement is checked locally by just looking at the label where corecursion happens. This paper then demonstrates how coinduction — in terms of both an informal pen-and-paper methodology as well as a formal program logic for proving equality of corecursive programs with or without side effects — can be seen as induction on the context. We thus avoid resorting to the least or greatest fixed point notions of lattice theory and domain theory to explain the duality between induction and coinduction (Gordon, 1994). Both the informal technique and formal system are sound in the sense that every syntactically-derived equality implies an observational equivalence: the program logic is proven sound with respect to an adequate denotational model of observational equivalence defined in terms of its operational semantics.

Having (co)inductive reasoning principles expressed within a calculus follows previous work (Curien & Herbelin, 2000) on defining a calculus that directly expresses the dualities commonly seen in logic, specifically (Downen & Ariola, 2023). For example, the duality between true and false computationally appears as the duality between a process that produces information and one that consumes information (Downen & Ariola, 2018). We think our work follows the spirit of Kozen & Silva (2017); the authors present several examples of the use of coinduction in informal-style mathematical arguments. This paper strives to put those arguments on solid ground that can be justified only in terms of computation. We believe our approach only requires the same mathematical skills already used by computer scientists to reason inductively over data structures, while still capturing the essential property of compositionality. Coinduction is explained in terms of subcomponents, much the same way structural induction is presented.

In addition to giving a compositional and computational foundation for coinduction, this paper studies both induction and coinduction proof principles in the setting of a language derived from classical logic à la Curry-Howard, and identifies the syntactic conditions that must be imposed on an argument to make it correct in the presence of computational effects. For example, it is well known that one needs to be careful applying induction in non-strict languages such as Haskell. For example, the optimization

$$x * 0 \stackrel{?}{=} 0$$

can be proved by traditional induction over the natural numbers, but it does not hold according to a call-by-name evaluation strategy because this proof does not account for the case of nontermination. Letting $\Omega$ stand for a non-terminating expression, notice that plugging in $\Omega$ for $x$ leads to an incorrect equality; $\Omega * 0 = 0$ claims that a non-terminating

expression $\Omega * 0$ is equal to a constant value. Even worse, if we consider other computational effects such as aborting a computation, then substituting `abort` 1 for $x$ leads to

$$(\texttt{abort}\ 1) * 0 = 0$$

seemingly equating 1 to 0.

Dually, strict languages such as OCaml suffer from the same kinds of problems where naïve coinduction is not always correct. For example, consider infinite stream $(x_0, x_1, x_2, x_3, \dots)$ with the two main projections:

$$\text{head}(x_0, x_1, x_2, \dots) = x_0 \qquad \text{tail}(x_0, x_1, x_2, \dots) = x_1, x_2, \dots$$

Now, intuitively, taking the *head* and *tail* of a stream and putting them back does nothing:

$$\text{head}\, xs, \text{tail}\, xs = xs$$

and this equation does indeed hold, under both call-by-name and call-by-value evaluation, both with or without side effects (as we will see in more detail later). So intuitively, we should be able to apply this equality a second time to expand out two places, right?

$$\text{head}\, xs, \text{head}(\text{tail}\, xs), \text{tail}(\text{tail}\, xs) \stackrel{?}{=} xs \tag{1.1}$$

As it turns out, this equation fails in call-by-value languages with side effects, similar to the problem with $x * 0 = 0$ in call-by-name. Of course, in the call-by-value setting, we need to be careful of timing considerations when handling infinite objects: an infinite stream cannot be fully evaluated in advance. So to support infinite streams, we ensure that the head and tail of the stream are only computed *on demand*, that is, at the last moment when they are required. As such, any pair $M, N$ of a head element $M$ and tail $N$ is treated as a first-class value, even if $M$ and $N$ have not been evaluated yet. Now, consider the partial stream value $0, \Omega$: asking for its head element returns 0, and asking for its tail does not return (it incurs the non-terminating computation $\Omega$). Plugging in $0, \Omega$ for $xs$ in (1.1) gives

$$\text{head}(0, \Omega), \text{head}(\text{tail}(0, \Omega)), \text{tail}(\text{tail}(0, \Omega)) = (0, \Omega)$$

This leads to a counter-example in call-by-value, where $\textbf{let}\, z = \text{tail}(0, \Omega)\, \textbf{in}\, 1$ does not terminate because $\text{tail}(0, \Omega) = \Omega$ which never returns a value that can be bound to $z$, but

$$\begin{aligned}
&\textbf{let}\, z = \text{tail}(\text{head}(0, \Omega), \text{head}(\text{tail}(0, \Omega)), \text{tail}(\text{tail}(0, \Omega)))\ \textbf{in}\, 1 \quad &= \\
&\textbf{let}\, z = \Omega, \Omega\ \textbf{in}\, 1 \quad &= \\
&1
\end{aligned}$$

In place of non-termination, plugging in $(0, \texttt{abort}\ 2)$ for $xs$ in (1.1) also serves as another example using abort as a side-effect:

$$\begin{aligned}
&\textbf{let}\, z = \text{tail}(0, \texttt{abort}\ 2)\, \textbf{in}\, 1 \quad &= \\
&\textbf{let}\, z = \text{tail}(\text{head}(0, \texttt{abort}\ 2), \text{head}(\text{tail}(0, \texttt{abort}\ 2)), \text{tail}(\text{tail}(0, \texttt{abort}\ 2)))\ \textbf{in}\, 1
\end{aligned}$$

where the left-hand side aborts with 2, but the right-hand side returns 1.

For a more practical example, consider these informally-defined operations on streams:

$$\begin{aligned}
\mathit{evens}\ (x_0, x_1, x_2, x_3, \dots) &= x_0, x_2, x_4, \dots \\
\mathit{odds}\ (x_0, x_1, x_2, x_3, \dots) &= x_1, x_3, x_5, \dots
\end{aligned}$$

$$merge\ (x_0, x_1, x_2, \dots)\ (y_0, y_1, y_2, \dots) = x_0, y_0, x_1, y_1, x_2, y_2, \dots$$

The *evens* function selects only the even elements of a stream, *odds* selects only the odd elements of a stream, and *merge* interleaves two streams together by alternating between them. It should be intuitive that selecting the even and odd elements of a stream and merging them back together is the same as the original stream:

$$merge\ (evens\ xs)\ (odds\ xs) \stackrel{?}{=} xs$$

We can prove this fact by conventional methods of coinduction, and this paper shows that it also holds true using our notion of *strong* structural coinduction under call-by-name evaluation whether or not *xs* contains side effects. However, strong structural coinduction is *not* sound in a call-by-value language with effects, and as a consequence the intuitive equality is incorrect. What goes wrong? The timing considerations of when the head or tail of a stream are computed become important in call-by-value, and need to be explicated. So if we rewrite *evens*, *odds*, and *merge* more formally as

$$\begin{aligned} evens\ xs &= head\ xs, odds\ (tail\ xs) \\ odds\ xs &= evens\ (tail\ xs) \\ merge\ xs\ ys &= head\ xs, head\ ys, merge\ (tail\ xs)\ (tail\ ys) \end{aligned}$$

then notice that *merge* applied to any two stream values *xs* and *ys* will *always* return a stream starting with at least two comma-separated elements. So if we consider the counter-example stream value $0, \Omega$ with exactly one comma, notice that $odds\ (0, \Omega) = evens\ \Omega$ which does not terminate. Thus, $merge\ (evens\ (0, \Omega))\ (odds\ (0, \Omega))$ doesn't terminate, too, which is immediately different from the value $0, \Omega$. As a second counter-example, consider the stream value $0, 1, 2, \Omega$ with three commas, we will return a value:

$$merge\ (evens\ (0, 1, 2, \Omega))\ (odds\ (0, 1, 2, \Omega)) = 0, 1, \Omega$$

but that value can be differentiated from the starting stream $0, 1, 2, \Omega$ by asking for the third element (2) via head(tail(tail *xs*)). Notice in each case, the stream returned by *merge* always has an even number of comma-separated elements; if the starting *xs* has an odd number of elements before $\Omega$, the last one is forgotten. As before, using an `abort` in place of $\Omega$ gives us alternative abort-based counter-examples. So plugging in the partial stream value 0, `abort` 1 causes $merge\ (evens\ (0, \texttt{abort}\ 1))\ (odds\ (0, \texttt{abort}\ 1))$ to immediately abort with 1 instead of returning some value, and plugging in 0, 1, 2, `abort` 3 returns the smaller partial stream 0, 1, `abort` 3.

The remainder of this paper will give a firm, unambiguous, computational foundation for reasoning about corecursive programs using structural coinduction, including the subtle timing implications when side effects are involved. As an example of an inductive type we take the canonical definition of natural numbers, and for coinductive types we consider streams, building on top of the abstract machine language from (Downen & Ariola, 2023), which defines a calculus of primitive recursion and corecursion. Here, we extend that calculus with an informal ("pen-and-paper") proof technique for structural coinduction as well as a formal logic for soundly deriving equalities between programs with control effects. While we focus on examples involving natural numbers and streams, the reasoning

techniques discussed here are applicable to other data and codata types. More specifically, we provide the following contributions:

- Section 2 provides examples of applying informal (co)inductive reasoning to programs which use (co)recursion to process (co)inductive types like numbers and streams.
- Section 3 introduces the differences between intensional and extensional equality in the presence of (co)inductive types, and gives a sound, formal program logic for reasoning (co)inductively about (co)recursive programs with control effects. It also discusses how to soundly generalize the induction principle for call-by-value, and soundly generalize the coinduction principle for call-by-name.
- Section 4 discusses the contrast in expressive power between the different (co)-inductive principles: restricted and universally sound versus unrestricted and conditionally sound. To do so, we derive a number of more familiar reasoning principles, such as strong induction on the numbers, bisimulation, and compositionality of coinduction.
- Section 5 provides a proof that the program logic in Section 3 is sound: syntactic formal proofs of equality imply semantic contextual equivalence, and more specifically, 0 is not equal to 1. This proof is modeled in terms of a logical relation based on the notion of *orthogonality* between producers and consumers.

## 2 (Co)Inductive Reasoning About (Co)Recursive Programs

Skilled functional programmers are quite adept at using induction, both for writing their programs and reasoning about them. For example, we can follow the inductive structure of the usual natural number type,

$$\begin{array}{c}\textbf{inductive data}\ \mathrm{Nat}\ \textbf{where}\\ \begin{array}{ll}\mathrm{zero}: & \mathrm{Nat}\\ \mathrm{succ}: & \mathrm{Nat}\to\mathrm{Nat}\end{array}\end{array}$$

to inductively define the addition $plus : \mathrm{Nat}\to\mathrm{Nat}\to\mathrm{Nat}$ by the patterns of Nat like so:

$$\begin{array}{c} plus\ \ \mathrm{zero}\ \ y = y\\ plus\ (\mathrm{succ}\ x)\ y = \mathrm{succ}(plus\ x\ y)\end{array}$$

Why is *plus* well-founded—meaning it never causes an infinite loop, and always returns a valid result for any valid arguments? Because its first argument always gets *smaller* (Burstall, 1969); the $x$ passed into the recursive call $plus\ x\ y$ is a piece of the original argument $\mathrm{succ}\ x$ from the call $plus\ (\mathrm{succ}\ x)\ y$ that triggered it (a property we can statically check in the definition).

### 2.1 Structural induction

To reason about functions like *plus* that take Nats as arguments, programmers can also reason by induction that follows the structure of Nat in the same way the code is written. For example, the very definition of *plus* is first built on the identity of addition, that

*plus* zero $y = y$ for any number $y$, so it holds by just calculation with no further verification required. However, addition's second identity law, *plus* $x$ zero $= x$ for any number $x$, cannot be directly calculated in the same way. Instead, matching the inductive structure of *plus* itself, we have to prove this property by cases on what that first argument $x$ might be.

**Example Theorem 2.1.** *For all $x$ of type* Nat*, plus $x$* zero $= x$.

**Proof** By induction on the structure of the value $x$

- $x =$ zero. We have: *plus* zero zero $=$ zero $= x$, by definition of *plus*.
- $x =$ succ $x'$. Assume the inductive hypothesis *plus* $x'$ zero $= x'$. From there,

$$\begin{aligned} \mathit{plus}\ (\mathrm{succ}\ x')\ \mathrm{zero} &= \mathrm{succ}(\mathit{plus}\ x'\ \mathrm{zero}) && \text{def. of } \mathit{plus} \\ &= \mathrm{succ}\ x' = x && \text{inductive hypothesis} \end{aligned}$$

■

Why is the proof of Example Theorem 2.1 well-founded—meaning it does not contain any vicious circle in its reasoning? In the inductive hypothesis, we assume that Example Theorem 2.1 is true for the *specific* $x'$ that is the predecessor of the $x$ we started with. As such, the cyclic reasoning always applies to a strictly smaller $x$, and the inductive hypothesis can never lead to a vicious cycle no matter how we use it, so no further checks are necessary to validate this proof. More formally, this inductive argument is justified because the set of natural numbers of type Nat is a *least* fixed point (Pierce, 2002): it is the *smallest* set containing zero and closed under succ.

### *2.2 Coinductive programs and proofs*

The correspondence between inductive type, inductive program, and inductive proof, all line up quite neatly in the functional paradigm, with each of them following exactly the same structure. Since *coinduction* is the logical dual of induction, shouldn't this correspondence naturally extend to coinductive structures like infinite streams? One can define the type of streams coinductively as the largest data type (*i.e., greatest* fixed point, or *final coalgebra*)

$$\begin{array}{c} \textbf{coinductive data}\ \mathrm{Stream}\ a\ \textbf{where} \\ \mathrm{Cons} : a \to \mathrm{Stream}\ a \to \mathrm{Stream}\ a \end{array}$$

built from the Cons constructor—appending an element to the front of another stream—without any base case for the empty stream. From there, coinductive functions—like *always* : $a \to$ Stream $a$ which returns the stream that always contains the same value of type $a$, or *iterate* : $(a \to a) \to a \to$ Stream $a$ that builds an infinite stream from some original $a$ by repeatedly applying a given function to it—can be defined cyclically like so:

$$\mathit{always}\ x = \mathrm{Cons}\ x\ (\mathit{always}\ x) \qquad \mathit{iterate}\ f\ x = \mathrm{Cons}\ x\ (\mathit{iterate}\ f\ (f\ x))$$

Why are *always* and *iterate* well-founded? The answer here is not so clear; at first glance, they look like infinite loops that never return a definite answer. However, one justification is that both definitions are *productive*: they always return a Cons before recursing. In other

words, the self-references of *always* and *iterate* are both found *inside* a Cons (that is, in the context Cons *first* ...). If we assume lazy evaluation of Cons this can be enough to prevent infinite loops for well-behaved observers of the stream; trying to access the "last" element of an infinite stream is not a well-behaved observer. While this justification may not be as self-evident as the structural induction of functions like *plus*, at least it is a property that can be syntactically checked in the specific definitions of *always* and *iterate*.

**Example Theorem 2.2.** *For all values x, iterate* $(\lambda y.y)\ x = always\ x$.

**Proof** Assume the coinductive hypothesis

$$iterate\ (\lambda y.y)\ x = always\ x\ .$$

From there,

$$\begin{aligned} iterate\ (\lambda y.y)\ x &= \text{Cons}\ x\ (iterate\ (\lambda y.y)\ ((\lambda y.y)\ x)) && \text{def. of } iterate \\ &= \text{Cons}\ x\ (iterate\ (\lambda y.y)\ x) && \beta\text{-reduction} \\ &= \text{Cons}\ x\ (always\ x) && \text{coinductive hypothesis} \\ &= always\ x && \text{def. of } always \end{aligned}$$

■

Why is the proof of Example Theorem 2.2 well-founded? Compared to the inductive proof, skepticism of this form of coinduction is more warranted. After all, the proof begins by immediately assuming the very fact it is trying to prove, with no stipulation! What's to stop us from this much simpler, but hopelessly vicious, "coinductive" proof of Example Theorem 2.2?

**Bad Proof** Assume the coinductive hypothesis $iterate\ (\lambda y.y)\ x = always\ x$. From the coinductive hypothesis, it follows that $iterate\ (\lambda y.y)\ x = always\ x$, as required. ◪

This bad proof is obviously invalid, even though it "proved" the goal through a trivial sequence of apparently valid steps (introducing a hypothesis and using it). What's the difference between the bad proof above and the good proof of Example Theorem 2.2? The good proof only tried to use the coinductive hypothesis "inside" a Cons, whereas the bad proof just nakedly used the coinductive hypothesis outside of any Cons. Thus, somehow a coinductive proof of this form must be very careful that certain hypotheses can only be used in certain contexts, whatever that means, even if they are a perfect match for the current goal.

The concern over even a trivial theorem like Example Theorem 2.2 shows the potential breakdown of the correspondence of coinductive types, coinductive programs, and coinductive proofs; at each step, our certainty in the basic structures wanes. Even if the intuition for distinguishing "good" from "bad" programs may be fraught, a formal system like a proof assistant might be up to the task of regulating context-sensitive uses of the coinductive hypothesis to verify a proof. But a human that needs to understand a proof with informal reasoning, which has no perfect overseer like a mental proof assistant, can quickly become overwhelmed as the theorems and proofs grow ever larger. No wonder why coinduction fills us with such trepidation.

Instead, what is needed is a style of coinductive reasoning which is not burdened by precariously implicit context-sensitive rules of validity. Or put another way, the context-sensitivity imposed by coinduction should be made an *explicit* part of the coinductive hypothesis, so that it may be used freely, and fearlessly, in any place that it fits.

The first step is to shift our view away from coinductively-defined data types, to co-inductively-defined *codata types* (Hagino, 1987). Rather than constructors, codata types define the basic observations, or projections, allowed on values of the type. For infinite streams, these are the head and tail projections that access the first element and the remainder of the stream, respectively, as described in the following declaration:

$$\begin{array}{c}\textbf{coinductive codata}\ \text{Stream}\ a\ \textbf{where}\\ \text{head} : \text{Stream}\ a \to a\\ \text{tail} : \text{Stream}\ a \to \text{Stream}\ a\end{array}$$

While abstractly these may be two views of the same isomorphic structure, they give us a different way to understand coinduction and the operational meaning of programs. With codata types, we define programs by matching on the structure of their projections, dual to the way function programmers define functions like *plus* by matching on the structure of constructors of data types. For example, the *always* and *iterate* functions can be rewritten in terms of *copatterns* (Abel *et al.*, 2013) like so:

$$\begin{array}{ll}\text{head}(\mathit{always}\ x) = x & \text{head}(\mathit{iterate}\ f\ x) = x\\ \text{tail}(\mathit{always}\ x) = \mathit{always}\ x & \text{tail}(\mathit{iterate}\ f\ x) = \mathit{iterate}\ f\ (f\ x)\end{array}$$

Here, head and tail are seen as projection *functions*, and the streams returned by *always* $x$ and *iterate* $f$ $x$ are defined by the two lines, describing what their head and tail is.

But copatterns alone aren't enough. We also need to *label* our context, so that the language itself is expressive enough to regulate how to control the use of coinduction to certain contexts. To do so, we have to move outside of pure functional programming, based on intuitionistic logic, to a more language based on *classical logic* with labels and jumps. One such language (Downen *et al.*, 2015) is modeled on the sequent calculus (Curien & Herbelin, 2000), which provides a syntax for writing contextual observations as first-class objects. In this sequent style, a Greek letter $\alpha$, $\beta$, ..., stands for an observer of values, and the *command* $\langle x \| \alpha \rangle$ says that the observer $\alpha$ is applied to $x$, or symmetrically, that the value $x$ is returned to $\alpha$.

Rather than viewing the Stream operations head and tail as functions, as we did above, we could instead view them as primitive ways to build new observations. So if $\alpha$ is expecting to observe a value of type $a$, then the *composition* head $\alpha$ observes a value of type Stream $a$ by taking its first element and passing it to $\alpha$. Similarly, if $\beta$ is expecting to observe a value of type Stream $a$, then tail $\beta$ observes a value of type Stream $a$ by discarding its first element and passing the rest to $\beta$. Putting them together, the observation tail(head $\beta$) should be read as first observing the tail of a stream and then applying the head to that result so that $\beta$ receives the second element of the stream. The two different views—head and tail as functions versus observations—are always equal to one another:

$$\langle \text{head}\ s \| \alpha \rangle = \langle s \| \text{head}\ \alpha \rangle \qquad \langle \text{tail}\ s \| \beta \rangle = \langle s \| \text{tail}\ \beta \rangle \tag{2.1}$$

In this observer-centric style, we can further refine *always* and *iterate* by labeling the full context in which they are observed in a command, $\langle always\ x \| \alpha \rangle$ and $\langle iterate\ f\ x \| \alpha \rangle$. The two definitions then follow by matching on the structure of the observer $\alpha$, which must be built by either a head or tail projection.

$$\begin{array}{ll} \langle always\ x \| \text{head}\ \beta \rangle = \langle x \| \beta \rangle & \langle iterate\ f\ x \| \text{head}\ \beta \rangle = \langle x \| \beta \rangle \\ \langle always\ x \| \text{tail}\ \alpha' \rangle = \langle always\ x \| \alpha' \rangle & \langle iterate\ f\ x \| \text{tail}\ \alpha' \rangle = \langle iterate\ f\ (f\ x) \| \alpha' \rangle \end{array}$$

Now, the fact that these corecursive functions are well-founded follows the same basic reasoning as the recursive function *plus*: all instances of self-reference are invoked with a *strictly smaller observer*. In particular, the observer $\alpha'$ in the corecursive call $\langle always\ x \| \alpha' \rangle$ is a piece of the original observer tail $\alpha'$ from the command $\langle always\ x \| \text{tail}\ \alpha' \rangle$. Similarly, the observer of the corecursive call $\langle iterate\ f\ (f\ x) \| \alpha' \rangle$ came from a piece of the observer in the proceeding command $\langle iterate\ f\ x \| \text{tail}\ \alpha' \rangle$. So copattern-matching over observers restores the symmetry between recursive functions (which *consume* inductively-defined arguments) and corecursive functions (which *produce* coinductively-defined results).

### 2.3 Structural coinduction

What about proofs involving these programs? Let's try to prove the analogous version of Example Theorem 2.2 but in the context of an observer labeled $\alpha$.

**Example Theorem 2.3.** *For all values x of type a and all observers* $\alpha$ *of type* Stream *a,* $\langle iterate\ (\lambda y.y)\ x \| \alpha \rangle = \langle always\ x \| \alpha \rangle$.

**Proof** By coinduction on the stream received by $\alpha$, *i.e.,* by induction on the structure of $\alpha$:

- $\alpha = \text{head}\ \beta$. We have: $\langle iterate\ (\lambda y.y)\ x \| \text{head}\ \beta \rangle = \langle x \| \beta \rangle = \langle always\ x \| \text{head}\ \beta \rangle$ by definition of *iterate* and *always*.
- $\alpha = \text{tail}\ \alpha'$. Assume the coinductive hypothesis

  $$\langle iterate\ (\lambda y.y)\ x \| \alpha' \rangle = \langle always\ x \| \alpha' \rangle.$$

  From there,

  $$\begin{array}{lll} \langle iterate\ (\lambda y.y)\ x \| \text{tail}\ \alpha' \rangle & = \langle iterate\ (\lambda y.y)\ ((\lambda y.y)\ x) \| \alpha' \rangle & \text{def. of } iterate \\ & = \langle iterate\ (\lambda y.y)\ x \| \alpha' \rangle & \beta\text{-reduction} \\ & = \langle always\ x \| \alpha' \rangle & \text{coinductive hypothesis} \\ & = \langle always\ x \| \text{tail}\ \alpha' \rangle & \text{def. of } always \end{array}$$

■

Notice how the proof of Example Theorem 2.3 above follows much closer the overall shape of the inductive proof of Example Theorem 2.1. First, the coinductive hypothesis is only introduced in the step for $\alpha = \text{tail}\ \alpha'$; as with induction, the coinductive hypothesis is not available to show the base case of $\alpha = \text{head}\ \beta$. Furthermore, the coinductive hypothesis $\langle iterate\ (\lambda y.y)\ x \| \alpha' \rangle = \langle always\ x \| \alpha' \rangle$ carries enough information to fully dictate the valid contexts in which it can be used. In particular, we can only assume the goal (that

*iterate* $(\lambda y.y)\ x$ is equal to *always* $x$) when observed by $\alpha'$, the specific ancestor to the original observer $\alpha = \text{tail}\ \alpha'$. There is no way to use the coinductive hypothesis to equate these two streams when seen by any other observer. In particular, the coinductive hypothesis doesn't even apply to the original goal $\langle \mathit{iterate}\ (\lambda y.y)\ x \| \alpha \rangle = \langle \mathit{always}\ x \| \alpha \rangle$, like we did in the bad coinductive proof, because $\alpha \neq \alpha'$. As such, even though the proof above is informal, there is no longer any ambiguity about its validity, so no further checks are necessary to avoid vicious cycles. Since it follows the structure of the context, we call it *structural coinduction*.

But have we proved the same result; are Example Theorems 2.2 and 2.3 logically the same? In order to compare the two, we can employ the notion of *observational equivalence*, which says that two terms are equal exactly when no observer can tell them apart. Spelled out in terms of labeled contexts, observational equivalence is the principle that, for any terms $M$ and $N$ (without a free reference to $\alpha$):

$$M = N \text{ if and only if, for all } \alpha, \langle M \| \alpha \rangle = \langle N \| \alpha \rangle$$

Applying this principle to Example Theorems 2.2 and 2.3, we know for all values $x$,

$$\mathit{iterate}\ (\lambda y.y)\ x = \mathit{always}\ x \text{ if and only if, for all } \alpha, \langle \mathit{iterate}\ (\lambda y.y)\ x \| \alpha \rangle = \langle \mathit{always}\ x \| \alpha \rangle$$

So the two theorems state the same equality, up to observational equivalence.

Note that we can derive the result of applying head and tail as functions to *iterate* via observational equivalence. Starting with a generic $\alpha$, we can convert these function applications to observations on top of $\alpha$ to match the definition of *iterate* as follows:

$$\begin{aligned} \langle \text{head}(\mathit{iterate}\ f\ x) \| \alpha \rangle &=^{\text{by } 2.1} \langle \mathit{iterate}\ f\ x \| \text{head}\ \alpha \rangle = \langle x \| \alpha \rangle \\ \langle \text{tail}(\mathit{iterate}\ f\ x) \| \alpha \rangle &=^{\text{by } 2.1} \langle \mathit{iterate}\ f\ x \| \text{tail}\ \alpha \rangle \ = \langle \mathit{iterate}\ f\ (f\ x) \| \alpha \rangle \end{aligned}$$

and thus by observational equivalence, we have

$$\text{head}(\mathit{iterate}\ f\ x) = x \tag{2.2}$$

$$\text{tail}(\mathit{iterate}\ f\ x) = \mathit{iterate}\ f\ (f\ x) \tag{2.3}$$

Notice that these equations derived by observational equivalence are exactly the same as the purely functional, copattern-matching definition of *iterate* that we gave above. In other words, the two copattern-based definitions—one in a functional style, and the other matching on the structure of a labeled observer—are equivalent.

Let's continue with one more example of structural coinduction. Here is a definition for *map*ping a function over all elements in an infinite stream, where we use head and tail as both part of the main coinductive observer on the left-hand side of the equations, as well as a function to be applied to the given stream we are mapping over on the right-hand sides.

$$\begin{aligned} \langle \mathit{map}\ f\ s \| \text{head}\ \beta \rangle &= \langle f\ (\text{head}\ s) \| \beta \rangle \\ \langle \mathit{map}\ f\ s \| \text{tail}\ \alpha' \rangle &= \langle \mathit{map}\ f\ (\text{tail}\ s) \| \alpha' \rangle \end{aligned}$$

Notice how, in the following proof, we can make use of observational equivalence in order to reason about head and tail applied as a function to *iterate*.

**Example Theorem 2.4.** *For all functions $f$ of type $A \to B$, values $x$ of type $A$, and observers $\alpha$ of type* Stream $A$, $\langle map\ f\ (iterate\ f\ x) \| \alpha \rangle = \langle iterate\ f\ (f\ x) \| \alpha \rangle$.

**Proof** By structural coinduction on the observer $\alpha$ (leaving the value $x$ generic):

- $\alpha = \text{head}\ \beta$.

$$\begin{aligned} \langle map\ f\ (iterate\ f\ x) \| \text{head}\ \beta \rangle &= \langle f(\text{head}(iterate\ f\ x)) \| \beta \rangle && \text{def. of } map \\ &= \langle f\ x \| \beta \rangle && \text{by (2.2)} \\ &= \langle iterate\ f\ (f\ x) \| \text{head}\ \beta \rangle && \text{def. of } iterate \end{aligned}$$

- $\alpha = \text{tail}\ \alpha'$. Assume the coinductive hypothesis

$$\langle map\ f\ (iterate\ f\ x) \| \alpha' \rangle = \langle iterate\ f\ (f\ x) \| \alpha' \rangle$$

for *all* values $x$ of type $A$.

$$\begin{aligned} \langle map\ f\ (iterate\ f\ x) \| \text{tail}\ \alpha' \rangle &= \langle map\ f\ (\text{tail}(iterate\ f\ x)) \| \alpha' \rangle && \text{def. of } map \\ &= \langle map\ f\ (iterate\ f\ (f\ x)) \| \alpha' \rangle && \text{by (2.3)} \\ &= \langle iterate\ f\ (f\ (f\ x)) \| \alpha' \rangle && \text{coinductive hypothesis} \\ & && \text{with } (f\ x) \text{ for } x \\ &= \langle iterate\ f\ (f\ x) \| \text{tail}\ \alpha' \rangle && \text{def. of } iterate \end{aligned}$$

■

### *2.4 Mutual coinduction*

Given a stream $s$, we can define mutually corecursive functions taking the elements of $s$ at even and odd positions as so:

$$\begin{aligned} \langle evens\ s \| \text{head}\ \beta \rangle &= \langle s \| \text{head}\ \beta \rangle & \langle odds\ s \| \text{head}\ \beta \rangle &= \langle s \| \text{tail}(\text{head}\ \beta) \rangle \\ \langle evens\ s \| \text{tail}\ \alpha' \rangle &= \langle odds\ (\text{tail}\ s) \| \alpha' \rangle & \langle odds\ s \| \text{tail}\ \alpha' \rangle &= \langle evens\ (\text{tail}\ s) \| \text{tail}\ \alpha' \rangle \end{aligned}$$

By observational equivalence and the definitions of *odds* and *evens*, we have:

$$odds\ s = evens\ (\text{tail}\ \ s) \tag{2.4}$$

$$\text{tail}(evens\ s)) = odds\ (\text{tail}\ \ s) \tag{2.5}$$

Merging two streams is defined as:

$$\begin{aligned} \langle merge\ s_1\ s_2 \| \text{head}\ \beta \rangle &= \langle s_1 \| \text{head}\ \beta \rangle \\ \langle merge\ s_1\ s_2 \| \text{tail}(\text{head}\ \beta) \rangle &= \langle s_2 \| \text{head}\ \beta \rangle \\ \langle merge\ s_1\ s_2 \| \text{tail}(\text{tail}\ \alpha') \rangle &= \langle merge\ (\text{tail}\ s_1)\ (\text{tail}\ s_2) \| \alpha' \rangle \end{aligned}$$

As an application of observational equivalence, we have

$$\text{tail}(\text{tail}(merge\ s_1\ s_2)) = merge(\text{tail}\ s_1)(\text{tail}\ s_2) \tag{2.6}$$

**Example Theorem 2.5.** *For all values $s_1$ and $s_2$ and observers $\alpha$ of type* Stream $A$, $\langle evens\ (merge\ s_1\ s_2) \| \alpha \rangle = \langle s_1 \| \alpha \rangle$ *and* $\langle odds\ (merge\ s_1\ s_2) \| \alpha \rangle = \langle s_2 \| \alpha \rangle$.

