Structures for Structural Recursion

Paul Downen Philip Johnson-Freyd Zena M. Ariola
University of Oregon, USA
{pdownen,philipjf,ariola}@cs.uoregon.edu

Abstract

Our goal is to develop co-induction from our understanding of induction, putting them on level ground as equal partners for reasoning about programs. We investigate several structures which represent well-founded forms of recursion in programs. These simple structures encapsulate reasoning by primitive and noetherian induction principles, and can be composed together to form complex recursion schemes for programs operating over a wide class of data and co-data types. At its heart, this study is guided by *duality*: each structure for recursion has a dual form, giving perfectly symmetric pairs of equal and opposite data and co-data types for representing recursion in programs. Duality is brought out through a framework presented in sequent style, which inherently includes control effects that are interpreted logically as classical reasoning principles. To accommodate the presence of effects, we give a calculus parameterized by a notion of strategy, which is strongly normalizing for a wide range of strategies. We also present a more traditional calculus for representing effect-free functional programs, but at the cost of losing some of the founding dualities.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Program and recursion schemes

Keywords Recursion; Induction; Coinduction; Duality; Structures; Classical Logic; Sequent Calculus; Strong Normalization

1. Introduction

Martin-Löf's type theory [5, 15] taught us that inductive definitions and reasoning are pervasive throughout proof theory, mathematics, and computer science. Inductive data types are used in programming languages like ML and Haskell to represent structures, and in proof assistants and dependently typed languages like Coq and Agda to reason about finite structures of arbitrary size. Mendler [17] showed us how to talk about recursive types and formalize inductive reasoning over arbitrary data structures. However, the foundation for the opposite to induction, co-induction, has not fared so well. Co-induction is a major concept in programming, representing endless processes, but it is often neglected, misunderstood, or mistreated. As articulated by McBride [19]:

> We are obsessed with foundations partly because we are aware of a number of significant foundational problems that we've got to get right before we can do anything realistic. The thing I would think of ... is coinduction and reasoning about corecursive processes. That's currently, in all major implementations of type theory, a disaster. And if we're going to talk about real systems, we've got to actually have something sensible to say about that.

The introduction of copatterns for coinduction [3] is a major step forward in rectifying this situation. Abel *et al.* emphasize that there is a dual view to inductive data types, in which the values of types

are defined by how they are used instead of how they are built, a perspective on *co-data types* first spurred on by Hagino [12]. Co-inductive co-data types are exciting because they may solve the existing problems with representing infinite objects in proof assistants like Coq [2].

The primary thrust of this work is to improve the understanding and treatment of co-induction, and to integrate both induction and co-induction into a cohesive whole for representing well-founded recursive programs. Our main tools for accomplishing this goal are the pervasive and overt duality and symmetry that runs through classical logic and the sequent calculus. By developing a representation of well-founded induction in a language for the classical sequent calculus, we get an equal and opposite version of well-founded co-induction "for free." Thus, the challenges that arise from using classical sequent calculus as a foundation for induction are just as well the challenges of co-induction, as the two are inherently developed simultaneously. Afterward, we translate the developments of induction and co-induction in the classical sequent calculus to a λ-calculus based language for effect-free programs, to better relate to the current practice of type theory and functional programming. As the λ-based style lacks symmetries present in the sequent calculus, some of the constructs for recursion are lost in translation. Unsurprisingly, the cost of an asymmetrical viewpoint is blindness to the complete picture revealed by duality.

Our philosophy is to emphasize the disentanglement of the recursion in types from the recursion in programs, to attain a language rich in both data and co-data while highlighting their dual symmetries. On the one hand, the Coq viewpoint is that *all* recursive types—both inductive and co-inductive—are represented as data types (positive types in polarized logic [16]), where induction

allows for infinitely deep destruction and co-induction allows for infinitely deep construction. On the other hand, the copattern approach [2, 3] represents inductive types as data and co-inductive types as co-data. In contrast, we take the view that separates the recursive definition of types from the types used for specifying recursive processing loops. Thereby, the types for representing the structure of a recursive process are given first-class status, defined on their own independently of any other programming construct. This makes the types more compositional, so that they may be combined freely in more ways, as they are not confined to certain restrictions about how they relate to data vs co-data or induction vs co-induction. More traditional views on the distinction between inductive and co-inductive programs come from different modes of use for the

same building blocks, emerging from particular compositions of several (co-)data types.

The primary calculus for recursion that we study corresponds to a classical logic, so it inherently contains *control effects* [11] that allow programs to abstract over their own control-flow—intuitionistic logic and effect-free functional programs are later considered as a special case. For that reason, the intended *evaluation strategy* for a program becomes an essential part of understanding its meaning: even terminating programs give different results for different strategies. For example, the functional program $length(\mathsf{Cons}\ (error\ \text{“}\mathtt{boom}\text{”})\ \mathsf{Nil})$ returns 1 under call-by-name (lazy) evaluation, but goes “boom” with an error under call-by-value (strict) evaluation. Therefore, a calculus that talks about the behavior of programs needs to consider the impact of the evaluation strategy. Again, we disentangle this choice from the calculus itself, boiling down the distinction as a *discipline* for substitution. We get a family of calculi, parameterized by this substitution discipline, for reasoning about the behavior of programs ultimately executed with some evaluation strategy. The issue of strong normalization is then shown uniformly over this family of calculi by specifying some basic requirements of the chosen discipline.

The bedrock on which we build our structures for recursion is the connection between logic and programming languages, and the cornerstone of the design is the duality permeating these programming concepts. Induction and co-induction are clearly dual, and to better highlight their symmetric opposition we base our language in the symmetric setting of the sequent calculus. Here, classicality is not just a feature, but an essential completion of the duality needed to fully express the connections between recursion and co-recursion. We consider several different types for representing recursion in pro-

grams based on the mathematical principles of *primitive* and *noetherian* recursion which are reflected as pairs of dual data and co-data types. As we will find, both of these different recursive principles have different strengths when considered programmatically: primitive recursion allows us to simulate seemingly infinite constructed objects, like potentially infinite lists in Coq or Haskell, whereas noetherian recursion admits type-erasure. In essence, we demonstrate how this parametric sequent calculus can be used as a core calculus and compilation target for establishing well-foundedness of recursive programs, via the computational interpretation of common principles of mathematical induction.

We begin by presenting some basic functional programs, including copatterns [3], in a sequent based syntax to illustrate how the sequent calculus gives a language for programming with structures and duality (Section 2) in which *all* types, including functions and polymorphism, are treated as user-defined data and co-data types (Section 3). Next, we develop two forms of well-founded recursion in types—based on primitive and noetherian recursion—along with specific data and co-data types for performing well-founded recursion in programs (Section 4). These two recursion schemes are incorporated into the sequent calculus language, and we demonstrate a rewriting theory that is strongly normalizing for well-typed programs and supports erasure of computationally irrelevant types at run-time (Section 5). Finally, we illustrate the natural deduction counterpart to our sequent calculus language, and show how the recursive constructs developed for classically effectful programs can be imported into a language for effect-free functional programming (Section 6).

2. Programming with Structures and Duality

Pattern-matching is an integral part of functional programming languages, and is a great boon to their elegance. However, the traditional language of pattern-matching can be lacking in areas, especially when we consider *dual* concepts that arise in all programs. For example, when defining a function by patterns, we can match on the structure of the *input*—the argument given to the function—but not its *output*—the observation being made about its result. In contrast, calculi inspired by the sequent calculus feature a more symmetric language which both highlights and restores this missing duality. Indeed, in a setting with such ingrained symmetry, maintaining dualities is natural. We now consider how concepts from functional programming translate to a sequent-based language, and how programs can leverage duality by writing basic functional programs in this symmetric setting.

Example 1. One of the most basic functional programs is the function that calculates the length of a list. We can write this $length$ function in a Haskell- or Agda-like language by pattern-matching over the structure of the given $\mathsf{List}\ a$ to produce a Nat:

$$\begin{array}{ll} \textbf{data}\ \mathsf{Nat}\ \textbf{where} & \textbf{data}\ \mathsf{List}\ a\ \textbf{where} \\ \quad \mathsf{Z} : \mathsf{Nat} & \quad \mathsf{Nil} : \mathsf{List}\ a \\ \quad \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat} & \quad \mathsf{Cons} : a \to \mathsf{List}\ a \to \mathsf{List}\ a \end{array}$$

$$\begin{array}{ll} length\ \mathsf{Nil} & = \mathsf{Z} \\ length\ (\mathsf{Cons}\, x\ xs) & = \textbf{let}\ y = length\ xs\ \textbf{in}\ \mathsf{S}\, y \end{array}$$

This definition of $length$ describes its result for every possible call. Similarly, we can define $length$ in the $\mu\tilde{\mu}$-calculus[1] [8], a language

based on Gentzen's sequent calculus, in much the same way. First, we introduce the types in question by data declarations in the sequent calculus:

$$\begin{array}{lr} \textbf{data}\ \mathsf{Nat}\ \textbf{where} & \\ \quad \mathsf{Z}: & \vdash \mathsf{Nat}\ | \\ \quad \mathsf{S}: & \mathsf{Nat} \vdash \mathsf{Nat}\ | \end{array} \qquad \begin{array}{lr} \textbf{data}\ \mathsf{List}(a)\ \textbf{where} & \\ \quad \mathsf{Nil}: & \vdash \mathsf{List}(a)\ | \\ \quad \mathsf{Cons}: & a, \mathsf{List}(a) \vdash \mathsf{List}(a)\ | \end{array}$$

While these declarations give the same information as before, the differences between these specific data type declarations are largely stylistic. Instead of describing the constructors in terms of a predefined function type, the shape of the constructors are described via *sequents*, replacing function arrows with entailment ($\vdash$) and commas for separating multiple inputs. Furthermore, the type of the main output produced by each constructor is highlighted to the right of the sequent between entailment and a vertical bar, as in $\vdash \mathsf{Nat}\ |$ or $\vdash \mathsf{List}(a)|$, and all other types describe the parameters that must be given to the constructor to produce this output. Thus, we can construct a list as either Nil or $\mathsf{Cons}(x, xs)$, much like in functional languages. Next, we define $length$ by specifying its behavior for every possible call:

$$\begin{aligned} \langle length \| \mathsf{Nil} \cdot \alpha \rangle &= \langle \mathsf{Z} \| \alpha \rangle \\ \langle length \| \mathsf{Cons}(x, xs) \cdot \alpha \rangle &= \langle length \| xs \cdot \tilde{\mu} y. \langle \mathsf{S}(y) \| \alpha \rangle \rangle \end{aligned}$$

The main difference is that we consider more than just the argument to $length$. Instead, we are describing the action of $length$ with its entire context by showing the behavior of a *command*, which connects together a producer and a consumer. For example, in the

command $\langle \mathsf{Z} \| \alpha \rangle$, Z is a *term* producing zero and α is a *co-term*—specifically a *co-variable*—that consumes that number. Besides co-variables, we have other co-terms that consume information. The call-stack $\mathsf{Nil} \cdot \alpha$ consumes a function by supplying it with Nil as its argument and consuming its returned result with α. The input abstraction $\tilde{\mu} y. \langle \mathsf{S}(y) \| \alpha \rangle$ names its input y before running the command $\langle \mathsf{S}(y) \| \alpha \rangle$, similarly to the context $\mathbf{let}\, y = \square \,\mathbf{in}\, \mathsf{S}(y)$ from the functional program.

[1] Note that symbols μ and $\tilde{\mu}$ used here are not related to recursion, but rather are binders for variables and their dual co-variables in the tradition of [6].

In functional programs, it is common to avoid explicitly naming the result of a recursive call, especially in such a short program. Instead, we would more likely define $length$ as:

$$\begin{aligned} length\ \mathsf{Nil} &= \mathsf{Z} \\ length\ (\mathsf{Cons}\, x\ xs) &= \mathsf{S}(length\ xs) \end{aligned}$$

We can mimic this definition in the sequent calculus as:

$$\begin{aligned} \langle length \| \mathsf{Nil} \cdot \alpha \rangle &= \langle \mathsf{Z} \| \alpha \rangle \\ \langle length \| \mathsf{Cons}(x, xs) \cdot \alpha \rangle &= \langle \mathsf{S}(\mu\beta.\langle length \| xs \cdot \beta \rangle) \| \alpha \rangle \end{aligned}$$

Note that to represent the functional call $length\ xs$ inside the successor constructor S, we need to make use of a new kind of term: the output abstraction $\mu\beta.\langle length \| xs \cdot \beta \rangle$ names its output channel β before running the command $\langle length \| xs \cdot \beta \rangle$, which calls $length$ with xs as the argument and β as the return consumer. In the $\mu\tilde{\mu}$-calculus, output abstractions are exactly dual to input abstractions, and defining $length$ in $\mu\tilde{\mu}$ requires us to name the recursive result as either an input or an output. *End example* 1.

