JFP, 58 pages, 2021. © Cambridge University Press 2021
doi:10.1017/xxxxx

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 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.

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). 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, in practice, 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 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 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 type system 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 domain theory to explain the duality between induction and coinduction (Gordon, 1994). Both the informal technique and formal system are proved consistent with a computational model where every syntactically-derived equality is a proof of observational equivalence.

Having (co)inductive reasoning principles expressed within a calculus follows our previous work on defining a calculus that directly expresses the dualities commonly seen in logic. 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, 2023, 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. Even though our approach does not need any mathematical sophistication it still captures 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 classical logic, 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 induction does not always work in non-strict languages such as Haskell. For example, the following optimization

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

can be proved by induction, but it does not hold according to a call-by-name evaluation strategy. Letting Ω stand for a non-terminating expression, notice that plugging in Ω 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?

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

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 Ω). Plugging in $0, \Omega$ for xs gives us 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

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

In place of non-termination, plugging in $0, (\texttt{abort}\ 2)$ for xs also serves as another example using abort as a side-effect, where the left-hand side $\textbf{let}\, z = \text{tail}(0, (\texttt{abort}\ 2))\, \textbf{in}\, 1$ aborts with 2, but the right-hand side returns 1.

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

$$\begin{aligned} \mathit{evens}\ (x_0, x_1, x_2, x_3, \dots) &= x_0, x_2, x_4, \dots \\ \mathit{odds}\ (x_0, x_1, x_2, x_3, \dots) &= x_1, x_3, x_5, \dots \\ \mathit{merge}\ (x_0, x_1, x_2, \dots)\ (y_0, y_1, y_2, \dots) &= x_0, y_0, x_1, y_1, x_2, y_2, \dots \end{aligned}$$

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:

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

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

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

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 $\mathit{odds}\ (0, \Omega) = \mathit{evens}\ \Omega$ which does not terminate. Thus, $\mathit{merge}\ (\mathit{evens}\ (0, \Omega))\ (\mathit{odds}\ (0, \Omega))$ 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:

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

but that value can be differentiated from the starting stream $0, 1, 2, \Omega$ by asking for the third element (2) via $\mathrm{head}(\mathrm{tail}(\mathrm{tail}\ \mathit{xs}))$. 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 Ω, the last one is forgotten. As before, using an `abort` in place of Ω gives us alternative abort-based counter-examples. So plugging in the partial stream value $0, \texttt{abort}\ 1$ causes $\mathit{merge}\ (\mathit{evens}\ (0, \texttt{abort}\ 1))\ (\mathit{odds}\ (0, \texttt{abort}\ 1))$ immediately aborts with 1 instead of returning some value, and plugging in $0, 1, 2, \texttt{abort}\ 3$ returns the smaller partial stream $0, 1, \texttt{abort}\ 3$.

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. However, the reasoning techniques discussed here are applicable to other data and codata types. More specifically, we have:

- Section 2 provides examples of applying (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. It also illustrates the flaws in taking an overly naïve view of the rules for (co)inductive reasoning—they can simultaneously be too strong and too weak in the broader context.
- Section 4 gives a sound, formal equational theory for reasoning (co)inductively about (co)recursive programs, which overcomes the flaws raised in Section 3: the theory is safe for both call-by-name and call-by-value evaluation, yet sufficient to prove interesting equalities. In certain cases, we can soundly enhance the equational theory further with even stronger versions of the inductive or coinductive rules.
- Section 5 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 6 provides a proof that the equational theories in Section 4 are sound: syntactic formal proofs of equality imply semantic observational 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:

$$\begin{aligned} plus\ \ \mathrm{zero}\ \ y &= y\\ plus\ (\mathrm{succ}\ x)\ y &= \mathrm{succ}(plus\ x\ y)\end{aligned}$$

Why *plus* is 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*; 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).

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 $plus\ \ \mathrm{zero}\ \ y = y$ for any number y, so it holds by just calculation with no further verification required. However, addition's second identity law, $plus\ x\ \ \mathrm{zero} = x$ 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, $plus\ x\ \ \mathrm{zero} = x$.

Proof By induction on the structure of the value x

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

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

■

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.

2.1 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

$$\textbf{coinductive data}\ \text{Stream}\ a\ \textbf{where}$$
$$\text{Cons} : a \to \text{Stream}\ a \to \text{Stream}\ a$$

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 $always : a \to \text{Stream}\ a$ which returns the stream that always contains the same value of type a, or $repeat : (a \to a) \to a \to \text{Stream}\ a$ that builds an infinite stream from some original a by repeatedly applying a given function to it—can be defined cyclically like so:

$$always\ x = \text{Cons}\ x\ (always\ x) \qquad repeat\ f\ x = \text{Cons}\ x\ (repeat\ f\ (f\ x))$$

Why are *always* and *repeat* well-founded? The answer here is not so clear; from a 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. Or in other words, the self-references of *always* and *repeat* 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 definitions of *always* and *repeat*.

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

Proof Assume the coinductive hypothesis

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

From there,

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

$= \mathit{always}\ x$ def. of *always*

■

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 $\mathit{repeat}\ (\lambda y.y)\ x = \mathit{always}\ x$. From the coinductive hypothesis, it follows that $\mathit{repeat}\ (\lambda y.y)\ x = \mathit{always}\ x$, 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 need 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 to ever larger scales. 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 coinductively-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:

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

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 *repeat* functions can be rewritten in

terms of *copatterns* (Abel *et al.*, 2013) like so:

$$\begin{aligned} \mathrm{head}(\mathit{always}\ x) &= x & \mathrm{head}(\mathit{repeat}\ f\ x) &= x \\ \mathrm{tail}(\mathit{always}\ x) &= \mathit{always}\ x & \mathrm{tail}(\mathit{repeat}\ f\ x) &= \mathit{repeat}\ f\ (f\ x) \end{aligned}$$

Here, head and tail are seen as projection *functions*, and the streams returned by *always* x and *repeat* $f\ x$ 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 is modeled on the sequent calculus (Downen *et al.*, 2015), which provides a syntax for writing contextual observations as first-class objects. In this sequent style, a Greek letter α, β, ..., stands for an observer of values, and the *command* $\langle x \| \alpha \rangle$ says that the observer α is applied to x, or symmetrically, that the value x is returned to α.

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 α is expecting to observe a value of type a, then the *composition* head α observes a value of type Stream a by taking its first element and passing it to α. Similarly, if β is expecting to observe a value of type Stream a, then tail β observes a value of type Stream a by discarding its first element and passing the rest to β. The two different views—head and tail as functions versus observations—are always equal to one another:

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

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

$$\begin{aligned} \langle \mathit{always}\ x \| \mathrm{head}\ \beta \rangle &= \langle x \| \beta \rangle & \langle \mathit{repeat}\ f\ x \| \mathrm{head}\ \beta \rangle &= \langle x \| \beta \rangle \\ \langle \mathit{always}\ x \| \mathrm{tail}\ \alpha' \rangle &= \langle \mathit{always}\ x \| \alpha' \rangle & \langle \mathit{repeat}\ f\ x \| \mathrm{tail}\ \alpha' \rangle &= \langle \mathit{repeat}\ f\ (f\ x) \| \alpha' \rangle \end{aligned}$$

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 α' in the corecursive call $\langle \mathit{always}\ x \| \alpha' \rangle$ is a piece of the original observer tail α' from the command $\langle \mathit{always}\ x \| \mathrm{tail}\ \alpha' \rangle$. Similarly, the observer of the corecursive call $\langle \mathit{repeat}\ f\ (f\ x) \| \alpha' \rangle$ came from a piece of the observer in the proceeding command $\langle \mathit{repeat}\ f\ x \| \mathrm{tail}\ \alpha' \rangle$. So copattern-matching over observers restores the symmetry between recursive functions (which *consume* inductively-defined arguments) and corecursive functions (which *produce* coinductively-defined results).

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 α.

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

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

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

$$\langle \mathit{repeat}\ (\lambda y.y)\ x \| \alpha' \rangle = \langle \mathit{always}\ x \| \alpha' \rangle.$$

From there,

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

■

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 $\langle \mathit{repeat}\ (\lambda y.y)\ x \| \alpha' \rangle = \langle \mathit{always}\ x \| \alpha' \rangle$ carries enough information to fully dictate the valid contexts in which it can be used. In particular, we can only assume the goal (that $\mathit{repeat}\ (\lambda y.y)\ x$ is equal to $\mathit{always}\ x$) when observed by α', 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 $\langle \mathit{repeat}\ (\lambda y.y)\ x \| \alpha \rangle = \langle \mathit{always}\ x \| \alpha \rangle$, 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 Theorem 2.2 and Example Theorem 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 α):

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

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

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

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 *repeat* via observational equivalence. Starting with a generic α, we can convert these function applications to observations on top of α to match the definition of *repeat* as follows:

$$\langle \text{head}(\mathit{repeat}\ f\ x) \| \alpha \rangle =^{\text{by } 2.1} \langle \mathit{repeat}\ f\ x \| \text{head}\,\alpha \rangle = \langle x \| \alpha \rangle$$

$$\langle \text{tail}(\mathit{repeat}\ f\ x) \| \alpha \rangle =^{\text{by } 2.1} \langle \mathit{repeat}\ f\ x \| \text{tail}\ \alpha \rangle \quad = \langle \mathit{repeat}\ f\ (f\ x) \| \alpha \rangle$$

and thus by observational equivalence, we have

$$\text{head}(\mathit{repeat}\ f\ x) = x \tag{2.2}$$
$$\text{tail}(\mathit{repeat}\ f\ x) = \mathit{repeat}\ f\ (f\ x) \tag{2.3}$$

Notice that these equations derived by observational equivalence are exactly the same as the purely functional, copattern-matching definition of *repeat* 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.

$$\langle \mathit{maps}\ f\ s \| \text{head}\ \beta \rangle = \langle f\ (\text{head}\ s) \| \beta \rangle$$
$$\langle \mathit{maps}\ f\ s \| \text{tail}\ \alpha' \rangle = \langle \mathit{maps}\ f\ (\text{tail}\ s) \| \alpha' \rangle$$

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 *repeat*.

Example Theorem 2.4. *For all functions f of type $A \to B$, values x of type A, and observers α of type* Stream A, $\langle \mathit{maps}\ f\ (\mathit{repeat}\ f\ x) \| \alpha \rangle = \langle \mathit{repeat}\ f\ (f\ x) \| \alpha \rangle$.

Proof By structural coinduction on the observer α (leaving the value x generic):

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

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

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

$$\langle \mathit{maps}\ f\ (\mathit{repeat}\ f\ x) \| \alpha' \rangle = \langle \mathit{repeat}\ f\ (f\ x) \| \alpha' \rangle$$

for *all* values x of type A.

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

■

Mutual coinduction

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

$$\langle evens\ s \| \text{head}\ \beta \rangle = \langle s \| \text{head}\ \beta \rangle \qquad \langle odds\ s \| \alpha \rangle = \langle evens\ (\text{tail}\ s) \| \alpha \rangle$$
$$\langle evens\ s \| \text{tail}\ \alpha' \rangle = \langle odds\ (\text{tail}\ s) \| \alpha' \rangle$$

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:[1]

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

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 α 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 α (leaving s_1 and s_2 generic):

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

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

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

- $\alpha = \text{tail}\ \alpha'$. 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.

$$\langle evens\ (merge\ s_1\ s_2) \| \text{tail}\ \alpha' \rangle$$

[1] The observation $\text{tail}(\text{head}\ \beta)$ should be read as first observing the tail of a strem and then applying the head to that result.

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

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

■

Strong coinduction

Let us try to prove that the property $\langle \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s)\|\alpha\rangle = \langle s\|\alpha\rangle$ holds for all values s and observers α 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 α. We can easily prove the property if $\alpha = \mathrm{head}(\beta)$. If $\alpha = \mathrm{tail}(\beta)$ then we proceed by case analysis on β. If $\beta = \mathrm{head}(\beta')$ the proof goes through without any issues. If $\beta = \mathrm{tail}(\beta')$ we need to prove $\langle \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s)\|\mathrm{tail}(\mathrm{tail}(\beta'))\rangle = \langle s\|\mathrm{tail}(\mathrm{tail}(\beta'))\rangle$ and note that the coinductive hypothesis is:

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

for a generic s. We then have:

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

At this point, we would like to apply the coinductive hypothesis 2.9, which however does not hold on β'. What we need is a strong version of coinduction. This is not surprising since the same issue comes up with induction. If $\alpha = \mathrm{tail}\ \beta$, we assume the property to hold not just for the immediate subcontext β but also for tail β' when α is $\mathrm{tail}(\mathrm{tail}\ \beta')$, as shown in the following proof.