**Proof** Both equalities can be proved at the same time by structural coinduction on $\alpha$ (leaving $s_1$ and $s_2$ generic):

- $\alpha = \text{head}\ \beta$.

$$\begin{aligned} \langle evens\ (merge\ s_1\ s_2) \| \text{head}\ \beta \rangle &= \langle merge\ s_1\ s_2 \| \text{head}\ \beta \rangle && \text{def. of } evens \\ &= \langle s_1 \| \text{head}\ \beta \rangle && \text{def. of } merge \end{aligned}$$

$$\begin{aligned} \langle odds\ (merge\ s_1\ s_2) \| \text{head}\ \beta \rangle &= \langle merge\ s_1\ s_2 \| \text{tail}(\text{head}\ \beta) \rangle && \text{def. of } odds \\ &= \langle s_2 \| \text{head}\ \beta \rangle && \text{def. of } merge \end{aligned}$$

- $\alpha = \text{tail}\ \alpha'$. Assume the coinductive hypotheses

$$\langle evens\ (merge\ s_1\ s_2) \| \alpha' \rangle = \langle s_1 \| \alpha' \rangle \tag{2.7}$$

$$\langle odds\ (merge\ s_1\ s_2) \| \alpha' \rangle = \langle s_2 \| \alpha' \rangle \tag{2.8}$$

for *all* values $s_1$ and $s_2$ of type Stream $A$.

$$\begin{aligned} &\langle evens\ (merge\ s_1\ s_2) \| \text{tail}\ \alpha' \rangle \\ &= \langle odds\ (\text{tail}(merge\ s_1\ s_2)) \| \alpha' \rangle && \text{def. of } evens \\ &= \langle evens\ (\text{tail}(\text{tail}(merge\ s_1\ s_2))) \| \alpha' \rangle && \text{by (2.4)} \\ &= \langle evens\ (merge\ (\text{tail}\ s_1)\ (\text{tail}\ s_2)) \| \alpha' \rangle && \text{by (2.6)} \\ &= \langle \text{tail}\ s_1 \| \alpha' \rangle && \text{coinductive hypothesis (2.7)} \\ &= \langle s_1 \| \text{tail}\ \alpha' \rangle && \text{tail observation (2.1)} \end{aligned}$$

$$\begin{aligned} &\langle odds\ (merge\ s_1\ s_2) \| \text{tail}\ \alpha' \rangle \\ &= \langle evens\ (\text{tail}(merge\ s_1\ s_2)) \| \text{tail}\ \alpha' \rangle && \text{def. of } odds \\ &= \langle odds\ (\text{tail}(\text{tail}(merge\ s_1\ s_2))) \| \alpha' \rangle && \text{def. of } evens \\ &= \langle odds\ (merge\ (\text{tail}\ s_1)\ (\text{tail}\ s_2)) \| \alpha' \rangle && \text{by (2.6)} \\ &= \langle \text{tail}\ s_2 \| \alpha' \rangle && \text{coinductive hypothesis (2.8)} \\ &= \langle s_2 \| \text{tail}\ \alpha' \rangle && \text{tail observation (2.1)} \end{aligned}$$

■

### *2.5 Strong coinduction*

Let us try to prove that the property $\langle merge\ (evens\ s)\ (odds\ s) \| \alpha \rangle = \langle s \| \alpha \rangle$ holds for all values $s$ and observers $\alpha$ of type Stream $A$. We will show the complete proof shortly. For now, we will focus on the problematic step. We do a proof by conduction on $\alpha$. We can easily prove the property if $\alpha = \text{head}(\beta)$. If $\alpha = \text{tail}(\beta)$ then we proceed by case analysis on $\beta$. If $\beta = \text{head}(\beta')$ the proof goes through without any issues. If $\beta = \text{tail}(\beta')$ we need to prove $\langle merge\ (evens\ s)\ (odds\ s) \| \text{tail}(\text{tail}(\beta')) \rangle = \langle s \| \text{tail}(\text{tail}(\beta')) \rangle$ and note that the coinductive

hypothesis is:

$$\langle merge\ (evens\ s)\ (odds\ s) \| \beta \rangle = \langle s \| \beta \rangle \tag{2.9}$$

for a generic $s$. We then have:

$$\begin{aligned}
&\langle merge\ (evens\ s)\ (odds\ s) \| \text{tail}(\text{tail}\ \beta') \rangle \\
&= \langle merge\ (\text{tail}(evens\ s))\ (\text{tail}(odds\ s)) \| \beta' \rangle && \text{def. of } merge \\
&= \langle merge\ (\text{tail}(evens\ s))\ (\text{tail}(evens\ (\text{tail}\ s))) \| \beta' \rangle && \text{by (2.4)} \\
&= \langle merge\ (odds\ (\text{tail}\ s))\ (odds\ (\text{tail}(\text{tail}\ s))) \| \beta' \rangle && \text{by (2.5)} \\
&= \langle merge\ (evens\ (\text{tail}(\text{tail}\ s)))\ (odds\ (\text{tail}(\text{tail}\ s))) \| \beta' \rangle && \text{by (2.4)}
\end{aligned}$$

At this point, we would like to apply the coinductive hypothesis 2.9, but it is fixed to $\beta$ and does not hold on $\beta'$. What we need instead is a strong version of coinduction. This is not surprising since the same issue comes up with induction. If $\alpha = \text{tail}(\text{tail}\ \beta')$, we assume the property holds not just for the immediate subcontext $\text{tail}\ \beta'$, but also for $\beta'$, too. We use this strengthened reasoning principle to break the following coinductive proof into more specific sub-cases.

**Example Theorem 2.6.** *For all values $s$ and observers $\alpha$ of type* Stream $A$, $\langle merge\ (evens\ s)\ (odds\ s) \| \alpha \rangle = \langle s \| \alpha \rangle$

**Proof** By strong coinduction on the structure of the observer $\alpha$ (where we leave the stream value $s$ generic):

- $\alpha = \text{head}\ \beta$.

$$\begin{aligned}
\langle merge\ (evens\ s)\ (odds\ s) \| \text{head}\ \beta \rangle &= \langle evens\ s \| \text{head}\ \beta \rangle && \text{def. of } merge \\
&= \langle s \| \text{head}\ \beta \rangle && \text{def. of } evens
\end{aligned}$$

- $\alpha = \text{tail}(\text{head}\ \beta')$.

$$\begin{aligned}
\langle merge\ (evens\ s)\ (odds\ s) \| \text{tail}(\text{head}\ \beta') \rangle &= \langle odds\ s \| \text{head}\ \beta' \rangle && \text{def. of } merge \\
&= \langle s \| \text{tail}(\text{head}\ \beta') \rangle && \text{def. of } odds
\end{aligned}$$

- $\beta = \text{tail}(\text{tail}\ \beta')$. Assume the coinductive hypothesis (CH)

$$\langle merge\ (evens\ s)\ (odds\ s) \| \beta' \rangle = \langle s \| \beta' \rangle \tag{2.10}$$

for *all* values $s$ of type Stream $A$.

$$\begin{aligned}
&\langle merge\ (evens\ s)\ (odds\ s) \| \text{tail}(\text{tail}\ \beta') \rangle \\
&= \langle merge\ (\text{tail}(evens\ s))\ (\text{tail}(odds\ s)) \| \beta' \rangle && \text{def. of } merge \\
&= \langle merge\ (\text{tail}(evens\ s))\ (\text{tail}(evens\ (\text{tail}\ s))) \| \beta' \rangle && \text{by (2.4)} \\
&= \langle merge\ (odds\ (\text{tail}\ s))\ (odds\ (\text{tail}(\text{tail}\ s))) \| \beta' \rangle && \text{by (2.5)} \\
&= \langle merge\ (evens\ (\text{tail}(\text{tail}\ s)))\ (odds\ (\text{tail}(\text{tail}\ s))) \| \beta' \rangle && \text{by (2.4)} \\
&= \langle \text{tail}(\text{tail}\ s) \| \beta' \rangle && \text{CH (2.10) with } (\text{tail}(\text{tail}\ s)) \text{ for } s \\
&= \langle s \| \text{tail}(\text{tail}\ \beta') \rangle && \text{by tail observation (2.1)}
\end{aligned}$$

*Commands (c), general terms (v), and general coterms (e)*:

$$\mathit{Command} \ni c ::= \langle v \| e \rangle \qquad \mathit{Term} \ni v, w ::= \mu\alpha.c \mid R \qquad \mathit{CoTerm} \ni e, f ::= \tilde{\mu}x.c \mid L$$

*Type-specific introductions of values on the right (R) and covalues on the left (L)*:

$$\mathit{Right} \ni R ::= \lambda x.v \mid \mathrm{zero} \mid \mathrm{succ}\, V \mid \mathbf{corec}\{\mathrm{head}\ \alpha \to e \mid \mathrm{tail}\ \beta \to \gamma.f\}\ \mathbf{with}\ V$$
$$\mathit{Left} \ni L ::= V \cdot E \mid \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\ x \to y.w\}\ \mathbf{with}\ E \mid \mathrm{head}\ E \mid \mathrm{tail}\ E$$

*Call-by-name values (V) and evaluation contexts (E)*:

$$\mathit{Value} \ni V ::= v \qquad\qquad \mathit{CoValue} \ni E ::= \alpha \mid L$$

*Call-by-value values (V) and evaluation contexts (E)*:

$$\mathit{Value} \ni V ::= x \mid R \qquad\qquad \mathit{CoValue} \ni E ::= e$$

*Operational rules:*

$$(\mu) \qquad \langle \mu\alpha.c \| E \rangle \mapsto c[E/\alpha]$$
$$(\tilde{\mu}) \qquad \langle V \| \tilde{\mu}x.c \rangle \mapsto c[V/x]$$
$$(\beta_{\to}) \qquad \langle \lambda x.v \| V \cdot E \rangle \mapsto \langle v[V/x] \| E \rangle$$
$$(\beta_{\mathrm{zero}}) \qquad \left\langle \mathrm{zero} \,\middle\|\, \begin{array}{l}\mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \qquad \mid \mathrm{succ}\ x \to y.w\} \\ \mathbf{with}\ E\end{array} \right\rangle \mapsto \langle v \| E \rangle$$
$$(\beta_{\mathrm{succ}}) \qquad \left\langle \mathrm{succ}\, V \,\middle\|\, \begin{array}{l}\mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \qquad \mid \mathrm{succ}\ x \to y.w\} \\ \mathbf{with}\ E\end{array} \right\rangle \mapsto \left\langle \mu\alpha. \left\langle V \,\middle\|\, \begin{array}{l}\mathbf{rec}\ \{\ \mathrm{zero} \to v \\ \qquad \mid \mathrm{succ}\ x \to y.w\} \\ \mathbf{with}\ \alpha\end{array} \right\rangle \,\middle\|\, \tilde{\mu}y.\langle w[V/x] \| E \rangle \right\rangle$$
$$(\beta_{\mathrm{head}}) \qquad \left\langle \begin{array}{l}\mathbf{corec}\{\ \mathrm{head}\ \alpha \to e \\ \qquad \mid \mathrm{tail}\ \beta \to \gamma.f\} \\ \mathbf{with}\ V\end{array} \,\middle\|\, \mathrm{head}\, E \right\rangle \mapsto \langle V \| e[E/\alpha] \rangle$$
$$(\beta_{\mathrm{tail}}) \qquad \left\langle \begin{array}{l}\mathbf{corec}\{\ \mathrm{head}\ \alpha \to e \\ \qquad \mid \mathrm{tail}\ \beta \to \gamma.f\} \\ \mathbf{with}\ V\end{array} \,\middle\|\, \mathrm{tail}\, E \right\rangle \mapsto \left\langle \mu\gamma.\langle V \| f[E/\beta] \rangle \,\middle\|\, \tilde{\mu}x. \left\langle \begin{array}{l}\mathbf{corec}\{\ \mathrm{head}\ \alpha \to e \\ \qquad \mid \mathrm{tail}\ \beta \to \gamma.f\} \\ \mathbf{with}\ x\end{array} \,\middle\|\, E \right\rangle \right\rangle$$

Fig. 1: Syntax and semantics of the uniform, (co)recursive abstract machine.

■

## 3 Intensional Versus Extensional Equality With (Co)Inductive Types

Before we lay out our formal rules of (co)inductive reasoning about the behavior of programs, we need to specify the language in which those programs are written. For the sake of illustration, we will use the abstract machine language with both recursion and corecursion defined in (Downen & Ariola, 2023) — which is an extension of (Curien & Herbelin, 2000) with primitive types for inductive numbers and coinductive streams, with full primitive (co)recursion, not just (co)iteration — because the symmetry of its syntax lets us express the duality of induction and coinduction most clearly. However, note that the important (co)-inductive reasoning principles below can be applied to other languages as well—provided the language can label points in the flow of control.

Types ($A$), typing environments ($\Gamma$), and typing judgements ($J$)

$$\begin{array}{rll}
\textit{Type} \ni A, B & ::= A \to B \mid \mathrm{Nat} \mid \mathrm{Stream}\, A & \\
\textit{Env} \ni & \Gamma ::= \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A & (\text{all } x \text{ and } \alpha \text{ in } \Gamma \text{ are distinct}) \\
\textit{Typing} \ni & \tau ::= c \mid v : A \mid e \div A & \\
\textit{Judge} \ni & J ::= \boxed{\Gamma \vdash \tau} & (FV(\tau) \subseteq AV(\Gamma))
\end{array}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma \vdash e \div A}{\Gamma \vdash \langle v \| e \rangle}\ \mathit{Cut}$$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \mathit{VarR} \qquad \frac{}{\Gamma, \alpha \div A, \Gamma' \vdash \alpha \div A}\ \mathit{VarL}$$

$$\frac{\Gamma, \alpha \div A \vdash c}{\Gamma \vdash \mu\alpha.c : A}\ \mathit{ActR} \qquad \frac{\Gamma, x : A \vdash c}{\Gamma \vdash \tilde{\mu} x.c \div A}\ \mathit{ActL}$$

$$\frac{\Gamma, x : A \vdash v : B}{\Gamma \vdash \lambda x.v : A \to B} \to R \qquad \frac{\Gamma \vdash V : A \quad \Gamma \vdash E \div B}{\Gamma \vdash V \cdot E \div A \to B} \to L$$

$$\frac{}{\Gamma \vdash \mathrm{zero} : \mathrm{Nat}}\ \mathrm{Nat}R_{\mathrm{zero}} \qquad \frac{\Gamma \vdash V : \mathrm{Nat}}{\Gamma \vdash \mathrm{succ}\, V : \mathrm{Nat}}\ \mathrm{Nat}R_{\mathrm{succ}}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : \mathrm{Nat}, y : A \vdash w : A \quad \Gamma \vdash E \div A}{\Gamma \vdash \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\, x \to y.w\}\ \mathbf{with}\ E \div \mathrm{Nat}}\ \mathrm{Nat}L$$

$$\frac{\Gamma \vdash E \div A}{\Gamma \vdash \mathrm{head}\, E \div \mathrm{Stream}\, A}\ \mathrm{Stream}L_{\mathrm{head}} \qquad \frac{\Gamma \vdash E \div \mathrm{Stream}\, A}{\Gamma \vdash \mathrm{tail}\, E \div \mathrm{Stream}\, A}\ \mathrm{Stream}L_{\mathrm{tail}}$$

$$\frac{\Gamma, \alpha \div A \vdash e \div B \quad \Gamma, \beta \div \mathrm{Stream}\, A, \gamma \div B \vdash f \div B \quad \Gamma \vdash V : B}{\Gamma \vdash \mathbf{corec}\{\mathrm{head}\, \alpha \to e \mid \mathrm{tail}\, \beta \to \gamma.f\}\ \mathbf{with}\ V : \mathrm{Stream}\, A}\ \mathrm{Stream}R$$

Fig. 2: Type system of the uniform, (co)recursive abstract machine.

Encoding $\lambda$-terms in the abstract machine language

$$\begin{aligned}
v\, w &:= \mu\alpha.\langle v \| w \cdot \alpha \rangle \\
\mathrm{head}\, v &:= \mu\alpha.\langle v \| \mathrm{head}\, \alpha \rangle \\
\mathrm{tail}\, v &:= \mu\alpha.\langle v \| \mathrm{tail}\, \alpha \rangle \\
\mathbf{let}\, x = v\, \mathbf{in}\, w &:= \mu\alpha.\langle v \| \tilde{\mu} x.\langle w \| \alpha \rangle \rangle \\
\mathbf{rec}\, v\, \mathbf{as}\{\dots\} &:= \mu\alpha.\langle v \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \alpha \rangle
\end{aligned}$$

Evaluating computations in constructors and destructors:

$$\begin{aligned}
v \cdot e &:= \tilde{\mu} x.\langle v \| \tilde{\mu} y.\langle \mu\alpha.\langle x \| y \cdot \alpha \rangle \| e \rangle \rangle && (v \notin \mathit{Value} \text{ or } e \notin \mathit{CoValue}) \\
\mathrm{succ}\, v &:= \mu\alpha.\langle v \| \tilde{\mu} x.\langle \mathrm{succ}\, x \| \alpha \rangle \rangle && (v \notin \mathit{Value}) \\
\mathbf{rec}\{\dots\}\ \mathbf{with}\ e &:= \tilde{\mu} x.\langle \mu\alpha.\langle x \| \mathbf{rec}\{\dots\}\ \mathbf{with}\ \alpha \rangle \| e \rangle && (e \notin \mathit{CoValue}) \\
\mathbf{corec}\{\dots\}\ \mathbf{with}\ v &:= \mu\alpha.\langle v \| \tilde{\mu} x.\langle \mathbf{corec}\{\dots\}\ \mathbf{with}\ x \| \alpha \rangle \rangle && (v \notin \mathit{Value})
\end{aligned}$$

Fig. 3: Syntactic sugar in the abstract machine language.

The syntax and operational semantics of our (co)recursive abstract machine language are given in Fig. 1. Computation occurs as a reduction of machine commands ($c$), which are made up of a term ($v$) interacting with a coterm ($e$). Intuitively, terms correspond to the expressions of a $\lambda$-calculus-like language and coterms correspond to continuations (or

evaluation contexts) that arise during computation. Of note, the machine is *uniform* in the sense that it can express either call-by-value or call-by-name evaluation with the same form of operational rules. The only difference between the two evaluation strategies is in the definitions of *values* ($V$), which denote the terms that may be bound to and substituted for variables, and *covalues* ($E$) which correspond to evaluation contexts. For example, the right-hand sides of the $\beta_{\text{succ}}$ and $\beta_{\text{tail}}$ rules seem to have two possible reductions via $\mu$ or $\tilde{\mu}$, but the definition of $V$ and $E$ will only permit one of them. In call-by-name, a $\tilde{\mu}$-abstraction is never a covalue, so the next step will be a $\tilde{\mu}$-reduction that computes the coterm side first; for $\beta_{\text{succ}}$ calculating the updated covalue for the **with**-clause of recursion, and for $\beta_{\text{tail}}$ this means immediately unrolling the corecursive loop again. In call-by-value, a $\mu$-abstraction is never a value, so the next step will be a $\mu$-reduction that computes the term side first; for $\beta_{\text{succ}}$ this means immediately unrolling the recursive loop again, and for $\beta_{\text{tail}}$ this means calculating the updated value for the **with**-clause of corecursion.

The type system is given in Fig. 2. We use $FV$ to denote the set of free variables an expression refers to, *e.g.,* $FV(v : A)$ for the free variables in a term $v$, $FV(e \div A)$ for the free variables in a co-term $e$, and $FV(c)$ for the free variables in a command. $AV(\Gamma)$ denotes the set of variables that have been assigned a type by $\Gamma$, *e.g.,* $AV(x_i : A_i, \ldots, \alpha_j \div B_j, \ldots) = \{x_i, \ldots, \alpha_j, \ldots\}$. Besides the ordinary function type $A \to B$, the (co)recursive abstract machine includes the types Nat of natural numbers, serving as a canonical example of an inductive type, and Stream $A$ of infinite streams containing $A$ elements, serving as a canonical example of a coinductive type. Note that in the style of the sequent calculus (Curien & Herbelin, 2000; Downen & Ariola, 2018), the constructs of these types are divided between the term and coterm sides of a command. For example, we include the usual abstraction $\lambda x.v$ from the $\lambda$-calculus, but instead of application we build a *call stack* $V \cdot E$ which accepts a function of type $A \to B$ when $V$ produces an $A$ and $E$ consumes a $B$. Similarly for numbers, we include the constructors zero and succ for building values of Nat, which are consumed by a **rec** continuation corresponding to the System T's recursor (Gödel, 1980). Symmetrically for streams, we instead have the *destructors* head and tail for building covalues of Stream $A$, which project out of a **corec** value that corecursively builds a stream, on-demand, one piece at a time. To check the types of these (co)terms and validity of commands, we use a typing environment $\Gamma$ that describes both the variables $x$ and covariables $\alpha$ in scope that can be referenced, along with their types, written $x : A$ and $\alpha \div A$, respectively. These variables are considered *free* in the underlying (co)term and command expressions, and they are *assigned* a type by the environment $\Gamma$. Notice that we make the simplifying assumption throughout this paper that environments $\Gamma$ never assign types to the same (co)variable $x$ or $\alpha$ more than once (*i.e.,* every $x$ or $\alpha$ bound by a $\Gamma$ are distinct), ruling out cases like $x : \text{Nat}, y : \text{Nat}, x : \text{Nat} \to \text{Nat}$.

Since this abstract machine language doesn't have an application like the $\lambda$-calculus, how can it express basic compositions like $f(g(x))$? These sorts of terms can be encoded thanks to the $\mu$- and $\tilde{\mu}$-abstractions in the machine language. For example, $f(g(x))$ can be written

$$\mu\alpha.\langle \mu\beta.\langle g \| x \cdot \beta \rangle \| \tilde{\mu} z.\langle f \| z \cdot \alpha \rangle \rangle$$

where the outer $\mu$ assigns the name $\alpha$ to the surrounding calling context of $f$, and $\tilde{\mu}$ gives a name to the computation $g(x)$ and invokes $f$ with that name and the return point $\alpha$. More generally, we can use the syntactic sugar given in Fig. 3 as macro-definitions for

all the usual expressions of $\lambda$-calculi, including applications $(v\ w)$, using head and tail directly as projections, **let**-bindings, and the recursor as a term. Notice how each of these macro-definitions uses $\mu$ to name the current evaluation context $\alpha$, in order to build a larger continuation. But what happens if we want to use a non-value term $v$ in a context like $v \cdot e$ or succ $v$ which is not allowed by the syntax of Fig. 1? Again, we can use $\mu$ and $\tilde{\mu}$ to give a name to non-(co)value expressions and follow the syntactic restrictions of the abstract machine. These additional macro-expansions are also shown in Fig. 3.

*Example* 3.1. As pointed out above, the syntactic sugar might help in better grasping the (co)recursors; we present next how to define the *plus* and *iterate* functions seen in the previous section. The reader might consult (Downen & Ariola, 2023) for a detailed explanation of their use.

The *plus* function is defined as

$$\lambda x.\lambda z.\mu\alpha.\langle x \| \mathbf{rec}\{\text{zero} \to z \mid \text{succ}\ \_ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \alpha\rangle$$

Running $\langle \mathit{plus}\ 2\ 2 \| \beta\rangle$ (with $1 = \text{succ zero}$ and $2 = \text{succ } 1$) in call-by-value becomes:

$$\begin{aligned}
\langle \mathit{plus} \| 2 \cdot 2 \cdot \beta\rangle &\mapsto\!\!\!\!\to && (\beta_{\to}, \mu)\\
\langle 2 \| \mathbf{rec}\{\text{zero} \to 2 \mid \text{succ}\ _ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \beta\rangle &\mapsto && (\beta_{\text{succ}})\\
\langle \mu\alpha.\langle 1 \| \mathbf{rec}\{\text{zero} \to 2 \mid \text{succ}\ _ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \alpha\rangle \| \tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle\rangle &\mapsto && (\mu)\\
\langle 1 \| \mathbf{rec}\{\text{zero} \to 2 \mid \text{succ}\ _ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle\rangle &\mapsto && (\beta_{\text{succ}})\\
\langle \mu\alpha.\langle \text{zero} \| \mathbf{rec}\{\text{zero} \to 2 \mid \text{succ}\ _ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \alpha\rangle \| \tilde{\mu}z.\langle \text{succ}\ z \| \tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle\rangle\rangle &\mapsto && (\mu)\\
\langle \text{zero} \| \mathbf{rec}\{\text{zero} \to 2 \mid \text{succ}\ _ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \tilde{\mu}z.\langle \text{succ}\ z \| \tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle\rangle\rangle &\mapsto && (\beta_{\text{zero}})\\
\langle 2 \| \tilde{\mu}z.\langle \text{succ}\ z \| \tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle\rangle\rangle &\mapsto\!\!\!\!\to && (\tilde{\mu})\\
\langle \text{succ}(\text{succ}\ 2) \| \beta\rangle &&&
\end{aligned}$$

Notice how at each recursive step the continuation gets updated: first $\beta$, then $\tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle$, and finally $\tilde{\mu}z.\langle \text{succ}\ z \| \tilde{\mu}y.\langle \text{succ}\ y \| \beta\rangle\rangle$.

The *iterate* function is expressed as

$$\lambda f.\lambda x.\mu\alpha.\langle \mathbf{corec}\{\text{head}\ \alpha \to \alpha \mid \text{tail}\ \beta \to \gamma.\tilde{\mu}x.\langle f \| x \cdot \gamma\rangle\}\ \mathbf{with}\ x \| \alpha\rangle$$

If *add2* stands for the function $\lambda x.\mu\alpha.\langle \text{succ succ}\ x \| \alpha\rangle$ then the even natural numbers can be represented as $\mu\alpha.\langle \mathit{iterate} \| \mathit{add2} \cdot \text{zero} \cdot \alpha\rangle$, and the third element of this stream is computed in call-by-value as follows (where $\mathit{iter2} = \{\text{head}\ \alpha \to \alpha \mid \text{tail}\ \beta \to \gamma.\tilde{\mu}x.\langle \mathit{add2} \| x \cdot \gamma\rangle\}$):

$$\begin{aligned}
\langle \mu\alpha.\langle \mathit{iterate} \| \mathit{add2} \cdot \text{zero} \cdot \alpha\rangle \| \text{tail}(\text{tail}(\text{head}(\alpha)))\rangle &\mapsto\!\!\!\!\to && (\mu, \beta_{\to})\\
\langle \mathbf{corec}\{\text{head}\ \alpha \to \alpha \mid \text{tail}\ \beta \to \gamma.\tilde{\mu}x.\langle \mathit{add2} \| x \cdot \gamma\rangle\}\ \mathbf{with}\ \text{zero} \| \text{tail}(\text{tail}(\text{head}(\alpha)))\rangle &\mapsto && (\beta_{\text{tail}})\\
\langle \mu\gamma.\langle \text{zero} \| \tilde{\mu}x.\langle \mathit{add2} \| x \cdot \gamma\rangle\rangle \| \tilde{\mu}x.\langle \mathbf{corec}\ \mathit{iter2}\ \mathbf{with}\ x \| \text{tail}(\text{head}(\alpha))\rangle\rangle &\mapsto && (\mu)\\
\langle \text{zero} \| \tilde{\mu}x.\langle \mathit{add2} \| x \cdot \tilde{\mu}x'.\langle \mathbf{corec}\ \mathit{iter2}\ \mathbf{with}\ x' \| \text{tail}(\text{head}(\alpha))\rangle\rangle\rangle &\mapsto\!\!\!\!\to &&\\
\langle \mathbf{corec}\ \mathit{iter2}\ \mathbf{with}\ 2 \| \text{tail}(\text{head}(\alpha))\rangle &\mapsto && (\beta_{\text{tail}})\\
\langle \mu\gamma.\langle 2 \| \tilde{\mu}x.\langle \mathit{add2} \| x \cdot \gamma\rangle\rangle \| \tilde{\mu}x.\langle \mathbf{corec}\ \mathit{iter2}\ \mathbf{with}\ x \| \text{head}\ \alpha\rangle\rangle &\mapsto && (\mu)\\
\langle 2 \| \tilde{\mu}x.\langle \mathit{add2} \| x \cdot \tilde{\mu}x.'\langle \mathbf{corec}\ \mathit{iter2}\ \mathbf{with}\ x' \| \text{head}\ \alpha\rangle\rangle\rangle &\mapsto\!\!\!\!\to &&\\
\langle \mathbf{corec}\ \mathit{iter2}\ \mathbf{with}\ 4 \| \text{head}(\alpha)\rangle &\mapsto && (\beta_{\text{head}})\\
\langle 4 \| \alpha\rangle &&&
\end{aligned}$$

Notice how at each co-recursive step it is not the continuation that gets updated but the internal seed: zero, 2, and 4.