We have seen how to write a recursive function by pattern-matching on the first argument, x, in a call-stack $x \cdot \alpha$. However, why should we be limited to only matching on the structure of the argument x? If the observations on the returned result must also follow a particular structure, why can't we match on α as well? Indeed, in a dually symmetric language, there is no such distinction. For example, the function call-stack itself can be viewed as a structure, so that a curried chain of function applications $f\ x\ y\ z$ is represented by the pattern $x \cdot y \cdot z \cdot \alpha$, which reveals the nested structure down the output side of function application, rather than the input side. Thus, the sequent calculus reveals a dual way of

thinking about information in programs phrased as *co-data*, in which observations follow predictable patterns, and values respond to those observations by matching on their structure. In such a symmetric setting, it is only natural to match on any structure appearing in *either* inputs or outputs.

Example 2. We can consider this view on co-data to understand programs with "infinite" objects. For example, infinite streams may be defined by the primitive projections *out* of streams:

$$\begin{aligned}&\mathbf{codata}\,\mathsf{Stream}(a)\,\mathbf{where}\\ &\quad\mathsf{Head}: \quad \mid \mathsf{Stream}(a) \vdash a\\ &\quad\mathsf{Tail}: \quad \mid \mathsf{Stream}(a) \vdash \mathsf{Stream}(a)\end{aligned}$$

Contrarily to data types, the type of the main input consumed by co-data constructors is highlighted to the left of the sequent in between a vertical bar and entailment, as in $\mid \mathsf{Stream}(a) \vdash$. The rest of the types describe the parameters that must be given to the constructor in order to properly consume this main input. For $\mathsf{Streams}$, the observation $\mathsf{Head}[\alpha]$ requests the head value of a stream which should be given to α, and $\mathsf{Tail}[\beta]$ asks for the tail of the stream which should be given to β.[2] We can now define a function $countUp$—which turns an x of type Nat into the infinite stream $x, \mathsf{S}(x), \mathsf{S}(\mathsf{S}(x)), \ldots$—by pattern-matching on the structure of observations on functions and streams:

$$\begin{aligned}\langle countUp \| x \cdot \mathsf{Head}[\alpha] \rangle &= \langle x \| \alpha \rangle\\ \langle countUp \| x \cdot \mathsf{Tail}[\beta] \rangle &= \langle countUp \| \mathsf{S}(x) \cdot \beta \rangle\end{aligned}$$

If we compare $countUp$ with $length$ in this style, we can see that there is no fundamental distinction between them: they are both defined by cases on their possible observations. The only point of

[2] We use square brackets as grouping delimiters in observations, like the head projection $\mathsf{Head}[\alpha]$ out of a stream, as opposed to round parentheses used as grouping delimiters in results, like the successor number $\mathsf{S}(y)$. This helps to disambiguate between results (terms) and observations (co-terms) in a way that is syntactically apparent independently of their context.

difference is that $length$ happens to match on its input data structure in its call-stack, whereas $countUp$ matches on its return co-data structure.

Abel *et al.* [3] have carried this intuition back into the functional paradigm. For example, we can still describe streams by their Head and Tail projections, and define $countUp$ through co-patterns:

$$\begin{aligned}
&\mathbf{codata}\ \mathsf{Stream}\ a\ \mathbf{where}\\
&\qquad \mathsf{Head} : \mathsf{Stream}\ a \to a\\
&\qquad \mathsf{Tail} : \mathsf{Stream}\ a \to \mathsf{Stream}\ a\\
&(countUp\ x).\,\mathsf{Head} = x\\
&(countUp\ x).\,\mathsf{Tail} \ = countUp\ (\mathsf{S}\ x)
\end{aligned}$$

This definition gives the functional program corresponding to the sequent version of $countUp$. So we can see that co-patterns arise naturally, in Curry-Howard isomorphism style, from the computational interpretation of Gentzen's sequent calculus.

Since a symmetric language is not biased against pattern-matching on inputs or outputs, and indeed the two are treated identically, there is nothing special about matching against *both* inputs and outputs simultaneously. For example, we can model infinite streams with possibly missing elements as $\mathsf{SkipStream}(a) = \mathsf{Stream}(\mathsf{Maybe}(a))$, where $\mathsf{Maybe}(a)$ corresponds to the Haskell

datatype with constructors $\mathsf{Nothing}$ and $\mathsf{Just}(x)$ for x of type a. Then we can define the empty skip stream which gives $\mathsf{Nothing}$ at every position, and the $countDown$ function that transforms $\mathsf{S}^n(\mathsf{Z})$ into the stream $\mathsf{S}^n(\mathsf{Z}), \mathsf{S}^{n-1}(\mathsf{Z}), \ldots, \mathsf{Z}, \mathsf{Nothing}, \ldots$:

$$\begin{aligned}
\langle empty \| \mathsf{Head}[\alpha] \rangle &= \langle \mathsf{Nothing} \| \alpha \rangle \\
\langle empty \| \mathsf{Tail}[\beta] \rangle &= \langle empty \| \beta \rangle \\
\\
\langle countDown \| x \cdot \mathsf{Head}[\alpha] \rangle &= \langle \mathsf{Just}(x) \| \alpha \rangle \\
\langle countDown \| \mathsf{Z} \cdot \mathsf{Tail}[\beta] \rangle &= \langle empty \| \beta \rangle \\
\langle countDown \| \mathsf{S}(x) \cdot \mathsf{Tail}[\beta] \rangle &= \langle countDown \| x \cdot \beta \rangle
\end{aligned}$$

End example 2.

Example 3. As opposed to the co-data approach to describing infinite objects, there is a more widely used approach in lazy functional languages like Haskell and proof assistants like Coq that still favors framing information as data. For example, an infinite list of zeroes is expressed in this functional style by an endless sequence of Cons:

$$zeroes = \mathsf{Cons}\,\mathsf{Z}\,zeroes$$

We could emulate this definition in sequent style as the expansion of $zero$ when observed by any α:

$$\langle zeroes \| \alpha \rangle = \langle \mathsf{Cons}(\mathsf{Z}, zeroes) \| \alpha \rangle$$

Likewise, we can describe the concatenation of two, possibly infinite lists in the same way, by pattern-matching on the call:

$$\begin{aligned}
\langle cat \| \mathsf{Nil} \cdot ys \cdot \alpha \rangle &= \langle ys \| \alpha \rangle \\
\langle cat \| \mathsf{Cons}(x, xs) \cdot ys \cdot \alpha \rangle &= \langle \mathsf{Cons}(x, \mu\beta.\langle cat \| xs \cdot ys \cdot \beta \rangle) \| \alpha \rangle
\end{aligned}$$

The intention is that, so long as we do not evaluate the sub-components of Cons eagerly, then α receives a result even if xs is an infinitely long list like $zeroes$. *End example* 3.

3. A Higher-Order Sequent Calculus

Based on our example programs in Section 2, we now flesh out more formally a higher-order language of the sequent calculus: the $\mu\tilde{\mu}$-calculus. The full syntax of this language is shown in Figure 1. The different components of programs in the $\mu\tilde{\mu}$-calculus can be understood by their relationship between opposing forces of input and output. A term, v, produces an output, a co-term, e, consumes

$$c \in Command ::= \langle v \| e \rangle$$

$$v \in Term ::= x \mid \mu\alpha.c \mid \mathsf{K}(\vec{e}, \vec{v}) \mid \mu(\mathsf{H}[\vec{s}, \vec{\alpha}].c| \ldots) \qquad e \in CoTerm ::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}[\mathsf{K}(\vec{a}, \vec{s}).c| \ldots] \mid \mathsf{H}[\vec{v}, \vec{e}]$$

$$A, B, C, D \in Type ::= a \mid \lambda a : k.B \mid A\ B \mid \mathsf{F}(\vec{A}) \mid \mathsf{G}(\vec{A}) \qquad k, l \in Kind ::= \star \mid k_1 \to k_2$$

Figure 1. The syntax of the higher-order $\mu\tilde{\mu}$-calculus.

an input, and a command, c, neither produces nor consumes, it just *runs*. Thus, we can consider commands to be the computational unit of the language: when we talk about running a program, it is a command which does the running, not a term.

To begin, we focus on the *core* of the $\mu\tilde{\mu}$-calculus, which includes just the substrate necessary for piping inputs and outputs to the appropriate places. In particular, we have two different forms of inputs and outputs: the implicit, unnamed inputs and outputs of terms and co-terms, and the explicit, named inputs and outputs introduced by variables (typically written x, y, z) and co-variables (typically written α, β, γ). Thus, besides variables and co-variables, the core $\mu\tilde{\mu}$-calculus includes the generic abstractions seen in Section 2,

$\mu\alpha.c$ and $\tilde{\mu}x.c$, which mediate between named and unnamed inputs and outputs. The output of the term $\mu\alpha.c$ is named α in the command c, and dually the input of the co-term $\tilde{\mu}x.c$ is named x in c.

Even though the core $\mu\tilde{\mu}$-calculus has not introduced any specific types yet, we can still consider its type system for ensuring proper communication between producers and consumers, shown in Figure 2. The (typed) free variables and co-variables are tracked in separate contexts, written Γ and Δ respectively, and the entailment ($\vdash$) separates inputs on the left from outputs on the right. Additionally, the context, Θ, for type variables (written a, b, c, d), being neither input nor output, adorn the turnstyle itself. Since programs of the $\mu\tilde{\mu}$-calculus are made up of three different forms of components, the typing rules use three different forms of sequents: $\Gamma \vdash_\Theta v : A|\Delta$ states that v is a term producing an output of type A, $\Gamma|e : A \vdash_\Theta \Delta$ states that e is a co-term consuming an input of type A, and $c : (\Gamma \vdash_\Theta \Delta)$ states that c is a well-typed command. The language of types and kinds is just the simply typed λ-calculus at the type level with $\star$ as the base kind, $\Theta \vdash A : k$ states that A is a type of kind k, and $\Theta \vdash A = B : k$ states that A and B are $\alpha\beta\eta$-equivalent types of kind k.

This core language does not include any baked-in types. Instead, all types are user-defined by a general declaration mechanism for (co-)data types introduced in [8], similar to the data declaration mechanisms of functional languages but generalized through duality. Data declarations introduce new constructed terms as well as a new *case abstraction* co-term that performs case analysis to destruct its input before deciding which branch to take similar to the context $\mathbf{case}\,\square\,\mathbf{of}\,\ldots$ in functional languages. Co-data declarations are exactly symmetric, introducing new constructed co-terms as well as a new case abstraction term that performs case analysis on its output

before deciding how to respond.

We already saw some example declarations previously for Nat, $\mathsf{List}(a)$, and $\mathsf{Stream}(a)$. As it turns out, *all* the basic types from functional programming languages follow the same pattern and can be declared as user-defined types. For example, pairs are defined as:

$$\begin{array}{c}\mathbf{data}\,(a:\star)\otimes(b:\star)\,\mathbf{where}\\ (\_,\_):\quad a,b\vdash a\otimes b|\end{array}$$

which says that building a pair of type $a \otimes b$ requires the terms v of type a and v' of type b, obtaining the constructed pair (v, v'). Destruction of pairs, expressed by the case abstraction co-term $\tilde{\mu}[(x, y).c]$, pattern-matches on its input pair before running the command c.