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

Proof By strong coinduction on the structure of the observer α (where we leave the stream value s generic):

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

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

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

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

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

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

for *all* values s of type Stream A.

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

■

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 an abstract machine language with both recursion and corecursion (Downen & Ariola, 2023), 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.

The syntax and operational semantics of our (co)recursive abstract machine language are given in Fig. 1, and its type system is in Fig. 2. 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 λ-calculus-like language and coterms correspond to continuations that arise during computation. Of note, the machine in Fig. 1 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

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.

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.

Besides the ordinary function type $A \to B$, the (co)recursive abstract machine (Fig. 2) 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 (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 λ-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

Types (A), typing environments (Γ), and typing judgements (J)

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma, x : A, \alpha \div B, \Gamma' \vdash \tau}{\Gamma, \alpha \div B, x : A, \Gamma' \vdash \tau}\ \textit{ExLR} \qquad \frac{\Gamma, x : A, y : B, \Gamma' \vdash \tau}{\Gamma, y : B, x : A, \Gamma' \vdash \tau}\ \textit{ExLL}$$

$$\frac{\Gamma, \alpha \div A, x : B, \Gamma' \vdash \tau}{\Gamma, x : B, \alpha \div A, \Gamma' \vdash \tau}\ \textit{ExRL} \qquad \frac{\Gamma, \alpha \div A, \beta \div B, \Gamma' \vdash \tau}{\Gamma, \beta \div B, \alpha \div A, \Gamma' \vdash \tau}\ \textit{ExRR}$$

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

Encoding λ-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:

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

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

at a time. To check the types of these (co)terms and validity of commands, we use a typing environment Γ that describes both the variables x and covariables α 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 *bound* by the environment Γ. Notice that we make the simplifying assumption throughout this paper that environments Γ never bind the same (co)variable x or α more than once (*i.e.,* every x or α bound by a Γ are distinct), ruling out cases like $x : \mathrm{Nat}, y : \mathrm{Nat}, x : \mathrm{Nat} \to \mathrm{Nat}$.[2]

Since this abstract machine language doesn't have an application like the λ-calculus, how can it express basic compositions like $f(g(x))$? These sorts of terms can be encoded thanks to the μ- 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 μ assigns the name α 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 α. More generally, we can use the syntactic sugar given in Fig. 3 as macro-definitions for all the usual expressions of λ-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 μ to name the current evaluation context α, 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 $\mathrm{succ}\ v$ which is not allowed by the syntax of Fig. 1? Again, we can utilize μ 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 *repeat* 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

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

The application *plus* 2 2 (with $2 = \mathrm{succ}(\mathrm{succ}\ \mathrm{zero})$) becomes:

$$\begin{aligned}
\mu\alpha.\langle 2 \| \mathbf{rec}\{\mathrm{zero} \to 2 \mid \mathrm{succ}\ _ \to y.\ \mathrm{succ}\ y\}\ \mathbf{with}\ \alpha \rangle &\mapsto\!\!\!\!\to \\
\mu\alpha.\langle 1 \| \mathbf{rec}\{\mathrm{zero} \to 2 \mid \mathrm{succ}\ _ \to y.\ \mathrm{succ}\ y\}\ \mathbf{with}\ \tilde{\mu} y.\langle \mathrm{succ}\ y \| \alpha \rangle \rangle &\mapsto\!\!\!\!\to \\
\mu\alpha.\langle \mathrm{zero} \| \mathbf{rec}\{\mathrm{zero} \to 2 \mid \mathrm{succ}\ _ \to y.\ \mathrm{succ}\ y\}\ \mathbf{with}\ \tilde{\mu} y.\langle \mathrm{succ}(\mathrm{succ}\ y) \| \alpha \rangle \rangle &\mapsto\!\!\!\!\to \\
\mu\alpha.\langle 2 \| \tilde{\mu} y.\langle \mathrm{succ}(\mathrm{succ}\ y) \| \alpha \rangle \rangle &\mapsto\!\!\!\!\to \\
\mu\alpha.\langle 4 \| \alpha \rangle &
\end{aligned}$$

Notice how at each recursive step the continuation gets updated: α, $\tilde{\mu} y.\langle \mathrm{succ}\ y \| \alpha \rangle$, and $\tilde{\mu} y.\langle \mathrm{succ}(\mathrm{succ}\ y) \| \alpha \rangle$.

[2] We use FV to denote the set of free variables an expression refers to, *e.g.,* $FV(v)$ for the free variables in a term or $FV(c)$ for the free variables in a command, and $BV(\Gamma)$ to denote the set of variables bound by the typing assignments in Γ. In the type system, we could interpret Γ as an unordered set with *at most one* type assignment to any x or α. However, the next section will allow for dependencies *within* Γ, and we aim to present the usual left-to-right dependency ordering inside the telescoping Γ. In preparation for these internal dependencies, we consider Γ as an ordered sequence, and use the exchange rules *ExRR*, *ExRL*, *ExLR*, and *ExLL* to swap the ordering of pairs that don't depend on one another.

The *repeat* 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 *double* stands for the function $\lambda x.\mu\alpha.\langle \text{succ succ } x \| \alpha\rangle$ then the even natural numbers can be represented as $\mu\alpha.\langle \mathit{repeat} \| \mathit{double} \cdot \text{zero} \cdot \alpha\rangle$, and the third element of this stream is computed as so:

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

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

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—relates many more programs by giving up determinism. One of the key allowances is that we can apply the reduction steps in *any* context (no matter how deep within the given expression) and in *any* direction (both forward and backward). Such an equational theory for the (co)recursive abstract machine is given in Fig. 4. All judgments have the form $\Gamma \vdash \Phi$, where Γ is the assumptions and Φ 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 $e = e' \div A$).

The bulk of the rules are dedicated to *compatibility*: the allowance that equalities may be applied in *any* context. Though there are many different compatibility 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 can be easily derived by induction on the typing derivation and the axioms *VarR* and *VarL*. Still, in the following, we will sometimes refer to these reflexivity rules:

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

Equational properties (Φ) and judgements (J):

$$\begin{array}{rl} \mathit{Prop} \ni \Phi ::= & c = c' \mid v = v' : A \mid e = e' \div A \\ \mathit{Env} \ni \Gamma ::= & \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A \qquad \text{(all } x \text{ and } \alpha \text{ bound by } \Gamma \text{ are distinct)} \\ \mathit{Judge} \ni J ::= & \boxed{\Gamma \vdash \Phi} \end{array}$$

Equivalence:

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

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

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

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

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

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

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

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

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

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

(Co)Variable Exchange:

$$\frac{\Gamma, x : A, \alpha \div B, \Gamma' \vdash \Phi}{\Gamma, \alpha \div B, x : A, \Gamma' \vdash \Phi}\ \mathit{ExLR} \qquad \frac{\Gamma, x : A, y : B, \Gamma' \vdash \Phi}{\Gamma, y : B, x : A, \Gamma' \vdash \Phi}\ \mathit{ExLL}$$

$$\frac{\Gamma, \alpha \div A, x : B, \Gamma' \vdash \Phi}{\Gamma, x : B, \alpha \div A, \Gamma' \vdash \Phi}\ \mathit{ExRL} \qquad \frac{\Gamma, \alpha \div A, \beta \div B, \Gamma' \vdash \Phi}{\Gamma, \beta \div B, \alpha \div A, \Gamma' \vdash \Phi}\ \mathit{ExRR}$$

Equality of computation:

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

Fig. 4: Intensional equational theory of computation.

where the double horizontal lines 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, $c \mapsto c'$:

$$\dfrac{\dfrac{\Gamma \vdash c}{\Gamma \vdash c = c}\ \mathit{Refl} \qquad c \mapsto c'}{\Gamma \vdash c = c'}\ \mathit{Red}$$

and we will simply write

$$\dfrac{\Gamma \vdash c \quad c \mapsto c'}{\Gamma \vdash c = c'}\ \mathit{Red}$$

The equational theory in Fig. 4 is *intensional* in the sense that 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.[3] In other words, the intensional theory of Fig. 4 only considers two expressions equal when they reduce to the same normal forms. For example, it is easy to show by reduction that $\mathit{plus}\ \mathrm{zero}\ x \mapsto\!\!\!\!\rightarrow x$ because *plus* was defined by recursion on its first argument, but $\mathit{plus}\ x\ \ \mathrm{zero}$ doesn't reduce at all, even though it is nonetheless equivalent to zero in any context. Thus,

$$\alpha \div \mathrm{Nat} \vdash \langle \mathit{plus} \| \mathrm{zero} \cdot x \cdot \alpha \rangle = \langle x \| \alpha \rangle$$

is derivable but

$$\alpha \div \mathrm{Nat} \vdash \langle \mathit{plus} \| x \cdot \mathrm{zero} \cdot \alpha \rangle = \langle x \| \alpha \rangle$$

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

3.2 Extensional equational theory — First attempt

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 define an *extensional* equational theory that 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.

To implement this idea of observational equality, we would like formal inference rules that embody these two relationships between different forms of equality.

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

[3] For this reason, the intensional equational theory is sometimes also called *definitional* equality.

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

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

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

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

Fig. 5: A naïve attempt at rules for extensionality and (co)induction.

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

Thankfully, the "only if" direction of both of these is implemented by the *Cut* compatibility rule already present in the intensional equational theory (Fig. 4):

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

If we already know two terms $\Gamma \vdash v = v' : A$ are equal, then for any other equal coterms $\Gamma \vdash e = e' \div A$, *Cut* lets us conclude that their pointwise combination gives equal commands $\Gamma \vdash \langle v \| e \rangle = \langle v' \| e' \rangle$. Dually, starting with two equal coterms, *Cut* lets us combine them with any equal terms to give equal commands.

The "if" direction is the difficult one because it introduces a "for all" in the premise. Taken literally, we would like formal inference rules which capture these notions of extensionality on the right (*ER*?) and left (*EL*?), but without the metatheoretic $\forall$:

$$\frac{\forall(\Gamma \vdash e = e' \div A). \quad \Gamma \vdash \langle v \| e \rangle = \langle v' \| e' \rangle}{\Gamma \vdash v = v' : A}\ \mathit{ER?} \qquad \frac{\forall(\Gamma \vdash v = v' : A). \quad \Gamma \vdash \langle v \| e \rangle = \langle v' \| e' \rangle}{\Gamma \vdash e = e' \div A}\ \mathit{EL?}$$

These informal deductions are implemented by the formal inference rules of the extensional theory given in Fig. 5.