Equational properties ($\Phi$), environments ($\Gamma$), hypotheses ($\Delta$), and judgements ($J$):

$$\begin{array}{rll}
\mathit{Prop} \ni \Phi ::= & c = c' \mid v = v' : A \mid e = e' \div A & \\
\mathit{Env} \ni \Gamma ::= & \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A & \text{(all } x \text{ and } \alpha \text{ assigned in } \Gamma \text{ are distinct)} \\
\mathit{Hyp} \ni \Delta ::= & \bullet \mid \Delta, \Phi & \\
\mathit{Judge} \ni J ::= & \boxed{\Gamma \mid \Delta \vdash \Phi} & (FV(\Delta) \cup FV(\Phi) \subseteq AV(\Gamma))
\end{array}$$

Equivalence:

$$\frac{\Gamma \mid \Delta \vdash c = c'}{\Gamma \mid \Delta \vdash c' = c}\ \mathit{Symm} \qquad \frac{\Gamma \mid \Delta \vdash c = c' \quad \Gamma \mid \Delta \vdash c' = c''}{\Gamma \mid \Delta \vdash c = c''}\ \mathit{Trans}$$

Operational equality:

$$\frac{\Gamma \mid \Delta \vdash c = c' \quad c' \mapsto c''}{\Gamma \mid \Delta \vdash c = c''}\ \mathit{Red}$$

Congruence (mirror the typing rules from Fig. 2):

$$\frac{\Gamma \mid \Delta \vdash v = v' : A \quad \Gamma \mid \Delta \vdash e = e' \div A}{\Gamma \mid \Delta \vdash \langle v \| e \rangle = \langle v' \| e' \rangle}\ \mathit{Cut}$$

$$\frac{}{\Gamma, x : A \mid \Delta \vdash x = x : A}\ \mathit{VarR} \qquad \frac{}{\Gamma, \alpha \div A \mid \Delta \vdash \alpha = \alpha \div A}\ \mathit{VarL}$$

$$\frac{\Gamma, \alpha \div A \mid \Delta \vdash c = c'}{\Gamma \mid \Delta \vdash \mu\alpha.c = \mu\alpha.c' : A}\ \mathit{ActR} \qquad \frac{\Gamma, x : A \mid \Delta \vdash c = c'}{\Gamma \mid \Delta \vdash \tilde{\mu}x.c = \tilde{\mu}x.c' \div A}\ \mathit{ActL}$$

$$\frac{\Gamma, x : A \mid \Delta \vdash v = v' : B}{\Gamma \mid \Delta \vdash \lambda x.v = \lambda x.v' : A \to B}\ {\to}R \qquad \frac{\Gamma \mid \Delta \vdash V = V' : A \quad \Gamma \mid \Delta \vdash E = E' \div B}{\Gamma \mid \Delta \vdash V \cdot E = V' \cdot E' \div A \to B}\ {\to}L$$

$$\frac{}{\Gamma \mid \Delta \vdash \mathrm{zero} = \mathrm{zero} : \mathrm{Nat}}\ \mathrm{Nat}R_{\mathrm{zero}} \qquad \frac{\Gamma \mid \Delta \vdash V = V' : \mathrm{Nat}}{\Gamma \mid \Delta \vdash \mathrm{succ}\ V = \mathrm{succ}\ V' : \mathrm{Nat}}\ \mathrm{Nat}R_{\mathrm{succ}}$$

$$\frac{\Gamma \mid \Delta \vdash v = v' : A \quad \Gamma, x : \mathrm{Nat}, y : A \mid \Delta \vdash w = w' : A \quad \Gamma \mid \Delta \vdash E = E' \div A}{\Gamma \mid \Delta \vdash \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\ x \to y.w\}\ \mathbf{with}\ E = \mathbf{rec}\{\mathrm{zero} \to v' \mid \mathrm{succ}\ x \to y.w'\}\ \mathbf{with}\ E' \div \mathrm{Nat}}\ \mathrm{Nat}L$$

$$\frac{\Gamma \mid \Delta \vdash E = E' \div A}{\Gamma \mid \Delta \vdash \mathrm{head}\ E = \mathrm{head}\ E' \div \mathrm{Stream}\ A}\ \mathrm{Stream}L_{\mathrm{head}} \qquad \frac{\Gamma \mid \Delta \vdash E = E' \div \mathrm{Stream}\ A}{\Gamma \mid \Delta \vdash \mathrm{tail}\ E = \mathrm{tail}\ E' \div \mathrm{Stream}\ A}\ \mathrm{Stream}L_{\mathrm{tail}}$$

$$\frac{\Gamma, \alpha \div A \mid \Delta \vdash e = e' \div B \quad \Gamma, \beta \div \mathrm{Stream}\ A, \gamma \div B \mid \Delta \vdash f = f' \div B \quad \Gamma \mid \Delta \vdash V = V' : B}{\Gamma \mid \Delta \vdash \begin{array}{l}\mathbf{corec}\{\mathrm{head}\ \alpha \to e \\ \quad \mid \mathrm{tail}\ \beta \to \gamma.f\}\ \mathbf{with}\ V\end{array} = \begin{array}{l}\mathbf{corec}\{\mathrm{head}\ \alpha \to e' \\ \quad \mid \mathrm{tail}\ \beta \to \gamma.f'\}\ \mathbf{with}\ V'\end{array} : \mathrm{Stream}\ A}\ \mathrm{Stream}R$$

Fig. 4: Intensional equational theory of computation.

### 3.1 Intensional equational theory

The machine's operational semantics in Fig. 1 only allows us to apply the reduction steps ($c \mapsto c'$) to the top-level of the given command, and only ever forward: the multi-step reduction $c_1 \mapsto\!\!\!\!\to c_n$ combines several individual steps together, $c_1 \mapsto c_2 \mapsto c_3 \mapsto \ldots \mapsto c_n$, but requires that all the arrows are pointed in the same direction. These two restrictions make

the operational semantics *deterministic*: it always marches forward in one path because there is never more than one choice of step to take.

In contrast, an equational theory—for reasoning about when two programs, or fragments of programs, have the same observable result—gives us more freedom to relate programs that appear to have the same behavior. One of the key allowances is that the reduction steps can be applied both forwards and backwards; *i.e.,* equality is symmetric. The other is that we can apply the reduction steps in *any* context (no matter how deep within the given expression); *i.e.,* equality is congruent. Such an equational theory for the (co)recursive abstract machine is given in Fig. 4[1]. All judgments have the form $\Gamma \mid \Delta \vdash \Phi$, where $\Gamma$ contains the (unordered) type assignments to free (co)variables, $\Delta$ contains the (unordered) hypothesized properties involving those free (co)variables, and $\Phi$ is the property being proved. The main properties are the base equalities for commands ($c = c'$) and (co)terms of some type $A$ ($v = v' : A$ and $e = e' \div A$). For now, the hypotheses $\Delta$ do not yet interact with the inference rules — they will soon play a crucial role in Section 3.2 — so as shorthand, we will write $\Gamma \vdash \Phi$ instead of $\Gamma \mid \bullet \vdash \Phi$ in examples.

This equational theory is the smallest equivalence relation that includes the operational semantics, letting us reason about which programs are equal up to execution. As a result, it is more discriminating than a purely external observer, and can distinguish between two definitions with the same input-output behavior depending on the way they are defined. For this reason, this kind of equational theory is sometimes called *intensional* (because it lacks extensionality or any non-trivial mathematical reasoning) or *definitional* (because it is based on the definition of code).

For example, it is easy to show by reduction that $\langle plus \| \text{zero} \cdot x \cdot \alpha \rangle \mapsto\!\!\!\!\rightarrow \langle x \| \alpha \rangle$ because *plus* was defined by recursion on its first argument (see Section 2), and therefore we have the following equality:

$$\alpha \div \text{Nat}, x : \text{Nat} \vdash \langle plus \| \text{zero} \cdot x \cdot \alpha \rangle = \langle x \| \alpha \rangle$$

However, $\langle plus \| x \cdot \text{zero} \cdot \alpha \rangle$ doesn't reduce at all, even though it is nonetheless equivalent to $x$ in any context. Thus,

$$\alpha \div \text{Nat}, x : \text{Nat} \vdash \langle plus \| x \cdot \text{zero} \cdot \alpha \rangle = \langle x \| \alpha \rangle$$

is not provable. Likewise, *iterate* $(\lambda x.x)$ $x$ is observationally equivalent to the stream *always* $x$, but the intensional theory considers them different because their definitions are too different.

The bulk of the rules are dedicated to *congruence*: the allowance that equalities may be applied in *any* context. Though there are many different congruence rules to spell out (accounting for the many different contexts that may appear), they thankfully reflect exactly the same structure as the type system. Each typing rule from Fig. 2 for checking a single command, term, or coterm has a corresponding rule of the same name in Fig. 4 which just compares two such expressions hereditarily.

Note that reflexivity of well-typed commands, terms, and coterms is not included as an inference rule in Fig. 4 because it holds by performing an induction on the typing derivation, and applying the appropriate congruence rules. Still, in the following, we will sometimes

[1] $FV(\Phi)$ stands for the free variables in a proposition $\Phi$, and $FV(\Delta, \Phi)$ is defined as $FV(\Delta) \cup FV(\Phi)$ with $FV(\bullet) = \emptyset$

refer to these reflexivity rules:

$$\dfrac{\Gamma \vdash c}{\Gamma \mid \Delta \vdash c = c}\ \mathit{Refl} \qquad \dfrac{\Gamma \vdash v : A}{\Gamma \mid \Delta \vdash v = v : A}\ \mathit{ReflR} \qquad \dfrac{\Gamma \vdash e \div A}{\Gamma \mid \Delta \vdash e = e \div A}\ \mathit{ReflL}$$

where the vertical dots indicate the rule is derivable.

The *Red* rule states that any reduction step of the operational semantics can be added onto another equality. Together with reflexivity, we can say that any well-typed command $c$ is equal to its next step, $c \mapsto c'$:

$$\dfrac{\Gamma \vdash c \quad \vdots\ \mathit{Refl} \quad \Gamma \mid \Delta \vdash c = c \qquad c \mapsto c'}{\Gamma \mid \Delta \vdash c = c'}\ \mathit{Red}$$

and we will simply write

$$\dfrac{\Gamma \vdash c \qquad c \mapsto c' \quad \vdots\ \mathit{Red}}{\Gamma \mid \Delta \vdash c = c'}$$

Whereas a (unary) type system interprets the free variable $x : A$ (and analogously, $\alpha \div A$) as one unknown value of type $A$, a (binary) equational theory interprets the free $x : A$ as *two* unknown values which are equal at type $A$. More concretely, we can understand the meaning of free (co)variables in terms of the following notion that substitution commutes with equality—substitution of equals into equals are equals:

$$\dfrac{\Gamma, x : A \mid \Delta \vdash c = c' \qquad \Gamma \mid \Delta \vdash V = V' : A}{\Gamma \mid \Delta \vdash c[V/x] = c'[V'/x]}\ \mathit{SubstL} \qquad \dfrac{\Gamma, \alpha \div A \mid \Delta \vdash c = c' \qquad \Gamma \mid \Delta \vdash E = E' \div A}{\Gamma \mid \Delta \vdash c[E/\alpha] = c'[E'/\alpha]}\ \mathit{SubstR} \tag{3.1}$$

With the rules we already have, we can use reduction to derive substitution of equal values for variables like so:

$$\dfrac{\Gamma \mid \Delta \vdash V = V' : A \qquad \dfrac{\Gamma, x : A \mid \Delta \vdash c = c'}{\Gamma \mid \Delta \vdash \tilde{\mu} x.c = \tilde{\mu} x.c' \div A}\ \mathit{ActL}}{\dfrac{\dfrac{\Gamma \mid \Delta \vdash \langle V \| \tilde{\mu} x.c \rangle = \langle V' \| \tilde{\mu} x.c' \rangle}{\Gamma \mid \Delta \vdash \langle V' \| \tilde{\mu} x.c' \rangle = \langle V \| \tilde{\mu} x.c \rangle}\ \mathit{Symm} \qquad \langle V \| \tilde{\mu} x.c \rangle \mapsto_{\tilde{\mu}} c[V/x]}{\dfrac{\Gamma \mid \Delta \vdash \langle V' \| \tilde{\mu} x.c' \rangle = c[V/x]}{\Gamma \mid \Delta \vdash c[V/x] = \langle V' \| \tilde{\mu} x.c' \rangle}\ \mathit{Symm} \qquad \langle V' \| \tilde{\mu} x.c' \rangle \mapsto_{\tilde{\mu}} c'[V'/x]}\ \mathit{Red}}\ \mathit{Red}}\ \mathit{Cut}$$

And the derivation of the dual substitution of covalues for covariables follows analogously to the above, using the dual $\mu$ activation and operational steps.

*Example* 3.2. Consider this basic application of **corec** which just (corecursively) forwards all observations onto some underlying stream *xs*:

$$\mathit{parrot}\ \mathit{xs} := \mathbf{corec}\{\mathrm{head}\ \alpha \to \mathrm{head}\ \alpha \mid \mathrm{tail}\ \alpha \to \gamma.\ \mathrm{tail}\ \gamma\}\ \mathbf{with}\ \mathit{xs}$$

Intuitively, *parrot xs* produces a stream that produces exactly the same elements as *xs*. We can understand *parrot* at a higher level in terms of these equations that show how it reacts

to head and tail observations:

$$\langle parrot\ xs \| \text{head}\ \beta \rangle = \langle xs \| \text{head}\ \beta \rangle \qquad \langle parrot\ xs \| \text{tail}\ \alpha \rangle = \langle parrot\ (\text{tail}\ xs) \| \alpha \rangle$$

Both of these equalities are derivable from the intensional equational theory by just applying the operational rules (and expanding any syntactic sugar from Fig. 3 as necessary). The head case follows directly from the $\beta_{\text{head}}$ step:

$$\langle parrot\ xs \| \text{head}\ \beta \rangle \mapsto \langle xs \| \text{head}\ \beta \rangle \qquad (\beta_{\text{head}})$$

The tail case is slightly more involved because its reduction depends on whether the command is evaluated according to the call-by-name or call-by-value. In the call-by-name operational semantics, we have the forward reduction (where according to Fig. 3, tail $xs$ corresponds to $\mu\gamma.\langle xs \| \text{tail}\ \gamma \rangle$):

$$\begin{aligned} \langle parrot\ xs \| \text{tail}\ \alpha \rangle &\mapsto \langle \text{tail}\ xs \| \tilde{\mu} xs'.\langle parrot\ xs' \| \alpha \rangle \rangle && (\beta_{\text{tail}}) \\ &\mapsto \langle parrot\ (\text{tail}\ xs) \| \alpha \rangle && (\tilde{\mu}) \end{aligned}$$

In the call-by-value operational semantics, we have this conversion instead:

$$\begin{aligned} \langle parrot\ xs \| \text{tail}\ \alpha \rangle &\mapsto \langle \text{tail}\ xs \| \tilde{\mu} xs'.\langle parrot\ xs' \| \alpha \rangle \rangle && (\beta_{\text{tail}}) \\ &\leftarrow\!\mapsto \langle \mu\beta.\langle \text{tail}\ xs \| \tilde{\mu} xs'.\langle parrot\ xs' \| \beta \rangle \rangle \| \alpha \rangle && (\mu) \\ &:= \langle parrot\ (\text{tail}\ xs) \| \alpha \rangle && (\textit{Fig. 3}) \end{aligned}$$

### 3.2 Extensional program logic

It's often unsatisfactory to only consider two expressions equal when they reduce to some common reduct; that misses out on far too many equalities. Instead, we will enhance the intensional equational theory with a program logic that is *extensional*, in the sense that it considers expressions equal when they appear to be the same from the outside. This means we will have to add additional rules for saying when two terms (or two coterms) are equal because they cannot be distinguished by some observer. But which observer is that? The other side of the command! Terms are observed by coterms, and vice versa. Therefore, the idea of extensionality in the abstract machine comes down to the idea that (co)terms of *any* type are equal *if and only if* they always form equal computations when interacting with equal counterparts of that type. The extensional program logic is given in Fig. 5. The distinctive feature of this theory is that it is applicable to both call-by-name and call-by-value. That is why some properties needed to be restricted.

*Propositions* - $\Phi$*:* We enrich the language of properties that we are proving by internalizing the implicit "for all" generalization made by the free $x : A$ and $\alpha \div A$ in the environment in terms of an explicit $\forall$ quantifier in the syntax of propositions. In addition to the same three cases of equality as before, we now have two dual forms of universal quantification as properties: $\forall x{:}A.\Phi$ generalizes the property $\Phi$ over all choices of equal values of type $A$ for $x$, and $\forall \alpha \div A.\Phi$ generalizes $\Phi$ over all choices of equal covalues for $\alpha$ of type $A$. The rules governing these two $\forall$ properties are given in Fig. 5 as well.

Universal quantifiers can be introduced by *IntroL* and *IntroR*, which state that $\forall$ internalizes a free (co)variable in the environment, and eliminated by *ElimL* and *ElimR*. The

Equational properties ($\Phi$), environments ($\Gamma$), hypotheses ($\Delta$), and judgements ($J$)::

$$
\begin{array}{rll}
\mathit{Prop} \ni \Phi ::= & c = c' \mid v = v' : A \mid e = e' \div A \mid \forall x{:}A.\Phi \mid \forall \alpha \div A.\Phi & \\
\mathit{Env} \ni \Gamma ::= & \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A & \text{(all } x \text{ and } \alpha \text{ assigned in } \Gamma \text{ are distinct)} \\
\mathit{Hyp} \ni \Delta ::= & \bullet \mid \Delta, \Phi & \\
\mathit{Judge} \ni J ::= & \boxed{\Gamma \mid \Delta \vdash \Phi} & (FV(\Delta) \cup FV(\Phi) \subseteq AV(\Gamma))
\end{array}
$$

Equational properties strict on $x$ (written $\Psi(x)$) and productive on $\alpha$ (written $\Psi(\alpha)$):

$$
\begin{array}{rll}
\mathit{StrictProp} \ni \Psi(x) ::= & \langle x \| E \rangle = \langle x \| E' \rangle & (x \notin FV(E) \cup FV(E')) \\
& \mid \forall y{:}B.\Psi(x) \mid \forall \beta \div B.\Psi(x) & (x \neq y) \\
& \mid \Phi \Rightarrow \Psi(x) \mid \Psi_1(x) \wedge \Psi_2(x) & (x \notin FV(\Phi)) \\
\mathit{ProdProp} \ni \Psi(\alpha) ::= & \langle V \| \alpha \rangle = \langle V' \| \alpha \rangle & (\alpha \notin FV(V) \cup FV(V')) \\
& \mid \forall y{:}B.\Psi(\alpha) \mid \forall \beta \div B.\Psi(\alpha) & (\alpha \neq \beta) \\
& \mid \Phi \Rightarrow \Psi(\alpha) \mid \Psi_1(\alpha) \wedge \Psi_2(\alpha) & (\alpha \notin FV(\Phi))
\end{array}
$$

$$\frac{}{\Gamma \mid \Delta, \Phi \vdash \Phi}\ Ax \qquad \frac{\Gamma \mid \Delta, \Phi' \vdash \Phi}{\Gamma \mid \Delta \vdash \Phi' \Rightarrow \Phi}\ IntroH \qquad \frac{\Gamma \mid \Delta \vdash \Phi' \Rightarrow \Phi \quad \Gamma \mid \Delta \vdash \Phi'}{\Gamma \mid \Delta \vdash \Phi}\ Lemm$$

$$\frac{\Gamma, x : A \mid \Delta \vdash \Phi}{\Gamma \mid \Delta \vdash \forall x{:}A.\Phi}\ IntroL \qquad \frac{\Gamma \mid \Delta \vdash \forall x{:}A.\Phi \quad \Gamma \mid \Delta \vdash V = V' : A}{\Gamma \mid \Delta \vdash \Phi[V/x = V'/x]}\ ElimL$$

$$\frac{\Gamma, \alpha \div A \mid \Delta \vdash \Phi}{\Gamma \mid \Delta \vdash \forall \alpha \div A.\Phi}\ IntroR \qquad \frac{\Gamma \mid \Delta \vdash \forall \alpha \div A.\Phi \quad \Gamma \mid \Delta \vdash E = E' \div A}{\Gamma \mid \Delta \vdash \Phi[E/\alpha = E'/\alpha]}\ ElimR$$

$$\frac{\Gamma \mid \Delta \vdash \Phi_1 \quad \Gamma \mid \Delta \vdash \Phi_2}{\Gamma \mid \Delta \vdash \Phi_1 \wedge \Phi_2}\ ConjI \qquad \frac{\Gamma \mid \Delta \vdash \Phi_1 \wedge \Phi_2}{\Gamma \mid \Delta \vdash \Phi_1}\ ConjE_1 \qquad \frac{\Gamma \mid \Delta \vdash \Phi_1 \wedge \Phi_2}{\Gamma \mid \Delta \vdash \Phi_2}\ ConjE_2$$

$$\frac{\Gamma, x : A \mid \Delta \vdash \langle x \| e \rangle = \langle x \| e' \rangle}{\Gamma \mid \Delta \vdash e = e' \div A}\ \sigma\tilde{\mu} \qquad \frac{\Gamma, \alpha \div A \mid \Delta \vdash \langle v \| \alpha \rangle = \langle v' \| \alpha \rangle}{\Gamma \mid \Delta \vdash v = v' : A}\ \sigma\mu$$

$$\frac{\Gamma, x : A, \beta \div B \mid \Delta \vdash \Psi[x \cdot \beta/\alpha]}{\Gamma, \alpha \div A \to B \mid \Delta \vdash \Psi(\alpha)}\ \omega{\to}$$

$$\frac{\Gamma \mid \Delta \vdash \Psi[\mathrm{zero}/x] \quad \Gamma, x : \mathrm{Nat} \mid \Delta, \Psi(x) \vdash \Psi[\mathrm{succ}\, x/x]}{\Gamma, x : \mathrm{Nat} \mid \Delta \vdash \Psi(x)}\ \omega\mathrm{Nat}$$

$$\frac{\Gamma, \beta \div A \mid \Delta \vdash \Psi[\mathrm{head}\, \beta/\alpha] \quad \Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \Psi(\alpha) \vdash \Psi[\mathrm{tail}\, \alpha/\alpha]}{\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta \vdash \Psi(\alpha)}\ \omega\mathrm{Stream}$$

Plus all the intensional equality rules from Fig. 4

Fig. 5: Extensional program logic.

notation $\Phi[V/x = V'/x]$ (and likewise $\Phi[E/\alpha = E'/\alpha]$) means to perform the substitution $[V/x]$ on the left-hand side of the equation in $\Phi$ and $[V'/x]$ on the right-hand side. For example, the base cases of this substitution are when $\Phi$ is just an equality; for a command equality, this looks like:

$$(c = c')[V/x = V'/x] := (c[V/x]) = (c'[V'/x])$$
$$(c = c')[E/\alpha = E'/\alpha] := (c[E/\alpha]) = (c'[E'/\alpha])$$

*ElimL* and *ElimR* generalize the substitution rules previously derived in 3.1 to instantiate the quantified (co)variables by any equal (co)values of the appropriate type:

$$\begin{array}{cc} \Gamma, x:A \mid \Delta \vdash \Phi & \Gamma \mid \Delta \vdash V = V' : A \\ \vdots & \\ \Gamma \mid \Delta \vdash \Phi[V/x = V'/x] & \textit{SubstL} \end{array} \qquad \begin{array}{cc} \Gamma, \alpha \div A \mid \Delta \vdash \Phi & \Gamma \mid \Delta \vdash E = E' \div A \\ & \vdots \\ \Gamma \mid \Delta \vdash \Phi[E/\alpha = E'/\alpha] & \textit{SubstR} \end{array} \tag{3.2}$$

which are derived from the *Intro* and *Elim* rules like so:

$$\dfrac{\dfrac{\Gamma, x:A \mid \Delta \vdash \Phi}{\Gamma \mid \Delta \vdash \forall x{:}A.\Phi}\textit{IntroL} \quad \Gamma \mid \Delta \vdash V = V' : A}{\Gamma \mid \Delta \vdash \Phi[V/x = V'/x]}\textit{ElimL} \qquad \dfrac{\dfrac{\Gamma, \alpha \div A \mid \Delta \vdash \Phi}{\Gamma \mid \Delta \vdash \forall \alpha \div A.\Phi}\textit{IntroR} \quad \Gamma \mid \Delta \vdash E = E' \div A}{\Gamma \mid \Delta \vdash \Phi[E/\alpha = E'/\alpha]}\textit{ElimR}$$

A special case of *Elim* is to reverse the *Intro* rules:

$$\begin{array}{cc} \Gamma \mid \Delta \vdash \forall x{:}A.\Phi & \\ \vdots & \\ \Gamma, x:A \mid \Delta \vdash \Phi & \textit{ElimL}_x \end{array} \qquad \begin{array}{cc} \Gamma \mid \Delta \vdash \forall \alpha \div A.\Phi & \\ \vdots & \\ \Gamma, \alpha \div A \mid \Delta \vdash \Phi & \textit{ElimR}_\alpha \end{array} \tag{3.3}$$

These can be derived from the *Elim* and *Var* rules by *weakening* the premise (adding additional (co)variable type assignments which are never used).[2] $\textit{ElimL}_x$ is derived like so:

$$\dfrac{\begin{array}{c}\Gamma \mid \Delta \vdash \forall x{:}A.\Phi \\ \vdots \\ \Gamma, x:A \mid \Delta \vdash \forall x{:}A.\Phi\end{array} \quad \dfrac{}{\Gamma, x:A \mid \Delta \vdash x = x : A}\textit{VarR}}{\Gamma, x:A \mid \Delta \vdash \Phi}\textit{ElimL}$$

Similar to the quantifiers, we also have plain propositional implication, written $\Phi' \Rightarrow \Phi$, for stating that the truth of $\Phi'$ implies the truth of $\Phi$. The rules governing $\Phi' \Rightarrow \Phi$ are *IntroH*, which introduces an implication that internalizes a hypothesis in the environment, and *Lemm*, which lets us eliminate an implication by proving its hypothesis in the style of instantiating a lemma. While propositional implication is not strictly necessary for the kinds of simple equalities we have considered thus far, their addition makes it possible for us to explore some more complex forms of reasoning that can all be derived from the same rules of structural (co)induction. For the same reason, we also introduce propositional conjunction, written $\Phi_1 \wedge \Phi_2$, to describe the compositionality of (co)induction. Propositional conjunction is introduced and eliminated with the familiar *ConjI* and *ConjE* rules.

*Strict and productive propositions - $\Psi(x)$ and $\Psi(\alpha)$:* In some rules, we need to impose some constraints on the use of (co)variables. This is because we aim for the program logic to be applicable in both the call-by-value and call-by-name setting. This requires careful attention to avoid equating a value with a non-value, as well as ensuring the same distinction for co-values. These restrictions are defined syntactically, and approximate the two dual notions of control flow and data flow:

- A property $\Psi(x)$ is *strict on $x$* when it uses $x$ directly with some covalue on both sides of its underlying equality, with the base case of a strict property on $x$ being

[2] Weakening follows by an induction on the given typing derivation and allowing for a larger $\Gamma$ in each axiom.

$\langle x \| E \rangle = \langle x \| E' \rangle$ where $x$ is not free in $E$ or $E'$. Intuitively, $\Psi(x)$ is some property which observes $x$ exactly once with a covalue, since all covalues are strict, forcing their input to be computed first before they act.

- Dually, a property $\Psi(\alpha)$ is *productive on* $\alpha$ when it immediately returns a value to $\alpha$ on both side of its underlying equality, with the base case of a productive property on $\alpha$ being $\langle V \| \alpha \rangle = \langle V' \| \alpha \rangle$ where $\alpha$ is not free in $V$ or $V'$. Intuitively, $\Psi(\alpha)$ is some property which produces exactly one value to $\alpha$.

The $\sigma\mu$ and $\sigma\tilde{\mu}$ rules implement the idea of observational equality, we would like formal inference rules that embody these two relationships between different forms of equality.

- $\Gamma \mid \Delta \vdash v = v' : A$ *if and only if* $\Gamma \mid \Delta \vdash \langle v \| e \rangle = \langle v' \| e' \rangle$ for all $\Gamma \mid \Delta \vdash e = e' \div A$.
- $\Gamma \mid \Delta \vdash e = e' \div A$ *if and only if* $\Gamma \mid \Delta \vdash \langle v \| e \rangle = \langle v' \| e' \rangle$ for all $\Gamma \mid \Delta \vdash v = v' : A$.

Thankfully, the "only if" direction of both of these can be derived by the *Cut* congruence rule already present in the intensional equational theory (Fig. 4):

$$\frac{\Gamma \mid \Delta \vdash v = v' : A \quad \Gamma \mid \Delta \vdash e = e' \div A}{\Gamma \mid \Delta \vdash \langle v \| e \rangle = \langle v' \| e' \rangle}\ \mathit{Cut}$$

If we already know two terms $\Gamma \mid \Delta \vdash v = v' : A$ are equal, then for any other equal coterms $\Gamma \mid \Delta \vdash e = e' \div A$, *Cut* lets us conclude that their pointwise combination gives equal commands $\Gamma \mid \Delta \vdash \langle v \| e \rangle = \langle v' \| e' \rangle$. Dually, starting with two equal coterms, *Cut* lets us combine them with any equal terms to give equal commands. Whereas the "if" direction is implemented by the $\sigma\mu$ and $\sigma\tilde{\mu}$ rules, which establish a logical equivalence between equality of commands versus equality of (co)terms. These rules say that any two terms (dually coterms) are equal when they form equal commands when interacting with a generic covariable (dually variable). The $\sigma$ rules allow the derivation of the extensional $\eta$ rules for $\mu$ and $\tilde{\mu}$ (Herbelin, 2005):

$$\frac{\begin{array}{c}\Gamma \vdash v : A \\ \vdots\ \eta_\mu \end{array}}{\Gamma \mid \Delta \vdash \mu\alpha.\langle v \| \alpha \rangle = v : A} \qquad \frac{\begin{array}{c}\Gamma \vdash e \div A \\ \vdots\ \eta_{\tilde{\mu}} \end{array}}{\Gamma \mid \Delta \vdash \tilde{\mu}x.\langle x \| e \rangle = e \div A}$$

They can be derived from the $\sigma\mu$, $\sigma\tilde{\mu}$ inference rules and $\mu\tilde{\mu}$ reductions. $\eta_\mu$ is derived as:

$$\frac{\begin{array}{c}\Gamma \vdash v : A \\ \vdots\ \mathit{Cut}, \mathit{VarL}, \mathit{ActR} \\ \Gamma, \alpha \div A \vdash \langle \mu\alpha.\langle v \| \alpha \rangle \| \alpha \rangle \qquad \langle \mu\alpha.\langle v \| \alpha \rangle \| \alpha \rangle \mapsto_\mu \langle v \| \alpha \rangle \\ \vdots\ \mathit{Red} \end{array}}{\Gamma \mid \Delta \vdash \mu\alpha.\langle v \| \alpha \rangle = v : A}\ \sigma\mu$$

Analogously for $\eta_{\tilde{\mu}}$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash e \div A}{\vdots\ Cut, VarR, ActL}}{\Gamma, x : A \vdash \langle x \| \tilde{\mu} x.\langle x \| e \rangle \rangle \qquad \langle x \| \tilde{\mu} x.\langle x \| e \rangle \rangle \mapsto_{\tilde{\mu}} \langle x \| e \rangle}}{\vdots\ Red}}{\dfrac{\Gamma, x : A \mid \Delta \vdash \langle x \| \tilde{\mu} x.\langle x \| e \rangle \rangle = \langle x \| e \rangle}{\Gamma \mid \Delta \vdash \tilde{\mu} x.\langle x \| e \rangle = e \div A}\ \sigma\tilde{\mu}}$$

Note that the $\mu$ and $\tilde{\mu}$ reduction apply to both call-by-name and call-by-value since (co)-variables are considered values in these strategies. This means that the $\eta_{\mu}$ and $\eta_{\tilde{\mu}}$ axioms are sound in both semantics.