Furthermore, we can declare the type for functions as:

$$\begin{array}{c}\mathbf{codata}\,(a:\star)\to(b:\star)\,\mathbf{where}\\ \_\cdot\_:\quad a|a\to b\vdash b\end{array}$$

This co-data declaration says that building a function call-stack of type $a \to b$ requires a term v of type a and a co-term e of type b, obtaining the constructed stack $v \cdot e$. Destruction of call-stacks, expressed by the case abstraction term $\mu(x \cdot \alpha.c)$, pattern-matches on its output stack before running c. Note that this is an alternative representation of functions to λ-abstractions in functional languages, but an equivalent one. Indeed, the two views of functions are mutually definable:

$$\lambda x.v = \mu(x \cdot \alpha.\langle v \| \alpha\rangle) \qquad \mu(x \cdot \alpha.c) = \lambda x.\mu\alpha.c$$

Here, we generalize the declaration mechanism from [8] to

include higher-order types and quantified type variables. The general forms of (non-recursive) data and co-data declaration in the $\mu\tilde{\mu}$-calculus are given in Figure 3, and the associated typing rules in Figure 4. In addition to the rule for determining when the (co-)data types $\mathsf{F}(\vec{A})$ and $\mathsf{G}(\vec{A})$ are well-kinded, we also have the left and right rules for typing (co-)data structures and case abstractions. By instantiating the (co-)data type constructors at the types $\vec{A}$, we must substitute $\vec{A}$ for all possible occurrences of the parameters $\vec{a}$ in the declaration. Furthermore, the chosen instances $\vec{D}$ for the quantified type variables $\vec{d_i}$, which annotate the constructor, must also be substituted for their occurrences in other types. With this in mind, the rules for construction (the $\mathsf{F}R\mathsf{K}_i$ and $\mathsf{G}L\mathsf{H}_i$ rules) check that the sub-(co-)terms and quantified types of a structure have the expected instantiated types, whereas the rules for deconstruction ($\mathsf{F}L$ and $\mathsf{G}R$) extend the typing contexts with the appropriately typed (co-)variables and type variables.

This form of (co-)data type declaration lets us express not only existential quantification—as in Haskell and Coq—but also universal quantification as well:

$$\mathbf{data}\ \exists(a : \star \to \star)\ \mathbf{where} \qquad \mathbf{codata}\ \forall(a : \star \to \star)\ \mathbf{where}$$
$$\mathsf{Pack} : \quad a\ b \vdash_{b:\star} \exists a| \qquad \mathsf{Spec} : \quad |\forall a \vdash_{b:\star} a\ b$$

Notice that these general patterns give us the expected typing rules:

$$\frac{c : \Gamma \vdash_{\Theta, b:\star} \alpha : A\ b, \Delta}{\Gamma \vdash_\Theta \mu(\mathsf{Spec}^{b:\star}[\alpha].c) : \forall A|\Delta} \qquad \frac{\Theta \vdash B : \star \quad \Gamma|e : A\ B \vdash_\Theta \Delta}{\Gamma|\,\mathsf{Spec}^B[e] : \forall A \vdash_\Theta \Delta}$$

$$\frac{\Theta \vdash B : \star \quad \Gamma \vdash_\Theta v : A\ B|\Delta}{\Gamma \vdash_\Theta \mathsf{Pack}^B(v) : \exists A|\Delta} \qquad \frac{c : \Gamma, x : A\ b \vdash_{\Theta, b:\star} \Delta}{\Gamma|\tilde{\mu}[\mathsf{Pack}^{b:\star}(x).c] : \exists A \vdash_\Theta \Delta}$$

Using a recursively-defined case abstraction with deep pattern-matching, we can now represent $length$ in the $\mu\tilde{\mu}$-calculus:

$$length = \mu(\mathsf{Nil} \cdot \alpha.\langle \mathsf{Z} \| \alpha \rangle \\ |\mathsf{Cons}(x, xs) \cdot \alpha.\langle length \| xs \cdot \tilde{\mu} y.\langle \mathsf{S}(y) \| \alpha \rangle \rangle)$$

Furthermore, the deep pattern-matching can be mechanically translated to the shallow case analysis for (co-)data types:

$$length = \mu(xs \cdot \alpha.\, \langle xs \| \tilde{\mu}[\mathsf{Nil}.\langle \mathsf{Z} \| \alpha \rangle \\ |\mathsf{Cons}(x, xs').\langle length \| xs' \cdot \tilde{\mu} y.\langle \mathsf{S}(y) \| \alpha \rangle \rangle]\rangle)$$

This case abstraction describes exactly the same specification as the definition for $length$ in Example 1.

$$\frac{}{\Gamma, x : A \vdash_\Theta x : A|\Delta}\ Var \qquad \frac{c : (\Gamma \vdash_\Theta \alpha : A, \Delta)}{\Gamma \vdash_\Theta \mu\alpha.c : A|\Delta}\ Act \qquad \frac{c : (\Gamma, x : A \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}x.c : A \vdash_\Theta \Delta}\ CoAct \qquad \frac{}{\Gamma|\alpha : A \vdash_\Theta \alpha : A, \Delta}\ CoVar$$

$$\frac{\Theta \vdash A = B : \star \quad \Gamma \vdash_\Theta v : A|\Delta}{\Gamma \vdash_\Theta v : B|\Delta}\ Eq \qquad \frac{\Gamma \vdash_\Theta v : A|\Delta \quad \Theta \vdash A : \star \quad \Gamma|e : A \vdash_\Theta \Delta}{\langle v \| e \rangle : (\Gamma \vdash_\Theta \Delta)}\ Cut \qquad \frac{\Theta \vdash A = B : \star \quad \Gamma|e : B \vdash_\Theta \Delta}{\Gamma|e : A \vdash_\Theta \Delta}\ CoEq$$

Figure 2. The type system for the core higher-order $\mu\tilde{\mu}$-calculus.

$$\textbf{data}\ \mathsf{F}(\overrightarrow{a : k})\ \textbf{where} \qquad\qquad \textbf{codata}\ \mathsf{G}(\overrightarrow{a : k})\ \textbf{where}$$

$$\begin{array}{lll} \mathsf{K}_1 : & \overrightarrow{B_1} \vdash_{\overrightarrow{d_1 : l_1}} \mathsf{F}(\vec{a})|\overrightarrow{C_1} & \qquad \mathsf{H}_1 : \quad \overrightarrow{B_1}|\,\mathsf{G}(\vec{a}) \vdash_{\overrightarrow{d_1 : l_1}} \overrightarrow{C_1} \\ \cdots & & \qquad \cdots \\ \mathsf{K}_n : & \overrightarrow{B_n} \vdash_{\overrightarrow{d_n : l_n}} \mathsf{F}(\vec{a})|\overrightarrow{C_n} & \qquad \mathsf{H}_n : \quad \overrightarrow{B_n}|\,\mathsf{G}(\vec{a}) \vdash_{\overrightarrow{d_n : l_n}} \overrightarrow{C_n} \end{array}$$

Figure 3. General form of declarations for user-defined data and co-data.

$$\frac{\overrightarrow{\Theta \vdash A : k}}{\Theta \vdash \mathsf{F}(\overrightarrow{A}) : \star} \qquad \frac{c_1 : (\Gamma, \overrightarrow{x : B_1\{\overrightarrow{A/a}\}} \vdash_{\Theta, \overrightarrow{d_1 : l_1\{\overrightarrow{A/a}\}}} \Delta, \overrightarrow{\alpha : C_1\{\overrightarrow{A/a}\}}) \quad \cdots}{\Gamma|\tilde{\mu}[\mathsf{K}_1^{\overrightarrow{d_1 : l_1}}(\vec{a}, \vec{x}).c_1| \ldots] : \mathsf{F}(\vec{A}) \vdash_\Theta \Delta}\ \mathsf{F}L$$

$$\frac{\overrightarrow{\Theta \vdash D : l_i\{\overrightarrow{A/a}\}} \quad \overrightarrow{\Gamma|e : C_i\{\overrightarrow{A/a}, \overrightarrow{D/d}\} \vdash_\Theta \Delta} \quad \overrightarrow{\Gamma \vdash_\Theta v : B_i\{\overrightarrow{A/a}, \overrightarrow{D/d}\}|\Delta}}{\Gamma \vdash_\Theta \mathsf{K}_i^{\vec{D}}(\vec{e}, \vec{v}) : \mathsf{F}(\vec{A})|\Delta}\ \mathsf{F}R\mathsf{K}_i$$

$$\frac{\overrightarrow{\Theta \vdash A : k}}{\Theta \vdash \mathsf{G}(\vec{A}) : \star} \qquad \frac{c_1 : (\Gamma, x : B_1\{\overrightarrow{A/a}\} \vdash_{\Theta, \overrightarrow{d_1 : l_1\{\overrightarrow{A/a}\}}} \alpha : C_1\{\overrightarrow{A/a}\}, \Delta) \quad \cdots}{\Gamma \vdash_\Theta \mu(\mathsf{H}_1^{\overrightarrow{d_1 : l_1}}[\vec{x}, \vec{a}].c_1| \ldots) : \mathsf{G}(\vec{A})|\Delta}\ \mathsf{G}R$$

$$\frac{\overrightarrow{\Theta \vdash D : l_i\{\overrightarrow{A/a}\}} \quad \overrightarrow{\Gamma \vdash_\Theta v : B_i\{\overrightarrow{A/a}, \overrightarrow{D/d}\}|\Delta} \quad \overrightarrow{\Gamma|e : C_i\{\overrightarrow{A/a}, \overrightarrow{D/d}\} \vdash_\Theta \Delta}}{\Gamma|\,\mathsf{H}_i^{\vec{D}}[\vec{v}, \vec{e}] : \mathsf{G}(\vec{A}) \vdash_\Theta \Delta}\ \mathsf{G}L\mathsf{H}_i$$

Figure 4. Typing rules for non-recursive, user-defined data and co-data types.

In each of the examples in Section 2, we were only concerned with writing recursive programs, but have not showed that they always terminate. Termination is especially important for proof assistants and dependently typed languages, which rely on the absence of infinite loops for their logical consistency. If we consider the programs in Examples 1 and 2, then termination appears fairly straightforward by structural recursion *somewhere* in a function call: each recursive invocation of $length$ has a structurally smaller list for the argument, and each recursive invocation of $countUp$, and

$countDown$ has a smaller stream projection out of its returned result. However, formulating this argument in general turns out to be more complicated. Even worse, the "infinite data structures" in Example 3 do not have as clear of a concept of "termination:" $zeroes$ and concatenation could go on forever, if they are not given a bound to stop. To tackle these issues, we will phrase principles of well-founded recursion in the $\mu\tilde{\mu}$-calculus, so that we arrive at a core calculus capable of expressing complex termination arguments (parametrically to the chosen evaluation strategy) inside the calculus itself (see Section 5).

4. Well-Founded Recursion

There is one fundamental difficulty in ensuring termination for programs written in a sequent calculus style: even incredibly simple programs perform their structural recursion from *within* some larger overall structure. For example, consider the humble $length$ function from Example 1. The decreasing component in the definition of $length$ is clearly the list argument which gets smaller with each call. However, in the sequent calculus, the actual recursive invocation of $length$ is the *entire* call-stack. This is because the recursive call to $length$ does not return to its original caller, but to some place new. When written in a functional style, this information is implicit since the recursive call to $length$ is not a tail-call, but rather $\mathsf{S}(length\ xs)$. When written in a sequent style, this extra information becomes an explicit part of the function call structure, necessary to remember to increment the output of the function before ultimately returning. This means that we must carry around enough memory to store our ever increasing result amidst our ever decreasing recursion.

Establishing termination for sequent calculus therefore requires a more finely controlled language for specifying "what's getting smaller" in a recursive program, pointing out *where* the decreasing measure is hidden within recursive invocations. For this purpose, we adopt a type-based approach to termination checking [1]. Besides allowing us to abstract over termination-ensuring measures, we can also specify which parts of a complex type are used as part of the termination argument. As a consequence for handling simplistic functions like $length$, we will find that, for free, the calculus ends up as a robust language for describing more advanced recursion over structures, including lexicographic and mutual recursion over both data and co-data structures simultaneously.

In considering the type-based approach to termination in the sequent calculus, we identify two different styles for the type-level measure indices. The first is an exacting notion of index with a predictable structure matching the natural numbers and which we use to perform *primitive recursion*. This style of indexing gives us a tight control over the size of structures, allowing us to define types like the fixed-sized vectors of values from dependently typed languages as well as a direct encoding of "infinite" structures as found in lazy functional languages. The second is a looser notion that only tracks the upper bound of indices and which we use to perform *noetherian recursion*. This style of indexing is more in tune with typical structurally recursive programs like $length$ and also supports full run-time erasure of bounded indices while still maintaining termination of the index-erased programs.

4.1 Primitive Recursion

We begin with the more basic of the two recursion schemes: primitive recursion on a single natural number index. These natural number indices are used in types in two different ways. First, the indices act as an explicit measure in recursively defined (co-)data types, tracking the recursive sub-components of their structures in the types themselves. Second, the indices are abstracted over by the primitive recursion principle, allowing us to generalize over arbitrary indices and write looping programs.