The most generic version of extensionality is given by $\sigma\tilde{\mu}$ and $\sigma\mu$ rules in Fig. 5, 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). Note that, instead of quantifying over all terms or coterms, $\sigma\tilde{\mu}$ and $\sigma\mu$ introduce an implicit quantification over generic (co)values in the form of a new free (co)variable that does not appear anywhere else in the command. 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:

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

Thankfully, we do not need to add these two deductions as inference rules in the theory. The substitution of values for variables is derived like so:

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

$$\dfrac{\Gamma \vdash \langle V' \| \tilde{\mu}x.c'\rangle = c[V/x] \qquad \dfrac{\Gamma \vdash c[V/x] = \langle V' \| \tilde{\mu}x.c'\rangle}{\Gamma \vdash c[V/x] = c'[V'/x]}\ Symm \qquad \langle V' \| \tilde{\mu}x.c'\rangle \mapsto_{\tilde{\mu}} c'[V'/x]}{\Gamma \vdash c[V/x] = c'[V'/x]}\ Red$$

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

Similarly, the extensional η rules for μ and $\tilde{\mu}$ (Herbelin, 2005)

$$\dfrac{\Gamma \vdash v : A}{\Gamma \vdash \mu\alpha.\langle v \| \alpha\rangle = v : A}\ \eta_{\mu} \qquad\qquad \dfrac{\Gamma \vdash e \div A}{\Gamma \vdash \tilde{\mu}x.\langle x \| e\rangle = e \div A}\ \eta_{\tilde{\mu}}$$

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

$$\dfrac{\dfrac{\Gamma \vdash v : A}{\Gamma, \alpha \div A \vdash \langle \mu\alpha.\langle v \| \alpha\rangle \| \alpha\rangle}\ Cut, VarL, ActR \qquad \langle \mu\alpha.\langle v \| \alpha\rangle \| \alpha\rangle \mapsto_{\mu} \langle v \| \alpha\rangle}{\Gamma, \alpha \div A \vdash \langle \mu\alpha.\langle v \| \alpha\rangle \| \alpha\rangle = \langle v \| \alpha\rangle}\ Red}{\Gamma \vdash \mu\alpha.\langle v \| \alpha\rangle = v : A}\ \sigma\mu$$

The other inferences rules in Fig. 5 capture stronger forms of extensionality that are specific to a particular type. First, the $\sigma\!\rightarrow$ rule expresses a form of extensionality for functions in terms of call stacks. The only canonical covalue of type $A \rightarrow B$ has the form $V \cdot E$. $\sigma\!\rightarrow$ says that testing a property on a generic call stack $x \cdot \beta$ is sufficient to generalize that property over *all* α of type $A \rightarrow B$. We can derive the η axiom for functions in $\overline{\lambda}\mu\tilde{\mu}$[4]

$$\dfrac{\Gamma \vdash v : A \rightarrow B}{\Gamma \vdash \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle = v : A \rightarrow B}\ \eta_{\rightarrow}$$

by applying both the general $\sigma\mu$ and the function-specific $\sigma\!\rightarrow$ like so:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash v : A \rightarrow B}{\Gamma, y : A, \beta \div B \vdash \langle \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle \| y \cdot \beta\rangle} \quad \langle \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle \| y \cdot \beta\rangle \mapsto\!\!\!\!\rightarrow_{\beta_{\rightarrow}\mu} \langle v \| y \cdot \beta\rangle}{\Gamma, y : A, \beta \div B \vdash \langle \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle \| y \cdot \beta\rangle = \langle v \| y \cdot \beta\rangle}\ Refl, Reds}{\Gamma, \gamma \div A \rightarrow B \vdash \langle \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle \| \gamma\rangle = \langle v \| \gamma\rangle}\ \sigma\!\rightarrow}{\Gamma \vdash \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle = v : A \rightarrow B}\ \sigma\mu$$

The last two rules capture dual notions of structural (co)induction over the two (co)-inductive types Nat and Stream A. The σNat rule summarizes this deduction for proving a property Φ over any number x using an infinite number of premises:

$$\dfrac{\Gamma \vdash \Phi[\mathrm{zero}/x] \quad \Gamma \vdash \Phi[\mathrm{succ\ zero}/x] \quad \Gamma \vdash \Phi[\mathrm{succ(succ\ zero)}/x] \quad \ldots}{\Gamma, x : \mathrm{Nat} \vdash \Phi}$$

[4] This η axiom for functions is equivalent to the familiar η law of the λ-calculus, which can be seen by macro-expanding the syntactic sugar for application according to Fig. 3: $\lambda x.(v\ x) = \lambda x.\mu\alpha.\langle v \| x \cdot \alpha\rangle =_{\eta_{\rightarrow}} v$.

This deduction is justified from the reasoning that these values—zero, succ zero, succ(succ zero)—are *all the canonical values* of Nat; testing Φ on all of them is sufficient to generalize Φ over any x of type Nat. σNat 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:

$$\Phi(\text{zero}) \Rightarrow (\forall x{:}\text{Nat}.\Phi(x) \Rightarrow \Phi(\text{succ}\, x)) \Rightarrow (\forall x{:}\text{Nat}.\Phi(x)) \qquad (\sigma\text{Nat})$$

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 Φ on an arbitrary number to the next proof of Φ for the successor of that *same number*. Because this second step is a transformation, we first assume that the property Φ is true on a generic $x : \text{Nat}$ by placing Φ in the environment Γ of other assumptions, with the intention that the assumed Φ in Γ can be used to prove Φ with x replaced by succ x.

Structural coinduction for streams works in exactly the same way as structural induction for numbers—just with the roles of values and covalues reversed. The σStream rule summarises this deduction for proving a property Φ over any stream projection α using an infinite number of premises:

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

This deduction is justified by the reasoning that the listed projections—head β, tail(head β), tail(tail(head β))—cover *all the canonical covalues* of Stream A; testing Φ on all of them is sufficient to generalize Φ over any generic α of type Stream A. σStream summarizes this kind of argument in a finite form, avoiding the list of separate proofs for each of the infinitely possible projections. Whereas σNat corresponds to the usual induction axiom for the natural numbers, the σStream rule corresponds to the dual form of the coinduction axiom for proving a property holds for all observations of infinite streams:

$$\begin{aligned} &(\forall \beta \div A.\Phi(\text{head}\,\beta)) \\ \Rightarrow\ &(\forall \alpha \div \text{Stream}\, A.\Phi(\alpha) \Rightarrow \Phi(\text{tail}\,\alpha)) \qquad (\sigma\text{Stream}) \\ \Rightarrow\ &(\forall \alpha \div \text{Stream}\, A.\Phi(\alpha)) \end{aligned}$$

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

Before we illustrate applications of these dual notions of structural (co)induction, we first need to address two major weaknesses with our initial attempts at formalizing them using the rules in Fig. 5. Somehow, they are both simultaneously *too weak* and *too strong* to accomplish what we need.

3.2.1 (Co)Induction may be too weak *on its own*

Taken in isolation, the (co)inductive σ rules in Fig. 5 are actually much weaker for proving equalities then they may first appear. Of course, we will need to add a rule to actually use

the (co)inductive hypothesis introduced in the second premise of σNat and σStream:

$$\frac{}{\Gamma, \Phi \vdash \Phi}\ Ax$$

But this is still not enough to prove many simple equalities that we would expect.

For example, 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 all 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 projections:

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

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 β_{head} step:

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

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, $\mathrm{tail}\ \mathit{xs}$ corresponds to $\mu\gamma.\langle \mathit{xs} \| \mathrm{tail}\ \gamma \rangle$):

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

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

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

From the above equations, it should be clear that *parrot xs* is the same stream as *xs*. So we should be able to apply the rules from Fig. 5 to prove that for all streams $\mathit{xs} : \mathrm{Stream}\ A$, the equality $\mathit{parrot}\ \mathit{xs} = \mathit{xs} : \mathrm{Stream}\ A$ holds. The bottom of the derivation starts like this:

$$\dfrac{\dfrac{\mathit{xs} : \mathrm{Stream}\ A,\ \alpha \div \mathrm{Stream}\ A,\ \langle \mathit{parrot}\ \mathit{xs} \| \alpha \rangle = \langle \mathit{xs} \| \alpha \rangle \vdash \langle \mathit{parrot}\ \mathit{xs} \| \mathrm{tail}\ \alpha \rangle = \langle \mathit{xs} \| \mathrm{tail}\ \alpha \rangle \qquad \mathit{xs} : \mathrm{Stream}\ A,\ \beta \div A \vdash \langle \mathit{parrot}\ \mathit{xs} \| \mathrm{head}\ \beta \rangle = \langle \mathit{xs} \| \mathrm{head}\ \beta \rangle}{\mathit{xs} : \mathrm{Stream}\ A,\ \alpha \div \mathrm{Stream}\ A \vdash \langle \mathit{parrot}\ \mathit{xs} \| \alpha \rangle = \langle \mathit{xs} \| \alpha \rangle}\ \sigma\mathrm{Stream}}{\mathit{xs} : \mathrm{Stream}\ A \vdash \mathit{parrot}\ \mathit{xs} = \mathit{xs} : \mathrm{Stream}\ A}\ \sigma\mu$$

We begin by assuming some generic stream value $\mathit{xs} : \mathrm{Stream}\ A$ 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 $\alpha \div \mathrm{Stream}\ A$ expecting a stream. From here, we can apply the σStream coinductive rule, which leads to two proof obligations:

1. Show $\langle \mathit{parrot}\ \mathit{xs} \| \mathrm{head}\ \beta \rangle = \langle \mathit{xs} \| \mathrm{head}\ \beta \rangle$.

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

Step 1 is exactly one of the equations we proved above, which follows directly from β_{head}. Step 2, on the other hand, does not follow in the same way. The problem is that we cannot apply the coinductive hypothesis, because the internal state of the **corec** used to define *parrot* changes at each tail step. That is, we would like to put together the following equality:

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

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 are trying to apply coinduction first to prove an equality holds for all choices of *xs*.

The issue of generalizing the coinductive hypothesis is not something fundamentally new. The same thing happens with induction, where sometimes we need to generalize the inductive hypothesis to prove a fact. For example, *parrot* is dual to the following "no operation" continuation implemented as a loop, ultimately passing through any given number unchanged:

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

Clearly, $\langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle$ for any x. Yet, just adding the induction rules from Fig. 5 fail to prove this for the same reason we struggled with *parrot*. More specifically, we can prove the base case

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

we then need to show $\langle \text{succ}\ x \| noop\ \alpha \rangle = \langle \text{succ}\ x \| \alpha \rangle$ from the inductive hypothesis (IH) $\langle x \| noop\ \alpha \rangle = \langle x \| \alpha \rangle$. As before, we would like to put together the following equality:

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

The problem is that the induction hypothesis holds only for α, but we now need to apply it in a different context.

This problem does not show up when one does inductive proofs in λ-calculus since this quantification on 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 $\square$, but in the bigger context succ $\square$. The inductive hypothesis should be better expressed as: $\forall C[\square], C[plus\ x'\ \text{zero}] = C[x]$.

3.2.2 (Co)induction may be too strong for certain applications

Ultimately, the equational theory 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 3.2 (Consistency)**.** An equational theory for the (co)recursive abstract machine is *consistent* iff $\bullet \vdash \text{zero} = \text{succ}\, V : \text{Nat}$ and $\bullet \vdash \text{head}\, E = \text{tail}\, E' \div \text{Stream}\, A$ are *not* derivable.

As with most systems, equating 0 and 1 collapses the equational theory. Assuming $\text{zero} = \text{succ zero} : \text{Nat}$ 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 $\sigma\mu$:

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

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 $\text{head}\ \alpha = \text{tail}(\text{head}\ \alpha)$, 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:

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

Unfortunately, the rules given in Fig. 5 can lead to inconsistency in certain settings. Note how we used $\sigma{\to}$ to derive the $\eta_{\to}$ rule for functions above, which concludes that $\lambda x.\mu\alpha.\langle v \| x \cdot \alpha \rangle = v$ for *any* term v (so long as x and α are not captured). One consequence of this unrestricted $\eta_{\to}$ rule is the following equation:

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

This equality is fine under call-by-name evaluation but is inconsistent under call-by-value. 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

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

whereas on the right, we have

$$\langle \mu\delta.\langle \text{zero} \| \alpha \rangle \| \tilde{\mu} z.\langle \text{succ zero} \| \alpha \rangle \rangle \mapsto\!\!\!\!\to \langle \text{zero} \| \alpha \rangle$$

Therefore, $\sigma{\to}$ lets us derive $\text{zero} = \text{succ zero} : \text{Nat}$ using call-by-value evaluation.

The inconsistency of $\sigma{\to}$ in call-by-value should not be surprising. It is well known that unrestricted η equivalence is unsound in the call-by-value λ-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 λ-abstraction, but the η law requires $\Omega = \lambda x.(\Omega\ x)$. Instead, the sound version of the call-by-value η law only applies to values: $\lambda x.(V\ x) = V$.

To make $\sigma\!\rightarrow$ consistent with the call-by-value semantics, it can only apply to a restricted set of properties that avoids equating all non-value terms $M : A \to B$ with $\lambda x.(M\ x)$.

What may be more surprising is that both the inductive and coinductive principles σNat and σStream also suffer the same problem in dual ways. The coinductive σStream rule for infinite streams is similar in spirit to $\sigma\!\rightarrow$: 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. Specifically, σStream proves:

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

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

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

Dually, the inductive σNat rule matches over the possible shapes of a generic value $x : \mathrm{Nat}$. It can derive inconsistent equations under the dual evaluation strategy—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, the σNat rule proves 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), allows for the call-by-name value $\mu\_.\langle \mathrm{succ}\ \mathrm{zero} \| \alpha \rangle$ to be substituted for x in this equation, leading to the inconsistent equality $\alpha \div \mathrm{Nat} \vdash \langle \mathrm{succ}\ \mathrm{zero} \| \alpha \rangle = \langle \mathrm{zero} \| \alpha \rangle$.