The $\omega\!\to$ rule expresses a form of extensionality for functions in terms of call stacks. It states that the only canonical covalue of type $A \to B$ has the form $V \cdot E$, and testing a property on a generic call stack $x \cdot \beta$ is sufficient to generalize that property over *all* $\alpha$ of type $A \to B$. The rule allows the derivation of the following $\eta$ axiom for functions (Curien & Herbelin, 2000):

$$\dfrac{\dfrac{\Gamma \vdash V : A \to B}{\vdots\ \eta_{\to}}}{\Gamma \mid \Delta \vdash \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle = V : A \to B}$$

which is equivalent to the familiar $\eta$ law of the $\lambda$-calculus, as it can be seen by macro-expanding the syntactic sugar for application according to Fig. 3: $\lambda x.(V\ x) = \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle =_{\eta_{\to}} V$. The derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash V : A \to B}{\vdots}}{\Gamma, y : A, \beta \div B \vdash \langle \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle \| y \cdot \beta \rangle \quad \langle \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle \| y \cdot \beta \rangle \twoheadrightarrow_{\beta_{\to}\mu} \langle V \| y \cdot \beta \rangle}}{\vdots\ Refl, Reds}}{\dfrac{\dfrac{\Gamma, y : A, \beta \div B \mid \Delta \vdash \langle \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle \| y \cdot \beta \rangle = \langle V \| y \cdot \beta \rangle}{\Gamma, \gamma \div A \to B \mid \Delta \vdash \langle \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle \| \gamma \rangle = \langle V \| \gamma \rangle}\ \sigma\mu}{\Gamma \mid \Delta \vdash \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle = V : A \to B}\ \omega\!\to}$$

Second from the bottom, $\omega\!\to$ can be applied to $\gamma \div A \to B$ because the equation $\langle \lambda x.\mu\alpha.\langle V \| x \cdot \alpha \rangle \| \gamma \rangle = \langle V \| \gamma \rangle$ is productive on $\gamma$; both sides of the equation immediately produce a syntactic value to $\gamma$.

*Remark* 3.3. If we relax the restriction on the $\omega\!\to$ rule and instead allow it for any property $\Phi$ (which we'll refer to as $\sigma\!\to$), then it would be possible to conclude that

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \gamma \div A \to B \mid \Delta \vdash \langle \lambda x.\mu\alpha.\langle v \| x \cdot \alpha \rangle \| \gamma \rangle = \langle v \| \gamma \rangle}\ \sigma\!\to}{\Gamma \mid \Delta \vdash \lambda x.\mu\alpha.\langle v \| x \cdot \alpha \rangle = v : A \to B}\ \sigma\mu}$$

The problem is that $\langle v \| \gamma \rangle$ may not produce a single value to $\gamma$—$v$ might throw $\gamma$ away, as shown in the example below.

$$\beta \div \mathrm{Nat}, \gamma \div A \to B \vdash \langle \lambda x.\mu\alpha.\langle \mu\delta.\langle \mathrm{zero} \| \beta \rangle \| x \cdot \alpha \rangle \| \gamma \rangle = \langle \mu\delta.\langle \mathrm{zero} \| \beta \rangle \| \gamma \rangle : A \to B$$

This equality is fine under call-by-name evaluation but is inconsistent under call-by-value, wherein not all covalues of function type have the form $\alpha$ or $x \cdot \alpha$. For example, the coterm

$\tilde{\mu}\_.\langle \text{succ zero} \| \alpha \rangle$ lets us observe the difference call-by-value evaluation makes between the two sides of the equation. On the left, we have

$$\langle \lambda x.\mu\alpha.\langle \mu\delta.\langle \text{zero} \| \beta \rangle \| x \cdot \alpha \rangle \| \tilde{\mu}\_.\langle \text{succ zero} \| \beta \rangle \rangle \mapsto\!\!\!\!\rightarrow \langle \text{succ zero} \| \beta \rangle$$

whereas on the right, we have

$$\langle \mu\delta.\langle \text{zero} \| \beta \rangle \| \tilde{\mu}\_.\langle \text{succ zero} \| \beta \rangle \rangle \mapsto\!\!\!\!\rightarrow \langle \text{zero} \| \beta \rangle$$

Therefore, we would be able to derive zero = succ zero : Nat using call-by-value evaluation, making the theory inconsistent.

This inconsistency in call-by-value should not be surprising. It is well known that unrestricted $\eta$ equivalence is unsound in the call-by-value $\lambda$-calculus with general recursion or side effects. The usual counter-example is that the term $\Omega = (\lambda x.x)\ (\lambda x.x)$ is observationally different from a $\lambda$-abstraction, but the $\eta$ law requires $\Omega = \lambda x.(\Omega\ x)$. Instead, the sound version of the call-by-value $\eta$ law only applies to values: $\lambda x.(V\ x) = V$.

The *induction* rule $\omega$Nat summarizes the following deduction for proving a property $\Psi$ over any number $x$ using an infinite number of premises:

$$\frac{\Gamma \mid \Delta \vdash \Psi[\text{zero}/x] \quad \Gamma \mid \Delta \vdash \Psi[\text{succ zero}/x] \quad \Gamma \mid \Delta \vdash \Psi[\text{succ}(\text{succ zero})/x] \quad \ldots}{\Gamma, x : \text{Nat} \mid \Delta \vdash \Psi}$$

This deduction is justified from the reasoning that zero, succ zero, succ(succ zero)—are *all the canonical values* of Nat; testing $\Psi$ on all of them is sufficient to generalize $\Psi$ over any $x$ of type Nat. $\omega$Nat uses the usual structure of primitive induction on the numbers to summarize this kind of argument in a finite form, and can be understood as an inference rule representing the usual axiom of induction:

$$\Psi(\text{zero}) \Rightarrow (\forall x{:}\text{Nat}.\Psi(x) \Rightarrow \Psi(\text{succ } x)) \Rightarrow (\forall x{:}\text{Nat}.\Psi(x)) \qquad (\omega\text{Nat})$$

Rather than listing a separate proof for each number, just start with a proof for zero specifically, and give a *transformation* from a proof of $\Psi$ on an arbitrary number to the next proof of $\Psi$ for the successor of that *same number*. Because this second step is a transformation, we first assume that the property $\Psi$ is true on a generic $x$ : Nat by placing $\Psi$ in the environment $\Gamma$ of other assumptions, with the intention that the assumed $\Psi$ in $\Gamma$ can be used to prove $\Psi$ with $x$ replaced by succ $x$.

As an example of the application of induction, we would like to prove the following *deep* extensionality axiom for "trivial" uses of recursion (where a _ stands for an unused variable):

$$(\delta_{\text{Nat}}) \qquad \forall\alpha \div \text{Nat}.\ \mathbf{rec} \left\{ \begin{array}{l} \text{zero} \to \text{zero} \\ \text{succ}\ \_ \to y.\ \text{succ}\ y \end{array} \right\} \mathbf{with}\ \alpha = \alpha \div \text{Nat}$$

The above is saying that any generic observer $\alpha$ cannot tell the difference if a natural number is first broken down and rebuilt from scratch from the base case (zero) up. Let us use the following shorthand:

$$\mathit{noop}\ \alpha := \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\ \_ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \alpha$$

and proceed as follows:

$$\dfrac{\dfrac{\dfrac{\alpha \div \mathrm{Nat} \vdash \quad \langle \mathrm{zero} \| noop\ \alpha \rangle = \langle \mathrm{zero} \| \alpha \rangle}{\alpha \div \mathrm{Nat}, x : \mathrm{Nat} \mid \langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle \vdash \langle \mathrm{succ}\ x \| noop\ \alpha \rangle = \langle \mathrm{succ}\ x \| \alpha \rangle}\ \omega\mathrm{Nat}}{\alpha \div \mathrm{Nat}, x : \mathrm{Nat} \vdash \langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle}\ \sigma\tilde{\mu}}{\alpha \div \mathrm{Nat} \vdash noop\ \alpha = \alpha}\ IntroR$$
$$\vdash \forall \alpha . noop\ \alpha = \alpha$$

We can apply the induction rule $\omega$Nat here because the property $\langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle$ is strict on $x$. This is evident because both sides of the equation observe $x$ with a covalue (**rec** ... **with** $\alpha$ on the left and $\alpha$ on the right) in both call-by-name and -value.

We can just evaluate the left-hand-side to prove the base case:

$$\langle \mathrm{zero} \| noop\ \alpha \rangle = \langle \mathrm{zero} \| \alpha \rangle$$

What remains is to show $\langle \mathrm{succ}\ x \| noop\ \alpha \rangle = \langle \mathrm{succ}\ x \| \alpha \rangle$ from the inductive hypothesis (IH) $\langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle$. We would like to put together the following equality:

$$\begin{aligned} \langle \mathrm{succ}\ x \| noop\ \alpha \rangle &= \langle \mu\beta . \langle x \| noop\ \beta \rangle \| \tilde{\mu} y . \langle \mathrm{succ}\ y \| \alpha \rangle \rangle \\ &= \langle \mu\beta . \langle x \| \beta \rangle \| \tilde{\mu} y . \langle \mathrm{succ}\ y \| \alpha \rangle \rangle && (IH?) \\ &= \langle \mathrm{succ}\ x \| \alpha \rangle && (\eta_\mu, \tilde{\mu}) \end{aligned}$$

The problem is that the induction hypothesis holds only for $\alpha$, but we now need to apply it in a different context. This requires a generalization of the context:

$$\dfrac{\dfrac{\dfrac{\vdash \forall \alpha \div \mathrm{Nat} . \quad \langle \mathrm{zero} \| noop\ \alpha \rangle = \langle \mathrm{zero} \| \alpha \rangle}{x : \mathrm{Nat} \mid \forall \alpha \div \mathrm{Nat} . \langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle \vdash \forall \alpha \div \mathrm{Nat} . \langle \mathrm{succ}\ x \| noop\ \alpha \rangle = \langle \mathrm{succ}\ x \| \alpha \rangle}\ \omega\mathrm{Nat}}{x : \mathrm{Nat} \vdash \forall \alpha \div \mathrm{Nat} . \langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle}}{\vdots\ ElimR_\alpha}$$
$$\dfrac{\alpha \div \mathrm{Nat}, x : \mathrm{Nat} \vdash \langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle}{\alpha \div \mathrm{Nat} \vdash noop\ \alpha = \alpha}\ \sigma\tilde{\mu}\ \dfrac{}{\vdash \forall \alpha \div \mathrm{Nat} . noop\ \alpha = \alpha}\ IntroR$$

Note that the $\omega$ Nat rule can still be applied since quantifying over a strict property gives another strict property. Now we can instantiate the inductive hypothesis

$$\forall \alpha \div \mathrm{Nat} . \langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle$$

to the new context $\beta$: $\langle x \| noop\ \beta \rangle = \langle x \| \beta \rangle$.

The need to generalize over the context does not show up when one does inductive proofs in $\lambda$-calculus since the context is left implicit. In fact, let's go back to the proof of Example Theorem 2.1. Notice how we applied the inductive hypothesis not at the top level, which we can represent as $\Box$, but in the bigger context succ $\Box$. The inductive hypothesis should be better expressed as: $\forall C[\Box], C[plus\ x'\ \mathrm{zero}] = C[x']$.

Analogously, we can also prove the following "shallow" extensionality property $\eta$Nat that just look at the outermost structure of a numeric value or a stream projection:

$$(\eta_{\mathrm{Nat}}) \qquad \forall \alpha \div \mathrm{Nat}.\ \textbf{rec} \left\{ \begin{array}{l} \mathrm{zero} \to \mathrm{zero} \\ \mathrm{succ}\ y \to \_ . \mathrm{succ}\ y \end{array} \right\} \textbf{with}\ \alpha = \alpha \div \mathrm{Nat}$$

*Remark* 3.4. Note that the property $\forall\alpha.\langle x \| \mathit{noop}\ \alpha\rangle = \langle x \| \alpha\rangle$ is strict in $x$, since the recursor is a co-value in both call-by-value and call-by-name. If we remove that restriction and apply induction on an arbitrary proposition $\Phi$ we can derive inconsistent equations under call-by-name because it can equate any coterm $e \div \mathrm{Nat}$ with a **rec** covalue, whether or not $e$ itself is a covalue. For example, we would be able to prove this property:

$$\alpha \div \mathrm{Nat}, x : \mathrm{Nat} \vdash \langle x \| \mathbf{rec}\{\mathrm{zero} \to \mathrm{zero} \mid \mathrm{succ}\ \_ \to \mathrm{zero}\}\ \mathbf{with}\ \alpha\rangle = \langle \mathrm{zero} \| \alpha\rangle$$

The call-by-name version of the derived *SubstL* rule, see (3.1) and (3.2), allows for the call-by-name value $\mu\_.\langle \mathrm{succ\ zero} \| \alpha\rangle$ to be substituted for $x$ in this equation, leading to the inconsistent equality $\alpha \div \mathrm{Nat} \vdash \langle \mathrm{succ\ zero} \| \alpha\rangle = \langle \mathrm{zero} \| \alpha\rangle$ .

The *coinduction* rule $\omega$Stream specifies a form of structural coinduction for streams. It works in exactly the same way as structural induction for numbers—just with the roles of values and covalues reversed. The $\omega$Stream rule summarizes this deduction for proving a property $\Psi$ over any stream projection $\alpha$ using an infinite number of premises:

$$\frac{\Gamma, \beta \div A \vdash \Psi[\mathrm{head}\ \beta/\alpha] \quad \Gamma, \beta \div A \vdash \Psi[\mathrm{tail}(\mathrm{head}\ \beta)/\alpha] \quad \Gamma, \beta \div A \vdash \Psi[\mathrm{tail}(\mathrm{tail}(\mathrm{head}\ \beta))/\alpha] \quad \ldots}{\Gamma, \alpha \div \mathrm{Stream}\ A \vdash \Psi}$$

This deduction is justified by the reasoning that the listed projections—head $\beta$, tail(head $\beta$), tail(tail(head $\beta$))—cover *all the canonical covalues* of Stream $A$; testing $\Psi$ on all of them is sufficient to generalize $\Psi$ over any generic $\alpha$ of type Stream $A$. $\omega$Stream summarizes this kind of argument in a finite form, avoiding the list of separate proofs for each of the infinitely possible projections. Whereas $\omega$Nat corresponds to the usual induction axiom for the natural numbers, the $\omega$Stream rule corresponds to the dual form of the coinduction axiom for proving a property holds for all observations of infinite streams in both call-by-name and call-by-value:

$$\begin{aligned} & (\forall\beta \div A.\Psi(\mathrm{head}\ \beta)) \\ \Rightarrow\ & (\forall\alpha \div \mathrm{Stream}\ A.\Psi(\alpha) \Rightarrow \Psi(\mathrm{tail}\ \alpha)) && (\omega\mathrm{Stream}) \\ \Rightarrow\ & (\forall\alpha \div \mathrm{Stream}\ A.\Psi(\alpha)) \end{aligned}$$

Dual to induction on the numbers, we start with a proof for head $\beta$ specifically, and give a *transformation* from a proof of $\Psi$ on an arbitrary observation on streams to the next proof of $\Psi$ for the *same observation* on the tail of the stream. As before, this transformation is represented by assuming $\Psi$ holds for a generic $\alpha \div \mathrm{Stream}\ A$ by listing it in the environment $\Gamma$, which can then be used to derive a proof of $\Psi$ with $\alpha$ replaced by tail $\alpha$.

As an example of application of co-induction, we would like to prove the following *deep* extensional property of streams, which is the dual of $\delta_{\mathrm{Nat}}$:

$$(\delta_{\mathrm{Stream}}) \qquad \forall xs{:}\mathrm{Stream}\ A.\ \mathbf{corec} \left\{ \begin{array}{l} \mathrm{head}\ \alpha \to \mathrm{head}\ \alpha \\ \mathrm{tail}\ \_ \to \beta.\ \mathrm{tail}\ \beta \end{array} \right\} \mathbf{with}\ xs = xs : \mathrm{Stream}\ A$$

The above is saying that any generic stream $xs$ gives the same response when its projections are broken down and rebuilt from scratch from the base case (head $\alpha$) up. We should be able to apply the rules from Fig. 5 to prove it.

By using the shorthand $\mathit{parrot}\ xs := \mathbf{corec}\{\text{head}\ \alpha \to \text{head}\ \alpha \mid \text{tail}\ \alpha \to \gamma.\ \text{tail}\ \gamma\}\ \mathbf{with}\ xs$ as in Example 3.2, the bottom of the derivation starts like this:

$$\dfrac{\dfrac{xs : \text{Stream}\,A,\ \alpha \div \text{Stream}\,A \mid \langle \mathit{parrot}\ xs \| \alpha\rangle = \langle xs \| \alpha\rangle \vdash \quad \langle \mathit{parrot}\ xs \| \text{tail}\ \alpha\rangle = \langle xs \| \text{tail}\ \alpha\rangle}{\dfrac{xs : \text{Stream}\,A,\ \alpha \div \text{Stream}\,A \vdash \langle \mathit{parrot}\ xs \| \alpha\rangle = \langle xs \| \alpha\rangle}{xs : \text{Stream}\,A \vdash \mathit{parrot}\ xs = xs : \text{Stream}\,A}\ \mathit{IntroL}}\ \sigma\mu}{\vdash \forall xs : \text{Stream}\,A.\mathit{parrot}\ xs = xs : \text{Stream}\,A}\ \omega\text{Stream}$$

We begin by assuming some generic stream value $xs : \text{Stream}\,A$ is in scope. The first step (from the bottom up) applies $\sigma\mu$ to generalize equality of terms to an equality of commands, by introducing a generic continuation $\alpha \div \text{Stream}\,A$ expecting a stream. From here, we can apply the $\omega$Stream coinductive rule since we invoke $\alpha$ with a value. We continue with two proof obligations:

1. Show $\langle \mathit{parrot}\ xs \| \text{head}\ \beta\rangle = \langle xs \| \text{head}\ \beta\rangle$.
2. Show $\langle \mathit{parrot}\ xs \| \text{tail}\ \alpha\rangle = \langle xs \| \text{tail}\ \alpha\rangle$ follows from the coinductive hypothesis (CIH) $\langle \mathit{parrot}\ xs \| \alpha\rangle = \langle xs \| \alpha\rangle$.

Step 1 follows directly from $\beta_{\text{head}}$, as shown in Example 3.2. Step 2 proceeds as follows:

$$\begin{aligned}
\langle \mathit{parrot}\ xs \| \text{tail}\ \alpha\rangle &= \langle \mathit{parrot}\ (\text{tail}\ xs) \| \alpha\rangle && (\beta_{\text{tail}}\mu\tilde{\mu}) \\
&= \langle \text{tail}\ xs \| \alpha\rangle && (\mathit{CIH}?) \\
&= \langle xs \| \text{tail}\ \alpha\rangle && (\mu)
\end{aligned}$$

The coinductive hypothesis does not apply in the middle step, because it is already fixed for some previously-chosen $xs$, which is not the same as (tail $xs$) used here. What we need is the ability to *generalize the coinductive hypothesis*. Rather than introducing a generic stream $xs$ first and then applying coinduction, we should apply coinduction to prove an equality holds for all choices of $xs$, as shown below:

$$\dfrac{\dfrac{\alpha \div \text{Stream}\,A \mid \forall xs.\langle \mathit{parrot}\ xs \| \alpha\rangle = \langle xs \| \alpha\rangle \vdash \forall xs{:}\text{Stream}\,A.\ \ \langle \mathit{parrot}\ xs \| \text{tail}\ \alpha\rangle = \langle xs \| \text{tail}\ \alpha\rangle}{\dfrac{\alpha \div \text{Stream}\,A \vdash \forall xs{:}\text{Stream}\,A.\langle \mathit{parrot}\ xs \| \alpha\rangle = \langle xs \| \alpha\rangle}{\vdots\ \mathit{ElimL}_{xs}}}{\dfrac{xs : \text{Stream}\,A,\ \alpha \div \text{Stream}\,A \vdash \langle \mathit{parrot}\ xs \| \alpha\rangle = \langle xs \| \alpha\rangle}{xs : \text{Stream}\,A \vdash \mathit{parrot}\ xs = xs : \text{Stream}\,A}\ \mathit{IntroL}}\ \sigma\mu}{\vdash \forall xs{:}\text{Stream}\,A.\mathit{parrot}\ xs = xs : \text{Stream}\,A}\ \omega\ \text{Stream}$$

The base case is as before. For the co-inductive case, we have the following calculation in call-by-value and -name:

$$\begin{aligned}
&\langle \mathit{parrot}\ xs \| \text{tail}\ \alpha\rangle \\
&\mapsto \langle \mu\beta.\langle xs \| \text{tail}\ \beta\rangle \| \tilde{\mu}y.\langle \mathit{parrot}\ y \| \alpha\rangle\rangle && (\beta_{\text{tail}}) \\
&= \langle \mu\beta.\langle xs \| \text{tail}\ \beta\rangle \| \tilde{\mu}y.\langle y \| \alpha\rangle\rangle && (\mathit{CIH}[y/x]) \\
&= \langle \mu\beta.\langle xs \| \text{tail}\ \beta\rangle \| \alpha\rangle && (\eta_{\tilde{\mu}}) \\
&\mapsto \langle xs \| \text{tail}\ \alpha\rangle && (\mu)
\end{aligned}$$

Note that the generalization over $xs$ in the coinductive hypothesis is essential for instantiating $xs$ with the bound $y$ newly introduced by $\beta_{\text{tail}}$ reduction.

As we did for Nat, we can also derive the following "shallow" extensionality property that just look at the outermost structure of a stream projection:

$$(\eta_{\mathrm{Stream}}) \qquad \forall x{:}\mathrm{Stream}\,A.\ \mathbf{corec}\left\{\begin{array}{l}\mathrm{head}\,\alpha \to \mathrm{head}\,\alpha \\ \mathrm{tail}\,\beta \to \_.\,\mathrm{tail}\,\beta\end{array}\right\}\ \mathbf{with}\ x = x : \mathrm{Stream}\,A$$

*Remark* 3.5. The coinductive $\omega$Stream rule is similar in spirit to $\omega{\to}$: it matches over the possible shapes of a generic covalue $\alpha \div \mathrm{Stream}\,A$ in scope. The problem is that in call-by-value there are more values than the ones we considered. If we relax the restriction of productivity and allow the application of the rule to a generic proposition $\Phi$ we would then prove:

$$\alpha \div \mathrm{Nat}, \beta \div \mathrm{Stream}\,A \vdash \langle \mathbf{corec}\{\mathrm{head}\,\_{\to}\alpha \mid \mathrm{tail}\,\_{\to}\alpha.\alpha\}\ \mathbf{with}\ \mathrm{zero} \| \beta\rangle = \langle \mathrm{zero} \| \alpha\rangle$$

and yet the call-by-value version of the derived *SubstR* rule, see (3.1) and (3.2), lets us substitute $\tilde{\mu}\_.\langle \mathrm{succ}\,\mathrm{zero} \| \alpha\rangle$ as a covalue for $\beta$, leading to an inconsistent equality:

$$\langle \mathbf{corec}\{\mathrm{head}\,\_{\to}\alpha \mid \mathrm{tail}\,\_{\to}\alpha.\alpha\}\ \mathbf{with}\ \mathrm{zero} \| \tilde{\mu}\_.\langle \mathrm{succ}\,\mathrm{zero} \| \alpha\rangle\rangle = \langle \mathrm{succ}\,\mathrm{zero} \| \alpha\rangle = \langle \mathrm{zero} \| \alpha\rangle$$

### *3.3 Consistency of the extensional program logic*

Ultimately, the program logic is not useful if it derives inconsistent results. One very simplistic version of consistency is that 0 is different from any successor (like 1); and dually, we should also know that a head projection is different from a tail projection.

**Definition 3.6** (Consistency)**.** An equational theory or program logic for the (co)recursive abstract machine is *consistent* iff the following equalities are *not* derivable:

- $\vdash \mathrm{zero} = \mathrm{succ}\,V : \mathrm{Nat}$, and
- $\vdash \mathrm{head}\,E = \mathrm{tail}\,E' \div \mathrm{Stream}\,A$.

As with most systems, equating 0 and 1 collapses the notion of equality. Assuming $\mathrm{zero} = \mathrm{succ}\,\mathrm{zero} : \mathrm{Nat}$ lets us prove that any two terms $v$ and $w$ of type $A$ are equal by abstracting over the output $\alpha \div A$ in this derivation with $\sigma\mu$:

$$\begin{array}{lll}\langle v \| \alpha\rangle &= \langle \mathrm{zero} \| \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\,\_ \to \_.w\}\ \mathbf{with}\ \alpha\rangle & (\beta_{\mathrm{zero}}) \\ &= \langle \mathrm{succ}\,\mathrm{zero} \| \mathbf{rec}\{\mathrm{zero} \to v \mid \mathrm{succ}\,\_ \to \_.w\}\ \mathbf{with}\ \alpha\rangle & (\mathrm{zero} = \mathrm{succ}\,\mathrm{zero}) \\ &= \langle w \| \alpha\rangle & (\beta_{\mathrm{succ}}\mu\tilde{\mu})\end{array}$$

This forces every $v = w : A$ to hold, which we can use to equate any two commands and any two coterms of the same type, as well. Likewise, equating the head and tail projections leads to the same collapse, due to a similar derivation. Assuming $\mathrm{head}\,\alpha = \mathrm{tail}(\mathrm{head}\,\alpha)$, we can prove any two coterms $e$ and $f$ of type $A$ are equal by abstracting over the input $x : A$ via $\sigma\tilde{\mu}$ in this derivation:

$$\begin{array}{lll}\langle x \| e\rangle &= \langle \mathbf{corec}\{\mathrm{head}\,\_ \to e \mid \mathrm{tail}\,\_ \to \_.f\}\ \mathbf{with}\ x \| \mathrm{head}\,\alpha\rangle & (\beta_{\mathrm{head}}) \\ &= \langle \mathbf{corec}\{\mathrm{head}\,\_ \to e \mid \mathrm{tail}\,\_ \to \_.f\}\ \mathbf{with}\ x \| \mathrm{tail}(\mathrm{head}\,\alpha)\rangle & (\mathrm{head}\,\alpha = \mathrm{tail}(\mathrm{head}\,\alpha)) \\ &= \langle x \| f\rangle & (\beta_{\mathrm{tail}}\mu\tilde{\mu})\end{array}$$

$$\frac{\Gamma, x : A, \beta \div B \mid \Delta \vdash \Phi[x \cdot \beta / \alpha]}{\Gamma, \alpha \div A \to B \mid \Delta \vdash \Phi}\ \sigma\to$$

$$\frac{\Gamma, \beta \div A \mid \Delta \vdash \Phi[\mathrm{head}\ \beta / \alpha] \quad \Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \Phi \vdash \Phi[\mathrm{tail}\ \alpha / \alpha]}{\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta \vdash \Phi}\ \sigma\mathrm{Stream}$$

$$\frac{\Gamma \mid \Delta \vdash \Phi[\mathrm{zero}/x] \quad \Gamma, x : \mathrm{Nat} \mid \Delta, \Phi \vdash \Phi[\mathrm{succ}\ x/x]}{\Gamma, x : \mathrm{Nat} \mid \Delta \vdash \Phi}\ \sigma\mathrm{Nat}$$

Fig. 6: Unrestricted coinduction rules $\sigma\to$, $\sigma$Stream, and unrestricted induction rule $\sigma$Nat

By restricting induction to only apply to strict properties, and restricting coinduction to only productive properties, we get a single extensional program logic (parameterized by the definition of values and covalues) that is consistent in *both* call-by-value and call-by-name evaluation. See Section 5 for the proof of consistency.

**Theorem 3.7.** *The extensional program logic in Fig. 5 is consistent for both the call-by-name and call-by-value semantics.*

### *3.4 When is unrestricted (co)induction sound?*

Common folklore says that induction holds only in call-by-value, and thus dually coinduction should hold only in call-by-name. This is reflected in part through the restrictions defining strict versus productive properties in the "universally" sound extensional program logic given in Fig. 5. Call-by-value has a more permissive notion of covalue (any coterm is a call-by-value covalue), so the induction principle $\omega$Nat applies to more properties in the call-by-value logic than in the call-by-name one. Symmetrically, call-by-name has a more permissive notion of value (any term is a call-by-name value), so the coinduction principle $\omega$Stream applies to more properties in call-by-name than in call-by-value. In Fig. 6 we give the unrestricted reasoning rules.

For non-recursive types like $A \to B$ the full power of $\sigma\to$ can be recovered from the weaker $\omega\to$ in the right setting. In call-by-name, the productivity restriction of $\omega\to$ is not important since any term can be a value, and we break down any property to apply $\omega\to$ at the root. However, this difference in power between (co)induction in the two semantics is not quite enough to account for the true strength of call-by-value induction and call-by-name coinduction, because the strategy of breaking down the property in advance *weakens* the (co)inductive hypothesis. As a consequence, the fact that the sub-syntax of strict and productive properties includes $\forall$ quantifiers but *not* implications ($\Phi' \Rightarrow \Phi$) of any form means that choosing the "best" semantics still does not fully restore $\omega$Nat to $\sigma$Nat or $\omega$Stream to $\sigma$Stream.