Let's consider some examples of using natural number indices for the purpose of defining (co-)data types with recursive structures. We extend the (co-)type declaration mechanism seen previously with the ability to define new (co-)data types by primitive recursion over an index, giving a mechanism for describing recursive (co-)data types with statically tracked measures. Essentially, the constructors

are given in two groups—for the zero and successor cases—and may only contain recursive sub-components at the (strictly) previous index. For example, we may describe vectors of exactly N values of type A, $\mathsf{Vec}(N, A)$, as in dependently typed languages:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{data}\,\mathsf{Vec}(i : \mathsf{Ix}, a : \star)\,\mathbf{by}\text{ primitive recursion on } i} \\ \mathbf{where}\, i = 0 & \mathsf{Nil} : \quad \vdash \mathsf{Vec}(0, a)| \\ \mathbf{where}\, i = j + 1 & \mathsf{Cons} : \quad a, \mathsf{Vec}(j, a) \vdash \mathsf{Vec}(j + 1, a)| \end{array}$$

where Ix is the kind of type-level natural number indices. Nil builds an empty vector of type $\mathsf{Vec}(0, A)$, and $\mathsf{Cons}(v, v')$ extends the vector $v' : \mathsf{Vec}(N, A)$ with another element $v : A$, giving us a vector with one more element of type $\mathsf{Vec}(N + 1, A)$. Other than these restrictions on the instantiations of $i : \mathsf{Ix}$ for vectors constructed by Nil and Cons, the typing rules for terms of $\mathsf{Vec}(N, A)$ follow the normal pattern for declared data types.[3] Destructing a vector diverges more from the usual pattern of non-recursive data types. Since the constructors of vector values are put in two separate groups, we have two separate case abstractions to consider, depending on whether the vector is empty or not. On the one hand, to destruct an empty vector, we only have to handle the case for Nil, as given by the co-term $\tilde{\mu}[\mathsf{Nil}.c]$. On the other, destructing a non-empty vector requires us to handle the Cons case, as given by the co-term $\tilde{\mu}[\mathsf{Cons}(x, xs).c]$. These co-terms are typed by the two left rules for Vec—one for both its zero and successor instances:

$$\frac{c : (\Gamma \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Nil}.c] : \mathsf{Vec}(0, A) \vdash_\Theta \Delta}\ \mathsf{Vec}\, L_0$$

$$\frac{c : (\Gamma, x : A, xs : \mathsf{Vec}(M, A) \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Cons}(x, xs).c] : \mathsf{Vec}(M+1, A) \vdash_\Theta \Delta}\ \mathsf{Vec}\, L_{+1}$$

As a similar example, we can define a less statically constrained list type by primitive recursion. The IxList indexed data type is just like Vec, except that the Nil constructor is available at both the zero and successor cases:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{IxList}(i : \mathsf{Ix}, a : \star)\ \mathbf{by}\ \text{primitive recursion on } i} \\
\mathbf{where}\ i = 0 & \mathsf{Nil} : & \vdash \mathsf{IxList}(0, a)| \\
\mathbf{where}\ i = j+1 & \mathsf{Nil} : & \vdash \mathsf{IxList}(j+1, a)| \\
 & \mathsf{Cons} : & a, \mathsf{IxList}(j, a) \vdash \mathsf{IxList}(j+1, a)|
\end{array}$$

Now, destructing a non-zero $\mathsf{IxList}(N+1, A)$ requires both cases, as given in the co-term $\tilde{\mu}[\mathsf{Nil}.c|\mathsf{Cons}(x, xs).c']$. IxList has three right rules for building terms: for Nil at both 0 and $M+1$ and for Cons. It also has two left rules: one for case abstractions handling the constructors of the 0 case and another for the $M+1$ case.

To write looping programs over these indexed recursive types, we use a recursion scheme which abstracts over the index occurring anywhere within an arbitrary type. As the types themselves are defined by primitive recursion over a natural number, the recursive structure of programs will also follow the same pattern. The trick then is to embody the primitive induction principle for proving a proposition P over natural numbers:

$$P[0] \wedge (\forall j : \mathbb{N}.P[j] \to P[j+1]) \to (\forall i : \mathbb{N}.P[i])$$

[3] We can have a vector with an abstract index if we don't yet know what shape it has, as with the variable x or abstraction $\mu\alpha.c$ of type $\mathsf{Vec}(i, A)$.

and likewise the refutation of such a statement, as is given by any specific counter-example—$n : \mathbb{N} \wedge \overline{P[n]} \to \overline{(\forall i : \mathbb{N}.P[i])}$—into logical rules of the sequent calculus.[4] By the usual reading of sequents, proofs come to the right of entailment ($\vdash A$ means "A is true"), whereas refutations come to the left ($A \vdash$ means "A is false"). Because we will have several recursion principles, we denote this particular one with a type named $\mathsf{Inflate}$, so that the primitive recursive proposition $\forall i : \mathbb{N}.P[i]$ is written as the type $\mathsf{Inflate}(\lambda i : \mathsf{Ix}.A)$ with the inference rules:

$$\frac{\vdash A\,0 \quad A\,j \vdash_{j:\mathsf{Ix}} A\,(j+1)}{\vdash \mathsf{Inflate}(A)} \qquad \frac{\vdash M : \mathsf{Ix} \qquad A\,M \vdash}{\mathsf{Inflate}(A) \vdash}$$

We use this translation of primitive induction into logical rules as the basis for our primitive recursive *co-data type*. The refutation of primitive recursion is given as a specific *counter-example*, so the co-term is a specific construction. Whereas, proof by primitive recursion is a *process* given by *cases*, the term performs case analysis over its observations. The canonical counter-example is described by the co-data type declaration for $\mathsf{Inflate}$:

$$\mathbf{codata}\ \mathsf{Inflate}(a : \mathsf{Ix} \to \star)\ \mathbf{where}$$
$$\mathsf{Up} : \quad \mid \mathsf{Inflate}(a) \vdash_{j:\mathsf{Ix}} a\ j$$

The general mechanism for co-data automatically generates the left rule for constructing the counter-example, and a right rule for extracting the parts of this construction. However, to give a recursive process for $\mathsf{Inflate}$, we need an additional right rule that gives us access to the recursive argument by performing case analysis on the particular index. This scheme for primitive recursion is expressed by the term $\mu(\mathsf{Up}^{0:\mathsf{Ix}}[\alpha].c_0 \mid \mathsf{Up}^{j+1:\mathsf{Ix}}[\alpha](x).c_1)$ which performs case analysis on type-level indices at *run-time*, and which can access the

recursive result through the extra variable x in the successor pattern $\mathsf{Up}^{j+1:\mathsf{Ix}}[\alpha](x)$. This term has the typing rule:

$$\frac{c_0 : (\Gamma \vdash_\Theta \alpha : A\ 0, \Delta) \quad c_1 : (\Gamma, x : A\ j \vdash_{\Theta, j:\mathsf{Ix}} \alpha : A\ (j+1), \Delta)}{\Gamma \vdash_\Theta \mu(\mathsf{Up}^{0:\mathsf{Ix}}[\alpha].c_0 | \ \mathsf{Up}^{j+1:\mathsf{Ix}}[\alpha](x).c_1) : \mathsf{Inflate}(A) | \Delta}$$

Terms of type $\mathsf{Inflate}\, i : \mathsf{Ix}.A$ (which is shorthand for the type $\mathsf{Inflate}(\lambda i : \mathsf{Ix}.A)$) describe a process which is able to produce $A\{N/i\}$, for any index N, by stepwise producing $A\{0/i\}$, $A\{1/i\}$, $\ldots$, $A\{N/i\}$ and piping the previous output to the recursive input

[4] We use the overbar notation, $\overline{P}$, to denote that the proposition P is false. The use of this notation is to emphasize that we are not talking about negation as a logical connective, but rather the *dual* to a proof that P is true, which is a refutation of P demonstrating that it is false.

x of the next step, thus "inflating" the index in the result arbitrarily high. The index of the particular step being handled is part of the constructor pattern, so that the recursive case abstraction knows which branch to take. In contrast, co-terms of type $\mathsf{Inflate}\, i : \mathsf{Ix}.A$ *hide* the particular index at which they can consume an input, thereby forcing their input to work for any index.

By just applying duality in the sequent calculus and flipping everything about the turnstyles, we get the opposite notion of primitive recursion as a data type. In particular, we get the data declaration describing a dual type, named $\mathsf{Deflate}$:

$$\mathbf{data}\ \mathsf{Delfate}(a : \mathsf{Ix} \to \star)\ \mathbf{where}$$
$$\mathsf{Down} : \quad a\ j \vdash_{j:\mathsf{Ix}} \mathsf{Deflate}(a)|$$

The general mechanism for data automatically generates the right rule for constructing an index-witnessed example case, and a left rule for extracting the index and value from this structure. Further, as before we need an additional left rule for performing self-referential recursion for consuming such a construction:

$$\frac{c_1 : (\Gamma, x : A\ (j+1) \vdash_{\Theta, j:\mathsf{Ix}} \alpha : A\ j, \Delta) \quad c_0 : (\Gamma, x : A\ 0 \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Down}^{0:\mathsf{Ix}}(x).c_0|\mathsf{Down}^{j+1:\mathsf{Ix}}(x)[\alpha].c_1] : \mathsf{Deflate}(A) \vdash_\Theta \Delta}$$

Dual to before, the recursive output sink can be accessed through the co-variable α in the pattern $\mathsf{Down}^{j+1:\mathsf{Ix}}(x)[\alpha]$. The terms of type $\mathsf{Deflate}\, i : \mathsf{Ix}.A$ hide the particular index at which they produce an output. In contrast, it is now the co-terms of the type $\mathsf{Deflate}\, i : \mathsf{Ix}.A$ which describe a process which is able to consume $A\{N/i\}$ for any choice of N in steps by consuming $A\{N/i\}, \ldots, A\{0/i\}$ and piping the previous input to the recursive output α of the next step, thus "deflating" the index in the input down to 0.

4.2 Noetherian Recursion

We now consider the more complex of the two recursion schemes: noetherian recursion over well-ordered indices. As opposed to ensuring a decreasing measure by matching on the specific structure of the index, we will instead quantify over arbitrary indices that are less than the current one. In other words, the details of what these indices look like are not important. Instead, they are used as arbitrary upper bounds in an ever decreasing chain, which stops when we run out of possible indices below our current one as guaranteed by the well-foundedness of their ordering. Intuitively, we may jump by leaps and bounds down the chain, until we run out of places to move. Qualitatively, this different approach to recursion measures allows us to abstract parametrically over the index, and generalize so strongly over the difference in the steps to the point where the particular chosen index is unknown. Thus, because a process receiving a bounded index has so little knowledge of what it looks like, the index cannot influence its action, thereby allowing us to totally erase bounded indices during run-time.

Now let's see how to define some types by noetherian recursion on an ordered index. Unlike primitive recursion, we do not need to consider the possible cases for the chosen index. Instead, we quantify over any index which is *less* than the given one. For example, recall the recursive definition of the Nat data type from Example 1. We can be more explicit about tracking the recursive sub-structure of the constructors by indexing Nat with some ordered type, and ensuring that each recursive instance of Nat has a *smaller* index, so that we may define natural numbers by noetherian recursion over ordered indices from a new kind called Ord:

$$\mathbf{data}\ \mathsf{Nat}(i : \mathsf{Ord})\ \mathbf{by}\ \text{noetherian recursion on}\ i\ \mathbf{where}$$

$$\mathsf{Z}: \quad \vdash \mathsf{Nat}(i)|$$
$$\mathsf{S}: \quad \mathsf{Nat}(j) \vdash_{j<i} \mathsf{Nat}(i)|$$

Note that the kind of indices less than i is denoted by $< i$, and we write $j < i$ as shorthand for $j : (< i)$. Noetherian recursion in types is surprisingly more straightforward than primitive recursion, and more closely follows the established pattern for data type declarations:

$$\frac{}{\Gamma \vdash_\Theta \mathsf{Z} : \mathsf{Nat}(N)|\Delta}\ \mathsf{Nat}R\,\mathsf{Z}$$

$$\frac{\Theta \vdash M < N \quad \Gamma \vdash_\Theta v : \mathsf{Nat}(M)|\Delta}{\Gamma \vdash_\Theta \mathsf{S}^M(v) : \mathsf{Nat}(N)|\Delta}\ \mathsf{Nat}R\,\mathsf{S}$$

Z builds a $\mathsf{Nat}(N)$ for any Ord index N, and $\mathsf{S}^M(v)$ builds an incremented $\mathsf{Nat}(N)$ out of a $\mathsf{Nat}(M)$, when $M < N$. To destruct a $\mathsf{Nat}(N)$, for any index N, we have the one case abstraction that handles both the Z and S cases:

$$\frac{c_0 : (\Gamma \vdash_\Theta \Delta) \quad c_1 : (\Gamma, x : \mathsf{Nat}(j) \vdash_{\Theta, j<N} \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Z}.c_0|\mathsf{S}^{j<N}(x).c_1] : \mathsf{Nat}(N) \vdash_\Theta \Delta}\ \mathsf{Nat}L$$

Like the case abstraction for tearing down an existentially constructed value, the pattern for S introduces the free type variable j which stands for an arbitrary index less than N.