4 Sound Theories of (Co)Induction

Since a naïve view of (co)inductive principles won't do—it simultaneously misses some obvious equalities while creating some inconsistent ones—we now consider a more nuanced version of (co)induction. First (in Section 4.1), we will make it possible to strengthen the (co)inductive hypothesis when necessary by enriching the language of properties that we can prove. Second (in Section 4.2), we will strategically weaken the rules of (co)induction to a restricted form that is *always* consistent, no matter the evaluation strategy or the effects that might be involved in computation. Third (in Section 4.3), we will consider the circumstances when the strong, unrestricted versions of induction and coinduction are sound, and can be consistently added to the equational theory to derive even more principles of equality for (co)inductive types.

4.1 Logical equational properties

Previously in Section 3.2.1, we found that even some simple equalities require that we strengthen the (co)inductive hypothesis. There is a material difference between saying:

1. for all *xs*, we can prove that an equality involving *xs* holds by (co)induction, versus
2. we can prove by (co)induction that, for all *xs*, the same equality holds.

$$Prop \ni \Phi ::= c = c' \mid v = v' : A \mid e = e' \div A \mid \Phi' \Rightarrow \Phi \mid \forall x{:}A.\Phi \mid \forall \alpha \div A.\Phi \mid \Phi_1 \wedge \Phi_2$$
$$Env \ni \Gamma ::= \bullet \mid \Gamma, x : A \mid \Gamma, \alpha \div A \mid \Gamma, \Phi$$

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

$$\frac{\Gamma, x : A \vdash \Phi}{\Gamma \vdash \forall x{:}A.\Phi}\ IntroL \qquad \frac{\Gamma \vdash \forall x{:}A.\Phi}{\Gamma, x : A \vdash \Phi}\ ElimL \qquad \frac{\Gamma, \alpha \div A \vdash \Phi}{\Gamma \vdash \forall \alpha \div A.\Phi}\ IntroR \qquad \frac{\Gamma \vdash \forall \alpha \div A.\Phi}{\Gamma, \alpha \div A \vdash \Phi}\ ElimR$$

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

$$\frac{}{\Gamma, \mathrm{succ}\, V = \mathrm{zero} : \mathrm{Nat}, \Gamma' \vdash \Phi}\ \mathrm{Nat}Coher$$

$$\frac{}{\Gamma, \mathrm{head}\, E = \mathrm{tail}\, E' \div \mathrm{Stream}\, A, \Gamma' \vdash \Phi}\ \mathrm{Stream}Coher$$

$$\frac{\Gamma, x : A, \Phi, \Gamma' \vdash \Phi' \quad x \notin FV(\Phi)}{\Gamma, \Phi, x : A, \Gamma' \vdash \Phi'}\ ExLA \qquad \frac{\Gamma, \Phi, x : A, \Gamma' \vdash \Phi' \quad x \notin FV(\Phi)}{\Gamma, x : A, \Phi, \Gamma' \vdash \Phi'}\ ExAL$$

$$\frac{\Gamma, \alpha \div A, \Phi, \Gamma' \vdash \Phi' \quad \alpha \notin FV(\Phi)}{\Gamma, \Phi, \alpha \div A, \Gamma' \vdash \Phi'}\ ExRA \qquad \frac{\Gamma, \Phi, \alpha \div A, \Gamma' \vdash \Phi' \quad \alpha \notin FV(\Phi)}{\Gamma, \alpha \div A, \Phi, \Gamma' \vdash \Phi'}\ ExAR$$

$$\frac{\Gamma, \Phi, \Phi', \Gamma' \vdash \Phi''}{\Gamma, \Phi', \Phi, \Gamma' \vdash \Phi''}\ ExAA$$

Fig. 6: Logical rules for equational properties.

The subtle difference is in the order between the "for all" introducing *xs* and the application of (co)induction. In 1, a generic *xs* is chosen first, becoming permanently fixed in the (co)inductive hypothesis introduced by the next step. In 2, (co)induction is tried first, which introduces a (co)inductive hypothesis that is still generic over all choices of *xs*. In our previous examples, we were following option 1.

Rather than accounting for this extra generality by complicating the (co)inductive rules themselves, we can instead just enrich the language of properties that we are proving. Importantly, we can internalize the implicit "for all" generalization made by bound $x : A$ and $\alpha \div A$ in the environment in terms of an explicit $\forall$ quantifier in the syntax of propositions. This generalization of *Prop*, among others, is shown in Fig. 6.

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 Φ over all choices of equal values of type A for x, and $\forall \alpha \div A.\Phi$ generalizes Φ over all choices of equal covalues for α of type A. The rules governing these two $\forall$ properties are given in Fig. 6 as well. Universal quantifiers can be introduced by *IntroL* and *IntroR*, which state that $\forall$ internalizes a bound (co)variable in the environment. An established quantifier can be eliminated by *ElimL* and *ElimR*, which according to the substitution rules given in 3.1 the quantified (co)-variables can be instantiated by any equal (co)values of the appropriate type. The 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 all equations in Φ and $[V'/x]$ on the right-hand side. For example, the

base cases of this substitution are when Φ is just an equality; for a command equality, this looks like:

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

Similar to the quantifiers, we also have plain propositional implication, written $\Phi' \Rightarrow \Phi$, for stating that the truth of Φ' implies the truth of Φ. 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.

To see the utility of propositional quantifiers consider the following two equations for "trivial" uses of recursion and corecursion (where a _ binding is an unused (co)variable):

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

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

Intuitively, these are *deep* extensionality axioms for the recursor and corecursor. Any generic observer α cannot tell the difference if a natural number is first broken down and rebuilt from scratch from the base case (zero) up. Likewise, any generic stream x gives the same response when its projections are broken down and rebuilt from scratch from the base case (head α) up. How do we actually prove δStream? Let us assume we can prove the following:

$$\alpha \div \text{Stream}\ A \vdash \forall x{:}\text{Stream}\ A.\langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x \| \alpha \rangle = \langle x \| \alpha \rangle \tag{4.1}$$

where we make use of the shorthand $cdeep := \{\text{head}\ \alpha \to \text{head}\ \alpha \mid \text{tail}\ \_ \to \beta.\ \text{tail}\ \beta\}$, we can then proceed as so:

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha \div \text{Stream}\ A \vdash \forall x{:}\text{Stream}\ A.\langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x \| \alpha \rangle = \langle x \| \alpha \rangle}{\alpha \div \text{Stream}\ A, x : \text{Stream}\ A \vdash \langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x \| \alpha \rangle = \langle x \| \alpha \rangle}\ ElimL}{x : \text{Stream}\ A, \alpha \div \text{Stream}\ A \vdash \langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x \| \alpha \rangle = \langle x \| \alpha \rangle}\ ExRL}{x : \text{Stream}\ A \vdash \mathbf{corec}\ cdeep\ \mathbf{with}\ x = x : \text{Stream}\ A}\ \sigma\mu}{\vdash \forall x{:}\text{Stream}\ A.\ \mathbf{corec}\ cdeep\ \mathbf{with}\ x = x : \text{Stream}\ A}\ IntroL$$

Now, let's go back to the proof of 4.1; we apply the coinduction principle *first* to the $\forall$-generalized property before introducing the $x : \text{Stream}\ A$ it binds. This introduces two premises for the base case (when α is instantiated with a head projection) and the co-inductive case (when α is instantiated with a tail projection). These two premises can be finished through the following calculations:

- $[\text{head}\,\beta/\alpha]$ we have the calculation:

$$\langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x \| \text{head}\,\beta\rangle \mapsto \langle x \| \text{head}\,\beta\rangle \qquad (\beta_{\text{head}})$$

- $[\text{tail}\,\beta/\alpha]$ assume $\forall x{:}\text{Stream}\,A.\langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x\|\beta\rangle = \langle x\|\beta\rangle$ as the coinductive hypothesis (CIH), so we have the following calculation (in call-by-value and -name):

$$\begin{aligned}
&\langle \mathbf{corec}\ cdeep\ \mathbf{with}\ x \| \text{tail}\,\beta\rangle \\
&\mapsto \langle \mu\beta.\langle x\|\text{tail}\,\beta\rangle \| \tilde{\mu} y.\langle \mathbf{corec}\ cdeep\ \mathbf{with}\ y\|\beta\rangle\rangle && (\beta_{\text{tail}}) \\
&= \langle \mu\beta.\langle x\|\text{tail}\,\beta\rangle \| \tilde{\mu} y.\langle y\|\beta\rangle\rangle && (CIH[y/x]) \\
&= \langle \mu\beta.\langle x\|\text{tail}\,\beta\rangle \| \beta\rangle && (\eta_{\tilde{\mu}}) \\
&\mapsto \langle x\|\text{tail}\,\beta\rangle && (\mu)
\end{aligned}$$

  Note that the generalization over x in the coinductive hypothesis is essential for instantiating x (via *SubstL*) with the bound y newly introduced by β_{tail} reduction.

Analogous to the "deep" extensionality properties δNat and δStream of (co)recursion, we can also derive the following "shallow" extensionality properties ηNat and ηStream that just look at the outermost structure of a numeric value or a stream projection:

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

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

4.2 Universally sound rules of (co)induction

In Section 3.2.2, we found that some of the straightforward rules for implementing observational equivalence in Fig. 5 were inconsistent in certain settings. Specifically, the coinductive rules $\sigma{\to}$ and σStream which match on the structure of a covalue could be used to equate 0 and 1 in a call-by-value language by using a term that throws away its continuation and the ability for the consumer $\tilde{\mu}x.\langle 1\|\alpha\rangle$ to disregard its input analogous to the context $\mathbf{let}\,x=\Box\,\mathbf{in}\,1$ in functional languages. Dually, the inductive rule σNat, which matches on the structure of a value, causes the same problem in a call-by-name language.[5]

This problem is analogous to a counter-example in the call-by-value λ-calculus, where, as discussed previously, it is inconsistent to use the η law of functions to equate an infinite loop with a λ-abstraction. Likewise, our approach to ensure consistency is analogous to the resolution between function extensionality and call-by-value evaluation in the λ-calculus. Namely, the safe version of the η law in call-by-value is $\lambda x.(V\ x) = V$, which only applies to a value V, so that both sides of the equation are values. By generalizing this law to commands, we get the equation $\langle \lambda y.(V\ y)\|\alpha\rangle = \langle V\|\alpha\rangle$, which follows by β reduction if we expand the abstract observer $\alpha : A \to B$ to the only possible covalue which is a call stack of

[5] Note that, in contrast, the generic rules $\sigma\mu$ and $\sigma\tilde{\mu}$ cannot be used to derive such a counterexample to consistency. This is analogous to the fact that the extensionality rules for generic abstractions—$\mu\alpha.\langle v\|\alpha\rangle$ and $\tilde{\mu}x.\langle x\|e\rangle$—are both sound without additional restrictions for call-by-name and call-by-value (Downen & Ariola, 2014). This is because μ- and $\tilde{\mu}$-abstractions are not necessarily (co)values, unlike things like λ-abstractions that are always values in both the call-by-name and call-by-value semantics.

Properties strict on x ($\Psi(x)$), and properties productive on α ($\Psi(\alpha)$):

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

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

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

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

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

Plus all the intensional equality rules from Fig. 4 and the logical rules from Fig. 6.