This essential difference in reasoning power raises the question: are the unrestricted induction and coinduction principles ever safe? Thankfully, it turns out that the full (co)-induction rules $\sigma$Nat and $\sigma$Stream can be consistently added to the program logic, even in the presence of computational effects like first-class control, under the correct evaluation strategy (see Section 5).

**Definition 3.8** (Strong Program Logics)**.** The two strong program logics, which generalize the common extensional program logic (Fig. 5) with additional sound rules specifically for call-by-name and call-by-value reduction are:

- The *strong call-by-name program logic* extends the call-by-name instance of Fig. 5 with the $\sigma$Stream and $\sigma\to$ rules of coinduction from Fig. 6.
- The *strong call-by-value program logic* extends the call-by-value instance of Fig. 5 with the $\sigma$Nat rule of induction from Fig. 6.

**Theorem 3.9.** *The strong call-by-name and call-by-value program logics are consistent.*

## 4 The Strength of Strong (Co)Induction

Due to the lack of propositional implication, there are certain forms of inductive reasoning (for example, "strong" induction on the numbers) that are possible using $\sigma$Nat with a property $\Phi' \Rightarrow \Phi$ that cannot be derived from the $\omega$Nat—even in call-by-value. Likewise, there are certain forms of coinductive reasoning (for example, bisimulation) that are possible with $\sigma$Stream but cannot be derived from $\omega$Stream—even in call-by-name.

Next, we will explore the strength of full $\sigma$Nat and $\sigma$Stream versus the weaker $\omega$Nat and $\omega$Stream, and the use of structural (co)induction for encoding several different reasoning principles for (co)inductive types.

### *Compositionality of weak mutual (co)induction*

Before we get to the full strength of strong structural (co)induction, consider an example of what can be done with just the weak version all on its own. Mutual induction lets us prove two properties at the same time, where the correctness of each one depends simultaneously on the other. To prove $\Psi_1(x)$ and $\Psi_2(x)$ for all natural numbers $x$, there are two inductive cases: one showing $\Psi_1(\operatorname{succ} x)$ and the other showing $\Psi_2(\operatorname{succ} x)$. The two cases can be proved separately from one another, but each one gets to assume *both* inductive hypotheses $\Psi_1(x)$ and $\Psi_2(x)$ hold. This principle is especially useful for *generalizing the inductive hypotheses* in situations where we are only interested in $\Psi_1(x)$ at the end, but the proof of $\Psi_1(x)$ requires additional knowledge about $\Psi_2(x)$ during the inductive step.

This mutual induction reasoning principle can be derived by applying the weak induction rule $\omega$Nat on the conjunction $\Psi_1(x) \wedge \Psi_2(x)$ *first*, before splitting the two apart like so:

$$\dfrac{\dfrac{\Gamma \mid \Delta \vdash \Psi_1[\text{zero}/x] \quad \Gamma \mid \Delta \vdash \Psi_2[\text{zero}/x]}{\Gamma \mid \Delta \vdash \Psi_1[\text{zero}/x] \wedge \Psi_2[\text{zero}/x]}\,\mathit{ConjI} \qquad \dfrac{\begin{array}{c}\Gamma, x : \text{Nat} \mid \Delta, \Psi_1(x) \wedge \Psi_2(x) \vdash \Psi_1[\text{succ}\, x/x] \\ \Gamma, x : \text{Nat} \mid \Delta, \Psi_1(x) \wedge \Psi_2(x) \vdash \Psi_2[\text{succ}\, x/x]\end{array}}{\Gamma, x : \text{Nat} \mid \Delta, \Psi_1(x) \wedge \Psi_2(x) \vdash \Psi_1[\text{succ}\, x/x] \wedge \Psi_2[\text{succ}\, x/x]}\,\mathit{ConjI}}{\Gamma, x : \text{Nat} \mid \Delta \vdash \Psi_1(x) \wedge \Psi_2(x)}\,\omega\text{Nat}$$

This application of the weak $\omega$Nat is allowed because the conjunction of two strict properties $\Psi_1(x) \wedge \Psi_2(x)$ is also strict on $x$.

Since the rules for induction and coinduction mirror each other, we can encode mutual (weak) coinduction on streams in the exact same way using the $\omega$Stream and *ConjI* rules.

This mutual weak coinduction rule looks like:

$$\dfrac{\dfrac{\Gamma, \beta \div A \mid \Delta \vdash \Psi_1[\text{head}\,\beta/\alpha] \quad \Gamma, \beta \div A \mid \Delta \vdash \Psi_2[\text{head}\,\beta/\alpha]}{\Gamma, \beta \div A \mid \Delta \vdash (\Psi_1 \wedge \Psi_2)[\text{head}\,\beta/\alpha]}\,\mathit{ConjI} \quad \dfrac{\Gamma, \alpha \div \text{Stream}\,A \mid \Delta, \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash \Psi_1[\text{tail}\,\alpha/\alpha] \quad \Gamma, \alpha \div \text{Stream}\,A \mid \Delta, \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash \Psi_2[\text{tail}\,\alpha/\alpha]}{\Gamma, \alpha \div \text{Stream}\,A \mid \Delta, \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash (\Psi_1 \wedge \Psi_2)[\text{tail}\,\alpha/\alpha]}\,\mathit{ConjI}}{\Gamma, \alpha \div \text{Stream}\,A \mid \Delta \vdash \Psi_1(\alpha) \wedge \Psi_2(\alpha)}\,\omega\text{Stream}$$

For example, we can apply this rule to formalize our previous mutually-coinductive proof of Example Theorem 2.5 about *evens* and *odds* like so:[3]

$$\begin{array}{rl}
\beta \div A \vdash & \forall s_1 \forall s_2.\langle \mathit{evens}\ (\mathit{merge}\ s_1\ s_2) \| \text{head}\,\beta\rangle = \langle s_1 \| \text{head}\,\beta\rangle \\
\beta \div A \vdash & \forall s_1 \forall s_2.\ \langle \mathit{odds}\ (\mathit{merge}\ s_1\ s_2) \| \text{head}\,\beta\rangle = \langle s_2 \| \text{head}\,\beta\rangle \\
\alpha \div \text{Stream}\,A \mid \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash & \forall s_1 \forall s_2.\ \ \langle \mathit{evens}\ (\mathit{merge}\ s_1\ s_2) \| \text{tail}\,\alpha\rangle = \langle s_1 \| \text{tail}\,\alpha\rangle \\
\alpha \div \text{Stream}\,A \mid \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash & \forall s_1 \forall s_2.\ \ \langle \mathit{odds}\ (\mathit{merge}\ s_1\ s_2) \| \text{tail}\,\alpha\rangle = \langle s_2 \| \text{tail}\,\alpha\rangle \\
& \vdots\ \omega\text{Stream}, \mathit{ConjI} \\
\alpha \div \text{Stream}\,A \vdash & \forall s_1 \forall s_2.\langle \mathit{evens}\ (\mathit{merge}\ s_1\ s_2) \| \alpha\rangle = \langle s_1 \| \alpha\rangle \\
& \wedge \forall s_1 \forall s_2.\ \langle \mathit{odds}\ (\mathit{merge}\ s_1\ s_2) \| \alpha\rangle = \langle s_2 \| \alpha\rangle
\end{array}$$

Where the two coinductive hypotheses are:

$$\begin{aligned}
\Psi_1(\alpha) &= \forall s_1 \forall s_2.\langle \mathit{evens}\ (\mathit{merge}\ s_1\ s_2) \| \alpha\rangle = \langle s_1 \| \alpha\rangle \\
\Psi_2(\alpha) &= \forall s_1 \forall s_2.\langle \mathit{odds}\ (\mathit{merge}\ s_1\ s_2) \| \alpha\rangle = \langle s_2 \| \alpha\rangle
\end{aligned}$$

Both of these two propositions are productive on $\alpha$ because on the right side they are given $s_i$ which is always a value, and on the left side they are immediately matched on by *evens* or *odds* (which are represented by a **corec** which is itself a value). From here, the calculations showing all four required equalities follow the same steps as the informal proof in Example Theorem 2.5. Since only the weak form of coinduction is used, this fact about *evens* and *odds* holds true in languages with side effects under *both* call-by-name *and* call-by-value evaluation.

Notice that, for both mutual induction and coinduction, the strong rules $\sigma$Stream or $\sigma$Nat are only needed to verify fundamentally non-productive or non-strict propositions $\Phi_1 \wedge \Phi_2$, respectively.

### *Strong induction on the naturals*

In contrast to mutual induction, which can be derived from $\omega$Nat, the traditional notion of *strong induction* on the natural numbers really requires the full $\sigma$Nat. How can we formalize the derivation of strong induction? First, define the ordering relation on numbers in terms of the following equality and translation of the usual *minus* function (replacing negative results with zero) specified as follows:

$$M \leq N : \text{Nat} := \mathit{minus}\ M\ N = \text{zero} : \text{Nat}$$

$$\begin{aligned}
\langle \mathit{minus} \| x \cdot \text{zero} \cdot \alpha\rangle &= \langle x \| \alpha\rangle \\
\langle \mathit{minus} \| \text{succ}\,x \cdot \text{succ}\,y \cdot \alpha\rangle &= \langle \mathit{minus} \| x \cdot y \cdot \alpha\rangle \\
\langle \mathit{minus} \| \text{zero} \cdot \text{succ}\,y \cdot \alpha\rangle &= \langle \text{zero} \| \alpha\rangle
\end{aligned}$$

[3] Note that while *evens* (*merge* $s_1$ $s_2$) and *odds* (*merge* $s_1$ $s_2$) are not syntactically values, they both simplify to a value in both call-by-value and call-by-name. So we can get the equivalent productive property by simplifying the two equations, applying $\omega$Stream, and then expanding back to this form.

We then write $\forall y \leq x : \mathrm{Nat} \,.\, \Phi$ as shorthand for the property $\forall y : \mathrm{Nat} \,.\, y \leq x : \mathrm{Nat} \Rightarrow \Phi$. Applying $\sigma\mathrm{Nat}$ to the free $x$ in this property gives:

$$\frac{\Gamma \vdash \forall y \leq \mathrm{zero} : \mathrm{Nat} \,.\, \Phi \quad \Gamma, x : \mathrm{Nat} \mid \forall y \leq x : \mathrm{Nat} \,.\, \Phi \vdash \forall y \leq \mathrm{succ}\, x : \mathrm{Nat} \,.\, \Phi}{\Gamma, x : \mathrm{Nat} \vdash \forall y \leq x : \mathrm{Nat} \,.\, \Phi}\, \sigma\mathrm{Nat}$$

Since we can derive the properties $\forall y \leq \mathrm{zero} : \mathrm{Nat} \,.\, y = \mathrm{zero} : \mathrm{Nat}$ (by definition of $\leq$) and $\forall x : \mathrm{Nat} \,.\, x \leq x : \mathrm{Nat}$ (by induction with $\sigma\mathrm{Nat}$), we can specialize the above application to derive the following simpler statement of strong induction on the naturals:

$$\frac{\begin{array}{c}\Gamma \vdash \Phi[\mathrm{zero}/x] \quad \Gamma, x : \mathrm{Nat} \mid \forall y \leq x : \mathrm{Nat} \,.\, \Phi \vdash \Phi[\mathrm{succ}\, x/x] \\ \vdots \\ \Gamma, x : \mathrm{Nat} \vdash \Phi\end{array}}{}$$

Notice that we can *never* use $\omega\mathrm{Nat}$ for this derivation, even if $\Phi$ happens to be strict on $x$. Why not? Because the property to which we apply induction,

$$\forall y : \mathrm{Nat}. \; y \leq \mathrm{succ}\, x : \mathrm{Nat} \implies \Phi$$

includes an implication where the inducted-upon $x$ is referenced to the *left* of $\implies$, which is not allowed in properties that are strict on $x$.

### *Strong coinduction on streams*

As with induction on the natural numbers, we can derive the dual notion of strong coinduction on infinite streams. First, define the ordering relation on stream *projections* as:

$$Q \leq R \div \mathrm{Stream}\, A := \mathit{depth}\, Q \leq \mathit{depth}\, R : \mathrm{Nat}$$

where $\mathit{depth}\, Q$ computes the depth of any stream projection $Q$, effectively converting $\mathrm{tail}^n(\mathrm{head}\, \alpha)$ to $\mathrm{succ}^n \mathrm{zero}$:

$$\mathit{depth}\, Q := \mu\alpha.\langle \mathbf{corec}\{\mathrm{head}\, \alpha \to \alpha \mid \mathrm{tail}\, \_ \to \gamma.\tilde{\mu}y.\langle \mathrm{succ}\, y \| \gamma \rangle\} \;\mathbf{with}\; \mathrm{zero} \| Q \rangle$$

As before, we write the quantification $\forall \beta \leq \alpha \div \mathrm{Stream}\, A.\Phi$ as shorthand for $\forall \beta \div \mathrm{Stream}\, A.\ \beta \leq \alpha \div \mathrm{Stream}\, A \Rightarrow \Phi$. Applying $\sigma\mathrm{Stream}$ to this property gives:

$$\frac{\begin{array}{c}\Gamma, \delta \div A \mid \Delta \vdash \forall \beta \leq \mathrm{head}\, \delta \div \mathrm{Stream}\, A.\Phi \\ \Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \forall \beta \leq \alpha \div \mathrm{Stream}\, A.\Phi \vdash \forall \beta \leq \mathrm{tail}\, \alpha \div \mathrm{Stream}\, A.\Phi\end{array}}{\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta \vdash \forall \beta \leq \alpha \div \mathrm{Stream}\, A.\Phi}\, \sigma\mathrm{Stream}$$

Analogous to strong induction on the naturals, we can use this application to derive the following simpler statement of strong coinduction on streams:

$$\frac{\begin{array}{c}\Gamma, \delta \div A \mid \Delta \vdash \Phi[\mathrm{head}\, \delta/\alpha] \\ \Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \forall \beta \leq \alpha \div \mathrm{Stream}\, A.\Phi[\beta/\alpha] \vdash \Phi[\mathrm{tail}\, \alpha/\alpha] \\ \vdots \\ \Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta \vdash \Phi\end{array}}{}$$

From this, we can derive the following special case of strong coinduction, where we must show the first $n+1$ base cases (for $\mathrm{head}\, \beta$, $\mathrm{tail}(\mathrm{head}\, \beta)$, $\ldots \mathrm{tail}^n(\mathrm{head}\, \beta)$) directly, and then

take the $n+1^{th}$ tail projection in the coinductive case:

$$\frac{\Gamma, \beta \div A \mid \Delta \vdash \Phi[\text{head}\,\beta/\alpha] \ldots \Gamma, \beta \div A \mid \Delta \vdash \Phi[\text{tail}^n(\text{head}\,\beta)/\alpha] \quad \Gamma, \alpha \div \text{Stream}\,A \mid \Delta, \Phi \vdash \Phi[\text{tail}^{n+1}\alpha/\alpha]}{\Gamma, \alpha \div \text{Stream}\,A \mid \Delta \vdash \Phi}$$

The above principle can prove that

$$\alpha \div \text{Stream}\,A \vdash \forall s : \text{Stream}\,A.\langle \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s) \| \alpha \rangle = \langle s \| \alpha \rangle \tag{4.1}$$

by stepping by 2. We prove the property for the base cases (head $\beta$ and tail(head $\beta$)) and then prove the coinductive case

$$\alpha \div \text{Stream}\,A \vdash \forall s : \text{Stream}\,A.\langle \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s) \| \text{tail}(\text{tail}\,\beta) \rangle = \langle s \| \text{tail}(\text{tail}\,\beta) \rangle$$

assuming that the property holds for $\beta$. This principle captures the proof of Example Theorem 2.6, with the difference that it avoids the case analysis. From (4.1), we can then prove

$$\forall s : \text{Stream}\,A.\ \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s) = s : \text{Stream}\,A \tag{4.2}$$

by *IntroL*, $\sigma\mu$, *ElimR*.

### *Bisimulation on streams*

To conclude our exploration, we now turn to one of the most commonly used principles for reasoning about coinductive structures—*bisimulation*—which allows us to prove two objects are equal whenever they are related by *any* valid bisimulation relation of our choosing. The traditional principle of bisimulation on streams can be represented by the following inference rule, where the property $\Phi$ (with free variables $s_1$ and $s_2$) stands for an arbitrary relationship between two streams $s_1$ and $s_2$:

$$\frac{\begin{array}{l}\Gamma, s_1 : \text{Stream}\,A, s_2 : \text{Stream}\,A \mid \Delta, \Phi \vdash \text{head}\,s_1 = \text{head}\,s_2 : A \\ \Gamma, s_1 : \text{Stream}\,A, s_2 : \text{Stream}\,A \mid \Delta, \Phi \vdash \Phi[\text{tail}\,s_1/s_1, \text{tail}\,s_2/s_2]\end{array}}{\Gamma, s_1 : \text{Stream}\,A, s_2 : \text{Stream}\,A \mid \Delta, \Phi \vdash s_1 = s_2 : \text{Stream}\,A}\ \mathit{Bisim}$$

The two assumptions confirm that $\Phi$ is a valid bisimulation relation: $\Phi$ only relates streams with equal heads, and is closed under tail projection. We show that this principle is also subsumed by the strong coinduction rule $\sigma$Stream. We are going to prove

$$\Gamma, \alpha : \text{Stream}\,A \mid \Delta \vdash \forall s_1, s_2 : \text{Stream}\,A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle \tag{4.3}$$

Where we use the shorthand $\forall s_1, s_2 : \text{Stream}\,A.\Phi$ to stand for multiple quantifications of the same type $\forall s : \text{Stream}\,A.\forall s' : \text{Stream}\,A.\Phi$. From the above the goal follows:

$$\frac{\begin{array}{c}\Gamma, \alpha : \text{Stream}\,A \mid \Delta \vdash \forall s_1, s_2 : \text{Stream}\,A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle \\ \vdots\ \mathit{ElimL}_{s_1,s_2}, \mathit{Ax}, \mathit{Lemm} \\ \Gamma, s_1 : \text{Stream}\,A, s_2 : \text{Stream}\,A, \alpha \div \text{Stream}\,A \mid \Delta, \Phi \vdash \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle\end{array}}{\Gamma, s_1 : \text{Stream}\,A, s_2 : \text{Stream}\,A \mid \Delta, \Phi \vdash s_1 = s_2 : \text{Stream}\,A}\ \sigma\mu$$

We are proving property 4.3 by strong coinduction ($\sigma$Stream):

- For the head case, we must show that

$$\Gamma, \alpha \div A \mid \Delta \vdash \forall s_1, s_2 : \mathrm{Stream}\, A.\Phi \Rightarrow \langle \mathrm{head}\, s_1 \| \alpha \rangle = \langle \mathrm{head}\, s_2 \| \alpha \rangle$$

  The first bisimulation assumption already guarantees that $\mathrm{head}\, s_1 = \mathrm{head}\, s_2 : A$ whenever $\Phi$ holds on $s_1$ and $s_2$, so this sub-goal follows directly from the congruence rules, as shown below:

$$\begin{array}{c} \Gamma, s_1 \colon \mathrm{Stream}\, A, s_2 \colon \mathrm{Stream}\, A, \Phi \vdash \mathrm{head}\, s_1 = \mathrm{head}\, s_2 : A \\ \vdots\ \mathit{WeakR}, \mathit{Cut}, \mathit{VarL} \\ \Gamma, \beta \div A, s_1 \colon \mathrm{Stream}\, A, s_2 \colon \mathrm{Stream}\, A, \Phi \vdash \langle \mathrm{head}\, s_1 \| \beta \rangle = \langle \mathrm{head}\, s_2 \| \beta \rangle \\ \vdots\ \mathit{IntroH}, \mathit{IntroL} \\ \Gamma, \beta \div A \vdash \forall s_1, s_2 \colon \mathrm{Stream}\, A.\Phi \Rightarrow \langle \mathrm{head}\, s_1 \| \beta \rangle = \langle \mathrm{head}\, s_2 \| \beta \rangle \end{array}$$

- For the tail case, from the coinductive hypothesis (referred to locally as *CIH*)

$$\forall s_1, s_2 : \mathrm{Stream}\, A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle \qquad (\mathit{CIH})$$

  we must show

$$\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \mathit{CIH} \vdash \forall s_1, s_2 : \mathrm{Stream}\, A.\Phi \Rightarrow \langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle \mathrm{tail}\, s_2 \| \alpha \rangle$$

  The second bisimulation assumption guarantees that $\Phi[\mathrm{tail}\, s_1/s_1, \mathrm{tail}\, s_2/s_2]$ holds as well. Therefore, substituting $\mathrm{tail}\, s_1$ and $\mathrm{tail}\, s_2$ in the coinductive hypothesis gives the required result $\langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle \mathrm{tail}\, s_2 \| \alpha \rangle$. More precisely, using the shorthand

$$\begin{aligned} \Gamma_{\mathit{CIH}} &:= \Gamma, \alpha \div \mathrm{Stream}\, A \\ \Gamma_{\mathit{Sim}} &:= \Gamma, s_1 : \mathrm{Stream}\, A, s_2 : \mathrm{Stream}\, A \\ \Gamma' &:= \Gamma, \alpha \div \mathrm{Stream}\, A, s_1 : \mathrm{Stream}\, A, s_2 : \mathrm{Stream}\, A \end{aligned}$$

  we can derive the goal of the coinductive step by weakening the given bisimulation premise $\Gamma_{\mathit{Sim}} \mid \Delta, \Phi \vdash \Phi[\mathrm{tail}\, s_1/s_1, \mathrm{tail}\, s_2/s_2]$ as follows:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_{\mathit{CIH}} \mid \Delta, \mathit{CIH} \vdash \forall s_1, s_2 \colon \mathrm{Stream}\, A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle}\ \mathit{Ax}}{\vdots\ \mathit{ElimL}, \mathit{ReflL}}}{\Gamma_{\mathit{CIH}} \mid \Delta, \mathit{CIH} \vdash \forall s_2 \colon \mathrm{Stream}\, A.\Phi[\mathrm{tail}\, s_1/s_1] \Rightarrow \langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle}\quad \dfrac{\Gamma_{\mathit{Sim}} \mid \Delta, \Phi \vdash \Phi[\mathrm{tail}\, s_1/s_1, \mathrm{tail}\, s_2/s_2]}{\vdots\ \mathit{WeakL}}}{\vdots\ \mathit{ElimL}, \mathit{ReflL}}$$

$$\dfrac{\Gamma' \mid \Delta, \mathit{CIH} \vdash \Phi[\mathrm{tail}\, s_1/s_1, \mathrm{tail}\, s_2/s_2] \Rightarrow \langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle \mathrm{tail}\, s_2 \| \alpha \rangle \qquad \Gamma' \mid \Delta, \mathit{CIH}, \Phi \vdash \Phi[\mathrm{tail}\, s_1/s_1, \mathrm{tail}\, s_2/s_2]}{\Gamma' \mid \Delta, \mathit{CIH} \vdash \langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle \mathrm{tail}\, s_2 \| \alpha \rangle}\ \mathit{Lemm}$$

$$\dfrac{\Gamma' \mid \Delta, \mathit{CIH} \vdash \langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle \mathrm{tail}\, s_2 \| \alpha \rangle}{\vdots\ \mathit{IntroL}}$$

$$\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \mathit{CIH} \vdash \forall s_1, s_2 \colon \mathrm{Stream}\, A.\Phi \Rightarrow \langle \mathrm{tail}\, s_1 \| \alpha \rangle = \langle \mathrm{tail}\, s_2 \| \alpha \rangle$$

## 5 Consistency of the Program Logic

We've seen the (strong) program logics used to encode and prove a variety of different reasoning principles and program equalities. But how do we know if and when the syntactic rules in Figs. 5 and 6 imply real equivalences between the results of programs? Applying $\beta$ reductions may be easy enough to believe since they correspond to actual steps of execution, but what about the (co)induction rules? They do not correspond to steps taken by the abstract

machine, and we have already seen counterexamples where some of them, like $\sigma$Nat and $\sigma$Stream, can be inconsistent in certain contexts.

In order to prove that the syntactic program logics are consistent, we will show that they are all approximations of a more general notion of *observational equivalence* (also known as *contextual equivalence* (Pitts, 1997*b*)), defined directly in terms of the behavior of running programs. First, we need to characterize the valid stopping points where computation has ended and we can observe the last attempt at communication between a constructor or observer and some free (co)variable. Two such commands are considered equivalent if the top-level structure is the same (ignoring anything deeper, since the computation is finished).

**Definition 5.1** (Observable)**.** The set of *observable typing environments* ($\Theta$) and *observable commands* ($d$) is

$$\begin{aligned} \mathit{ObsEnv} \ni \quad & \Theta ::= \bullet \mid \Theta, \alpha \div \mathrm{Nat} \mid \Theta, x : \mathrm{Stream}\, A \mid \Theta, x : A \to B \\ \mathit{ObsCommand} \ni \quad & d ::= \langle \mathrm{zero} \| \alpha \rangle \mid \langle \mathrm{succ}\, V \| \alpha \rangle \mid \langle x \| \mathrm{head}\, E \rangle \mid \langle x \| \mathrm{tail}\, E \rangle \mid \langle x \| V \cdot E \rangle \end{aligned}$$

The weak equivalence relation on observable commands, $d \sim d'$, is:

$$\begin{array}{lll} \langle \mathrm{zero} \| \alpha \rangle \sim \langle \mathrm{zero} \| \alpha \rangle & \langle x \| \mathrm{head}\, E \rangle \sim \langle x \| \mathrm{head}\, E' \rangle & \langle x \| V \cdot E \rangle \sim \langle x \| V' \cdot E' \rangle \\ \langle \mathrm{succ}\, V \| \alpha \rangle \sim \langle \mathrm{succ}\, V' \| \alpha \rangle & \langle x \| \mathrm{tail}\, E \rangle \sim \langle x \| \mathrm{tail}\, E' \rangle & \end{array}$$

This weak equivalence relation is extended to any two commands, $c \approx c'$, via computation: $c \approx c'$ if and only if there are observable commands $d$, $d'$ such that $c \mapsto\!\!\!\!\rightarrow d \sim d' \leftarrow\!\!\!\!\mapsto c'$.

**Definition 5.2** (Observational Equivalence)**.** *Typed observational equivalence* is defined as:

1. $\Gamma \vdash c_1 \approx c_2$ iff $\Gamma \vdash c_i$ and for all contexts $C$, $\Theta \vdash C[c_i]$ implies $C[c_1] \approx C[c_2]$.
2. $\Gamma \vdash v_1 \approx v_2 : A$ iff $\Gamma \vdash v_i : A$ and for all contexts $C$, $\Theta \vdash C[v_i]$ implies $C[v_1] \approx C[v_2]$.
3. $\Gamma \vdash e_1 \approx e_2 \div A$ iff $\Gamma \vdash e_i \div A$ and for all contexts $C$, $\Theta \vdash C[e_i]$ implies $C[e_1] \approx C[e_2]$.

where a context $C$ is any command with a hole (written $\Box$) somewhere in it. Filling the context (written $C[c]$, $C[v]$, or $C[e]$) means replacing the hole $\Box$ by the given sub-expression, potentially capturing that sub-expression's free variables.

Observational equivalence is particularly interesting since it is a *consistent, computational congruence* by definition:

**Congruence** Meaning it is a *reflexive*, *transitive*, and *symmetric* equivalence relation, which is also compatible with all contexts of the appropriate type. For example, if $\Gamma \vdash v_1 \approx v_2 : A$, and $C$ is a context such that $\Gamma \vdash C[v_i]$ is a well-typed command, then $\Gamma \vdash C[v_1] \approx C[v_2]$ holds by definition, because contexts compose.

**Computational** In the sense that it is closed under the reductions of the operational semantics: if $\Gamma \vdash c_1 \approx c_2$ and $c_i \mapsto\!\!\!\!\rightarrow c_i'$ then $\Gamma \vdash c_1' \approx c_2'$.

**Consistent** As per Definition 3.6. $\bullet \vdash \mathrm{zero} \approx \mathrm{succ}\, V : \mathrm{Nat}$ does not hold, due to the counterexample context $\langle \Box \| \alpha \rangle$; both $\alpha \div \mathrm{Nat} \vdash \langle \mathrm{zero} \| \alpha \rangle$ and $\alpha \div \mathrm{Nat} \vdash \langle \mathrm{succ}\, V \| \alpha \rangle$ are well-typed, irreducible commands in an observable environment, and yet $\langle \mathrm{zero} \| \alpha \rangle \not\sim$

$\langle \text{succ}\, V \| \alpha \rangle$. Similarly, $\bullet \vdash \text{head}\, E \approx \text{tail}\, E' \div \text{Stream}\, A$ does not hold, due to the counterexample context $\langle x \| \square \rangle$ in the observable environment $x : \text{Stream}\, A$.

In fact, observational equivalence is the *coarsest* such relation (Harper, 2016, Theorem 46.6), meaning that any other relation with these properties are included in Definition 5.2. So, proving that another computational congruence relation is consistent (such as the syntactic theories in Fig. 5 and Definition 3.8) can be reduced to proving they are included within observational equivalence.

Our primary goal, then, is to prove that these syntactic theories of equality all imply observational equivalence, from which their consistency falls out as a corollary. To do so, we will generalize the model of (co)inductive types from (Downen & Ariola, 2023)—based on the techniques of *classical realizability* (Krivine, 2005), *(bi)orthogonality* (Girard, 1987; Munch-Maccagnoni, 2009), $\top\top$-closure (Pitts, 2000), and *symmetric candidates* (Barbanera & Berardi, 1994) for proving strong normalization of classical calculi—from unary predicates describing safety to binary relations describing equivalence.

### 5.1 Orthogonal relations and equality candidates

The safety model of (Downen & Ariola, 2023, Section 6) is a logical relation built around the idea of *orthogonality* (Girard, 1987; Pitts, 2000; Munch-Maccagnoni, 2009): a safety predicate (written $\perp\!\!\!\perp$) classifying when producers ($v$) and consumers ($e$) can safely interact with one another in a command ($\langle v \| e \rangle \in \perp\!\!\!\perp$). Here we are interested in equality, in the sense that two commands have equivalent behavior when run. As such, we need to generalize orthogonality beyond the unary safety predicate $c \in \perp\!\!\!\perp$ on one command, and instead consider a binary equivalence relation $c \perp\!\!\!\perp c'$ between two commands.

We can now give our main definition of binary orthogonality $c \perp\!\!\!\perp c'$ serving in terms of the untyped weak equivalence among commands. Orthogonality, in turn, lets us describe a semantics for typed equality as a certain pair of relations between terms and coterms. This forms the potential (*i.e., candidate* (Girard, 1972)) denotations of types, so each type of our language can be interpreted as a particular candidate of equality.