We can consider some other examples of (co-)data types defined by noetherian recursion. The definition of finite lists is just an annotated version of the definition from Example 1:

$$\textbf{data}\ \mathsf{List}(i : \mathsf{Ord}, a : \star)\ \textbf{by}\ \text{noetherian recursion on}\ i\ \textbf{where}$$
$$\mathsf{Nil}: \quad \vdash \mathsf{List}(i, a)|$$

$$\mathsf{Cons}: \quad a, \mathsf{List}(j,a) \vdash_{j<i} \mathsf{List}(i,a)|$$

Furthermore, the infinite streams from Example 2 can also be defined as a co-data type by noetherian recursion:

$$\mathbf{codata}\ \mathsf{Stream}(i:\mathsf{Ord}, a:\star)\ \mathbf{by}\ \text{noetherian recursion on}\ i\ \mathbf{where}$$
$$\mathsf{Head}: \quad |\,\mathsf{Stream}(i,a) \vdash a$$
$$\mathsf{Tail}: \quad |\,\mathsf{Stream}(i,a) \vdash_{j<i} \mathsf{Stream}(j,a)$$

Recursive co-data types follow the dual pattern as data types, with finitely built observations and values given by case analysis on their observations. For $\mathsf{Stream}(N,A)$, we can always ask for the Head of the stream if we have some use for an input of type A, and we can ask for its tail if we can use an input of type $\mathsf{Stream}(M,A)$, for some smaller index $M < N$:

$$\frac{\Gamma|e : A \vdash_\Theta \Delta}{\Gamma|\,\mathsf{Head}[e] : \mathsf{Stream}(N,A) \vdash_\Theta \Delta}\ \mathsf{Stream}L\,\mathsf{Head}$$

$$\frac{\Theta \vdash M < N \quad \Gamma|e : \mathsf{Stream}(M,A) \vdash_\Theta \Delta}{\Gamma|\,\mathsf{Tail}^M[e] : \mathsf{Stream}(N,A) \vdash_\Theta \Delta}\ \mathsf{Stream}L\,\mathsf{Tail}$$

Whereas a $\mathsf{Stream}(N,A)$ value is given by pattern-matching on these two possible observations:

$$\frac{c : (\Gamma \vdash_\Theta \alpha : A, \Delta) \quad c' : (\Gamma \vdash_{\Theta, j<N} \beta : \mathsf{Stream}(j,A), \Delta)}{\Gamma|\mu(\mathsf{Head}[\alpha].c|\mathsf{Tail}^{j<N}[\beta].c') : \mathsf{Stream}(N,A) \vdash_\Theta \Delta}\ \mathsf{Stream}R$$

As before, to write looping programs over recursive types with bounded indices, we use an appropriate recursion scheme for abstracting over the type index. The proof principle for noetherian induction by a well-founded relation $<$ on a set of ordinals $\mathbb{O}$ is:

$$(\forall j : \mathbb{O}.(\forall i < j.P[i]) \to P[j]) \to (\forall i : \mathbb{O}.P[i])$$

which can be made more uniform by introducing an upper-bound to the quantifier in the conclusion as well as in the hypothesis:

$$(\forall j < n.(\forall i < j.P[i]) \to P[j]) \to (\forall i < n. \to P[i])$$

Likewise, a disproof of this argument is again a witness of a counter-example within the chosen bound. We can then translate these principles into inference rules in the sequent calculus, where we represent this new recursion scheme by a co-data type Ascend:

$$\frac{\mathsf{Ascend}(j, A) \vdash_{j<N} A\ j}{\vdash \mathsf{Ascend}(N, A)} \qquad \frac{\vdash M < N \quad A\ M \vdash}{\mathsf{Ascend}(N, A) \vdash}$$

Note that we will write $\mathsf{Ascend}\, i < N.A$ as shorthand for the type $\mathsf{Ascend}(N, \lambda i : \mathsf{Ord}.A)$. We use a similar reading of these rules as a basis for noetherian recursion as we did for primitive recursion. A refutation is still a specific counter-example, so it is represented as a constructed co-term, whereas a proof is a process so is given as a term defined by matching on its observation. Thus, we declare Ascend as a co-data type of the form:

$$\begin{aligned}&\mathbf{codata}\ \mathsf{Ascend}(i : \mathsf{Ord}, a : \mathsf{Ord} \to \star)\ \mathbf{where}\\ &\quad \mathsf{Rise} : \quad \mid \mathsf{Ascend}(i, a) \vdash_{j<i} a\ j\end{aligned}$$

Again, the general mechanism for co-data types tells us how to construct the counter-example with Rise, and destruct it by simple case analysis. The recursive form of case analysis is given manually as the term $\mu(\mathsf{Rise}^{j<N}[\alpha](x).c)$, where x in the pattern is a self-referential variable standing in for the term itself. The typing rule for this recursive case analysis restricts access to itself by making the type of the self-referential variable have a smaller upper bound:

$$\frac{c : (\Gamma, x : \mathsf{Ascend}(j, A) \vdash_{\Theta, j<N} \alpha : A\ j, \Delta)}{\Gamma \vdash_\Theta \mu(\mathsf{Rise}^{j<N}[\alpha](x).c) : \mathsf{Ascend}(N, A) | \Delta}$$

In essence, the terms of type $\mathsf{Ascend}\, i < N.A$ describe a process which is capable of producing $A\{M/i\}$ for any $M < N$ by leaps and bounds: an output of type $A\{M/i\}$ is built up by repeating the same process whenever it is necessary to ascending to an index under M. In contrast, and similar to primitive recursion, co-terms of type $\mathsf{Ascend}\, i < N.A$ hide the chosen index, forcing their input to work for any index.

As always, the symmetry of sequents points us to the dual formulation of noetherian recursion in programs. Specifically, we get the dual data type, named $\mathsf{Descend}$, with the following data declaration and additional typing rule for recursive case analysis:

$$\mathbf{data}\,\mathsf{Descend}(i : \mathsf{Ord}, a : \mathsf{Ord} \to \star)\,\mathbf{where}$$
$$\mathsf{Fall} : \quad a\ j \vdash_{j<i} \mathsf{Descend}(i, a)|$$

$$\frac{c : (\Gamma, x : A\ j \vdash_{\Theta, j<N} \alpha : \mathsf{Descend}(j, A), \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c] : \mathsf{Descend}(N, A) \vdash_{\Theta} \Delta}$$

Now that the roles are reversed, the terms of $\mathsf{Descend}\, i < N.A$ hide the chosen index M at which they can produce a result of type $A\{M/i\}$. Instead, the co-terms of $\mathsf{Descend}\, i < N.A$ consuming $A\{M/i\}$ for any index $M < N$: an input of type $A\{M/i\}$ is broken down by repeating the same process whenever it is necessary to descend from an index under M.

5. A Parametric Sequent calculus with Recursion

We now flesh out the rest of the system for recursive types and structures for representing recursive programs in the sequent calculus. The core rules for kinding and sorting, which accounts for both forms of type-level indices, are given in Figure 5. The rules for the inequality of Ord, $M < N$, are enough to derive expected facts like $\vdash 4 < 6$, but not so strong that they force us to consider Ord types above ∞. Specifically, the requirement that every Ord has a larger successor, $M < M + 1$, *only* when there is an upper bound already established, $M < N$, prevents us from introducing $\infty < \infty + 1$. Additionally, we have two sorts of kinds, those

of *erasable* types, $\square$, and *non-erasable* types, $\blacksquare$. Types (of kind $\star$) for program-level (co-)values and Ord indices are erasable, because they cannot influence the behavior of a program, whereas the Ix indices are used to drive primitive recursion, and cannot be erased. Thus, this sorting system categorizes the distinction between erasable and non-erasable type annotations found in programs.

Before admitting a user-defined (co-)data type into the system, we need to check that its declaration actually denotes a meaningful type. For the non-recursive (co-)data declarations, like those in Figure 3, this well-formedness check just confirms that the sequent associated to each constructor K_i or H_i is well-formed, given by a derivation of $(\overrightarrow{B_i} \vdash_{\overrightarrow{a:k},\overrightarrow{d_i:l_i}} \overrightarrow{C_i})\,\mathbf{seq}$ from Figure 5. When checking for well-formedness of (co-)data types defined by primitive induction on $i : \mathsf{Ix}$, as with the general form

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{data}\,\mathsf{F}(i : \mathsf{Ix}, \overrightarrow{a : k})\,\mathbf{by}\ \text{primitive recursion on } i} \\
\mathbf{where}\, i = 0 & \mathsf{K}_1 : \quad \overrightarrow{B_1} \vdash_{\overrightarrow{d_1:l_1}} \mathsf{F}(0, \vec{a}) | \overrightarrow{C_1} \qquad \ldots \\
\mathbf{where}\, i = j + 1 & \mathsf{K}'_1 : \quad \overrightarrow{B'_1} \vdash_{\overrightarrow{d'_1:l'_1}} \mathsf{F}(j+1, \vec{a}) | \overrightarrow{C'_1} \quad \ldots
\end{array}$$

the $i = 0$ case proceeds by checking that the sequents are well-formed for each constructor $K_1 \ldots$ without referencing i, $(\overrightarrow{B_1} \vdash_{\overrightarrow{a:k},\overrightarrow{d_1:l_1}} \overrightarrow{C_1})\,\mathbf{seq}$, and in the $i = j + 1$ case we check each $(\overrightarrow{B'_1} \vdash_{j:\mathsf{Ix},\overrightarrow{a:k},\overrightarrow{d'_1:l'_1}} \overrightarrow{C'_1})\,\mathbf{seq}$ with the extra rule

$$\frac{\overrightarrow{\Theta, j : \mathsf{Ix}, \Theta' \vdash A : k}}{\Theta, j : \mathsf{Ix}, \Theta' \vdash \mathsf{F}(j, \vec{A}) : \star}$$

Intuitively, in the $i = j + 1$ case the sequents for the constructors may additionally refer to smaller instances $\mathsf{F}(j, \vec{A})$ of the type being

defined. If the declaration is well-formed, we add the typing rules for F similarly to a non-recursive (co-)data type. The difference is that the constructors for the $i = 0$ and $i = j + 1$ case build a structure of type $\mathsf{F}(0, \overrightarrow{A})$ and $\mathsf{F}(M + 1, \overrightarrow{A})$ with M substituted for j, respectively. Additionally, there are two case abstractions: one of type $\mathsf{F}(0, \overrightarrow{A})$ that only handles constructors of the $i = 0$ case, and one of type $\mathsf{F}(M + 1, \overrightarrow{A})$ that only handles constructors of the $i = j + 1$ case. Similarly, when checking for well-formedness of (co-)data types $\mathsf{F}(i : \mathsf{Ord}, \overrightarrow{a : k})$ defined by noetherian induction on $i : \mathsf{Ord}$, we get to assume the type is defined for smaller indices:

$$\frac{\Theta, i : \mathsf{Ord}, \Theta' \vdash M < i \quad \overrightarrow{\Theta, i : \mathsf{Ord}, \Theta' \vdash A : k}}{\Theta, i : \mathsf{Ord}, \Theta' \vdash \mathsf{F}(M, \overrightarrow{A}) : \star}$$

Intuitively, the sequents for the constructors may refer to $\mathsf{F}(M, \overrightarrow{A})$, so long as they introduce quantified type variables $\overrightarrow{d : l}$ such that $\overrightarrow{a : k}, \overrightarrow{d : l} \vdash M < i$. Other than this, the typing rules for structures and case statements are exactly the same as for non-recursive (co-)data types.

We also give the rewriting theory for the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus in Figure 6, which is parameterized by the strategy $\mathcal{S}$. Since the classical sequent calculus inherently admits control effects, the result of a program can completely change depending on the strategy—$\langle length \| \mathsf{Rise}^2[\mathsf{Cons}(\mu\delta.\langle 13 \| \alpha\rangle, \mathsf{Nil}), \alpha]\rangle$ results in $\langle 1 \| \alpha\rangle$ under call-by-name evaluation and $\langle 13 \| \alpha\rangle$ under call-by-value—so that the parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus is actually a family of related but different rewriting theories for reasoning about different evaluation strategies, thus enabling strategy-independent reasoning. The choice of strategy is given as the syntactic notions of *value* and *co-value*: $\mathcal{S}$

is the subset of terms $V \in Value$ and $E \in CoValue$ which may be substituted for (co-)variables. In other words, the strategy refines the range of significance for (co-)variables by limiting what they might stand in for, and in this way it resolves the conflict between both the μ- and $\tilde{\mu}$-abstractions [6]. For example, the strategies for call-by-value and call-by-name evaluation are shown in Figure 8, and a strategy respresenting call-by-need evaluation is representable this way as well [8].