Fig. 7: Weak extensional equational theory.

the form $x \cdot \beta$:

$$\langle \lambda y.(V\ y) \| x \cdot \beta \rangle := \langle \lambda y.\mu\gamma.\langle V \| y \cdot \gamma \rangle \| x \cdot \beta \rangle \mapsto_{\beta_\to} \langle \mu\gamma.\langle V \| x \cdot \gamma \rangle \| \beta \rangle \mapsto_{\beta_\mu} \langle V \| x \cdot \beta \rangle$$

Our goal is to make sure that we can apply the coinduction principle to the equation $\langle \lambda y.(V\ y) \| \alpha \rangle = \langle V \| \alpha \rangle$ to reveal the above reduction *only* when V is a value. To thread the needle, we use the restricted rules in Fig. 7, which introduce weak (co)induction principles $\omega{\to}$, $\omega\mathrm{Nat}$, and $\omega\mathrm{Stream}$, together with all the intensional equality rules from Fig. 4 and the logical rules from Fig. 6. Note that the overall structure of these rules is the same as the "strong" versions $\sigma{\to}$, $\sigma\mathrm{Nat}$, and $\sigma\mathrm{Stream}$ from Fig. 5. The only difference is that the weakened (co)induction rules in Fig. 7 only apply to a subset of properties, Ψ, rather than any arbitrary property Φ, reminiscent of (Pédrot & Tabareau, 2017). 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 $\langle x \| E \rangle = \langle x \| E' \rangle$ 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* α when it immediately returns a value to α on both side of its underlying equality, with the base case of a productive property on α being $\langle V \| \alpha \rangle = \langle V' \| \alpha \rangle$ where α is not free in V or V'. Intuitively, $\Psi(\alpha)$ is some property which produces exactly one value to α.

By restricting induction to only apply to strict properties, and restricting coinduction to only productive properties, we get a single extensional equational theory (parameterized by the definition of values and covalues) that is consistent in *both* call-by-value and call-by-name evaluation. See Section 6 for the proof of consistency.

Theorem 4.1. *The weak extensional equational theory in Fig. 7 is consistent for both the call-by-name and call-by-value semantics.*

To see how the counterexamples against consistency are ruled out, consider again the derivation of the η axiom for functions in the extensional equational theory. By replacing the strong extensionality rule $\sigma\!\rightarrow$ with the weaker one $\omega\!\rightarrow$, we can only prove the restricted version of η equality which equates a value to a λ-abstraction:

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

Second from the bottom, $\omega\!\rightarrow$ 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 γ; both sides of the equation immediately produce a syntactic value to γ. This rule does not apply if we generalize V to be an arbitrary term v, because then $\langle v \| \gamma\rangle$ may not produce a single value to γ—v might throw γ away or capture and invoke γ multiple times.

Similarly, note that the deep extensionality axioms δ_{Nat} and δ_{Stream} are still provable using the weak (co)induction rules. Crucially, the properties that embody these axioms are strict and productive, respectively, on the necessary (co)variable. Recall in the proof we gave above for δ_{Stream}—and the implied dual proof of δ_{Nat}—the (co)inductive rule is applied to these two universally-quantified equations:

$$\alpha \div \text{Stream}\, A \vdash \forall x{:}\text{Stream}\, A.\ \langle \mathbf{corec}\{\text{head}\ \alpha \to \text{head}\ \alpha \mid \text{tail}\ \_ \to \beta.\ \text{tail}\ \beta\}\ \mathbf{with}\ x \| \alpha\rangle = \langle x \| \alpha\rangle$$
$$x : \text{Nat} \vdash \forall \alpha \div \text{Nat}. \qquad \langle x \| \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\ \_ \to y.\ \text{succ}\ y\}\ \mathbf{with}\ \alpha\rangle = \langle x \| \alpha\rangle$$

The first property is productive on α because both sides of the root equation pass a value (**corec** . . . **with** x on the left, and x on the right) to α according to both call-by-value and call-by-name semantics. Since quantifying over a productive property gives another productive property, ωStream applies to $\alpha \div \text{Stream}\, A$ in the first line. Likewise, the second property is strict on x because both sides of the root equation observe x with a covalue (**rec** . . . **with** α on the left and α on the right) in both call-by-name and -value. Thus, ωNat applies to $x : \text{Nat}$ in the second line. Because both of these crucial steps pass the additional requirements of the weakened (co)induction rules, the derivation of δ_{Stream} and δ_{Nat} according to Fig. 7 proceeds exactly as in Section 4.1, replacing σStream with ωStream and σNat with ωNat.

4.3 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 equational

Strong call-by-name equational theory: extends the weak extensional theory for call-by-name reduction from Fig. 7 with these rules:

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

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

Strong call-by-value equational theory: extends the weak extensional theory for call-by-value reduction from Fig. 7 with this rule:

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

Fig. 8: Two strong extensional equational theories — with unrestricted coinduction rules $\sigma\to$ and σStream that are sound in call-by-name, and an unrestricted induction rule σNat that is sound in call-by-value.

theory given from Fig. 7. Call-by-value has a more permissive notion of covalue (any coterm is a call-by-value covalue), so the induction principle ωNat applies to more properties in the call-by-value equational theory 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 ωStream applies to more properties in call-by-name than in call-by-value.

For non-recursive types like $A \to B$, this is enough to recover the full power of $\sigma\to$ 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 ωNat to σNat or ωStream to σStream.

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 σNat and σStream can be consistently added to the equational theory, even in the presence of computational effects like first-class control, under the correct evaluation strategy. (see Section 6).

Definition 4.2 (Strong Equational Theories)**.** The two strong extensional equational theories, which generalize the common weak extensional equational theory (Fig. 7) with additional sound rules specifically for call-by-name and call-by-value reduction, respectively, are shown in Fig. 8.

- The *strong call-by-name equational theory* extends the call-by-name instance of Fig. 7 with the σStream and $\sigma\to$ rules of coinduction from Fig. 5.
- The *strong call-by-value equational theory* extends the call-by-value instance of Fig. 7 with the σNat rule of induction from Fig. 5.

Theorem 4.3. *The strong call-by-name and call-by-value equational theories are consistent.*

5 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 σNat with a property $\Phi' \Rightarrow \Phi$ that cannot be derived from the ωNat—even in call-by-value. Likewise, there are certain forms of coinductive reasoning (for example, bisimulation) that are possible with σStream but cannot be derived from ωStream—even in call-by-name.

Next, we will explore the strength of full σNat and σStream versus the weaker ωNat and ωStream, 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 $\Psi_1(\mathrm{succ}\, x)$ and the other showing $\Psi_2(\mathrm{succ}\, x)$. 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 ωNat on the conjunction $\Psi_1(x) \wedge \Psi_2(x)$ *first*, before splitting the two apart like so:

$$\dfrac{\dfrac{\Gamma \vdash \Psi_1(\mathrm{zero}/x) \quad \Gamma \vdash \Psi_2(\mathrm{zero})/x}{\Gamma \vdash \Psi_1(\mathrm{zero}/x) \wedge \Psi_2(\mathrm{zero}/x)}\, \mathit{ConjI} \quad \dfrac{\begin{array}{c}\Gamma, x : \mathrm{Nat}, \Psi_1(x) \wedge \Psi_2(x) \vdash \Psi_1(\mathrm{succ}\, x/x) \\ \Gamma, x : \mathrm{Nat}, \Psi_1(x) \wedge \Psi_2(x) \vdash \Psi_2(\mathrm{succ}\, x/x)\end{array}}{\Gamma, x : \mathrm{Nat}, \Psi_1(x) \wedge \Psi_2(x) \vdash \Psi_1(\mathrm{succ}\, x/x) \wedge \Psi_2(\mathrm{succ}\, x/x)}\, \mathit{ConjI}}{\Gamma, x : \mathrm{Nat} \vdash \Psi_1(x) \wedge \Psi_2(x)}\, \omega\mathrm{Nat}$$

This application of the weak ωNat 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 ωStream and *ConjI* rules. This mutual weak coinduction rule looks like:

$$\dfrac{\begin{array}{c}\Gamma, \beta \div A \vdash \Psi_1(\mathrm{head}\, \beta/\alpha) \\ \Gamma, \beta \div A \vdash \Psi_2(\mathrm{head}\, \beta/\alpha) \\ \Gamma, \alpha \div \mathrm{Stream}\, A, \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash \Psi_1(\mathrm{tail}\, \alpha/\alpha) \\ \Gamma, \alpha \div \mathrm{Stream}\, A, \Psi_1(\alpha) \wedge \Psi_2(\alpha) \vdash \Psi_1(\mathrm{tail}\, \alpha/\alpha)\end{array}}{\Gamma, \alpha \div \mathrm{Stream}\, A \vdash \Psi_1(\alpha) \wedge \Psi_2(\alpha)}\, \omega\mathrm{Stream}, \mathit{ConjI}$$

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:[6]

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

Where the two coinductive hypotheses are:

$$\Psi_1(\alpha) = \forall s_1 \forall s_2.\langle evens\ (merge\ s_1\ s_2) \| \alpha\rangle = \langle s_1 \| \alpha\rangle$$
$$\Psi_2(\alpha) = \forall s_1 \forall s_2.\langle odds\ (merge\ s_1\ s_2) \| \alpha\rangle = \langle s_2 \| \alpha\rangle$$

Both of these two propositions are productive on α 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 σStream or σNat 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 ωNat, the traditional notion of *strong induction* on the natural numbers really requires the full σNat. How can we formalize the derivation of strong induction? First, define the ordering relation on numbers in terms of the following equality:

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

We then write $\forall y{\leq}x{:}\text{Nat}.\Phi$ as shorthand for the property $\forall y{:}\text{Nat}.\ y \leq x : \text{Nat} \Rightarrow \Phi$. Applying σNat to this gives:

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

Since $\forall y{\leq}\text{zero}{:}\text{Nat}.\ y = \text{zero} : \text{Nat}$ is derivable (by definition of $\leq$) as well as $\forall x{:}\text{Nat}.\ x \leq x : \text{Nat}$ (by induction with σNat), we can specialize the above application to derive the following simpler statement of strong induction on the naturals:

$$\frac{\Gamma \vdash \Phi[\text{zero}/x] \quad \Gamma, x{:}\,\text{Nat}, \forall y{\leq}x{:}\text{Nat}.\Phi \vdash \Phi[\text{succ}\,x/x]}{\Gamma, x{:}\,\text{Nat} \vdash \Phi}$$

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

Notice that we can *never* use ωNat for this derivation, even if Φ happens to be strict on x. Why not? Because the property to which we apply induction,

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

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:

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

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

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

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

$$\frac{\Gamma, \delta \div A \vdash \forall\beta{\leq}\text{head}\ \delta \div \text{Stream}\, A.\Phi \qquad \Gamma, \alpha \div \text{Stream}\, A, \forall\beta{\leq}\alpha \div \text{Stream}\, A.\Phi \vdash \forall\beta{\leq}\text{tail}\ \alpha \div \text{Stream}\, A.\Phi}{\Gamma, \alpha \div \text{Stream}\, A \vdash \forall\beta{\leq}\alpha \div \text{Stream}\, A.\Phi}$$

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

$$\frac{\Gamma, \delta \div A \vdash \Phi[\text{head}\ \delta/\alpha] \qquad \Gamma, \alpha \div \text{Stream}\, A, \forall\beta{\leq}\alpha \div \text{Stream}\, A.\Phi[\beta/\alpha] \vdash \Phi[\text{tail}\ \alpha/\alpha]}{\Gamma, \alpha \div \text{Stream}\, A \vdash \Phi}$$

From this, we can derive the following special case of strong coinduction, where we must show the first $n+1$ base cases (for head β, tail(head β), ... $\text{tail}^n(\text{head}\ \beta)$) directly, and then take the $n+1^{th}$ tail projection in the coinductive case:

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

The above principle can prove that

$$\alpha \div \text{Stream}\, A \vdash \forall s{:}\text{Stream}\, A.\langle \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s) \| \alpha\rangle = \langle s \| \alpha\rangle \tag{5.1}$$

by stepping by 2. We prove the property for the base cases (head β and tail(head β)) 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 β. This principle captures the proof of Example Theorem 2.6, with the difference that it avoids the case analysis. From (5.1), we can then

prove

$$\forall s{:}\mathrm{Stream}\, A.\ \mathit{merge}\ (\mathit{evens}\ s)\ (\mathit{odds}\ s) = s : \mathrm{Stream}\, A \tag{5.2}$$

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 relationship of our choosing. The traditional principle of bisimulation on streams can be represented by the following inference rule, where the property Φ (with free variables s_1 and s_2) stands for an arbitrary relationship between two streams s_1 and s_2:

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

The two assumptions confirm that Φ is a valid bisimulation relation: Φ 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 σStream. We are going to prove

$$\Gamma, \alpha : \mathrm{Stream}\, A \vdash \forall s_1, s_2{:}\ \mathrm{Stream}\, A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle \tag{5.3}$$