**Definition 5.3** (Orthogonality)**.** The *equivalence pole* $\perp\!\!\!\perp$ is the untyped equivalence relation on arbitrary commands ($c \approx c'$) given in terms of weak equivalence of observable commands ($d \sim d'$) from Definition 5.1:

$$\begin{aligned} c \perp\!\!\!\perp c' &:= c \approx c' \\ &:= \exists d, d' .\, c \mapsto\!\!\!\!\rightarrow d \sim d' \mathrel{\leftarrow\!\!\!\!\mapsto} c' \end{aligned}$$

*Orthogonality* of two binary relations $\mathbb{A}^+ \subseteq \mathit{Term}^2$ and $\mathbb{A}^- \subseteq \mathit{CoTerm}^2$ is defined as:[4]

$$\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^- := \forall v\ \mathbb{A}^+ v', e\ \mathbb{A}^- e' .\, \langle v \| e \rangle \perp\!\!\!\perp \langle v' \| e' \rangle$$

We write $\mathbb{A}^{+\perp\!\!\!\perp}$ to denote the largest coterm relation orthogonal to the term relation $\mathbb{A}^+$, and symmetrically write $\mathbb{A}^{-\perp\!\!\!\perp}$ to denote the largest term relation orthogonal to the coterm

[4] Note that we denote membership of a binary relation $\mathbb{R} \subseteq \mathbb{X} \times \mathbb{Y}$ as an infix operation $x\ \mathbb{R}\ y$ instead of set membership notation $(x, y) \in \mathbb{R}$. Furthermore, we use $Y^2$ as shorthand for the product $Y \times Y$

relation $\mathbb{A}^-$, which are respectively defined as:

$$e\ \mathbb{A}^{+\perp\!\!\!\perp}\ e' := \forall v\ \mathbb{A}^+ v'.\ \langle v \| e\rangle \perp\!\!\!\perp \langle v' \| e'\rangle$$
$$v\ \mathbb{A}^{-\perp\!\!\!\perp}\ v' := \forall e\ \mathbb{A}^- e'.\ \langle v \| e\rangle \perp\!\!\!\perp \langle v' \| e'\rangle$$

**Definition 5.4** (Candidates)**.** A *pre-candidate* is any pair $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ where $\mathbb{A}^+$ is a binary relation on terms, and $\mathbb{A}^-$ is a binary relation on coterms, *i.e.,*

$$\mathbb{A} \in \wp(Term^2) \times \wp(CoTerm^2)\ .$$

A *sound* (pre-)candidate $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ satisfies the following *soundness* requirement:

- *Soundness:* every combination of $\mathbb{A}^+$-related terms $v\ \mathbb{A}^+\ v'$ and $\mathbb{A}^-$-related coterms $e\ \mathbb{A}^-\ e'$ forms $\perp\!\!\!\perp$-equivalent commands $\langle v \| e\rangle \perp\!\!\!\perp \langle v' \| e'\rangle$.

A *complete* (pre-)candidate $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ satisfies these two *completeness* requirements:

- *Positive completeness:* if $\langle v \| E\rangle \perp\!\!\!\perp \langle v' \| E'\rangle$ for all $\mathbb{A}^-$-related covalues $E\ \mathbb{A}^-\ E'$, then $v\ \mathbb{A}^+\ v'$ are related by $\mathbb{A}^+$.
- *Negative completeness:* if $\langle V \| e\rangle \perp\!\!\!\perp \langle V' \| e'\rangle$ for all $\mathbb{A}^+$-related values $V\ \mathbb{A}^+\ V'$, then $e\ \mathbb{A}^-\ e'$ are related by $\mathbb{A}^-$.

An *equality candidate* is any sound and complete pre-candidate. $\mathscr{PC}$ denotes the set of all pre-candidates, $\mathscr{SC}$ denotes the set of sound ones, $\mathscr{CC}$ the set of complete ones, and $\mathscr{EC}$ denotes the set of all equality candidates.

As notation, given any pre-candidate $\mathbb{A}$, we will always write $\mathbb{A}^+$ to denote the first component of $\mathbb{A}$ and $\mathbb{A}^-$ to denote the second one, so that $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$. Given a binary relation on terms $\mathbb{A}^+$, we will occasionally write the sound candidate $(\mathbb{A}^+, \{\})$ as just $\mathbb{A}^+$ when the difference is clear from the context (notice that $(\mathbb{A}^+, \{\})$ is trivially sound by definition, but is incomplete). Likewise, we will occasionally write the sound candidate $(\{\}, \mathbb{A}^-)$ as just the binary coterm relation $\mathbb{A}^-$ when unambiguous. The common case of the empty set $\{\}$—which could be read as either the empty set of terms or the empty set of coterms—denotes the same sound candidate $(\{\}, \{\})$ according to either reading.

### *5.2 Dual lattices and completion*

With its two halves—one describing terms the other coterms—candidates provide multiple views on the relationship between types. These appear in the unary case of typed (co)terms, as in (Downen & Ariola, 2023, Definition 6.5), and carry over, essentially unchanged, to binary relationships, too. In particular, the set of candidates supports two separate, but complementary, lattice structures with different orderings; one based on a *refinement* notion of plain containment, and the other based on a notion of *subtyping* from programming languages.

**Definition 5.5** (Refinement and Subtyping)**.** There are two ways of ordering pre-candidates: *refinement* (denoted by $\mathbb{A} \sqsubseteq \mathbb{B}$ meaning "$\mathbb{A}$ refines $\mathbb{B}$" and "$\mathbb{B}$ extends $\mathbb{A}$) and *subtyping*

(denoted by $\mathbb{A} \leq \mathbb{B}$ meaning "$\mathbb{A}$ is a subtype of $\mathbb{B}$" and "$\mathbb{B}$ is a supertype of $\mathbb{A}$"), defined as:

$$(\mathbb{A}^+, \mathbb{A}^-) \sqsubseteq (\mathbb{B}^+, \mathbb{B}^-) := (\mathbb{A}^+ \subseteq \mathbb{B}^+) \text{ and } (\mathbb{A}^- \subseteq \mathbb{B}^-)$$
$$(\mathbb{A}^+, \mathbb{A}^-) \leq (\mathbb{B}^+, \mathbb{B}^-) := (\mathbb{A}^+ \subseteq \mathbb{B}^+) \text{ and } (\mathbb{A}^- \supseteq \mathbb{B}^-)$$

Refinement and subtyping both define a complete lattice on pre-candidates with the following unions and intersections for refinement ($\sqcup, \sqcap$) and subtyping ($\vee, \wedge$), defined over any set of pre-candidates $\{\mathbb{A}_i\}_i \subseteq \mathscr{PC}$ as:

$$\textstyle\bigsqcup_i(\mathbb{A}_i^+, \mathbb{A}_i^-) := (\bigcup_i \mathbb{A}_i^+, \bigcup_i \mathbb{A}_i^-) \qquad \bigvee_i(\mathbb{A}_i^+, \mathbb{A}_i^-) := (\bigcup_i \mathbb{A}_i^+, \bigcap_i \mathbb{A}_i^-)$$
$$\textstyle\bigsqcap_i(\mathbb{A}_i^+, \mathbb{A}_i^-) := (\bigcap_i \mathbb{A}_i^+, \bigcap_i \mathbb{A}_i^-) \qquad \bigwedge_i(\mathbb{A}_i^+, \mathbb{A}_i^-) := (\bigcap_i \mathbb{A}_i^+, \bigcup_i \mathbb{A}_i^-)$$

where $\bigcup$ and $\bigcap$ denote the union and intersection of binary relations, respectively.

There are many other ways in which refinement and subtyping differ from one another, and reveal different structures of equality candidates. Of note, the orthogonality operation distributes over the two orderings in completely opposite directions.

**Property 5.6** (Orthogonal Ordering)**.** *Given any pre-candidates $\mathbb{A}$ and $\mathbb{B}$:*

1. Antitonicity: *If $\mathbb{A} \sqsubseteq \mathbb{B}$ then $\mathbb{A}^{\perp\!\!\perp} \sqsupseteq \mathbb{B}^{\perp\!\!\perp}$.*
2. Monotonicity: *If $\mathbb{A} \leq \mathbb{B}$ then $\mathbb{A}^{\perp\!\!\perp} \leq \mathbb{B}^{\perp}$.*

Furthermore, the way the union and intersection operations in the two lattices preserve (or fail to preserve) soundness and completeness conditions also differ.

**Property 5.7** (Sound and Complete Lattices)**.** *Given any subset $\{\mathbb{A}_i\}_i \subseteq \mathscr{SC}$ of sound candidates and $\{\mathbb{B}_i\}_i \subseteq \mathscr{CC}$ complete candidates:*

1. *$\bigwedge_i \mathbb{A}_i$ and $\bigvee_i \mathbb{A}_i$ are sound, but $\bigwedge_i \mathbb{B}_i$ and $\bigvee_i \mathbb{B}_i$ may be incomplete.*
2. *$\bigsqcap_i \mathbb{A}_i$ is sound, but $\bigsqcup_i \mathbb{A}_i$ may be unsound.*
3. *$\bigsqcup_i \mathbb{B}_i$ is complete, but $\bigsqcap_i \mathbb{B}_i$ may be incomplete.*

In order to build the interpretation of (co)inductive types, we need a complete lattice of *equality candidates*, not just a lattice of pre-candidates, that preserves both soundness *and* completeness. Since subtyping naturally gives us a complete sub-lattice of sound candidates, we will begin there, with the subgoal of filling in the missing parts of a sound candidate to generate the fully completed equality candidate.

Beginning with some initial starting point, completeness demands that we include all other relationships which are compatible with what is already there. Since completeness only tests potential (co)term relations *w.r.t* the *(co)values* already related by a candidate, we will have to isolate these (co)values as part of our testing criteria. For this purpose, the *(co)value restriction* $\mathbb{A}^v$ of a candidate $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ includes only those values and covalues related by $\mathbb{A}$, defined as:

$$v\ \mathbb{A}^{v+}\ v' := v\ \mathbb{A}^+\ v' \text{ and } v, v' \in \mathit{Value} \qquad e\ \mathbb{A}^{v-}\ e' := e\ \mathbb{A}^-\ e' \text{ and } e, e' \in \mathit{CoValue}$$

We can use this (co)value restriction to form a complete equality candidate by interleaving it with the orthogonality operation. But there is a dual choice in our starting point: the positive viewpoint uses the *values* related by $\mathbb{A}$ as the defining axioms to define the equality candidate, and the negative viewpoint uses the *covalues* related by $\mathbb{A}$ as the defining axioms.

**Definition 5.8** (Positive and Negative Candidates)**.** Given a sound candidate $\mathbb{A}$, the *positive* and *negative* constructions of equality candidates around $\mathbb{A}$ are respectively defined as:

$$\mathrm{Pos}(\mathbb{A}) := (\mathbb{A}^{+}, \mathbb{A}^{+v\perp})^{v\perp v\perp} \qquad \mathrm{Neg}(\mathbb{A}) := (\mathbb{A}^{-v\perp}, \mathbb{A}^{-})^{v\perp v\perp}$$

The positive and negative viewpoints give complementary equality candidates. By starting with the values first, Pos gives a smaller equality candidate (*w.r.t* subtyping) compared to Neg. In fact, these two are the canonically *largest* and *smallest* equality candidates that extend any sound starting point. This fact lets us modify the subtyping lattice to preserve *both* soundness and completeness in both directions.

**Lemma 5.9** (Positive & Negative Completion)**.** *For any sound candidate* $\mathbb{A}$*,* $\mathrm{Pos}(\mathbb{A})$ *is the* smallest sound and complete extension *of* $\mathbb{A}^{v}$ w.r.t *subtyping, and* $\mathrm{Neg}(\mathbb{A})$ *is the* largest sound and complete extension *of* $\mathbb{A}^{v}$ w.r.t *subtyping. In other words, both* $\mathrm{Pos}(\mathbb{A})$ *and* $\mathrm{Neg}(\mathbb{A})$ *are equality candidates such that* $\mathrm{Pos}(\mathbb{A}) \sqsupseteq \mathbb{A}^{v}$ *and* $\mathrm{Neg}(\mathbb{A}) \sqsupseteq \mathbb{A}^{v}$*, and given any other equality candidate* $\mathbb{C} \sqsupseteq \mathbb{A}^{v}$*,*

$$\mathrm{Pos}(\mathbb{A}) \leq \mathbb{C} \leq \mathrm{Neg}(\mathbb{A})$$

*Proof sketch* The proof follows the same structure as in (Downen & Ariola, 2023, Lemma 6.7) extended from sets to binary relations, which uses the facts that $\mathrm{Pos}(\mathbb{A})$ and $\mathrm{Neg}(\mathbb{A})$ are fixed points of $\_^{v\perp}$ and, furthermore, that the set of these fixed points is exactly the set of all equality candidates (Downen *et al.*, 2020, Property 9). □

**Definition 5.10** (Equality Candidate Lattice)**.** Equality candidates form a complete lattice *w.r.t* subtyping whose unions ($\curlyvee$) and intersections ($\curlywedge$) are (Downen *et al.*, 2019):

$$\curlywedge_i \mathbb{A}_i := \mathrm{Neg}(\bigwedge_i \mathbb{A}_i) \qquad \curlyvee_i \mathbb{A}_i := \mathrm{Pos}(\bigvee_i \mathbb{A}_i)$$

Notice that the least equality candidate *w.r.t* subtyping is $\mathrm{Pos}\{\} = (\{\}, \mathit{CoValue}^2)^{\perp v\perp}$ and the greatest one is $\mathrm{Neg}\{\} = (\mathit{Value}^2, \{\})^{\perp v\perp}$.

From this perspective, we can re-describe the positive and negative completions in terms of the subtyping lattice of equality candidates. As per Lemma 5.9, Pos and Neg are the intersection and union (respectively) of all extensions of a restricted sound candidate $\mathbb{A}^{v}$:

$$\mathrm{Pos}(\mathbb{A}) = \curlywedge\{\mathbb{C} \in \mathscr{EC} \mid \mathbb{C} \sqsupseteq \mathbb{A}^{v}\} \qquad \mathrm{Neg}(\mathbb{A}) = \curlyvee\{\mathbb{C} \in \mathscr{EC} \mid \mathbb{C} \sqsupseteq \mathbb{A}^{v}\}$$

As a corollary of Lemma 5.9 and the definition of Pos and Neg, we get the following facts that let us reason about positively and negatively constructed equality candidates.

**Property 5.11** (Positive & Negative Invariance)**.** *For any sound candidates* $\mathbb{A}$ *and* $\mathbb{B}$*:*

- *If* $\mathbb{A}$ *and* $\mathbb{B}$ *relate the same values, then* $\mathrm{Pos}(\mathbb{A}) = \mathrm{Pos}(\mathbb{B})$.
- *If* $\mathbb{A}$ *and* $\mathbb{B}$ *relate the same covalues, then* $\mathrm{Neg}(\mathbb{A}) = \mathrm{Neg}(\mathbb{B})$.

**Property 5.12** (Strong Positive & Negative Completeness)**.** *For any sound candidate* $\mathbb{A}$*:*

- $E \ \mathrm{Pos}(\mathbb{A})^{-} \ E'$ *if and only if* $\langle V \| E \rangle \perp\!\!\!\perp \langle V' \| E' \rangle$ *for all* $V \ \mathbb{A}^{+} \ V'$.
- $V \ \mathrm{Neg}(\mathbb{A})^{+} \ V'$ *if and only if* $\langle V \| E \rangle \perp\!\!\!\perp \langle V' \| E' \rangle$ *for all* $E \ \mathbb{A}^{-} \ E'$.

**Property 5.13.** *For any set of equality candidates* $\{\mathbb{A}_i\}_i$*:*

1. *If* $e \ \mathbb{A}_i^{-} \ e'$ *for* some *i, then* $e \curlywedge_i \mathbb{A}_i \ e'$. *If* $V \ \mathbb{A}_i^{+} \ V'$ *for* all *i, then* $V \curlywedge_i \mathbb{A}_i \ V'$.
2. *If* $v \ \mathbb{A}_i \ v'$ *for* some *i, then* $v \curlyvee_i \mathbb{A}_i \ v'$. *If* $E \ \mathbb{A}_i^{-} \ E'$ *for* all *i, then* $E \curlyvee_i \mathbb{A}_i \ E'$.

### *5.3 Interpretation of types and properties*

We now have enough infrastructure to define the model of observational equivalence—as shown in Fig. 7—by interpreting each syntactic entity (types, properties, environments, and judgements) into its semantic counterpart.

Each syntactic type $A$ is interpreted as an equality candidate, denoted by $[\![A]\!]$, which is defined by induction on the syntax of $A$. This interpretation has three main cases—one for each type constructor—which are all defined in the style of Knaster-Tarski (Knaster, 1928; Tarski, 1955) fixed points in the subtyping lattice of equality candidates:

- A function type $A \to B$ is interpreted as the equality candidate relating the *fewest* covalues possible, while still relating any two call stacks built from $[\![A]\!]$-related arguments and $[\![B]\!]$-related return continuations. Dually, this is the equality candidate relating the *most* values possible, as long as they have equivalent behavior when observed by those previously described related call stacks.
- The number type Nat is interpreted as the equality candidate relating the *fewest* terms possible, while still relating 0 to itself, and ensuring that the successors of any two related values are still related. Dually, this is the equality candidate relating the *most* covalues possible, as long as they respond the same to any of those related numbers.
- A stream type Stream $A$ is interpreted as the equality candidate relating the *fewest* covalues possible, while still relating any two head projections with $[\![A]\!]$-related continuations, and ensuring that the tail of any two related Stream $A$ projection are still related. Dually, this equality candidate relates the *most* terms possible, as long as they have equivalent behavior when observed by those related stream projections.

Each typing environment $\Gamma$ is interpreted as a binary relation on substitutions, $[\![\Gamma]\!]$, both of which replace some variables with values, and some covariables with covalues. The interpretation of $\Gamma$ (written $\rho \ [\![\Gamma]\!] \ \rho'$) relates two such substitutions $\rho$ and $\rho'$ that abide by all three of the following criteria:

Interpretation of types $[\![\_]\!] : \mathit{Type} \to \mathscr{EC}$

$$[\![A \to B]\!] := \bigvee \{\mathbb{C} \in \mathscr{EC} \mid \forall\, V\ [\![A]\!]\ V', E\ [\![B]\!]\ E'.\ (V \cdot E)\ \mathbb{C}\ (V' \cdot E')\}$$
$$[\![\mathrm{Nat}]\!] := \bigwedge \{\mathbb{C} \in \mathscr{EC} \mid \mathrm{zero}\ \ \mathbb{C}\ \ \mathrm{zero}$$
$$\text{and}\quad \forall\, V\ \mathbb{C}\ V'.\ (\mathrm{succ}\, V)\ \mathbb{C}\ (\mathrm{succ}\, V')\}$$
$$[\![\mathrm{Stream}\, A]\!] := \bigvee \{\mathbb{C} \in \mathscr{EC} \mid \forall\, E\ [\![A]\!]\ E'.\ (\mathrm{head}\, E)\ \mathbb{C}\ (\mathrm{head}\, E')$$
$$\text{and}\quad \forall\, E\ \mathbb{C}\ E'.\ (\mathrm{tail}\, E)\ \mathbb{C}\ (\mathrm{tail}\, E')\}$$

Interpretation of environments $[\![\_]\!] : \mathit{Env} \to \wp(\mathit{Subst}^2)$

$$\mathit{Subst} \ni \rho ::= V/x, \ldots, E/\alpha, \ldots$$
$$\varepsilon\ [\![\bullet]\!]\ \varepsilon := \text{trivially true}$$
$$\rho[V/x]\ [\![\Gamma, x : A]\!]\ \rho'[V'/x] := \rho\ [\![\Gamma]\!]\ \rho' \text{ and } V\ [\![A]\!]\ V'$$
$$\rho[E/\alpha]\ [\![\Gamma, \alpha \div A]\!]\ \rho'[E'/\alpha] := \rho\ [\![\Gamma]\!]\ \rho' \text{ and } E\ [\![A]\!]\ E'$$

Interpretation of properties $[\![\_]\!] : \mathit{Prop} \to \wp(\mathit{Subst}^2)$

$$\rho\ [\![c = c']\!]\ \rho' := c[\rho] \perp\!\!\!\perp c'[\rho']$$
$$\rho\ [\![v = v' : A]\!]\ \rho' := v[\rho]\ [\![A]\!]\ v'[\rho']$$
$$\rho\ [\![e = e' \div A]\!]\ \rho' := e[\rho]\ [\![A]\!]\ e'[\rho']$$
$$\rho\ [\![\forall x : A.\Phi]\!]\ \rho' := \forall\, V\ [\![A]\!]\ V'.\ \rho[V/x]\ [\![\Phi]\!]\ \rho'[V'/x]$$
$$\rho\ [\![\forall \alpha \div A.\Phi]\!]\ \rho' := \forall\, E\ [\![A]\!]\ E'.\ \rho[E/\alpha]\ [\![\Phi]\!]\ \rho'[E'/\alpha]$$
$$\rho\ [\![\Phi \Rightarrow \Phi']\!]\ \rho' := \rho\ [\![\Phi]\!]\ \rho' \text{ implies } \rho\ [\![\Phi']\!]\ \rho'$$
$$\rho\ [\![\Phi \wedge \Phi']\!]\ \rho' := \rho\ [\![\Phi]\!]\ \rho' \text{ and } \rho\ [\![\Phi']\!]\ \rho'$$

Interpretation of hypotheses $[\![\_]\!] : \mathit{Hyp} \to \wp(\mathit{Subst}^2)$

$$\rho\ [\![\bullet]\!]\ \rho := \text{trivially true}$$
$$\rho\ [\![\Delta, \Phi]\!]\ \rho' := \rho\ [\![\Delta]\!]\ \rho' \text{ and } \rho\ [\![\Phi]\!]\ \rho'$$

Interpretation of judgements $[\![\_]\!] : \mathit{Judge} \to \{\mathrm{true}, \mathrm{false}\}$

$$[\![\Gamma \mid \Delta \vdash \Phi]\!] := [\![\Gamma]\!] \cap [\![\Delta]\!] \subseteq [\![\Phi]\!]$$

Fig. 7: Model of observational equivalence in the abstract machine.

- For each variable $x$ of type $A$ in the environment $\Gamma$, both $\rho$ and $\rho'$ must substitute some value for $x$ (call them $x[\rho] = V$ and $x[\rho'] = V'$, respectively), such that $V$ and $V'$ are related by the interpretation of $A$.
- For each covariable $x$ of type $A$ in the environment $\Gamma$, both $\rho$ and $\rho'$ must substitute some covalue for $\alpha$ (call them $\alpha[\rho] = E$ and $\alpha[\rho'] = E'$, respectively) such that $E$ and $E'$ are related by the interpretation of $A$.
- For each property $\Phi$ assumed in the environment $\Gamma$, the interpretation of $\Phi$ must be true when given both $\rho$ and $\rho'$. Or in other words, $\Phi$ must relate $\rho$ and $\rho'$.

Singular syntactic properties $\Phi$ — as well as collections of hypotheses $\Delta$ — are interpreted as a binary predicate deciding whether or not that property holds under a given pair of

substitutions. This is equivalent to interpreting $\Phi$ or $\Delta$ as a binary relation on substitutions, just like we did for typing environments, that identifies which substitutions make the property true. The interpretation of these properties as relations comes in three different flavors, which correspond to the three different roles served by each specific $\Phi$:

- *Equalities*: There are three different forms of equalities. Two commands are considered equal under a pair of substitutions when they are $\perp\!\!\!\perp$-related after applying the left substitution to the left command and the right substitution to the right command. Similarly, two (co)terms are considered equal at a type $A$ under a pair of substitutions when applying those substitutions leads to $[\![A]\!]$-related (co)terms.
- *Quantifiers*: Universal quantifiers signify that a property holds under any possible extension allowed by the type of the quantified (co)variable. Universal quantification over a variable, $\forall x{:}A.\Phi$, relates two substitutions when $\Phi$ does, after extending the substitutions with any pair $[\![A]\!]$-related values for $x$. Universal quantification over a covariable is defined in the same way.
- *Logical connectives*: The logical connectives of implication ($\Phi \Rightarrow \Phi'$) and conjunction ($\Phi \wedge \Phi'$) are interpreted directly for each pair of substitutions. Equivalently, we can say that $[\![\Phi \wedge \Phi']\!]$ means $[\![\Phi]\!] \cap [\![\Phi']\!]$ using the intersection of relations ($\cap$) that we've used previously, and $[\![\Phi \Rightarrow \Phi']\!]$ means $[\![\Phi]\!] \implies [\![\Phi']\!]$ where ($\implies$) denotes the implication of relations.

Speaking more broadly, we can generalize the universal quantification and environment extension from Fig. 7 to range over pre-candidates that lie outside the syntactic type system. This generality will be needed as we simplify away the extraneous elements of a type that we don't need to consider while proving a property. For any pre-candidate $\mathbb{A}$ and binary substitution relations $\gamma$ and $\phi$, the two universal quantifiers and environment extensions are:

$$\rho\ (\forall x{:}\mathbb{A}.\phi)\ \rho' := \forall\ V\ \mathbb{A}\ V'.\ \rho[V/x]\ \phi\ \rho'[V'/x]$$
$$\rho\ (\forall \alpha \div \mathbb{A}.\phi)\ \rho' := \forall\ E\ \mathbb{A}\ E'.\ \rho[E/\alpha]\ \phi\ \rho'[E'/\alpha]$$
$$\rho\ (\gamma, x : \mathbb{A})\ \rho' := \rho\ \gamma\ \rho' \text{ and } x[\rho]\ \mathbb{A}\ x[\rho']$$
$$\rho\ (\gamma, \alpha \div \mathbb{A})\ \rho' := \rho\ \gamma\ \rho' \text{ and } \alpha[\rho]\ \mathbb{A}\ \alpha[\rho']$$

Last but not least are judgements of the form $\Gamma \mid \Delta \vdash \Phi$, which are interpreted as just true or false statements. The syntactic entailment $\vdash$ is interpreted as the boolean test for relational implication $\subseteq$, so that $[\![\Gamma \mid \Delta \vdash \Phi]\!]$ whenever the environment $[\![\Gamma]\!]$ combined with the constraints in $[\![\Delta]\!]$ implies the property $[\![\Phi]\!]$. In other words, we can understand $[\![\Gamma \mid \Delta \vdash \Phi]\!]$ pointwise as the equivalent statement

$$[\![\Gamma \mid \Delta \vdash \Phi]\!] = \forall\ \rho\ [\![\Gamma]\!]\ \rho'.\ \text{if } \rho\ [\![\Delta]\!]\ \rho' \text{ then } \rho\ [\![\Phi]\!]\ \rho'$$

that $\rho\ [\![\Phi]\!]\ \rho'$ holds for all possible substitutions $\rho\ [\![\Gamma]\!]\ \rho'$ given by the typing environment and satisfying the pre-condition $\rho\ [\![\Delta]\!]\ \rho'$. This implicational interpretation of entailment gives rise to some useful structure to reason about the semantics of judgements.

**Property 5.14.** *For any pre-candidate $\mathbb{A}$ and binary substitution relations $\gamma$, $\phi$, and $\phi'$:*

$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!] \qquad [\![\forall x{:}A.\Phi]\!] = \forall x{:}[\![A]\!].[\![\Phi]\!] \qquad \gamma, x : \mathbb{A} \subseteq \phi = \gamma \subseteq \forall x{:}\mathbb{A}.\ \phi$$

$$[\![\Gamma, \alpha \div A]\!] = [\![\Gamma]\!], \alpha \div [\![A]\!] \quad [\![\forall \alpha \div A.\Phi]\!] = \forall \alpha \div [\![A]\!].[\![\Phi]\!] \quad \gamma, \alpha \div \mathbb{A} \subseteq \phi = \gamma \subseteq \forall \alpha \div \mathbb{A}.\ \phi$$
$$[\![\Delta, \Phi]\!] = [\![\Delta]\!] \cap [\![\Phi]\!] \quad [\![\Phi \Rightarrow \Phi']\!] = [\![\Phi]\!] \implies [\![\Phi']\!] \quad (\gamma \cap \phi) \subseteq \phi' = \gamma \subseteq (\phi \implies \phi')$$

$$(\gamma, x : \mathbb{A}) \cap [\![\Delta]\!] = (\gamma \cap [\![\Delta]\!]), x : \mathbb{A} \qquad (\textit{if } x \notin FV(\Delta))$$
$$(\gamma, \alpha \div \mathbb{A}) \cap [\![\Delta]\!] = (\gamma \cap [\![\Delta]\!]), \alpha \div \mathbb{A} \qquad (\textit{if } \alpha \notin FV(\Delta))$$

*Furthermore, for any related $\rho\ \gamma\ \rho'$, we have related extensions $\rho[V/x]\ (\gamma, x : \mathbb{A})\ \rho[V'/x]$ for all $V\ \mathbb{A}\ V'$, and $\rho[E/\alpha]\ (\gamma, \alpha : \mathbb{A})\ \rho[E'/\alpha]$ for all $E\ \mathbb{A}\ E'$.*

### 5.4 Universal consistency of weak (co)induction

We now turn to justifying inductive and coinductive reasoning in terms of the above model. One key component is that (co)induction seeks to reason about a type by only considering the concrete structures of a type. For an inductive type like Nat, that means we want to consider only the zero and succ cases of values, and ignore the rest. Dually for coinductive types like Stream $A$ and $A \to B$, we want to consider only the head and tail cases of stream covalues and only the stack $V \cdot E$ cases for function covalues.

The first step in this direction is to notice that certain universal properties need to consider fewer cases for positively and negatively complete equality candidates. A *strict property on $x$* holds for all related values of $\mathrm{Pos}(\mathbb{A})$ exactly when it holds on only the values related by $\mathbb{A}$. Dually, a *productive property on $\alpha$* holds for all related covalues of $\mathrm{Neg}(\mathbb{A})$ exactly when it holds on only the covalues related by $\mathbb{A}$. Note that this fact does not depend on the evaluation strategy of the language, but is instead ensured by the strictness or productivity of the underlying property.

**Lemma 5.15** ((De)Constructive (Co)Induction)**.** *For any sound candidate $\mathbb{A}$ and substitution relation $\gamma$:*

1. $\gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi(x)]\!]$ *if and only if* $\gamma, x : \mathbb{A} \subseteq [\![\Psi(x)]\!]$, *and*
2. $\gamma, \alpha \div \mathrm{Neg}(\mathbb{A}) \subseteq [\![\Psi(\alpha)]\!]$ *if and only if* $\gamma, \alpha \div \mathbb{A} \subseteq [\![\Psi(\alpha)]\!]$.