The reduction rules are derived from the core theory of substitution in $\mu\tilde{\mu}_{\mathcal{S}}$ (the top rules of Figure 6), plus rules derived from generic β and η principles for every (co-)data type. Of note are the ς rules, first appearing in Wadler's dual calculus [20], and which we derive from the $\beta\eta$ principles for any (co-)data type [8]. The general lifting rules for (co-)data types are described by the lifting contexts

$$\frac{}{\Theta \vdash 0 : \mathsf{Ix}} \qquad \frac{\Theta \vdash M : \mathsf{Ix}}{\Theta \vdash M+1 : \mathsf{Ix}} \qquad \frac{}{\Theta \vdash 0 < \infty} \qquad \frac{\Theta \vdash M < \infty}{\Theta \vdash M+1 < \infty} \qquad \frac{\Theta \vdash M < N}{\Theta \vdash M < M+1} \qquad \frac{\Theta \vdash M < N \quad \Theta \vdash N < N'}{\Theta \vdash M < N'}$$

$$\frac{a : k \notin \Theta'}{\Theta, a : k, \Theta' \vdash a : k} \qquad \frac{\Theta, a : k_1 \vdash B : k_2 \quad \Theta \vdash k_2 : \Box}{\Theta \vdash \lambda a : k.B : k_1 \to k_2} \qquad \frac{\Theta \vdash A : k_1 \to k_2 \quad \Theta \vdash B : k_1}{\Theta \vdash A\ B : k_2} \qquad \frac{\Theta \vdash M < N \quad \Theta \vdash N : \mathsf{Ord}}{\Theta \vdash M : \mathsf{Ord}} \qquad \frac{}{\Theta \vdash \infty : \mathsf{Ord}}$$

$$\frac{\Theta \vdash k : \Box}{\Theta \vdash k : \blacksquare} \qquad \frac{\Theta \vdash k_1 : \blacksquare \quad \Theta \vdash k_2 : \Box}{\Theta \vdash k_1 \to k_2 : \Box} \qquad \frac{}{\Theta \vdash \star : \Box} \qquad \frac{}{\Theta \vdash \mathsf{Ix} : \blacksquare} \qquad \frac{}{\Theta \vdash \mathsf{Ord} : \Box} \qquad \frac{\Theta \vdash N : \mathsf{Ord}}{\Theta \vdash (< N) : \Box}$$

$$\frac{}{(\vdash)\ \mathbf{seq}} \qquad \frac{\Theta \vdash A : \star \quad (\Gamma \vdash_\Theta \Delta)\ \mathbf{seq}}{(\Gamma, x : A \vdash_\Theta \Delta)\ \mathbf{seq}} \qquad \frac{\Theta \vdash A : \star \quad (\Gamma \vdash_\Theta \alpha : A, \Delta)\ \mathbf{seq}}{(\Gamma \vdash_\Theta \alpha : A, \Delta)\ \mathbf{seq}} \qquad \frac{\Theta \vdash k : \blacksquare \quad (\vdash_\Theta)\ \mathbf{seq}}{(\vdash_{\Theta, a:k})\ \mathbf{seq}}$$

Figure 5. Kinding, sorting, and well-formed typing sequents.

$$\langle \mu\alpha.c \| E \rangle \to_{\mu_E} c\{E/\alpha\} \qquad \langle V \| \tilde{\mu}x.c \rangle \to_{\tilde{\mu}_V} c\{V/x\} \qquad \mu\alpha.\langle v \| \alpha \rangle \to_{\eta_\mu} v \qquad \tilde{\mu}x.\langle x \| e \rangle \to_{\eta_{\tilde{\mu}}} e$$

$$\langle \mathsf{K}^{\vec{B}}(\vec{E}, \vec{V}) \| \tilde{\mu}[\mathsf{K}^{\overrightarrow{b:k}}(\vec{a}, \vec{x}).c | \ldots] \rangle \to_{\beta^{\mathsf{K}}_{\mathcal{S}}} c\{\overrightarrow{B/b}, \overrightarrow{E/\alpha}, \overrightarrow{V/x}\} \qquad \langle \mu(\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}, \vec{\alpha}].c | \ldots) \| \mathsf{H}^{\vec{B}}[\vec{V}, \vec{E}] \rangle \to_{\beta^{\mathsf{H}}_{\mathcal{S}}} c\{\overrightarrow{B/b}, \overrightarrow{V/x}, \overrightarrow{E/\alpha}\}$$

$$C^{\mathsf{K}}_\varsigma ::= \mathsf{K}^{\vec{B}}(\vec{E}, \Box, \vec{e}, \vec{v}) \mid \mathsf{K}^{\vec{B}}(\vec{E}, \vec{V}, \Box, \vec{v}) \qquad C^{\mathsf{H}}_\varsigma ::= \mathsf{H}^{\vec{B}}[\vec{V}, \Box, \vec{v}, \vec{e}] \mid \mathsf{H}^{\vec{B}}[\vec{V}, \vec{E}, \Box, \vec{e}]$$

$$C^{\mathsf{K}}_\varsigma[v] \to_{\varsigma^{\mathsf{K}}_{\mathcal{S}}} \mu\alpha.\langle v \| \tilde{\mu}y.\langle C^{\mathsf{K}}_\varsigma[y] \| \alpha \rangle \rangle \qquad C^{\mathsf{H}}_\varsigma[v] \to_{\varsigma^{\mathsf{H}}_{\mathcal{S}}} \tilde{\mu}x.\langle v \| \tilde{\mu}y.\langle x \| C^{\mathsf{H}}_\varsigma[y] \rangle \rangle \qquad \textbf{where } v \notin Value$$

$$C^{\mathsf{K}}_\varsigma[e] \to_{\varsigma^{\mathsf{K}}_{\mathcal{S}}} \mu\alpha.\langle \mu\beta.\langle C^{\mathsf{K}}_\varsigma[\beta] \| \alpha \rangle \| e \rangle \qquad C^{\mathsf{H}}_\varsigma[e] \to_{\varsigma^{\mathsf{H}}_{\mathcal{S}}} \tilde{\mu}x.\langle \mu\beta.\langle x \| C^{\mathsf{H}}_\varsigma[\beta] \rangle \| e \rangle \qquad \textbf{where } e \notin CoValue$$

Figure 6. Parametric rewriting theory for $\mu\tilde{\mu}_{\mathcal{S}}$.

$$\mu(\mathsf{Rise}^{j<N}[\alpha](x).c) \to \mu(\mathsf{Rise}^{i<N}[\alpha].c\{i/j, \mu(\mathsf{Rise}^{j<i}[\alpha](x).c)/x\}) \quad \tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c] \to \tilde{\mu}[\mathsf{Fall}^{i<N}(x).c\{i/j, \tilde{\mu}[\mathsf{Fall}^{j<i}(x)[\alpha].c]\}]$$

$$\langle V \| \mathsf{Up}^0[E] \rangle \to c_0\{E/\alpha\} \quad \langle V \| \mathsf{Up}^{M+1}[E] \rangle \to \langle \mu\beta.\langle V \| \mathsf{Up}^M[\beta] \rangle \| \tilde{\mu}x.c_1\{M/j, E/\alpha\} \rangle \quad \textbf{where } V = \mu(\mathsf{Up}^0[\alpha].c_0 | \mathsf{Up}^{j+1}[\alpha](x).c_1)$$

$$\langle \mathsf{Down}^0(V) \| E \rangle \to c_0\{V/x\} \quad \langle \mathsf{Down}^{M+1}(V) \| E \rangle \to \langle \mu\alpha.c_1\{M/j, V/x\} \| \tilde{\mu}y.\langle \mathsf{Down}^M(y) \| E \rangle \rangle \quad \textbf{where } E = \tilde{\mu}[\mathsf{Down}^0.c_0 | \mathsf{Down}^{j+1}(x)[\alpha].c_1]$$

Figure 7. Rewriting theory for recursion in $\mu\tilde{\mu}_{\mathcal{S}}$.

$$V \in Value_{\mathcal{V}} ::= x \mid \mathsf{K}(\vec{e}, \vec{V}) \mid \mu(\mathsf{H}^{\overline{b;k}}[\vec{x}, \vec{\alpha}].c[\dots) \qquad V \in Value_{\mathcal{N}} ::= v$$
$$E \in CoValue_{\mathcal{V}} ::= e \qquad E \in CoValue_{\mathcal{N}} ::= \alpha \mid \tilde{\mu}[\mathsf{K}^{\overline{b;k}}(\vec{\alpha}, \vec{x}).c[\dots] \mid \mathsf{H}[\vec{v}, \vec{E}]$$

Figure 8. The call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) strategies.

$C^{\mathsf{K}}_{\varsigma}$ and $C^{\mathsf{H}}_{\varsigma}$ for each (co-)constructor, and their role is to bring work to the top of a command, so that it can take over.

To implement recursion in the rewriting theory, we use the additional rules shown in Figure 7. The recursive case abstractions for Ascend and Descend are simplified by "unrolling" their loop: the recursive abstraction reduces to a non-recursive one by substituting itself inward—with a tighter upper bound—for the recursive variable. Intuitively, this index-unaware loop unrolling is possible because the actual chosen index *doesn't matter*, the loop must do the same thing each time around regardless of the value of the index. Contrarily, the Inflate and Deflate recursors operate strictly stepwise: they will always go from step 10 to 9 and so on to 0. The indices used in the constructor really do matter, because they can influence the behavior of the program. This fact forces us to "unroll" the loop while pattern-matching on structures like $\mathsf{Up}^{M+1}[E]$ in tandem, unlike noetherian recursion where the two steps can be performed independently.

We also have a restriction on reduction, following the motto "don't touch unreachable branches," to ensure strong normalization. Reduction may normally occur in all contexts, except for reduction inside a case abstraction which requires an additional *reachability* caveat about the kinds of quantified types introduced by pattern matching. This restriction prevents unnecessary infinite unrolling that would otherwise occur in simple commands like $\langle length \| \mathsf{Rise}^{i}[\alpha] \rangle$. Intuitively, the reachability caveat prevents re-

duction inside a case abstraction which introduces type variables that might be *impossible* to instantiate, like $i < 0$ or $j < i$. The reductions following the reachability caveat are defined as:

$$\frac{c \to c' \quad b : \overrightarrow{k \to}(< N) \in \Theta \implies N = \infty \vee N = M + 1}{\mu(\mathsf{H}^{\Theta}[\vec{x}, \vec{\alpha}].c| \ldots) \to \mu(\mathsf{H}^{\Theta}[\vec{x}, \vec{\alpha}].c'| \ldots)}$$

$$\frac{c \to c' \quad b : \overrightarrow{k \to}(< N) \in \Theta \implies N = \infty \vee N = M + 1}{\tilde{\mu}[\mathsf{K}^{\Theta}(\vec{\alpha}, \vec{x}).c| \ldots] \to \tilde{\mu}[\mathsf{K}^{\Theta}(\vec{\alpha}, \vec{x}).c'| \ldots]}$$

We also define the type erasure operation on programs, $Erase(c)$, which removes all types from constructors and patterns in c with an erasable kind, while leaving intact the unerasable $\mathsf{I}\times$ types. The corresponding type-erased $\mu\tilde{\mu}_S$-calculus is the same, except that the reachability caveat is enhanced to never reduce inside case abstractions. This means that every step of a type-erased command is justified by the same step in the original command, so that type-erasure cannot introduce infinite loops.

To demonstrate strong normalization, we use a combination of techniques. Giving a semantics for types based on Barbanera and

Berardi's symmetric candidates [4], a variant of Girard's reducibility candidates [9], as well as Krivine's classical realizability [13], an application of bi-orthogonality, establishes strong normalization of well-typed commands. Of note, the strong normalization of well-typed commands is parameterized by a strategy, which is enabled by the parameterization of the rewriting theory. Thus, instead of showing strong normalization of these related rewriting theories one-by-one, we establish strong normalization in one fell swoop by characterizing the properties of a strategy that are important for strong normalization. First, the chosen strategy $\mathcal{S}$ must be *stable*, meaning that (co-)values are closed under reduction and substitution, and non-(co-)values are closed under substitution and ς reduction. Second, $\mathcal{S}$ must be *focalizing*, meaning that (co-)variables, structures built from other (co-)values, and case abstractions must all be (co-)values. The latter criteria comes from focalization in logic [7, 16, 21]—each criterion comes from an inference rule for typing a (co-)value in focus.