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

$$\dfrac{\dfrac{\Gamma, \alpha : \mathrm{Stream}\, A \vdash \forall s_1, s_2{:}\ \mathrm{Stream}\, A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle}{\Gamma, s_1{:}\ \mathrm{Stream}\, A, s_2{:}\ \mathrm{Stream}\, A, \Phi, \alpha \div \mathrm{Stream}\, A \vdash \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle}\mathit{ElimL}, \mathit{Ax}, \mathit{Lemm}}{\Gamma, s_1{:}\ \mathrm{Stream}\, A, s_2{:}\ \mathrm{Stream}\, A, \Phi \vdash s_1 = s_2 : \mathrm{Stream}\, A}\ \sigma\mu$$

We are proving property 5.3 by strong coinduction (σStream):

- For the head case, we must show that

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

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

  $$\dfrac{\dfrac{\Gamma, s_1{:}\ \mathrm{Stream}\, A, s_2{:}\ \mathrm{Stream}\, A, \Phi \vdash \mathrm{head}\, s_1 = \mathrm{head}\, s_2 : A}{\Gamma, \beta \div A, s_1{:}\ \mathrm{Stream}\, A, s_2{:}\ \mathrm{Stream}\, A, \Phi \vdash \langle \mathrm{head}\, s_1 \| \beta \rangle = \langle \mathrm{head}\, s_2 \| \beta \rangle}\mathit{Compat}}{\Gamma, \beta \div A \vdash \forall s_1, s_2{:}\ \mathrm{Stream}\, A.\Phi \Rightarrow \langle \mathrm{head}\, s_1 \| \beta \rangle = \langle \mathrm{head}\, s_2 \| \beta \rangle}\mathit{IntroH}, \mathit{IntroL}$$

- 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

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

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

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

we can derive the goal of the coinductive step by weakening (written *Weak*) the given bisimulation premise $\Gamma, \Gamma_{\mathit{Sim}} \vdash \Phi[\text{tail}\, s_1/s_1, \text{tail}\, s_2/s_2]$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \Gamma_{\mathit{CIH}} \vdash \forall s_1, s_2 \colon \text{Stream}\, A.\Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle}\,\mathit{Ax}}{\Gamma, \Gamma_{\mathit{CIH}}, s_1 \colon \text{Stream}\, A, s_2 \colon \text{Stream}\, A \vdash \Phi \Rightarrow \langle s_1 \| \alpha \rangle = \langle s_2 \| \alpha \rangle}\,\mathit{ElimL}}{\Gamma' \vdash \Phi[\text{tail}\, s_1/s_1, \text{tail}\, s_2/s_2] \Rightarrow \langle \text{tail}\, s_1 \| \alpha \rangle = \langle \text{tail}\, s_2 \| \alpha \rangle}\,\mathit{SubstL} \quad \dfrac{\Gamma, \Gamma_{\mathit{Sim}} \vdash \Phi[\text{tail}\, s_1/s_1, \text{tail}\, s_2/s_2]}{\Gamma' \vdash \Phi[\text{tail}\, s_1/s_1, \text{tail}\, s_2/s_2]}\,\mathit{Weak}}{\Gamma, \Gamma_{\mathit{CIH}}, \Gamma_{\mathit{Sim}} \vdash \langle \text{tail}\, s_1 \| \alpha \rangle = \langle \text{tail}\, s_2 \| \alpha \rangle}\,\mathit{Lemm}}{\Gamma, \Gamma_{\mathit{CIH}} \vdash \forall s_1, s_2 \colon \text{Stream}\, A.\Phi \Rightarrow \langle \text{tail}\, s_1 \| \alpha \rangle = \langle \text{tail}\, s_2 \| \alpha \rangle}\,\mathit{Intro}$$

6 Consistency of the Equational Theory

We've seen the (strong) equational theories used to encode and prove a variety of different reasoning principles and program equalities. But how do we know that the syntactic rules in Figs. 7 and 8 imply real equivalences between the results of programs? Applying β 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 σNat and σStream, can be inconsistent in certain contexts.

In order to prove that the syntactic equational theories are consistent, we will show that they are all approximations of a more general notion of *observational equivalence*, defined directly in terms of the behavior of running programs.

Definition 6.1 (Observable)**.** The set of *observable typing environments* (Θ) and *observable commands* (d) is

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

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

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

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 6.2 (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 $C[e_1] \approx C[e_2]$.

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 call 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 3.2. $\bullet \vdash \mathrm{zero} \approx \mathrm{succ}\, V : \mathrm{Nat}$ 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 $\langle \mathrm{zero} \| \alpha \rangle \not\sim \langle \mathrm{succ}\, V \| \alpha \rangle$. Similarly, $\bullet \vdash \mathrm{head}\, E \approx \mathrm{tail}\, E' \div \mathrm{Stream}\, A$ does not hold, due to the counterexample context $\langle x \| \Box \rangle$ in the observable environment $x : \mathrm{Stream}\, A$.

As such, any other relation included within it (such as the syntactic equational theories in Fig. 7 and Definition 4.2) must also be consistent.

Our primary goal, then, is to prove that these syntactic equational theories 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 *(bi)orthogonality* and *symmetric candidates* techniques for strong normalization of classical calculi (Downen *et al.*, 2020, 2019)—from unary predicates to binary relations.

6.1 Orthogonal relations and equality candidates

The safety model of (Downen & Ariola, 2023) is built around the idea of *orthogonality* between producers and consumers. At the individual level, orthogonality classifies when a term (v) and a coterm (e) can safely interact with one another, written $v \perp\!\!\!\perp e$. This orthogonal notion of safe interactions, $\perp\!\!\!\perp$, can be viewed as either a term-coterm relation, or equivalently as a set of safe commands, such that $v \perp\!\!\!\perp e$ and $\langle v \| e \rangle \in \perp\!\!\!\perp$ mean the same thing. At the group level, orthogonality checks when every combination among some terms ($\mathbb{A}^+$) and some coterms ($\mathbb{A}^-$) are safe; *i.e.,* $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$ between sets means $v \perp\!\!\!\perp e$ for every $v \in \mathbb{A}^+$ and $e \in \mathbb{A}^-$. Notice the analogy to types, which classify *both* terms and coterms such that any combination forms a safe command. To that end, type safety means that any well-typed command $\Gamma \vdash c$ and any substitution σ described by Γ yield a safe command $c[\sigma] \in \perp\!\!\!\perp$.

But equational theories say something more than just instances when a single command is safe to run; they say when 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. Similarly, we will not ask when a single producer and consumer can safely interact, but rather when a pair of producers (v and v') and a pair of consumers (e and e') form equivalent interactions, of the form $\langle v \| e \rangle \perp\!\!\!\perp \langle v' \| e' \rangle$. As before, we can generalize orthogonality from the individual level to the group level by checking when a candidate notion of term equivalence ($\mathbb{A}^+$) and coterm equivalence ($\mathbb{A}^-$) always combine to form $\perp\!\!\!\perp$-equivalent commands; *i.e.,* $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$ between binary relations means $\langle v \| e \rangle \perp\!\!\!\perp \langle v' \| e' \rangle$ for every $v \; \mathbb{A}^+ v'$ and $e \; \mathbb{A}^- e'$.[7] Again, notice the analogy to the use of types in an equational theory: a type classifies both an equality relation between terms and an equality relation between coterms, such that any two commands made of equal terms and equal coterms of the same type must themselves be equal (as described by the *Cut* rule for equality).

With this overall structure in mind, we can now give our main definition of equivalence $c \perp\!\!\!\perp c'$ serving as the basis of orthogonality, and the semantics for equality 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*) denotations of types, so each type of our language can be interpreted as a particular candidate of equality.

Definition 6.3 (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 6.1:

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

Orthogonality between a pair of terms (v, v') and a pair of co-terms (e, e') is defined as:

$$(v, v') \perp\!\!\!\perp (e, e') := \langle v \| e \rangle \perp\!\!\!\perp \langle v' \| e' \rangle$$

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

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

Note how orthogonality between individual pairs $(v, v') \perp\!\!\!\perp (e, e')$ exactly coincides with the special case for orthogonality between singleton relations $\{(v, v')\} \perp\!\!\!\perp \{(e, e')\}$.

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

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

Definition 6.4 (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.,*

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

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

[7] 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 $Y \times Y$

- *Soundness:* the two sides of the $\mathbb{A}$ are orthogonal to one another, $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$. In other words, 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 $\mathbb{A}$ (the term relation $\pi_1(\mathbb{A})$) and $\mathbb{A}^-$ to denote the second one (the coterm relation $\pi_2(\mathbb{A})$). As a shorthand, given an equality candidate $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$, we write $v\ \mathbb{A}\ v'$ to mean $v\ \mathbb{A}^+\ v'$ and likewise $e\ \mathbb{A}\ e'$ to mean $e\ \mathbb{A}^-\ e'$. 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.

Intuitively, the soundness and completeness criteria of equality candidates correspond to our original principles of observational equality from Section 3.2 in an empty environment:

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

Soundness is equivalent to the "only if" direction of both points, which ensures that the *Cut* rule of the equational theory gives a true observational equivalence between commands. In contrast, we formalized the "if" direction using the $\sigma\tilde{\mu}$ and $\sigma\mu$ rules. In the special case where the outer environment is empty, these rules are:

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

The free (co)variable of type A in the premise stands for not just a single unknown (co)value of type A, but any *pair* of equivalent (co)values of type A where one is used in the left-hand side of the equation and the other on the right-hand side. In other words, these two inference rules correspond to the two following statements about the interpretation of type A as an equality candidate $[\![A]\!]$ (to be fully defined later in Section 6.3):

($\sigma\tilde{\mu}$) Suppose, for all related values V $[\![A]\!]$ V' of $[\![A]\!]$, we know that

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

Then e $[\![A]\!]$ e' are related coterms of $[\![A]\!]$.

($\sigma\mu$) Suppose, for all related covalues E $[\![A]\!]$ E' of $[\![A]\!]$, we know that

$$\langle v \| \alpha \rangle [E/\alpha] = \langle v \| E \rangle \perp\!\!\!\perp \langle v' \| E' \rangle = \langle v' \| \alpha \rangle [E'/\alpha]$$

Then v $[\![A]\!]$ v' are related terms of $[\![A]\!]$.

Note that these two interpretations of the $\sigma\tilde{\mu}$ and $\sigma\mu$ inference rules are logically equivalent to the positive and negative completeness criteria of Definition 6.4. In particular, the restriction that we only need to check a given (co)term against just the (co)values in a candidate corresponds to the fact that we only ever substitute a (co)value for a (co)variable. Thus, to ensure that $\sigma\tilde{\mu}$ and $\sigma\mu$ are valid, we need to assume that this limited test against only (co)values constitutes sufficient evidence to conclude that a pair of (co)terms are indeed related by a candidate.

We describe next another way to describe equality candidates.

6.2 Dual lattices and completion

By describing two polar-opposite sides—terms and coterms—there are several ways we can view candidates. These multiple perspectives appear when we are just interested in plain sets of typed (co)terms, as in (Downen & Ariola, 2023), 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 langauges.

Definition 6.5 (Refinement and Subtyping)**.** There are two ways of ordering pre-candidates: *refinement* (denoted by $\mathbb{A} \sqsubseteq \mathbb{B}$) and *subtyping* (denoted by $\mathbb{A} \leq \mathbb{B}$), defined as:

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

where $\subseteq$ denotes implication of binary relations: $\mathbb{R}_1 \subseteq \mathbb{R}_2$ when $x\ \mathbb{R}_1\ y$ implies $x\ \mathbb{R}_2\ y$. Symmetrically, we say $\mathbb{A}$ *extends* $\mathbb{B}$ (written $\mathbb{A} \sqsupseteq \mathbb{B}$) when $\mathbb{B} \sqsubseteq \mathbb{A}$, and $\mathbb{A}$ is a *supertype* of $\mathbb{B}$ (written $\mathbb{A} \geq \mathbb{B}$) when $\mathbb{B} \leq \mathbb{A}$.