**Proof** We use Property 5.14 to prove $\gamma, x : \mathbb{A} \subseteq [\![\Psi(x)]\!]$ and $\gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi(x)]\!]$ are equivalent statements generically for all $\gamma$ by induction on the syntax of $\Psi(x)$:

- $\langle x \| E \rangle = \langle x \| E' \rangle$ where $x$ is not free in $E$ or $E'$. First, note that $\mathbb{A} \sqsubseteq \mathrm{Pos}(\mathbb{A})$, so that $\forall x{:}\mathrm{Pos}(\mathbb{A}).[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!]$ implies $\forall x{:}\mathbb{A}.[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!]$ via this inclusion. Furthermore, $\forall x{:}\mathbb{A}.[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!]$ means

  $$\langle x \| E \rangle [V/x] = \langle V \| E \rangle \perp\!\!\!\perp \langle V' \| E' \rangle = \langle x \| E' \rangle [V'/x]$$

  for all $V\ \mathbb{A}\ V'$ (since $E[V/x] = E$ and $E'[V'/x] = E'$), and thus $E\ \mathbb{A}^{v\perp\!\!\!\perp}\ E'$ by the definition of orthogonality. Therefore $E\ \mathrm{Pos}(\mathbb{A})\ E'$ by Property 5.12, and thus

  $$\langle x \| E \rangle [V/x] = \langle V \| E \rangle \perp\!\!\!\perp \langle V' \| E' \rangle = \langle x \| E' \rangle [V'/x]$$

  for any $V\ \mathrm{Pos}(\mathbb{A})\ V'$, which means $\forall x{:}\mathrm{Pos}(\mathbb{A}).[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!]$. In other words,

  $$\forall x{:}\mathbb{A}.[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!] = \forall x{:}\mathrm{Pos}(\mathbb{A}).[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!]$$

are equivalent substitution relations, and thus more generally

$$\begin{aligned}\gamma, x : \mathbb{A} \subseteq [\![\langle x \| E \rangle = \langle x \| E' \rangle]\!] &= \gamma \subseteq \forall x{:}\mathbb{A}.[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!] \\ &= \gamma \subseteq \forall x{:}\mathrm{Pos}(\mathbb{A}).[\![\langle x \| E \rangle = \langle x \| E' \rangle]\!] \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\langle x \| E \rangle = \langle x \| E' \rangle]\!]\end{aligned}$$

- $\forall y{:}B.\Psi(x)$ where $y \neq x$. Applying Property 5.14:

$$\begin{aligned}\gamma, x : \mathbb{A} \subseteq [\![\forall y{:}B.\Psi(x)]\!] &= \gamma, x : \mathbb{A} \subseteq \forall y{:}[\![B]\!].[\![\Psi(x)]\!] \\ &= \gamma, x : \mathbb{A}, y : [\![B]\!] \subseteq [\![\Psi(x)]\!] \\ &= \gamma, y : [\![B]\!], x : \mathbb{A} \subseteq [\![\Psi(x)]\!] && (x \neq y) \\ &= \gamma, y : [\![B]\!], x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi(x)]\!] && (IH) \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}), y : [\![B]\!] \subseteq [\![\Psi(x)]\!] && (x \neq y) \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq \forall y{:}[\![B]\!].[\![\Psi(x)]\!] \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\forall y{:}B.\Psi(x)]\!]\end{aligned}$$

- $\forall \alpha \div B.\Psi(x)$. Follows by permuting the bindings of $x$ and $\alpha$ and applying the inductive hypothesis to $\gamma$ extended with $\alpha \div [\![B]\!]$ analogously to the previous case.
- $\Phi \Rightarrow \Psi(x)$ where $x$ is not free in $\Phi$. Applying Property 5.14:

$$\begin{aligned}\gamma, x : \mathbb{A} \subseteq [\![\Phi \Rightarrow \Psi(x)]\!] &= \gamma, x : \mathbb{A} \subseteq ([\![\Phi]\!] \implies [\![\Psi(x)]\!]) \\ &= (\gamma, x : \mathbb{A}) \cap [\![\Phi]\!] \subseteq [\![\Psi(x)]\!] \\ &= (\gamma \cap [\![\Phi]\!]), x : \mathbb{A} \subseteq [\![\Psi(x)]\!] && (x \notin FV(\Phi)) \\ &= (\gamma \cap [\![\Phi]\!]), x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi(x)]\!] && (IH) \\ &= (\gamma, x : \mathrm{Pos}(\mathbb{A})) \cap [\![\Phi]\!] \subseteq [\![\Psi(x)]\!] && (x \notin FV(\Phi)) \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Phi]\!] \implies [\![\Psi(x)]\!] \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Phi \Rightarrow \Psi(x)]\!]\end{aligned}$$

- $\Psi_1(x) \wedge \Psi_2(x)$. Note that $[\![\Psi_1(x) \wedge \Psi_2(x)]\!] = [\![\Psi_1(x)]\!] \cap [\![\Psi_2(x)]\!]$ so

$$\begin{aligned}&\gamma, x : \mathbb{A} \subseteq [\![\Psi_1(x) \wedge \Psi_2(x)]\!] \\ &= \gamma, x : \mathbb{A} \subseteq [\![\Psi_1(x)]\!] \cap [\![\Psi_2(x)]\!] \\ &= (\gamma, x : \mathbb{A} \subseteq [\![\Psi_1(x)]\!]) \text{ and } (\gamma, x : \mathbb{A} \subseteq [\![\Psi_2(x)]\!]) \\ &= (\gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi_1(x)]\!]) \text{ and } (\gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi_2(x)]\!]) && (IH) \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi_1(x)]\!] \cap [\![\Psi_2(x)]\!] \\ &= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Psi_1(x) \wedge \Psi_2(x)]\!]\end{aligned}$$

The "if" direction for property 2 follows analogously to the above using Property 5.12 for $\mathrm{Neg}(\mathbb{A})$ in the base case of an equality $\langle V \| \alpha \rangle = \langle V' \| \alpha \rangle$. ■

Lemma 5.15 is enough to prove the extensional rule $\omega\!\rightarrow$ for function types, since the interpretation $[\![A \to B]\!]$ corresponds exactly to a negatively-constructed type. As the largest equality candidate which relates call stacks built from related parts, we can isolate these call stacks as a negative type.

**Property 5.16** (Negative Functions). $[\![A \to B]\!] = \mathrm{Neg}([\![A]\!] \odot [\![B]\!])$ *where* $\mathbb{A} \odot \mathbb{B}$ *is the least relation on covalues such that:*

$$(V \cdot E)\ (\mathbb{A} \odot \mathbb{B})\ (V' \cdot E') := V\ \mathbb{A}\ V' \text{ and } E\ \mathbb{B}\ E'$$

**Lemma 5.17** ($\omega\to$). *Given* $\alpha, \beta, x \notin FV(\Delta)$, $[\![\Gamma, \alpha \div A \to B \mid \Delta \vdash \Psi(\alpha)]\!]$ *if and only if* $[\![\Gamma, x : A, \beta \div B \mid \Delta \vdash \Psi(x \cdot \beta)]\!]$.

**Proof** By viewing $[\![A \to B]\!]$ in terms core call-stack relation $[\![A]\!] \odot [\![B]\!]$ (Property 5.16), and using the fact that $\alpha \notin FV(\Delta)$ to commute the hypothesis with the binding (Property 5.14),

$$\begin{aligned}
[\![\Gamma, \alpha \div A \to B \mid \Delta \vdash \Psi(\alpha)]\!] &= ([\![\Gamma]\!], \alpha \div [\![A \to B]\!]) \cap [\![\Delta]\!] \subseteq [\![\Psi(\alpha)]\!] \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), \alpha \div [\![A \to B]\!] \subseteq [\![\Psi(\alpha)]\!] \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), \alpha \div \mathrm{Neg}([\![A]\!] \odot [\![B]\!]) \subseteq [\![\Psi(\alpha)]\!]
\end{aligned}$$

we learn from Lemma 5.15 that the quantification over $\alpha \div \mathrm{Neg}([\![A]\!] \odot [\![B]\!])$ is equivalent to the same quantification over call stacks:

$$\begin{aligned}
&([\![\Gamma]\!] \cap [\![\Delta]\!]), \alpha \div \mathrm{Neg}([\![A]\!] \odot [\![B]\!]) \subseteq [\![\Psi(\alpha)]\!] \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), \alpha \div [\![A]\!] \odot [\![B]\!] \subseteq [\![\Psi(\alpha)]\!] && \text{(Lemma 5.15)} \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), x : [\![A]\!], \beta \div [\![B]\!]. \subseteq [\![\Psi(x \cdot \beta)]\!] && (x, \beta \notin AV(\Gamma) \cup FV(\Delta) \cup FV(\Delta)) \\
&= ([\![\Gamma]\!], x : [\![A]\!], \beta \div [\![B]\!]) \cap [\![\Delta]\!] \subseteq [\![\Psi(x \cdot \beta)]\!] \\
&= [\![\Gamma, x : A, \beta \div B]\!] \cap [\![\Delta]\!] \subseteq [\![\Psi(x \cdot \beta)]\!] \\
&= [\![\Gamma, x : A, \beta \div B \mid \Delta \vdash \Psi(x \cdot \beta)]\!]
\end{aligned}$$

■

We can perform a similar inversion on the (co)inductive types, although not all at once. Rather, this bottom-up redefinition of natural numbers and streams must work incrementally. Beginning with the most extreme starting point (the least equality candidate $\mathrm{Pos}\{\}$ for inductive numbers and the greatest equality candidate $\mathrm{Neg}\{\}$ for coinductive streams), we iteratively build toward the final answer one step at a time. For the natural numbers, we use these interpretations of the zero and succ constructors as relations between values built by those constructors

$$\mathrm{zero}\ [\![\mathrm{zero}]\!]\ \mathrm{zero} := \text{trivially true} \qquad (\mathrm{succ}\ V)\ [\![\mathrm{succ}]\!](\mathbb{A})\ (\mathrm{succ}\ V') := V\ \mathbb{A}\ V'$$

in order to define larger and larger approximations of the $[\![\mathrm{Nat}]\!]$ equality candidate:

$$[\![\mathrm{Nat}]\!]_0 := \mathrm{Pos}\{\} \qquad [\![\mathrm{Nat}]\!]_{i+1} := \mathrm{Pos}([\![\mathrm{zero}]\!] \vee [\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i))$$

At the limit, the union of all under-approximations $\bigvee_i [\![\mathrm{Nat}]\!]_i$ is the Kleene-style fixed point definition (Kleene, 1971) of natural numbers. Thankfully, the dual construction of coinductive streams can be done in exactly the same way, just working from the other direction of the subtyping lattice. With these interpretations of the head and tail projections as relations between covalues built by those destructors

$$(\mathrm{head}\ E)\ [\![\mathrm{head}]\!](\mathbb{A})\ (\mathrm{head}\ E') := E\ \mathbb{A}\ E' \qquad (\mathrm{tail}\ E)\ [\![\mathrm{tail}]\!](\mathbb{A})\ (\mathrm{tail}\ E') := E\ \mathbb{A}\ E'$$

we can define smaller and smaller approximations of $[\![\mathrm{Stream}\, A]\!]$:

$$[\![\mathrm{Stream}\, A]\!]_0 := \mathrm{Neg}\{\} \qquad [\![\mathrm{Stream}\, A]\!]_{i+1} := \mathrm{Neg}([\![\mathrm{head}]\!]([\![A]\!]) \wedge [\![\mathrm{tail}]\!]([\![\mathrm{Stream}\, A]\!]_i))$$

Here, the intersection of over-approximations $\curlywedge_i [\![\mathrm{Stream}\, A]\!]_i$ is the dual Kleene-style fixed point definition of streams. These incremental fixed points define the same equality candidate as the Tarski-style fixed points from Fig. 7.

**Lemma 5.18** (Positive Numbers & Negative Streams)**.** *Under both call-by-value and call-by-name evaluation,*

$$[\![\mathrm{Nat}]\!] = \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i \qquad\qquad [\![\mathrm{Stream}\, A]\!] = \bigcurlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i$$

*Proof sketch* Generalizing the proof from (Downen & Ariola, 2023, Lemma 6.13) from sets to binary relations requires the analogous facts (Downen & Ariola, 2023, Lemmas 1.19 and 1.22) that

$$\bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i = \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i \qquad\qquad \bigcurlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i = \bigwedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i$$

The key to demonstrating that these two instances of unions of numbers and intersections of streams are equal is in showing that we can fully observe a constructed number or a stream projection of any size.

For numbers, notice $\bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i$ relates the following instance of the recursor to itself:

$$\mathbf{rec}_\infty := \mathbf{rec}\{\mathrm{zero} \to \mathrm{zero} \mid \mathrm{succ}\ \_ \to x.x\}\ \mathbf{with}\ \alpha \qquad\qquad \mathbf{rec}_\infty \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i\ \mathbf{rec}_\infty$$

Then, given any $V\ \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i\ \mathbf{rec}_\infty\ V'$, we can use the fact that $\langle V \| \mathbf{rec}_\infty \rangle \perp\!\!\!\perp \langle V' \| \mathbf{rec}_\infty \rangle$ to trace the reductions of the commands and show that $V\ [\![\mathrm{Nat}]\!]_i V'$ for some $i^{th}$ finite approximation.

Streams follow a similar logic. Notice that $\curlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i$ relates this stream to itself for any $V\ [\![A]\!]\ V'$:

$$\mathbf{corec}_\infty[V] := \mathbf{corec}\{\mathrm{head}\ \alpha \to \alpha \to \mathrm{tail}\ \_ \to \gamma.\gamma\}\ \mathbf{with}\ V$$

$$\mathbf{corec}_\infty[V] \bigcurlywedge_{i=0}^{\infty} [\![\mathrm{Stream}\, A]\!]_i\ \mathbf{corec}_\infty[V']$$

Then, given any $V\ [\![A]\!]\ V'$ and $E\ \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i\ \mathbf{rec}_\infty\ E'$, we can use the fact that $\langle \mathbf{corec}_\infty[V] \| E \rangle \perp\!\!\!\perp \langle \mathbf{corec}_\infty[V'] \| E' \rangle$ to trace the reductions of the commands and show that $E\ [\![\mathrm{Stream}\, A]\!]_i E'$ for some $i^{th}$ finite approximation.

It follows that these provide another definition of the least equality candidate closed under zero and succ, and the greatest equality candidate closed under head and tail, respectively. Since there can be only one least/greatest equality candidate satisfying the same closure condition, they must be the same as the ones in Fig. 7. □

The incremental nature of the Kleene-style redefinitions makes it easy to reason (co)-inductively over the $i^{th}$ approximation steps. This way, we can show that the premises to the

(co)inductive inference rules $\omega$Nat and $\omega$Stream are interpreted as equivalent statements to their conclusions.

**Lemma 5.19** ($\omega$Nat)**.** *Given* $x \notin FV(\Delta)$, $[\![\Gamma, x : \mathrm{Nat} \mid \Delta \vdash \Psi(x)]\!]$ *if and only if* $[\![\Gamma \mid \Delta \vdash \Psi(\mathrm{zero})]\!]$ *and* $[\![\Gamma, x : \mathrm{Nat} \mid \Delta, \Psi(x) \vdash \Psi(\mathrm{succ}\, x)]\!]$.

**Proof** The "only if" direction follows immediately, since the relations zero $[\![\mathrm{Nat}]\!]$ zero and $(\mathrm{succ}\, V)$ $[\![\mathrm{Nat}]\!]$ $(\mathrm{succ}\, V')$ hold for any $V$ $[\![\mathrm{Nat}]\!]$ $V'$.

For the "if" direction, assume $[\![\Gamma \mid \Delta \vdash \Psi(\mathrm{zero})]\!]$ and $[\![\Gamma, x : \mathrm{Nat} \mid \Delta, \Psi(x) \vdash \Psi(\mathrm{succ}\, x)]\!]$ hold, and we will show that $[\![\Gamma, x : \mathrm{Nat} \mid \Delta \vdash \Psi(x)]\!]$ holds, too. From Property 5.14 and Lemmas 5.15 and 5.18, it suffices to show that the following equivalent proposition holds:

$$\begin{aligned}
[\![\Gamma, x : \mathrm{Nat} \mid \Delta \vdash \Psi(x)]\!] &= ([\![\Gamma]\!], x : [\![\mathrm{Nat}]\!]) \cap [\![\Delta]\!] \subseteq [\![\Psi(x)]\!] \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), x : [\![\mathrm{Nat}]\!] \subseteq [\![\Psi(x)]\!] && \text{(Property 5.14)} \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), x : \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i \subseteq [\![\Psi(x)]\!] && \text{(Lemma 5.18)} \\
&= ([\![\Gamma]\!] \cap [\![\Delta]\!]), x : \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}]\!]_i \subseteq [\![\Psi(x)]\!] && \text{(Lemma 5.15)}
\end{aligned}$$

Let $\gamma = [\![\Gamma]\!] \cap [\![\Delta]\!]$, and we can now proceed by proving each individual approximation $\gamma, x : [\![\mathrm{Nat}]\!]_i \subseteq [\![\Psi(x)]\!]$ by induction on $i$.

- (Base case: 0) $[\![\mathrm{Nat}]\!]_0 = \mathrm{Pos}\{\}$, so we must show $\gamma, x : \mathrm{Pos}\{\} \subseteq [\![\Psi(x)]\!]$. By Lemma 5.15, this statement is equivalent to $\gamma, x : \{\} \subseteq [\![\Psi(x)]\!]$, which is vacuously true since there are no possible choices for $x$ in the empty pre-candidate $\{\}$.
- (Inductive case: $i+1$) $[\![\mathrm{Nat}]\!]_i = \mathrm{Pos}([\![\mathrm{zero}]\!] \vee [\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i))$. By Lemma 5.15, these statements

  $$\begin{aligned}
  &\gamma, x : [\![\mathrm{Nat}]\!]_{i+1} \subseteq [\![\Psi(x)]\!] \\
  &= \gamma, x : \mathrm{Pos}([\![\mathrm{zero}]\!] \vee [\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i)) \subseteq [\![\Psi(x)]\!] \\
  &= \gamma, x : [\![\mathrm{zero}]\!] \vee [\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i) \subseteq [\![\Psi(x)]\!] \\
  &= (\gamma, x : [\![\mathrm{zero}]\!] \subseteq [\![\Psi(x)]\!]) \text{ and } (\gamma, x : \mathrm{succ}([\![\mathrm{Nat}]\!]_i) \subseteq [\![\Psi(x)]\!])
  \end{aligned}$$

  are equivalent, and we must show that they hold. Since zero $[\![\mathrm{zero}]\!]$ zero is the only related values of $[\![\mathrm{zero}]\!]$, the assumption $[\![\Gamma \mid \Delta \vdash \Psi(\mathrm{zero})]\!]$ is equivalent to $\gamma, x : [\![\mathrm{zero}]\!] \subseteq [\![\Psi(x)]\!]$. Similarly, $(\mathrm{succ}\, V)$ $[\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i)$ $(\mathrm{succ}\, V')$ are the only related values of $[\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i)$ for any $V$ $[\![\mathrm{Nat}]\!]_i$ $V'$. By the inductive hypothesis, we know $\gamma, x : [\![\mathrm{Nat}]\!]_i \subseteq [\![\Phi(x)]\!]$. Because $[\![\mathrm{Nat}]\!]_i \leq [\![\mathrm{Nat}]\!]$, the assumption $[\![\Gamma, x : \mathrm{Nat} \mid \Delta, \Phi(x) \vdash \Phi(\mathrm{succ}\, x)]\!]$ implies $([\![\Gamma]\!], x : [\![\mathrm{Nat}]\!]_i) \cap \Delta \subseteq [\![\Phi(\mathrm{succ}\, x)]\!]$ which is equivalent to $\gamma, x : [\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i) \subseteq [\![\Phi(x)]\!]$. Therefore, the equivalent statements

  $$\begin{aligned}
  &(\gamma, x : [\![\mathrm{zero}]\!] \subseteq [\![\Psi(x)]\!]) \text{ and } (\gamma, x : \mathrm{succ}([\![\mathrm{Nat}]\!]_i) \subseteq [\![\Psi(x)]\!]) \\
  &= \gamma, x : [\![\mathrm{Nat}]\!]_{i+1} \subseteq [\![\Psi(x)]\!]
  \end{aligned}$$

  hold.

So since $\gamma, x : [\![\mathrm{Nat}]\!]_i \subseteq [\![\Psi(x)]\!]$ holds for every $i$, so too does

$$\gamma, x : \bigvee_{i=0}^{\infty} [\![\mathrm{Nat}_i]\!] \subseteq [\![\Psi(x)]\!] = [\![\Gamma, x : \mathrm{Nat} \mid \Delta \vdash \Psi(x)]\!]$$

■

**Lemma 5.20** ($\omega$Stream)**.** *Given* $\alpha, \beta \notin FV(\Delta)$, $[\![\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta \vdash \Psi(\alpha)]\!]$ *if and only if* $[\![\Gamma, \beta \div A \mid \Delta \vdash \Psi(\mathrm{head}\, \beta)]\!]$ *and* $[\![\Gamma, \alpha \div \mathrm{Stream}\, A \mid \Delta, \Psi(\alpha) \vdash \Psi(\mathrm{tail}\, \alpha)]\!]$.

**Proof** Analogous to Lemma 5.19. The "only if" direction is immediate since $(\mathrm{head}\, E)\ [\![\mathrm{Stream}\, A]\!]\ (\mathrm{head}\, E')$ holds for any $E\ [\![A]\!]\ E'$ and $(\mathrm{tail}\, E)\ [\![\mathrm{Stream}\, A]\!]\ (\mathrm{tail}\, E')$ holds for any $E\ [\![\mathrm{Stream}\, A]\!]\ E'$. For the "if" direction, it suffices to show the equivalent statement $([\![\Gamma]\!] \cap [\![\Delta]\!]), \alpha \div \bigwedge_i [\![\mathrm{Stream}\, A]\!]_i \subseteq [\![\Psi(\alpha)]\!]$ holds via Property 5.14 and Lemmas 5.15 and 5.18, which follows by induction on $i$ using Lemma 5.15 in a similar manner as in Lemma 5.19. ■

From this semantics of the main (co)inductive principles, we can prove soundness with respect to the model, analogous to (Downen & Ariola, 2023), which in turn lets us derive the fact that the universal program logic is a consistent approximation of observational equivalence in both call-by-name and call-by-value evaluation.

**Theorem 5.21** (Soundness)**.** *If* $\Gamma \mid \Delta \vdash \Phi$ *is derivable in the extensional program logic, then* $[\![\Gamma \mid \Delta \vdash \Phi]\!]$ *is true for both call-by-value and call-by-name evaluation.*

**Lemma 5.22.** $\alpha\ [\![\mathrm{Nat}]\!]\ \alpha$*, and* $x\ [\![\mathrm{Stream}\, A]\!]\ x$ *and* $x\ [\![A \to B]\!]\ x$ *for any* $\alpha$ *and* $x$*.*

**Proof** $\langle x \| V \cdot E \rangle \perp\!\!\!\perp \langle x \| V' \cdot E' \rangle$ by definition of $\perp\!\!\!\perp$, so that $x\ \ \mathrm{Neg}([\![A]\!] \odot [\![B]\!])\ \ x$ by Property 5.12, and thus $x\ [\![A \to B]\!]\ x$ by Property 5.16.

Dually, both $\langle \mathrm{zero} \| \alpha \rangle \perp\!\!\!\perp \langle \mathrm{zero} \| \alpha \rangle$ and $\langle \mathrm{succ}\, V \| \alpha \rangle \perp\!\!\!\perp \langle \mathrm{succ}\, V' \| \alpha \rangle$ by definition of $\perp\!\!\!\perp$. As a result, we have $\alpha\ [\![\mathrm{Nat}]\!]_i\ \alpha$ for all $i$: the case for $[\![\mathrm{Nat}]\!]_0 = \mathrm{Pos}\{\}$ is trivial since all covalues are related by $\mathrm{Pos}\{\}$, and the case for $[\![\mathrm{Nat}]\!]_{i+1} = \mathrm{Pos}([\![\mathrm{zero}]\!] \vee [\![\mathrm{succ}]\!] [\![\mathrm{Nat}]\!]_i)$ follows from the previously mentioned fact about $\perp\!\!\!\perp$ and Property 5.12. Finally, $\alpha\ \curlyvee_i [\![\mathrm{Nat}]\!]_i\ \alpha$ by Property 5.13, and thus $\alpha\ [\![\mathrm{Nat}]\!]\ \alpha$ by Lemma 5.18.

The fact that $x\ [\![\mathrm{Stream}\, A]\!]\ x$ follows from Properties 5.12 and 5.13 and Lemma 5.18 similarly to the above, using the fact that $\langle x \| \mathrm{head}\, E \rangle \perp\!\!\!\perp \langle x \| \mathrm{head}\, E' \rangle$ and $\langle x \| \mathrm{tail}\, E \rangle \perp\!\!\!\perp \langle x \| \mathrm{tail}\, E' \rangle$ by definition of $\perp\!\!\!\perp$. ■

**Theorem 5.23.** *In the extensional program logic, the following holds for both call-by-name and call-by-value evaluation:*

1. *If* $\Gamma \vdash c = c'$ *then* $\Gamma \vdash c \approx c'$.
2. *If* $\Gamma \vdash v = v' : A$ *then* $\Gamma \vdash v \approx v' : A$.
3. *If* $\Gamma \vdash e = e' \div A$ *then* $\Gamma \vdash e \approx e' \div A$.

**Proof** Suppose $\Gamma \vdash c = c'$ (the cases for $\Gamma \vdash v = v' : A$ and $\Gamma \vdash e = e' \div A$ are analogous) and let $C$ be any context such that $\Theta \vdash C[c]$ and $\Theta \vdash C[c']$, and thus $\Theta \vdash C[c] = C[c']$ by

congruence. From Theorem 5.21 it must be that $[\![\Theta \vdash C[c] = C[c']]\!]$, *i.e.,* for any substitution $\rho\ [\![\Theta]\!]\ \rho'$, then $C[c][\rho] \perp\!\!\!\perp C[c'][\rho]$. Note that all (co)variable type assignments in $\Theta$ have the form $\alpha \div \mathrm{Nat}$, $x : \mathrm{Stream}\, A$, and $x : A \to B$, so by Lemma 5.22, we know that the $[\![\Theta]\!]$ relates the identity substitution to itself. Thus a valid instance of $[\![\Theta \vdash C[c] = C[c']]\!]$ is just $C[c] \perp\!\!\!\perp C[c']$, meaning $C[c] \mapsto\!\!\!\!\rightarrow d \sim d' \leftarrow\!\!\!\!\mapsto C[c']$. In other words, we know $\Gamma \vdash c \approx c'$ by definition of observational equivalence. ■

**Theorem 3.7.** *The extensional program logic in Fig. 5 is consistent for both the call-by-name and call-by-value semantics.*

**Proof** A corollary of Theorem 5.23, since observational equivalence is a consistent congruence by definition. ■

### *5.5 Strong call-by-value induction and call-by-name coinduction*

In the general case, we need to interleave a (positive or negative) completion while building up a (co)inductive equality candidate like $[\![\mathrm{Nat}]\!]_i$ or $[\![\mathrm{Stream}\, A]\!]_i$. But in the specific case where the evaluation strategy lines up nicely, we get a much simpler definition for call-by-value inductive types and call-by-name coinductive types.

**Lemma 5.24** (Strict Construction of Naturals)**.** *Under call-by-value evaluation,* $V\ [\![\mathrm{Nat}]\!]\ V'$ *if and only if* $V = V' = \mathrm{succ}^n\ \mathrm{zero}$ *for some n. Furthermore* $[\![\mathrm{Nat}]\!] = \mathrm{Pos}(\mathbb{N})$ *under call-by-value evaluation, where* $\mathbb{N}$ *is the reflexive relation on only the hereditary numeric constructions,* i.e., *the smallest binary relation such that* $(\mathrm{succ}^n\ \mathrm{zero})\ \mathbb{N}\ (\mathrm{succ}^n\ \mathrm{zero})$.

**Proof** Let $deep_{\mathrm{Nat}} = \mathbf{rec}\{\mathrm{zero} \to \mathrm{zero} \mid \mathrm{succ}\ \_ \to y.\ \mathrm{succ}\ y\}$ **with** $\alpha$, and note that $\alpha \div \mathrm{Nat} \vdash deep_{\mathrm{Nat}} \div \mathrm{Nat}$ is a well-typed covalue, so by reflexivity it is equal to itself at type Nat. Adequacy (Theorem 5.21) then ensures that $\alpha \div [\![\mathrm{Nat}]\!] \subseteq [\![deep_{\mathrm{Nat}} = deep_{\mathrm{Nat}} \div \mathrm{Nat}]\!]$ and since $\alpha\ [\![\mathrm{Nat}]\!]\ \alpha$ (Lemma 5.22), we know specifically that $deep_{\mathrm{Nat}}\ [\![\mathrm{Nat}]\!]\ deep_{\mathrm{Nat}}$. From the soundness of $[\![\mathrm{Nat}]\!]$, we know that $V\ [\![\mathrm{Nat}]\!]\ V'$ implies $\langle V \| deep_{\mathrm{Nat}} \rangle \perp\!\!\!\perp \langle V' \| deep_{\mathrm{Nat}} \rangle$, or in other words $\langle V \| deep_{\mathrm{Nat}} \rangle \mapsto\!\!\!\!\rightarrow d \sim d' \leftarrow\!\!\!\!\mapsto \langle V' \| deep_{\mathrm{Nat}} \rangle$ In call-by-value, the only such values that satisfy this relationship are $V = \mathrm{succ}^n\ \mathrm{zero}$ and $V' = \mathrm{succ}^{n'}\ \mathrm{zero}$, for some $n, n'$ iterations of the successor. Specifically, $\mu$-abstractions are not values in call-by-value, and the only other choices for values all lead to computations that get stuck at some unobservable command.