Theorem 1. *For any stable and focalizing strategy $\mathcal{S}$, if $c : \Gamma \vdash_\Theta \Delta$ and $(\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}$, then c is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus. Furthermore, $Erase(c)$ is strongly normalizing in the type-erased $\mu\tilde{\mu}_{\mathcal{S}}$-calculus.*

Note that the call-by-name, call-by-value, and call-by-need strategies from [8] are all stable and focalizing, so that as a corollary, we achieve strong normalization for these particular instances of the parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus. Furthermore, the "maximally" non-deterministic strategy—attained by letting every term be a value and every co-term be a co-value—is also stable and focalizing, which gives another account of strong normalization for the symmetric λ-calculus [14] as a corollary.

5.1 Encoding Recursive Programs via Structures

To see how to encode basic recursive definitions into the sequent calculus using the primitive and noetherian recursion principles, we revisit the previous examples from Section 2. We will see how the intuitive argument for termination can be represented using the type indices for recursion in various ways.

Example 4. Recall the $length$ function from Example 1, as written in sequent-style. As we saw, we could internalize the definition for $length$ into a recursively-defined case abstraction that describes each possible behavior. Using the noetherian recursion principle in the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, we can give a more precise and non-recursive definition for $length$:

$$
\begin{aligned}
&length : \forall a : \star.\, \mathsf{Ascend}\, i < \infty.\, \mathsf{List}(i, a) \to \mathsf{Nat}(i) \\
&length = \mu(\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[\mathsf{Nil} \cdot \gamma](r)].\langle \mathsf{Z} \| \gamma \rangle \\
&\qquad\qquad |\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[\mathsf{Cons}^{j<i}(x, xs) \cdot \gamma](r)]. \\
&\qquad\qquad\quad \langle r \| \mathsf{Rise}^j[xs \cdot \tilde{\mu} y.\langle \mathsf{S}^j(y) \| \gamma \rangle] \rangle)
\end{aligned}
$$

The difference is that the polymorphic nature of the $length$ function is made explicit in System F-style, and the recursion part of the function has been made internal through the Ascend co-data type. Going further, we may unravel the deep patterns into shallow case analysis, giving annotations on the introduction of every co-variable:

$$
\begin{aligned}
&length = \mu(\mathsf{Spec}^a[\alpha^{\mathsf{Ascend}\, i<\infty.\, \mathsf{List}(i,a)\to\mathsf{Nat}(i)}]. \\
&\langle \mu(\mathsf{Rise}^{i<\infty}[\beta^{\mathsf{List}(i,a)\to\mathsf{Nat}(i)}](r^{\mathsf{Ascend}\, j<i.\, \mathsf{List}(j,a)\to\mathsf{Nat}(j)}). \\
&\quad \langle \mu([xs^{\mathsf{List}(i,a)} \cdot \gamma^{\mathsf{Nat}(i)}].\, \langle xs| \\
&\qquad |\tilde{\mu}[\mathsf{Nil}.\langle \mathsf{Z} \| \gamma \rangle \\
&\qquad\quad |\mathsf{Cons}^{j<i}(x^a, ys^{\mathsf{List}(j,a)}).\langle r \| \mathsf{Rise}^j[ys \cdot \tilde{\mu} y^{\mathsf{Nat}(j)}.\langle \mathsf{S}^j(y) \| \gamma \rangle]\rangle]\rangle)|
\end{aligned}
$$

$|\beta\rangle)|$
$|\alpha\rangle)$

Although quite verbose, this definition spells out all the information we need to verify that $length$ is well-typed and well-founded: no guessing required. Furthermore, this core definition of $length$ is entirely in terms of shallow case analysis, making reduction straightforward to implement. Since the correctness of programs is ensured for this core form, which can be elaborated from the deep pattern-matching definition mechanically, we will favor the more concise pattern-matching forms for simplicity in the remaining examples. *End example* 4.

Example 5. Recall the $countUp$ function from Example 2. When we attempt to encode this function into the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, we run into a new problem: the indices for the given number and the resulting stream do not line up since one grows while the other shrinks. To get around this issue, we mask the index of the given natural number using the dual form of noetherian recursion, and say that $\mathsf{ANat} = \mathsf{Descend}\, i < \infty.\, \mathsf{Nat}(i)$. We can then describe $countUp$ as a function from ANat to a $\mathsf{Stream}(i, \mathsf{ANat})$ by noetherian recursion on i:

$$countUp : \mathsf{Ascend}\, i < \infty.\, \mathsf{ANat} \to \mathsf{Stream}(i, \mathsf{ANat})$$
$$countUp = \mu(\mathsf{Rise}^{i<\infty}[x \cdot \mathsf{Head}[\alpha]](r).\langle x \| \alpha \rangle$$
$$|\mathsf{Rise}^{i<\infty}[\mathsf{Fall}^{j<i}(x) \cdot \mathsf{Tail}^{k<i}[\beta]](r).$$
$$\langle r \| \mathsf{Rise}^{k}[\mathsf{Fall}^{j+1}(\mathsf{S}^{j}(x)) \cdot \beta] \rangle)$$

End example 5.

Example 6. The previous example shows how infinite streams

may be modeled by co-data. However, recall the other approach to infinite objects mentioned in Example 3. Unfortunately, an infinitely constructed list like $zeroes$ would be impossible to define in terms of noetherian recursion: in order to use the recursive argument, we need to come up with an index smaller than the one we are given, but since lists are a data type their observations are inscrutable and we have no place to look for one. As it turns out, though, primitive recursion is set up in such a way that we can make headway. Defining infinite lists to be $\mathsf{InfList}(a) = \mathsf{Inflate}\, i : \mathsf{Ix}\,.\,\mathsf{IxList}(i, a)$, we can encode $zeroes$ as:

$$
\begin{aligned}
&zeroes : \mathsf{InfList}(\mathsf{Nat}(0)) \\
&zeroes = \mu(\mathsf{Up}^0[\alpha^{\mathsf{IxList}(0,\mathsf{Nat})}].\langle \mathsf{Nil} \| \alpha \rangle \\
&\qquad\qquad |\mathsf{Up}^{i+1}[\alpha^{\mathsf{IxList}(i+1,\mathsf{Nat})}](r^{\mathsf{IxList}(i,\mathsf{Nat})}).\langle \mathsf{Cons}(\mathsf{Z}, r) \| \alpha \rangle)
\end{aligned}
$$

Even more, we can define the concatenation of infinitely constructed lists in terms of primitive recursion as well. We give a wrapper, cat, that matches the indices of the incoming and outgoing list structure, and a worker, cat', that performs the actual recursion:

$$
\begin{aligned}
&cat : \forall a : \star.\, \mathsf{InfList}(a) \to \mathsf{InfList}(a) \to \mathsf{InfList}(a) \\
&cat = \langle \mu(\mathsf{Spec}^a[xs \cdot ys \cdot \mathsf{Up}^i[\alpha]]. \\
&\qquad\qquad \langle xs \| \mathsf{Up}^i[\tilde{\mu} zs.\langle cat' \| \mathsf{Up}^i[zs \cdot ys \cdot \alpha] \rangle] \rangle)
\end{aligned}
$$

$$
\begin{aligned}
&cat' : \forall a : \star.\, \mathsf{Inflate}\, i : \mathsf{Ix}\,.\,\mathsf{IxList}(i, a) \to \mathsf{InfList}(a) \to \mathsf{IxList}(i, a) \\
&cat' = \mu(\mathsf{Spec}^a[\mathsf{Up}^0[\mathsf{Nil} \cdot ys \cdot \alpha]].\langle \mathsf{Nil} \| \alpha \rangle \\
&\qquad |\mathsf{Spec}^a[\mathsf{Up}^{i+1}[\mathsf{Nil} \cdot ys \cdot \alpha](r)].\langle ys \| \mathsf{Up}^{i+1}[\alpha] \rangle \\
&\qquad |\mathsf{Spec}^a[\mathsf{Up}^{i+1}[\mathsf{Cons}(x, xs) \cdot ys \cdot \alpha](r)]. \\
&\qquad\qquad \langle \mathsf{Cons}(x, \mu\beta.\langle r \| xs \cdot ys \cdot \beta \rangle) \| \alpha \rangle)
\end{aligned}
$$

If we would like to stick with the "finite objects are data, infinite objects are co-data" mantra, we can write a similar concatenation function over possibly terminating streams:

$$\begin{aligned}&\mathbf{codata}\ \mathsf{StopStream}(i < \infty, a : \star)\ \mathbf{where}\\ &\quad \mathsf{Head}: \quad \mid \mathsf{StopStream}(i, a) \vdash a\\ &\quad \mathsf{Tail}: \quad \mid \mathsf{StopStream}(i, a) \vdash_{j<i} 1, \mathsf{StopStream}(j, a)\end{aligned}$$

A $\mathsf{StopStream}(i, a)$ object is like a $\mathsf{Stream}(i, a)$ object except that asking for its Tail might fail and return the unit value instead, so it represents an infinite or finite stream of one or more values. This co-data type makes essential use of multiple conclusions, which

are only available in a language for classical logic. We can now write a general recursive definition of concatenation in terms of the StopStream co-data type:

$$\langle cat \| xs \cdot ys \cdot \mathsf{Head}[\alpha] \rangle = \langle xs \| \mathsf{Head}[\alpha] \rangle$$
$$\langle cat \| xs \cdot ys \cdot \mathsf{Tail}[\delta, \beta] \rangle = \langle cat \| \mu\gamma.\langle xs \| \mathsf{Tail}[\tilde{\mu}[().\langle ys \| \beta \rangle], \gamma] \rangle \cdot ys \cdot \beta \rangle$$

This function encodes into a similar pair of worker-wrapper values, where now a possibly infinite list is represented as a terminating stream $\mathsf{InfList}(a) = \mathsf{Ascend}\, i < \infty.\, \mathsf{StopStream}(i, a)$:

$$cat' : \forall a : \star.\, \mathsf{Ascend}\, i < \infty.\ \mathsf{StopStream}(i, a) \to \mathsf{InfList}(a) \to \mathsf{StopStream}(i, a)$$
$$cat' = \mu(\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[xs \cdot ys \cdot \mathsf{Head}[\alpha]](r)).\langle xs \| \alpha \rangle \mid \mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[xs \cdot ys \cdot \mathsf{Tail}^{j<i}[\delta, \beta]](r)).\ \langle r \| \mathsf{Rise}^j[\mu\gamma.\langle xs \| \mathsf{Tail}^j[\tilde{\mu}[().\langle ys \| \mathsf{Rise}^j[\beta] \rangle], \gamma] \rangle \cdot ys \cdot \beta] \rangle)$$

End example 6.

Intermezzo 1. It is worth pointing out why our encoding for "infinite" data structures, like *zeroes*, avoids the problem underlying the lack of subject reduction for co-induction in Coq [18]. Intuitively, the root of the problem is that Coq's co-inductive objects are non-extensional, since the interaction between case analysis and the co-fixpoint operator effectively allows these objects to notice if they are being discriminated or not. In contrast, we take the extensional view that the presence or absence of case analysis, in *all* of its various forms, is unobservable. To ensure strong normalization, the basic observation is instead a specific message that advertises to the object exactly how deep it would like to go, thus restoring extensionality and putting a limit on unfolding. *End intermezzo* 1.