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:

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

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

These two orderings and lattices lift up the implicit structure of plain term or coterm relations, and carry them over to the richer structure of pre-candidates themselves. For example, in the same way that we generalized orthogonality from individuals ($(v, v') \perp\!\!\!\perp (e, e')$) to entire sets or relations on individuals ($\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$), we can generalize orthogonality yet again to full candidates combining a term relation with a coterm relation. In this sense, two pre-candidates, $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ and $\mathbb{B} = (\mathbb{B}^+, \mathbb{B}^-)$, are orthogonal when their underlying positive and negative halves are orthogonal to one another like so:

$$(\mathbb{A}^+, \mathbb{A}^-) \perp\!\!\!\perp (\mathbb{B}^+, \mathbb{B}^-) := \mathbb{A}^+ \perp\!\!\!\perp \mathbb{B}^- \text{ and } \mathbb{B}^+ \perp\!\!\!\perp \mathbb{A}^-$$

Unlike orthogonality between a pair of terms and coterms or between a term relation and a coterm relation, wherein two different types of things are compared, orthogonality between two pre-candidates $\mathbb{A}$ and $\mathbb{B}$ is commutative, $\mathbb{A} \perp\!\!\!\perp \mathbb{B}$ if and only if $\mathbb{B} \perp\!\!\!\perp \mathbb{A}$. Furthermore, given any candidate $\mathbb{A}$, we can find it's canonical orthogonal counterpart $\mathbb{A}^{\perp\!\!\!\perp}$:

$$(\mathbb{A}^+, \mathbb{A}^-)^{\perp\!\!\!\perp} := (\mathbb{A}^{-\perp\!\!\!\perp}, \mathbb{A}^{+\perp\!\!\!\perp})$$

which is the *largest* pre-candidate *w.r.t refinement* that is orthogonal to $\mathbb{A}$. In other words, $\mathbb{A} \perp\!\!\!\perp \mathbb{A}^{\perp\!\!\!\perp}$ is always guaranteed, and given any other orthogonal pre-candidate $\mathbb{A} \perp\!\!\!\perp \mathbb{B}$, then $\mathbb{B} \sqsubseteq \mathbb{A}^{\perp\!\!\!\perp}$. Note, however, $\mathbb{A}^{\perp\!\!\!\perp}$ may have no meaningful *subtyping* relationship to another orthogonal $\mathbb{A} \perp\!\!\!\perp \mathbb{B}$.

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 6.6 (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 two lattices preserve (or fail to preserve) soundness and completeness conditions are also completely different from one another.

Property 6.7 (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.*

Interestingly, the subtyping lattice always preserves soundness, but not completeness. In contrast, one direction of refinement preserves only soundness, and the other direction preserves only completeness. At the end of the day, we are only interested in equality candidates, which are both sound and complete. In order to build the interpretation of (co)-inductive types, we will 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 lattice of sound candidates, we will begin there, with the subgoal

of filling in the missing parts of a sound but incomplete 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. Note that 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, and the negative viewpoint uses the *covalues* related by $\mathbb{A}$ as the defining axioms.

Definition 6.8 (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:

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

The positive and negative viewpoints give complementary results. 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 6.9 (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) extended from sets to binary relations, which uses the facts that $\mathrm{Pos}(\mathbb{A})$ and $\mathrm{Neg}(\mathbb{A})$ are fixed points of $\_^{v\perp\!\!\!\perp}$ and, furthermore, that the set of these fixed points is exactly the set of all equality candidates (Downen *et al.*, 2020). □

Definition 6.10 (Equality Candidate Lattice)**.** Equality candidates form a complete lattice *w.r.t* subtyping (Downen *et al.*, 2019) whose unions ($\curlyvee$) and intersections ($\curlywedge$) are defined as:

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

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

From this perspective, we can re-describe the positive and negative completions in terms of the subtyping lattice of equality candidates. As per Lemma 6.9, Pos and Neg are the intersection and union (respectively) of all extensions of a restricted sound candidate $\mathbb{A}^{\nu}$:

$$\mathrm{Pos}(\mathbb{A}) = \bigwedge\{\mathbb{C} \in \mathscr{EC} \mid \mathbb{C} \sqsupseteq \mathbb{A}^{\nu}\} \qquad \mathrm{Neg}(\mathbb{A}) = \bigvee\{\mathbb{C} \in \mathscr{EC} \mid \mathbb{C} \sqsupseteq \mathbb{A}^{\nu}\}$$

As a corollary of Lemma 6.9 and the definition of Pos and Neg, we get the following facts that let us reason about positively and negatively constructed equality candidates.

Property 6.11 (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 6.12 (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 6.13. *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* $E \curlyvee_i \mathbb{A}_i\ E'$.

6.3 Interpretation of types and properties

We now have enough infrastructure to define the model of observational equivalence—as shown in Fig. 9—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 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 when given 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

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

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

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

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

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

$$[\![\Gamma \vdash \Phi]\!] := [\![\Gamma]\!] \subseteq [\![\Phi]\!]$$

Fig. 9: Model of observational equivalence in the abstract machine.

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 Γ 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 Γ (written $\rho\ [\![\Gamma]\!]\ \rho'$) relates two such substitutions ρ and ρ' that abide by all three of the following criteria:

- For each variable x of type A in the environment Γ, both ρ and ρ' 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 Γ, both ρ and ρ' must substitute some covalue for α (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 Φ assumed in the environment Γ, the interpretation of Φ must be true when given both ρ and ρ'. Or in other words, Φ must relate ρ and ρ'.

In Figs. 2 and 4, we limit ourselves to environments that only assign types to a distinct collection of (co)variables (*i.e.,* no x or α bound by Γ is assigned a type more than once). However, notice how the interpretation of environments Γ in Fig. 9 will still properly respect the scope of (co)variables which "shadow" older ones of the name previously to the left of Γ. For instance, this still ensures the validity of extension—if $\rho\ [\![\Gamma]\!]\ \rho'$ and $V\ [\![A]\!]\ V'$ then $\rho[V/x]\ [\![\Gamma, x : A]\!]\ \rho'[V'/x]$, and similar for covariables—even if the "new" (co)variable already appears in Γ. As such, if there are somehow several type assignments for the same (co)variable, only the right-most one is relevant. Similarly, each assumed property in the environment only depends on the previous choices made for substitutions to the left of it. For example, we have $[\lambda y.y/x]\ [\![x : \mathrm{Nat}, x = \mathrm{zero} : \mathrm{Nat}, x : \mathrm{Nat} \to \mathrm{Nat}]\!]\ [\lambda y.y/x]$ where we substitute a function of type $\mathrm{Nat} \to \mathrm{Nat}$ for x because the right-most type assignment is the final relevant one, and in addition, there is a valid substitution for the preceding environment $[\mathrm{zero}/x]\ [\![x : \mathrm{Nat}, x = \mathrm{zero} : \mathrm{Nat}]\!]\ [\mathrm{zero}/x]$ that gets overridden. In contrast, the environment $[\![x : \mathrm{Nat}, \mathrm{succ}\, x = \mathrm{zero} : \mathrm{Nat}, x : \mathrm{Nat} \to \mathrm{Nat}]\!]$ relates no environments, not even $[\lambda y.y/x]$ to itself, because the preceeding $[\![x : \mathrm{Nat}, \mathrm{succ}\, x = \mathrm{zero} : \mathrm{Nat}]\!]$ is empty since no choice for $x : \mathrm{Nat}$ satisfies $\mathrm{succ}\, x = \mathrm{zero}$.

From the last point above, we can see that syntactic properties Φ must be interpreted as a binary predicate deciding whether or not that property holds under a given pair of substitutions. This is equivalent to interpreting Φ 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 Φ:

- *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 Φ 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. 9 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 γ and ϕ, 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 $\Gamma \vdash \Phi$, 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 $\Gamma \vdash \Phi$ whenever the environment $[\![\Gamma]\!]$ implies the property Φ. In other words, we can understand $[\![\Gamma \vdash \Phi]\!]$ pointwise as the equivalent statement

$$[\![\Gamma \vdash \Phi]\!] = \forall\ \rho\ [\![\Gamma]\!]\ \rho'.\ \rho\ [\![\Phi]\!]\ \rho' \text{ holds}$$

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

Property 6.14. *For any pre-candidate $\mathbb{A}$ and binary substitution relations γ, ϕ, and ϕ':*

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

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 $E\ \mathbb{A}\ E'$.

6.4 Universal adequacy 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 α* 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 6.15 ((De)Constructive (Co)Induction)**.** *For any sound candidate* $\mathbb{A}$ *and substitution relation* γ*:*

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 6.14 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 γ 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 6.12, 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 6.14:

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

- $\forall \alpha \div B.\Psi(x)$. Follows by permuting the bindings of x and α and applying the inductive hypothesis to γ extended with with $\alpha \div [\![B]\!]$ analogously to the previous case.
- $\Phi \Rightarrow \Psi(x)$ where x is not free in Φ. Applying Property 6.14:

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

$$= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Phi]\!] \implies [\![\Psi(x)]\!]$$
$$= \gamma, x : \mathrm{Pos}(\mathbb{A}) \subseteq [\![\Phi \Rightarrow \Psi(x)]\!]$$

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

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

The "if" direction for property 2 follows analogously to the above using Property 6.12 for $\mathrm{Neg}(\mathbb{A})$ in the base case of an equality $\langle V \| \alpha \rangle = \langle V' \| \alpha \rangle$. ■

Lemma 6.15 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 6.16 (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 6.17 ($\omega\!\rightarrow$)**.** $[\![\Gamma, \alpha \div A \to B \vdash \Psi(\alpha)]\!]$ *if and only if* $[\![\Gamma, x : A, \beta \div B \vdash \Psi(x \cdot \beta)]\!]$.

Proof By viewing $[\![A \to B]\!]$ in terms core call-stack relation $[\![A]\!] \odot [\![B]\!]$ (Property 6.16),

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

we learn from Lemma 6.15 that the quantification over $\alpha \div \mathrm{Neg}([\![A]\!] \odot [\![B]\!])$ is equivalent to the same quantification over call stacks:

$$\begin{aligned}
&[\![\Gamma]\!], \alpha \div \mathrm{Neg}([\![A]\!] \odot [\![B]\!]) \subseteq [\![\Psi(\alpha)]\!] \\
&= [\![\Gamma]\!], \alpha \div [\![A]\!] \odot [\![B]\!] \subseteq [\![\Psi(\alpha)]\!] && \text{(Lemma 6.15)} \\
&= [\![\Gamma]\!], x : [\![A]\!], \beta \div [\![B]\!]. \subseteq [\![\Psi(x \cdot \beta)]\!] && (x, \beta \notin FV(\Gamma) \cup FV(\Psi)) \\
&= [\![\Gamma, x : A, \beta \div B]\!] \subseteq [\![\Psi(x \cdot \beta)]\!] \\
&= [\![\Gamma, x : A, \beta \div B \vdash \Psi(x \cdot \beta)]\!]
\end{aligned}$$

■

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

$$\text{zero}\ \llbracket\text{zero}\rrbracket\ \text{zero} := \text{trivially true} \qquad (\text{succ}\, V)\ \llbracket\text{succ}\rrbracket(\mathbb{A})\ (\text{succ}\, V') := V\ \mathbb{A}\ V'$$

in order to define larger and larger approximations of the $\llbracket\text{Nat}\rrbracket$ equality candidate:

$$\llbracket\text{Nat}\rrbracket_0 := \text{Pos}\{\} \qquad \llbracket\text{Nat}\rrbracket_{i+1} := \text{Pos}(\llbracket\text{zero}\rrbracket \vee \llbracket\text{succ}\rrbracket(\llbracket\text{Nat}\rrbracket_i))$$

At the limit, the union of all under-approximations $\bigvee_i \llbracket\text{Nat}\rrbracket_i$ is the Kleene-style fixed point definition 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

$$(\text{head}\, E)\ \llbracket\text{head}\rrbracket(\mathbb{A})\ (\text{head}\, E') := E\ \mathbb{A}\ E' \qquad (\text{tail}\, E)\ \llbracket\text{tail}\rrbracket(\mathbb{A})\ (\text{tail}\, E') := E\ \mathbb{A}\ E'$$

we can define smaller and smaller approximations of $\llbracket\text{Stream}\, A\rrbracket$:

$$\llbracket\text{Stream}\, A\rrbracket_0 := \text{Neg}\{\} \qquad \llbracket\text{Stream}\, A\rrbracket_{i+1} := \text{Neg}(\llbracket\text{head}\rrbracket(\llbracket A\rrbracket) \wedge \llbracket\text{tail}\rrbracket(\llbracket\text{Stream}\, A\rrbracket_i))$$

Here, the intersection of over-approximations $\bigwedge_i \llbracket\text{Stream}\, A\rrbracket_i$ 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. 9.