To see that $n = n'$, consider what happens if $n \neq n'$, and suppose (without loss of generality) that $n < n'$. Here is a family of well-typed covalues that peel off $n$ successors:

$$minus_0 := \alpha \qquad minus_{n+1} := \mathbf{rec}\{\mathrm{zero} \to \mathrm{zero} \mid \mathrm{succ}\ x \to \_.x\}\ \mathbf{with}\ minus_n$$

So that, for any $m \leq m'$, $\langle \mathrm{succ}^{m'}\ \mathrm{zero} \| minus_m \rangle \mapsto\!\!\!\!\rightarrow \langle \mathrm{succ}^{m'-m}\ \mathrm{zero} \| \alpha \rangle$. Note again that $\alpha \div \mathrm{Nat} \vdash minus_n \div \mathrm{Nat}$ is a well-typed covalue, so that it is equal to itself by reflexivity, and thus by adequacy (Theorem 5.21) and Lemma 5.22, $minus_n\ [\![\mathrm{Nat}]\!]\ minus_n$. From soundness of $[\![\mathrm{Nat}]\!]$, it follows that the following inconsistent equivalence holds

$$\langle \mathrm{succ}^n\ \mathrm{zero} \| minus^n \rangle \mapsto\!\!\!\!\rightarrow \langle \mathrm{zero} \| \alpha \rangle \sim \langle \mathrm{succ}(\mathrm{succ}^{n'-n-1}\ \mathrm{zero}) \| \alpha \rangle \leftarrow\!\!\!\!\mapsto \langle \mathrm{succ}^{n'} \| minus^n \rangle$$

which contradicts the definition of $\sim$. Therefore, $n = n'$, and thus $V\ [\![\mathrm{Nat}]\!]\ V'$ if and only if $V = V' = \mathrm{succ}^n\ \mathrm{zero}$ exactly.

In other words, $\mathbb{N} = [\![\mathrm{Nat}]\!]^{v+}$, and so $\mathrm{Pos}(\mathbb{N}) = \mathrm{Pos}([\![\mathrm{Nat}]\!]^{v+}) = \mathrm{Pos}([\![\mathrm{Nat}]\!])$ by Property 5.11, and $\mathrm{Pos}([\![\mathrm{Nat}]\!]) = [\![\mathrm{Nat}]\!]$, because $[\![\mathrm{Nat}]\!]$ is already a complete equality candidate. ■

**Lemma 5.25** (Strict Destruction of Streams)**.** *Under call-by-name evaluation,* $E\ [\![\mathrm{Stream}\,A]\!]\ E'$ *if and only if* $E = \mathrm{tail}^n(\mathrm{head}\,E_1)$ *and* $E' = \mathrm{tail}^n(\mathrm{head}\,E_1')$ *for some n and* $E\ [\![A]\!]\ E'$*. Furthermore* $[\![\mathrm{Stream}\,A]\!] = \mathrm{Neg}(\mathbb{S}([\![A]\!]))$ *under call-by-value evaluation, where* $\mathbb{S}([\![A]\!])$ *is the reflexive relation on only the hereditary stream projections,* i.e., *the smallest binary relation such that* $(\mathrm{tail}^n(\mathrm{head}\,E))\ \mathbb{S}(\mathbb{A})\ (\mathrm{tail}^n(\mathrm{head}\,E'))$ *if and only if* $E\ \mathbb{A}\ E'$.

**Proof** Analogous to the proof for Lemma 5.24. Using the value $deep_{\mathrm{Stream}} = \mathbf{corec}\{\mathrm{head}\,\alpha \to \alpha \mid \mathrm{tail}\,\_ \to \beta.\,\mathrm{tail}\,\beta\}\ \mathbf{with}\ x$, which has the type $x : \mathrm{Stream}\,A \vdash deep_{\mathrm{Stream}\,A}$, we can conclude that $E\ [\![\mathrm{Stream}\,A]\!]\ E'$ if and only if $E = \mathrm{tail}^n(\mathrm{head}\,E_0)$ and $E' = \mathrm{tail}^{n'}(\mathrm{head}\,E_0')$ for some $E_0\ [\![A]\!]\ E_0'$. Furthermore, it must be that $n = n'$, because we can peel off $n$ tail projections using the value

$$raise_0 := x \qquad raise_{n+1} := \mathbf{corec}\{\mathrm{head}\,\alpha \to \mathrm{head}\,\alpha \mid \mathrm{tail}\,\beta \to \_.\beta\}\ \mathbf{with}\ raise_n$$

which derives an inconsistent equivalence $\langle x \| \mathrm{head}\,E_0 \rangle \sim \langle x \| \mathrm{tail}(\mathrm{tail}^{n'-n-1}(\mathrm{head}\,E_0')) \rangle$ that contradicts the definition of $\perp\!\!\!\perp$. Therefore, $[\![\mathrm{Stream}\,A]\!] = \mathrm{Neg}(\mathbb{S}([\![A]\!]))$. ■

These simpler definitions for $[\![\mathrm{Nat}]\!]$ and $[\![\mathrm{Stream}\,A]\!]$ make it possible to verify the stronger (co)inductive rules $\sigma$Nat and $\sigma$Stream, which do not place any restrictions on the kinds of properties they may prove.

**Lemma 5.26** ($\sigma$Nat)**.**
$[\![\Gamma, x : \mathrm{Nat} \vdash \Phi]\!]$ *if and only if* $[\![\Gamma \vdash \Phi[\mathrm{zero}/x]]\!]$ *and* $[\![\Gamma, x : \mathrm{Nat}, \Phi \vdash \Phi[\mathrm{succ}\,x/x]]\!]$.

**Proof** By Lemmas 5.15 and 5.24, the meaning of $[\![\Gamma, x : \mathrm{Nat} \vdash \Phi]\!]$ is equivalent to:

$$\begin{aligned}[\![\Gamma, x : \mathrm{Nat} \vdash \Phi]\!] &= [\![\Gamma]\!], x : [\![\mathrm{Nat}]\!] \subseteq [\![\Phi]\!] \\ &= [\![\Gamma]\!], x : \mathrm{Pos}(\mathbb{N}) \subseteq [\![\Phi]\!] \\ &= [\![\Gamma]\!], x : \mathbb{N} \subseteq [\![\Phi]\!]\end{aligned}$$

Which can be proved equivalent to $[\![\Gamma]\!] \subseteq [\![\Phi[\mathrm{zero}/x]]\!]$ and $[\![\Gamma]\!], x : \mathbb{N} \subseteq [\![\Phi[\mathrm{succ}\,x/x]]\!]$ by an ordinary induction on the numeric constructions in $\mathbb{N}$. ■

**Lemma 5.27** ($\sigma$Stream)**.** $[\![\Gamma, \alpha \div \mathrm{Stream}\,A \vdash \Phi]\!]$ *if and only if* $[\![\Gamma, \beta \div A \vdash \Phi[\mathrm{head}\,\beta/\alpha]]\!]$ *and* $[\![\Gamma, \alpha \div \mathrm{Stream}\,A, \Phi \vdash \Phi[\mathrm{tail}\,\alpha/\alpha]]\!]$.

**Proof** Analogous to Lemma 5.26 by duality using Lemmas 5.15 and 5.25. ■

**Theorem 5.28** (Soundness)**.** *If* $\Gamma \vdash \Phi$ *is derivable in the strong call-by-value program logic, then* $[\![\Gamma \vdash \Phi]\!]$ *is true under call-by-value evaluation. Likewise, If* $\Gamma \vdash \Phi$ *is derivable in the strong call-by-name program logic, then* $[\![\Gamma \vdash \Phi]\!]$ *is true under call-by-name evaluation.*

**Proof** The same as the proof of Theorem 5.21 with one additional case for $\sigma$Nat in call-by-value or $\sigma$Stream and $\sigma\to$ in call-by-name. ■

**Theorem 5.29.** *In the strong call-by-value program logic and operational semantics, and in the strong call-by-name program logic and operational semantics, the following hold:*

1. *If* $\Gamma \vdash c = c'$ *then* $\Gamma \vdash c \approx c'$.
2. *If* $\Gamma \vdash v = v' : A$ *then* $\Gamma \vdash v \approx v' : A$.
3. *If* $\Gamma \vdash e = e' \div A$ *then* $\Gamma \vdash e \approx e' \div A$.

**Theorem 3.9.** *The strong call-by-name and call-by-value program logics are consistent.*

**Proof** Both Theorems 3.9 and 5.29 are proved the same as Theorems 3.7 and 5.23, using the generalized Theorem 5.28 in place of Theorem 5.21. ■

## 6 Related Work

*Foundations and implementations of coinduction*

Coinduction has been heavily used in different domains: to prove security properties of low-level code (Leroy & Rouaix, 1998; Appel & Felty, 2000), to prove regular expressions containments (Henglein & Nielsen, 2011), to show language equivalence of a non-deterministic finite automata (Bonchi & Pous, 2013), to reason about software-defined networks (Foster *et al.*, 2015), and probabilistic functional programs (Lago *et al.*, 2014). The relation between coinductive reasoning and programming languages theory has been consolidated in (Hur *et al.*, 2012).

Coq is one of the few formal verifiers with a long history of native support for coinduction (Giménez, 1996; Chlipala, 2013). Yet, coinductive proof development in Coq is not easy: such proofs are not checked until they are completed, which is too late for Coq's interactive proof development. It is often said that coinductive proofs have a very different "feel." Much work on improving the mechanization of coinduction has been done in a form of structural coinduction in the Isabelle/HOL theorem prover (Traytel *et al.*, 2012) with the aim to improve the ease of use (Blanchette *et al.*, 2014, 2015). There, the built-in notion of coinductive proof is based on bisimulation, and so the implementation has support to automatically derive the bisimulation relation. In contrast, here we formulate structural coinduction directly on the shape of observations — with bisimulation as just one, optional, mode of use — so bisimulation relations never arise for many proofs. Instead, the closest direct implementation of structural coinduction as presented here is the implementation of copatterns in Agda (Abel *et al.*, 2013). Coinduction has also been brought to program verification in Dafny (Leino & Moskal, 2014) and Liquid Haskell (Mastorou *et al.*, 2022).

While we focus on methods of reasoning based on computation and formal classical logic, other approaches have been employed for reasoning about corecursive programs. From the domain-theoretic approach, Scott and de Bakker's fixed-point induction (Bakker, 1980) is one of the early examples. However, applying fixed-point induction is not so easy, because it requires knowledge of the CPO semantics of types and their properties. In its

place, other lemmas such as the take lemma (Bird & Wadler, 1988), and its improvement, the approximation lemma (Bird, 1998; Hutton & Gibbons, 2001), reframes the problem of observing infinite objects through a family of more familiar questions about induction on finite objects: two streams are equal if all their finite approximations are. Similarly, Mastorou *et al.* (2022) encodes coinduction in terms of induction by adding an index. Gibbons & Hutton (2005) give a survey of these other methods. The formalization here, in contrast, identifies and reifies the "inductive" nature inherent in the context of coinduction to use directly in the coinductive principle without encoding or a change of representation.

Another approach to coinduction involves the *hidden algebras* (Goguen & Malcolm, 1999) behind coinductive modules in the object-oriented paradigm. This has been used to formulate *circular coinductive proofs* for object-oriented behavior (Goguen *et al.*, 2000) and concurrent processes (Popescu & Gunter, 2010). Circular coinduction has been implemented in Coq (Endrullis *et al.*, 2013), and generalized to a form of *parameterized* coinductive proofs (Hur *et al.*, 2013).

### *Coinductive reasoning principles*

Using coinduction makes it possible to avoid working with numbers (Gordon, 1994). Instead, coinductive proofs are completely based on the structure of programs, analogous to bisimulation (Sangiorgi, 2009). Our notion of strong (co)induction also allows for local reasoning about valid applications of the (co)inductive hypothesis, which leads to a compositional development of (co)inductive proofs. Similarly, Paco (Hur *et al.*, 2013) aims to aid the development of coinductive proofs through both compositionality (local, not global, correctness criteria) and incrementality (new knowledge may be accumulated as the proof is developed). We showed how the strong version of our program logic encompasses well-known principles of strong induction and bisimulation of corecursive processes.

Corecursion—and the coinductive principles to reason about them—have also been generalized to capture common patterns that occur in programming but which make structural coinduction more difficult to verify. For example, consider the following usual definition of the infinite Fibonacci stream in Haskell:

$$\begin{aligned} \mathit{fibs} &= 0 : 1 : \mathit{sums}\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs}) \\ \mathit{sums}\ (x : \mathit{xs})\ (y : \mathit{ys}) &= (x + y) : (\mathit{sums}\ \mathit{xs}\ \mathit{ys}) \end{aligned}$$

From experience, we know this is a well-behaved infinite stream: we can access any particular number in finite time. However, the reason why is non-trivial, which can be more easily seen when translated as follows into the abstract machine language used here:

$$\begin{aligned} \langle \mathit{fibs} \| \mathrm{head}\ \alpha \rangle &= \langle 0 \| \alpha \rangle \\ \langle \mathit{fibs} \| \mathrm{tail}(\mathrm{head}\ \alpha) \rangle &= \langle 1 \| \alpha \rangle \\ \langle \mathit{fibs} \| \mathrm{tail}(\mathrm{tail}\ \alpha) \rangle &= \langle \mathit{sums} \| \mu\beta_1.\langle \mathit{fibs} \| \beta_1 \rangle \cdot \mu\beta_2.\langle \mathit{fibs} \| \mathrm{tail}\ \beta_2 \rangle \cdot \alpha \rangle \\ \\ \langle \mathit{sums} \| \mathit{xs} \cdot \mathit{ys} \cdot \mathrm{head}\ \alpha \rangle &= \langle \mathrm{head}\ \mathit{xs} + \mathrm{head}\ \mathit{ys} \| \alpha \rangle \\ \langle \mathit{sums} \| \mathit{xs} \cdot \mathit{ys} \cdot \mathrm{tail}\ \alpha \rangle &= \langle \mathit{sums} \| \mu\beta_1.\langle \mathit{xs} \| \mathrm{tail}\ \beta_1 \rangle \cdot \mu\beta_2.\langle \mathit{xs} \| \mathrm{tail}\ \beta_2 \rangle \cdot \alpha \rangle \end{aligned}$$

The trouble is that *fibs*'s coinductive case for $\mathrm{tail}(\mathrm{tail}\ \alpha)$ recursively references back to *fibs* with some syntactically unknown observers—$\beta_1$ and $\mathrm{tail}\ \beta_2$ respectively—which might be

far larger than the smaller case tail $\alpha$. Despite this, the reason *fibs* is well-founded has to do with the helper function *sums*: the application $\langle sums \| xs \cdot ys \cdot \alpha \rangle$ will replicate a stream projection of *exactly* $\alpha$'s length (*i.e.,* the same number of tail projections) to both *xs* and *ys*. A function with this special property is known as "friendly" in Isabelle/HOL (Blanchette *et al.*, 2015, 2017) and "abstemious" in Dafny (Leino & Moskal, 2014).

Since justifying these kinds of definitions are well-founded is already complex, reasoning about them is even more so. For example, consider this product function of two streams from (Blanchette *et al.*, 2015), also defined in terms of sums:

$$prods\ (x : xs)\ (y : ys) = (x \times y) : (sums\ (prods\ (x : xs)\ ys)\ (prods\ xs\ (y : ys)))$$

Both the sum and product of two streams should be commutative. It is straightforward enough to show commutativity of sums—$sums\ xs\ ys = sums\ ys\ xs$—directly because it is defined only in terms of itself and plain addition. However, *prod* is defined in terms of *sums*, which gets in the way of a bisimulation argument. Isabelle/HOL supports the notion of coinduction "up to" (Blanchette *et al.*, 2015, 2017) in order to better automate the bisimulation relation for these kinds of programs. We conjecture that the notion of structural coinduction developed here—which does not require bisimulation at all—can sidestep the issue entirely in this kind of example. In particular, translating the above function to the abstract machine language looks like:

$$\langle prods \| xs \cdot ys \cdot \text{head}\ \alpha \rangle = \langle x \times y \| \alpha \rangle$$
$$\langle prods \| xs \cdot ys \cdot \text{tail}\ \alpha \rangle = \langle sums \| \mu\beta_1.\langle prods \| xs \cdot \text{tail}\ ys \cdot \beta_1 \rangle \cdot \mu\beta_2.\langle prods \| \text{tail}\ xs \cdot ys \cdot \beta_2 \rangle \cdot \alpha \rangle$$

Suppose we accept this definition as well-founded because of *sums*' properties—either marking *sums* "friendly" or "abstemious" as above, or using sized types (Abel, 2006). Then we know that $\beta_1$ and $\beta_2$ will always be instantiated by another observation no bigger than $\alpha$ (*i.e.,* we know $\beta_1 \leq \alpha$ and $\beta_2 \leq \alpha$ according to Section 4). We could then proceed to prove

$$\langle prod \| xs \cdot ys \cdot \alpha \rangle = \langle prod \| ys \cdot xs \cdot \alpha \rangle$$

using our notion of strong coinduction on $\alpha$. The main case for $\alpha = \text{tail}\ \alpha'$ would then look like the following, with a coinductive hypothesis (*CIH*) applicable to *any* observation of $\alpha'$s size or smaller, which includes the $\beta_1 \leq \alpha'$ and $\beta_2 \leq \alpha'$ instantiated by *sums*:

$$\begin{aligned}
&\langle prods \| xs \cdot ys \cdot \text{tail}\ \alpha' \rangle \\
&= \langle sums \| \mu\beta_1.\langle prods \| xs \cdot \text{tail}\ ys \cdot \beta_1 \rangle \cdot \mu\beta_2.\langle prods \| \text{tail}\ xs \cdot ys \cdot \beta_2 \rangle \cdot \alpha' \rangle && (prods\ \text{def.}) \\
&= \langle sums \| \mu\beta_1.\langle prods \| \text{tail}\ ys \cdot xs \cdot \beta_1 \rangle \cdot \mu\beta_2.\langle prods \| \text{tail}\ xs \cdot ys \cdot \beta_2 \rangle \cdot \alpha' \rangle && (CIH, \beta_1 \leq \alpha') \\
&= \langle sums \| \mu\beta_1.\langle prods \| \text{tail}\ ys \cdot xs \cdot \beta_1 \rangle \cdot \mu\beta_2.\langle prods \| ys \cdot \text{tail}\ xs \cdot \beta_2 \rangle \cdot \alpha' \rangle && (CIH, \beta_2 \leq \alpha') \\
&= \langle sums \| \mu\beta_2.\langle prods \| ys \cdot \text{tail}\ xs \cdot \beta_2 \rangle \cdot \mu\beta_1.\langle prods \| \text{tail}\ ys \cdot xs \cdot \beta_1 \rangle \cdot \alpha' \rangle && (sums\ \text{commut.}) \\
&= \langle prods \| ys \cdot xs \cdot \text{tail}\ \alpha' \rangle && (prods\ \text{def.})
\end{aligned}$$

### *Logical relation and program equality*

Our overall approach to proving properties about programs using syntactic rules (the program logic) that are then shown to be part of a consistent-by-definition operational model follows the general approach of logical relations (Statman, 1985), Tait's method (Tait, 1967), and realizability (Kleene, 1945). However, we cannot use Tait's original method

as formulated, because types in our language classify both terms *and* coterms. Instead, we use the formulation of logical relations based on orthogonality between two opposing sets, which has been developed in multiple places including linear logic (Girard, 1987), classical realizability (Krivine, 2005), and $\top\top$-closed relations (Pitts, 2000, 1997a), and symmetric candidates (Barbanera & Berardi, 1994). A key feature of our model is the built-in notion that types are first modeled by a chosen set of values (for positive types) or covalues (for negative types). This generation from (co)values comes from a study of polarity and focusing in linear logic (Munch-Maccagnoni, 2009), which makes similar distinctions between call-by-value and call-by-name interpretations of types as call-by-push-value (Levy, 2001).

We also make use of the notion of *candidates* — an initial definition describing all possible models of types, whose interpretation will come later — as part of our proof. Traditionally, the candidate-based approach is used to prove properties about programs in languages that have impredicative polymorphism like system F (Girard, 1972). Here, we use the same idea to construct inductive and coinductive types by quantifying over all their possible approximations: either smaller subtypes in the case of induction or larger supertypes in the case of coinduction. These approximations are then assembled into their least or greatest fixed points using both the Knaster-Tarski (Knaster, 1928; Tarski, 1955) and Kleene (Kleene, 1971) constructions — which are equivalent by Lemma 5.18 — using the lattice structure present in the logical relations model corresponding to intersection and union types (Coppo & Dezani-Ciancaglini, 1978; Sallé, 1978; Pottinger, 1980).

## 7 Conclusion

This paper defines a language for providing a computational foundation of (co)inductive reasoning principles which brings out their duality. The impact of the evaluation strategy is also illustrated. Whereas induction does not fully work in call-by-name, co-induction has the same issues in call-by-value. The (co)inductive principles are derived from the definition of types in terms of *construction* or *destruction*, using *control flow* instead of bisimulation to guide the coinductive hypothesis. In the end, the logical dualities in computation—between data and codata; information flow and control flow—provide a unified framework for using and reasoning with (co)inductive types.

As future work, we would like to formalize more advanced notions of coinduction and bisimilarity (Pous & Sangiorgi, 2012) that relax the constraint that the processes need to proceed completely in sync, thus allowing one to compare processes that"almost" compute in the same way. We would also like to show that Paco's coinductive principles (Hur *et al.*, 2013) can also be encoded as an application of strong coinduction—giving a computational model for its proofs—where accumulated knowledge may be represented as the accumulator of a corecursive process.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2245516. The authors declare no competing interests.

## References


Abel, A. (2006) *A Polymorphic Lambda Calculus with Sized Higher-Order Types*. Ph.D. thesis, Ludwig-Maximilians-Universität München.

Abel, A., Pientka, B., Thibodeau, D. and Setzer, A. (2013) Copatterns: Programming infinite structures by observations. *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13, pp. 27–38. ACM.

Appel, A. W. and Felty, A. P. (2000) A semantic model of types and machine instuctions for proof-carrying code. Wegman, M. N. and Reps, T. W. (eds), *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000* pp. 243–253. ACM.

Bakker, J. W. d. (1980) *Mathematical Theory of Program Correctness*. Prentice-Hall, Inc.

Barbanera, F. and Berardi, S. (1994) A symmetric lambda calculus for "classical" program extraction. *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings* pp. 495–515.

Barwise, J. and Moss, L. (1997) Vicious circles. on the mathematics of non-wellfounded phenomena. *The Journal of Symbolic Logic* 1039–1040.

Bird, R. (1998) *Introduction to Functional Programming Using Haskell (second edition)*. Prentice-Hall, Inc.

Bird, R. and Wadler, P. (1988) *An Introduction to Functional Programming*. Prentice-Hall, Inc.

Blanchette, J. C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A. and Traytel, D. (2014) Truly modular (co)datatypes for isabelle/hol. Klein, G. and Gamboa, R. (eds), *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Lecture Notes in Computer Science 8558, pp. 93–110. Springer.

Blanchette, J. C., Popescu, A. and Traytel, D. (2015) Foundational extensible corecursion: a proof assistant perspective. Fisher, K. and Reppy, J. H. (eds), *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015* pp. 192–204. ACM.

Blanchette, J. C., Bouzy, A., Lochbihler, A., Popescu, A. and Traytel, D. (2017) Friends with benefits - implementing corecursion in foundational proof assistants. Yang, H. (ed), *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Lecture Notes in Computer Science 10201, pp. 111–140. Springer.

Bonchi, F. and Pous, D. (2013) Checking NFA equivalence with bisimulations up to congruence. Giacobazzi, R. and Cousot, R. (eds), *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013* pp. 457–468. ACM.

Burstall, R. M. (1969) Proving properties of programs by structural induction. *The Computer Journal* **12**(1):41–48.

Chlipala, P. (2013) *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

Coppo, M. and Dezani-Ciancaglini, M. (1978) A new type assignment for $\lambda$-terms. *Arch. Math. Log.* **19**(1):139–156.

Curien, P.-L. and Herbelin, H. (2000) The duality of computation. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00, pp. 233–243. ACM.

Downen, P. and Ariola, Z. M. (2018) A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming* **28**:e3.

Downen, P. and Ariola, Z. M. (2023) Classical (co)recursion: Mechanics. *Journal of Functional Programming* **33**:e4.

Downen, P., Johnson-Freyd, P. and Ariola, Z. M. (2015) Structures for structural recursion. *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP '15, pp. 127–139. ACM.

Downen, P., Ariola, Z. M. and Ghilezan, S. (2019) The duality of classical intersection and union types. *Fundamenta Informaticae* **170**(1-3):39–92.

Downen, P., Johnson-Freyd, P. and Ariola, Z. M. (2020) Abstracting models of strong normalization for classical calculi. *Journal of Logical and Algebraic Methods in Programming* **111**:100512.

Endrullis, J., Hendriks, D. and Bodin, M. (2013) Circular coinduction in Coq using bisimulation-up-to techniques. *Proceedings of the 4th International Conference on Interactive Theorem Proving*. ITP'13, p. 354–369. Springer-Verlag.

Foster, N., Kozen, D., Milano, M., Silva, A. and Thompson, L. (2015) A coalgebraic decision procedure for netkat. Rajamani, S. K. and Walker, D. (eds), *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* pp. 343–355. ACM.

Gibbons, J. and Hutton, G. (2005) Proof methods for corecursive programs. *Fundamenta Informaticae* **66**(04):353–366.

Giménez, E. (1996) An application of co-inductive types in coq: Verification of the alternating bit protocol. Berardi, S. and Coppo, M. (eds), *Types for Proofs and Programs* pp. 135–152. Springer Berlin Heidelberg.

Girard, J. Y. (1972) *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7.

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50**(1):1–101.

Gödel, K. (1980) On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosophical Logic* **9**(2):133–142.

Goguen, J., Lin, K. and Rosu, G. (2000) Circular coinductive rewriting. *Proceedings, Automated Software Engineering '00* 07.

Goguen, J. A. and Malcolm, G. (1999) Hidden coinduction: behavioural correctness proofs for objects. *Mathematical Structures in Computer Science* **9**(3):287–319.

Gordon, A. (1994) A tutorial on co-induction and functional programming. *Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland* pp. 78–95. Springer London.

Gordon, M. (2017) *Corecursion and coinduction: what they are and how they relate to recursion and induction*. https://www.cl.cam.ac.uk/archive/mjcg/Blog/WhatToDo/Coinduction.pdf.

Hagino, T. (1987) A typed lambda calculus with categorical type constructors. *Category Theory and Computer Science* pp. 140–157. Springer Berlin Heidelberg.

Harper, R. (2016) *Practical Foundations for Programming Languages*. 2nd edn. Cambridge University Press.

Henglein, F. and Nielsen, L. (2011) Regular expression containment: Coinductive axiomatization and computational interpretation. *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11, p. 385–398. Association for Computing Machinery.

Herbelin, H. (2005) *C'est maintenant qu'on calcule : Au coeur de la dualité*. Habilitation thesis, Université Paris 11.

Hur, C., Dreyer, D., Neis, G. and Vafeiadis, V. (2012) The marriage of bisimulations and kripke logical relations. Field, J. and Hicks, M. (eds), *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012* pp. 59–72. ACM.

Hur, C.-K., Neis, G., Dreyer, D. and Vafeiadis, V. (2013) The power of parameterization in coinductive proof. *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13, p. 193–206. Association for Computing Machinery.

Hutton, G. and Gibbons, J. (2001) The generic approximation lemma. *Information Processing Letters* **79**(08):197–201.

Kleene, S. C. (1945) On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic* **10**(4):109–124.

Kleene, S. C. (1971) *Introduction to Metamathematics*. Bibliotheca Mathematica, a Series of Monographs on Pure and. Wolters-Noordhoff.

Knaster, B. (1928) Un theoreme sur les functions d'ensembles. *Ann. Soc. Polon. Math.* **6**:133–134.
Kozen, D. and Silva, A. (2017) Practical coinduction. *Mathematical Structures in Computer Science* **27**(7):1132–1152.
Krivine, J.-L. (2005) Realizability in classical logic. *Interactive models of computation and program behaviour*, vol. 27, pp. 197–229. Société Mathématique de France.
Lago, U. D., Sangiorgi, D. and Alberti, M. (2014) On coinductive equivalences for higher-order probabilistic functional programs. Jagannathan, S. and Sewell, P. (eds), *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014* pp. 297–308. ACM.
Leino, K. R. M. and Moskal, M. (2014) Co-induction simply. Jones, C., Pihlajasaari, P. and Sun, J. (eds), *FM 2014: Formal Methods* pp. 382–398. Springer International Publishing.
Leroy, X. and Rouaix, F. (1998) Security properties of typed applets. MacQueen, D. B. and Cardelli, L. (eds), *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998* pp. 391–403. ACM.
Levy, P. B. (2001) *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London.
Mastorou, L., Papaspyrou, N. and Vazou, N. (2022) Coinduction inductively: Mechanizing coinductive proofs in liquid haskell. *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022, p. 1–12. Association for Computing Machinery.
Munch-Maccagnoni, G. (2009) Focalisation and classical realisability. *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings* pp. 409–423.
Pierce, B. C. (2002) *Types and Programming Languages*. 1st edn. The MIT Press.
Pitts, A. (1997a) A note on logical relations between semantics and syntax. *Logic Journal of IGPL* **5**(4):589–601.
Pitts, A. M. (1997b) Operationally-based theories of program equivalence. *Semantics and Logics of Computation* **14**:241.
Pitts, A. M. (2000) Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* **10**(3):321–359.
Popescu, A. and Gunter, E. L. (2010) Incremental pattern-based coinduction for process algebra and its Isabelle formalization. *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures*. FOSSACS'10, p. 109–127. Springer-Verlag.
Pottinger, G. (1980) A type assignment for the strongly normalizable $\lambda$-terms. Seldin, J. P. and Hindley, J. R. (eds), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 561–577. Academic Press.
Pous, D. and Sangiorgi, D. (2012) Enhancements of the bisimulation proof method. Sangiorgi, D. and Rutten, J. (eds), *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press.
Rutten, J. (2019) *The Method of Coalgebra: Exercises in coinduction*. CWI, Amsterdam, The Netherlands.
Sallé, P. (1978) Une extension de la théorie des types en lambda-calcul. Ausiello, G. and Böhm, C. (eds), *Fifth International Conference on Automata, Languages and Programming*. Lecture Notes in Computer Science 62, pp. 398–410. Springer-Verlag.
Sangiorgi, D. (2009) On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.* **31**(4).
Statman, R. (1985) Logical relations and the typed $\lambda$-calculus. *Information and control* **65**(2-3):85–97.
Tait, W. W. (1967) Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* **32**(2):198–212.
Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* **5**(2):285 – 309.
Traytel, D., Popescu, A. and Blanchette, J. C. (2012) Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012* pp. 596–605. IEEE Computer Society.