Lemma 6.18 (Positive Numbers & Negative Streams)**.** *Under both call-by-value and call-by-name evaluation,*

$$\llbracket\text{Nat}\rrbracket = \bigvee_{i=0}^{\infty} \llbracket\text{Nat}\rrbracket_i \qquad \llbracket\text{Stream}\, A\rrbracket = \bigwedge_{i=0}^{\infty} \llbracket\text{Stream}\, A\rrbracket_i$$

Proof sketch Generalizing the proof from (Downen & Ariola, 2023) from sets to binary relations requires the analogous fact that

$$\bigvee_{i=0}^{\infty} \llbracket\text{Nat}\rrbracket_i = \bigvee_{i=0}^{\infty} \llbracket\text{Nat}\rrbracket_i \qquad \bigwedge_{i=0}^{\infty} \llbracket\text{Stream}\, A\rrbracket_i = \bigwedge_{i=0}^{\infty} \llbracket\text{Stream}\, A\rrbracket_i$$

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 that $\bigvee_{i=0}^{\infty} \llbracket\text{Nat}\rrbracket_i$ relates the following instance of the recursor to itself:

$$\mathbf{rec}_\infty := \mathbf{rec}\{\text{zero} \to \text{zero} \mid \text{succ}\ \_ \to x.x\}\ \mathbf{with}\ \alpha \qquad \mathbf{rec}_\infty \bigvee_{i=0}^{\infty} \llbracket\text{Nat}\rrbracket_i\ \mathbf{rec}_\infty$$

Then, given any $V\ \bigvee_{i=0}^{\infty} \llbracket\text{Nat}\rrbracket_i\ \mathbf{rec}_\infty\ V'$, 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 $V\ \llbracket\text{Nat}\rrbracket_i V'$ for some i^{th} finite approximation.

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

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

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

Then, given any $V\ [\![A]\!]\ V'$ and $E\ \curlyvee_{i=0}^{\infty} [\![\text{Nat}]\!]_i\ \mathbf{rec}_\infty\ E'$, 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 $E\ [\![\text{Stream}\, A]\!]_i E'$ 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. 9. □

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 ωNat and ωStream are interpreted as equivalent statements to their conclusions.

Lemma 6.19 (ωNat)**.**
$[\![\Gamma, x : \text{Nat} \vdash \Psi(x)]\!]$ *if and only if* $[\![\Gamma \vdash \Psi(\text{zero})]\!]$ *and* $[\![\Gamma, x : \text{Nat}, \Psi(x) \vdash \Psi(\text{succ}\, x)]\!]$.

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

For the "if" direction, assume that $[\![\Gamma \vdash \Psi(\text{zero})]\!]$ and $[\![\Gamma, x : \text{Nat} \vdash \Psi(\text{succ}\, x)]\!]$ hold, and we will show that $[\![\Gamma, x : \text{Nat} \vdash \Psi(x)]\!]$ holds, too. From Lemmas 6.15 and 6.18, it suffices to show that $[\![\Gamma]\!], x : \bigvee_i [\![\text{Nat}]\!]_i \subseteq [\![\Psi(x)]\!]$ holds. We can now proceed by proving each individual approximation $[\![\Gamma]\!], x : [\![\text{Nat}]\!]_i \subseteq [\![\Psi(x)]\!]$ by induction on i.

- (Base case: prove it for $i = 0$) $[\![\text{Nat}]\!]_0 = \text{Pos}\{\}$, so we must show $[\![\Gamma]\!], x : \text{Pos}\{\} \subseteq [\![\Psi(x)]\!]$. By Lemma 6.15, 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: prove it for $i+1$) $[\![\text{Nat}]\!]_i = \text{Pos}([\![\text{zero}]\!] \vee [\![\text{succ}]\!]([\![\text{Nat}]\!]_i))$. By Lemma 6.15, these statements

  $$\begin{aligned}&[\![\Gamma]\!], x : \text{Pos}([\![\text{zero}]\!] \vee [\![\text{succ}]\!]([\![\text{Nat}]\!]_i)) \subseteq [\![\Psi(x)]\!]\\ &= [\![\Gamma]\!], x : [\![\text{zero}]\!] \vee [\![\text{succ}]\!]([\![\text{Nat}]\!]_i) \subseteq [\![\Psi(x)]\!]\\ &= ([\![\Gamma]\!], x : [\![\text{zero}]\!] \subseteq [\![\Psi(x)]\!]) \text{ and } ([\![\Gamma]\!], x : \text{succ}([\![\text{Nat}]\!]_i) \subseteq [\![\Psi(x)]\!])\end{aligned}$$

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

equivalent to $[\![\Gamma]\!], x : [\![\mathrm{succ}]\!]([\![\mathrm{Nat}]\!]_i) \subseteq [\![\Phi(x)]\!]$. Therefore, the equivalent statements

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

hold.

■

Lemma 6.20 (ωStream)**.** $[\![\Gamma, \alpha \div \mathrm{Stream}\, A \vdash \Psi(\alpha)]\!]$ *if and only if* $[\![\Gamma, \beta \div A \vdash \Psi(\mathrm{head}\, \beta)]\!]$ *and* $[\![\Gamma, \alpha \div \mathrm{Stream}\, A, \Psi(\alpha) \vdash \Psi(\mathrm{tail}\, \alpha)]\!]$.

Proof Analogous to Lemma 6.19. 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 $[\![\Gamma]\!], \alpha \div \bigwedge_i [\![\mathrm{Stream}\, A]\!]_i \subseteq [\![\Psi(\alpha)]\!]$ via Lemmas 6.15 and 6.18, which follows by induction on i using Lemma 6.15 in a similar manner as in Lemma 6.19. ■

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

Theorem 6.21 (Adequacy)**.** *If* $\Gamma \vdash \Phi$ *is derivable in the weak equational theory, then* $[\![\Gamma \vdash \Phi]\!]$ *is true for both call-by-value and call-by-name evaluation.*

Lemma 6.22. $\alpha\ [\![\mathrm{Nat}]\!]\ \alpha$*, and* $x\ [\![\mathrm{Stream}\, A]\!]\ x$ *and* $x\ [\![A \to B]\!]\ x$ *for any* α *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 6.12, and thus $x\ [\![A \to B]\!]\ x$ by Property 6.16.

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 6.12. Finally, $\alpha\ \curlyvee_i [\![\mathrm{Nat}]\!]_i\ \alpha$ by Property 6.13, and thus $\alpha\ [\![\mathrm{Nat}]\!]\ \alpha$ by Lemma 6.18.

The fact that $x\ [\![\mathrm{Stream}\, A]\!]\ x$ follows from Properties 6.12 and 6.13 and Lemma 6.18 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 6.23 (Observational Equivalence)**.** *In the weak equational theory, 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 compatibility of the universal equational theory. By adequacy (Theorem 6.21) it must be that $[\![\Theta \vdash C[c] = C[c']]\!]$, *i.e.,* for any substitution ρ $[\![\Theta]\!]$ ρ', then $C[c][\rho] \perp\!\!\!\perp C[c'][\rho]$. Note that all (co)variable type assignments in Θ have the form $\alpha \div \mathrm{Nat}$, $x : \mathrm{Stream}\, A$, and $x : A \to B$, so by Lemma 6.22, we know that the $[\![\Theta]\!]$ relates 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 4.1. *The weak extensional equational theory in Fig. 7 is consistent for both the call-by-name and call-by-value semantics.*

Proof A corollary of Theorem 6.23, since observational equivalence is a consistent congruence. ■

6.5 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 6.24 (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** α, 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 6.21) then ensures that $\alpha \div [\![\mathrm{Nat}]\!] \subseteq [\![deep_{\mathrm{Nat}} = deep_{\mathrm{Nat}} \div \mathrm{Nat}]\!]$ and since α $[\![\mathrm{Nat}]\!]$ α (Lemma 6.22), 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, μ-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 6.21) and Lemma 6.22, $minus_n$ $[\![\mathrm{Nat}]\!]$ $minus_n$. From soundness

of $[\![\mathrm{Nat}]\!]$, it follows that the following inconsistent equivalence holds

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

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 6.11, and $\mathrm{Pos}([\![\mathrm{Nat}]\!]) = [\![\mathrm{Nat}]\!]$, because $[\![\mathrm{Nat}]\!]$ is already a complete equality candidate. ■

Lemma 6.25 (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 6.24. Using the value $\mathit{deep}_{\mathrm{Stream}} = \mathbf{corec}\{\mathrm{head}\,\alpha \to \alpha \mid \mathrm{tail}\,\_ \to \beta.\,\mathrm{tail}\,\beta\}\ \mathbf{with}\ x$, which has the type $x : \mathrm{Stream}\,A \vdash \mathit{deep}_{\mathrm{Stream}\,A}$, 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

$$\mathit{raise}_0 := x \qquad \mathit{raise}_{n+1} := \mathbf{corec}\{\mathrm{head}\,\alpha \to \mathrm{head}\,\alpha \mid \mathrm{tail}\,\beta \to \_.\beta\}\ \mathbf{with}\ \mathit{raise}_n$$

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 $\bot\!\!\!\bot$. 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 σNat and σStream, which do not place any restrictions on the kinds of properties they may prove.

Lemma 6.26 (σNat)**.**
$[\![\Gamma, x : \mathrm{Nat} \vdash \Phi]\!]$ *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 6.15 and 6.24, 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 6.27 (σStream)**.** $[\![\Gamma, \alpha \div \mathrm{Stream}\,A \vdash \Phi]\!]$ *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 6.26 by duality using Lemmas 6.15 and 6.25. ■

Theorem 6.28 (Adequacy)**.** *If $\Gamma \vdash \Phi$ is derivable in the strong call-by-value equational theory, then $[\![\Gamma \vdash \Phi]\!]$ is true under call-by-value evaluation. Likewise, If $\Gamma \vdash \Phi$ is derivable in the strong call-by-name equational theory, then $[\![\Gamma \vdash \Phi]\!]$ is true under call-by-name evaluation.*

Proof The same as the proof of Theorem 6.21 with one additional case for σNat in call-by-value or σStream and $\sigma\to$ in call-by-name. ■

Theorem 6.29 (Observational Equivalence)**.** *In the strong call-by-value equational theory and operational semantics, and in the strong call-by-name equational theory and operational semantics, the following hold:*

1. *If $\Gamma \vdash c = c' : A$ then $\Gamma \vdash c \approx c' : A$.*
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 4.3. *The strong call-by-name and call-by-value equational theories are consistent.*

Proof Both Theorems 4.3 and 6.29 are proved the same as Theorems 4.1 and 6.23, using the generalized Theorem 6.28 in place of Theorem 6.21. ■

7 Related Work

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).

Coinduction has also been brought to program verification, it has been implemented in Dafny (Leino & Moskal, 2014), in Liquid Haskell (Mastorou *et al.*, 2022), and in Agda following the work of Abel *et al.* (Abel *et al.*, 2013). Coq is one of the few formal verifiers with a long history of native support for coinduction (Chlipala, 2013) (Giménez, 1996). 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."

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 fixpoint 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 as 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.

The advantage of coinduction is that it allows one to avoid working with numbers (Gordon, 1994), a proof is completely based on the structure of programs, analogously to the notion of a 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 equational theory encompasses well-known principles of strong induction and bisimulation of corecursive processes.

8 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 synch, 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.

References

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.

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.

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.

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

Downen, P. and Ariola, Z. M. (2014) The duality of construction. *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*. Lecture Notes in Computer Science 8410, pp. 249–269. Springer Berlin Heidelberg.

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.

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.

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

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.

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.

Kozen, D. and Silva, A. (2017) Practical coinduction. *Mathematical Structures in Computer Science* **27**(7):1132–1152.

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.

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.

Pédrot, P.-M. and Tabareau, N. (2017) An Effectful Way to Eliminate Addiction to Dependence. *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on* p. 12.

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.

Sangiorgi, D. (2009) On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.* **31**(4).