Example 7. We now consider an example with a more complex

recursive argument that makes non-trivial use of lexicographic induction. The Ackermann function can be written as:

$$\begin{aligned}
\langle ack \| \mathsf{Z} \cdot y \cdot \alpha \rangle &= \langle \mathsf{S}(y) \| \alpha \rangle \\
\langle ack \| \mathsf{S}(x) \cdot \mathsf{Z} \cdot \alpha \rangle &= \langle ack \| x \cdot \mathsf{S}(\mathsf{Z}) \cdot \alpha \rangle \\
\langle ack \| \mathsf{S}(x) \cdot \mathsf{S}(y) \cdot \alpha \rangle &= \langle ack \| \mathsf{S}(x) \cdot y \cdot \tilde{\mu} z. \langle ack \| x \cdot z \cdot \alpha \rangle \rangle
\end{aligned}$$

The fact that this function terminates follows by lexicographic induction on both arguments: to every recursive call of ack, either the first number decreases, or the first number stays the same and the second number decreases. This argument can be encoded into the basic noetherian recursion principle we already have by nesting it twice:

$$\begin{aligned}
ack : \mathsf{Ascend}\, i < \infty. \mathsf{Ascend}\, j < \infty. \mathsf{Nat}(i) \to \mathsf{Nat}(j) \to \mathsf{ANat} \\
ack = \mu(\mathsf{Rise}^{i<\infty}[\mathsf{Rise}^{j<\infty}[\mathsf{Z} \cdot y \cdot \alpha](r_2)](r_1). \langle \mathsf{Fall}^{j+1}(\mathsf{S}^j(y)) \| \alpha \rangle \\
\quad |\mathsf{Rise}^{i<\infty}[\mathsf{Rise}^{j<\infty}[\mathsf{S}^{i'<i}(x) \cdot \mathsf{Z} \cdot \alpha](r_2)](r_1). \\
\quad\quad \langle r_1 \| \mathsf{Rise}^{i'}[\mathsf{Rise}^{1}[x \cdot \mathsf{S}^0(\mathsf{Z}) \cdot \alpha]] \rangle \\
\quad |\mathsf{Rise}^{i<\infty}[\mathsf{Rise}^{j<\infty}[\mathsf{S}^{i'<i}(x) \cdot \mathsf{S}^{j'<j}(y) \cdot \alpha](r_2)](r_1). \\
\quad\quad \langle r_2 \| \mathsf{Rise}^{j'}[\mathsf{S}^{i'}(x) \cdot y \cdot \tilde{\mu}[\mathsf{Fall}^{k<\infty}(z). \\
\quad\quad\quad\quad \langle r_1 \| \mathsf{Rise}^{i'}[\mathsf{Rise}^{k}[x \cdot z \cdot \alpha]] \rangle]]]] \rangle)
\end{aligned}$$

Essentially, we get two recursive arguments from nesting Ascend:

$$\begin{aligned}
r_1 : \mathsf{Ascend}\, i' < i. \mathsf{Ascend}\, j < \infty. \mathsf{Nat}(i') \to \mathsf{Nat}(j) \to \mathsf{ANat} \\
r_2 : \mathsf{Ascend}\, j' < j. \mathsf{Nat}(i) \to \mathsf{Nat}(j') \to \mathsf{ANat}
\end{aligned}$$

The first recursive path r_1 can be taken whenever the first argument is smaller, in which case the second argument is arbitrary. The second recursive path r_2 can be taken whenever the second argument

is smaller and the first argument has the same index (the i in the type of r_2 matches the index of the original first argument to ack). Again, we find that the dual form noetherian recursion, Descend, is useful for masking the index of the output from ack. Furthermore, it is interesting to note that in the third case of ack, we must explicitly destruct the Descend-ed result from ack before performing the second recursive call. In practical terms, this forces the nested recursive call of the Ackermann function to be strict, even in a lazy language. *End example* 7.

6. Natural Deduction and Effect-Free Programs

So far, we have looked at a calculus for representing recursion via structures in sequent style, which corresponds to a classical logic and thus includes control effects [11]. Let's now briefly shift focus, and see how the intuition we gained from the sequent calculus can be reflected back into a more traditional core calculus for expressing functional-style recursion. The goal here is to see how the recursive principles we have developed in the sequent setting can be incorporated into a λ-calculus based language: using the traditional connection between natural deduction and the sequent calculus, we show how to translate our primitive and noetherian recursive types and programs into natural deduction style. In essence, we will consider a functional calculus based on an effect-free subset of the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus corresponding to *minimal* logic.

Essentially, the minimal restriction of the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus for representing effect-free functional programs follows a single mantra, based on the connection between classical and minimal logics: there is always *exactly* one conclusion. In the type system, this means that the sequent for typing terms has the more restricted

form $\Gamma \vdash_\Theta v : A$, where the active type on the right is no longer ambiguous and does not need to be distinguished with $|$, as is more traditional for functional languages. Notice that this limitation on the form of sequents impacts which type constructors we can express. For example, common sums and products, declared as

$$\begin{array}{rl} \multicolumn{2}{l}{\mathbf{data}\, a \oplus b\, \mathbf{where}} \\ \mathsf{Left}: & a \vdash a \oplus b| \\ \mathsf{Right}: & b \vdash a \oplus b| \end{array} \qquad \begin{array}{rl} \multicolumn{2}{l}{\mathbf{codata}\, a \,\&\, b\, \mathbf{where}} \\ \mathsf{Fst}: & |a \,\&\, b \vdash a \\ \mathsf{Snd}: & |a \,\&\, b \vdash b \end{array}$$

fit into this restricted typing discipline, because each of their (co-)constructors only ever involves one type to the right of entailment. However, the (co-)data types for representing more exotic connectives like subtraction and linear logic's par

$$\begin{array}{rl} \multicolumn{2}{l}{\mathbf{data}\, a - b\, \mathbf{where}} \\ \mathsf{Pause}: & a \vdash a - b|b \end{array} \qquad \begin{array}{rl} \multicolumn{2}{l}{\mathbf{codata}\, a \parr b\, \mathbf{where}} \\ \mathsf{Split}: & |a \parr b \vdash a, b \end{array}$$

do not fit, because they require placing two types to the right of entailment. In sequent style, this means these *minimal* data types can never contain a co-value, and *minimal* co-data types must always involve exactly one co-value for returning the unique result. In functional style, the data types are exactly the algebraic data types used in functional languages, with the corresponding constructors and case expressions, and the co-data types can be thought of as merging functions with records into a notion of abstract "objects" which compute and return a value when observed. For example, to observe a value of type $a \,\&\, b$, we could access the first component as a record field, $v.\mathsf{Fst}$, and we describe an object of this type by saying how it responds to all possible observations, $\{\mathsf{Fst} \Rightarrow v_1 | \mathsf{Snd} \Rightarrow v_2\}$, with the typing rules:

$$\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B \qquad\qquad \Gamma \vdash v : A \,\&\, B \quad \Gamma \vdash v : A \,\&\, B$$

$$\Gamma \vdash \{\mathsf{Fst} \Rightarrow v_1 | \, \mathsf{Snd} \Rightarrow v_2\} : A \,\&\, B \qquad \Gamma \vdash v.\mathsf{Fst} : A \qquad \Gamma \vdash v.\mathsf{Snd} : B$$

Likewise, the traditional λ-abstractions and type abstractions from System F can be expressed by objects of these form. Specifically, since they are user-definable, minimal co-data types with one constructor, $\mathsf{Call} : a | a \to b \vdash b$ and $\mathsf{Spec} : | \forall a \vdash_{b:\star} a\ b$, the abstractions can be given as syntactic sugar:

$$\lambda x^A.v = \{\mathsf{Call}[x^A] \Rightarrow v\} \qquad \Lambda b^\star.v = \{\mathsf{Spec}^{b:\star} \Rightarrow v\}$$

Thus, these objects also serve as "generalized λ-abstractions" [2] defined by shallow case analysis rather than deep pattern-matching.

The typing rules for recursive structures translated to functional style are shown in Figure 9, and the reduction rules for the calculus

$$\frac{\Gamma \vdash_\Theta v_0 : A\{0/i\} \quad \Gamma, x : A\{j/i\} \vdash_{\Theta, j:\mathsf{Ix}} v_1 : A\{j+1/i\}}{\Gamma \vdash_\Theta \{\mathsf{Up}^0 \Rightarrow v_0 | \, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\} : \mathsf{Inflate}\, i : \mathsf{Ix}.A} \qquad \frac{\Gamma \vdash_\Theta v : \mathsf{Inflate}\, i : \mathsf{Ix}.A \quad \Theta \vdash M : \mathsf{Ix}}{\Gamma \vdash_\Theta v.\,\mathsf{Up}^M : A\{M/i\}}$$

$$\frac{\Gamma \vdash_\Theta v : A\{M/i\} \quad \Theta \vdash M : \mathsf{Ix}}{\Gamma \vdash_\Theta \mathsf{Down}^M(v) : \mathsf{Deflate}\, i : \mathsf{Ix}.A} \qquad \frac{\Gamma \vdash_\Theta v : \mathsf{Deflate}\, i : \mathsf{Ix}.A \quad \Gamma, x : A\{0/i\} \vdash_\Theta v_0 : C \quad \Gamma, x : A\{j+1/i\} \vdash_{\Theta, j:\mathsf{Ix}} v_1 : A\{j/i\}}{\Gamma \vdash_\Theta \mathbf{loop}\, v\, \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 | \, \mathsf{Down}^{j+1}(x) \Rightarrow v_1 : C}$$

$$\frac{\Gamma, x : \mathsf{Ascend}\, i < j.A \vdash_{\Theta, j<N} v : A\{j/i\}}{\Gamma \vdash_\Theta \{\mathsf{Rise}^{j<N}(x) \Rightarrow v\} : \mathsf{Ascend}\, i < N.A} \qquad \frac{\Gamma \vdash_{\Theta, j<N} v : A\{j/i\}}{\Gamma \vdash_\Theta \{\mathsf{Rise}^{j<N} \Rightarrow v\} : \mathsf{Ascend}\, i < N.A} \qquad \frac{\Gamma \vdash_\Theta v : \mathsf{Ascend}\, i < N.A \quad \Theta \vdash M < N}{\Gamma \vdash_\Theta v.\,\mathsf{Rise}^M : A\{M/i\}}$$

Figure 9. Typing primitive and noetherian recursion in natural deduction style.

$$\{\mathsf{H}^{\overrightarrow{b:k}}[\overrightarrow{x}] \Rightarrow v' | \ldots\}.\mathsf{H}^{\overrightarrow{B}}[\overrightarrow{v}] \to v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\} \qquad \mathbf{case}\ \mathsf{K}^{\overrightarrow{B}}(\overrightarrow{v})\ \mathbf{of}\ \mathsf{K}^{\overrightarrow{b:k}}(\overrightarrow{x}) \Rightarrow v' | \ldots \to v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\}$$

$$\{\mathsf{Rise}^{j<N}(x) \Rightarrow v\} \to \{\mathsf{Rise}^{i<N} \Rightarrow v\{i/j, \{\mathsf{Rise}^{j<i}(x) \Rightarrow v\}/x\}\}$$

$$\{\mathsf{Up}^0 \Rightarrow v_0 | \, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^0 \to v_0 \qquad \{\mathsf{Up}^0 \Rightarrow v_0 | \, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^{M+1} \to v_1\{M/j, \{\mathsf{Up}^0 \Rightarrow v_0 | \, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^M/x\}$$

$$\mathbf{loop}\ \mathsf{Down}^0(v)\ \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 | \, \mathsf{Down}^{j+1}(x) \Rightarrow v_1 \to v_0\{v/x\}$$

$$\mathbf{loop}\ \mathsf{Down}^{M+1}(v)\ \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 | \, \mathsf{Down}^{j+1}(x) \Rightarrow v_1 \to \mathbf{loop}\ \mathsf{Down}^M(v_1\{M/j, v/x\})\ \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 | \, \mathsf{Down}^{j+1}(x) \Rightarrow v_1$$

Figure 10. Reduction rules for a natural deduction language with (co-)data types and recursion.

$$x^\flat = x \qquad (\mathsf{K}^{\overrightarrow{B}}(\overrightarrow{v}))^\flat = \mathsf{K}^{\overrightarrow{B}}(\overrightarrow{v^\flat}) \qquad (\mathbf{case}\, v\ \mathbf{of}\ \mathsf{K}^{\overrightarrow{b:k}}(\overrightarrow{x}) \Rightarrow v' | \ldots)^\flat = \mu\alpha.\langle v^\flat \| \tilde{\mu}[\mathsf{K}^{\overrightarrow{b:k}}(\overrightarrow{x}).\langle v'^\flat \| \alpha\rangle | \ldots]\rangle$$

$$(v'.\mathsf{H}^{\overrightarrow{B}}[\overrightarrow{v}])^\flat = \mu\alpha.\langle v'^\flat \| \mathsf{H}^{\overrightarrow{B}}[\overrightarrow{v^\flat}, \alpha]\rangle \quad \{\mathsf{H}^{\overrightarrow{b:k}}[\overrightarrow{x}] \Rightarrow v | \ldots\}^\flat = \mu(\mathsf{H}^{\overrightarrow{b:k}}[\overrightarrow{x}, \alpha].\langle v^\flat \| \alpha\rangle | \ldots) \quad \{\mathsf{Rise}^{j<N}(x) \Rightarrow v\}^\flat = \mu(\mathsf{Rise}^{j<N}[\alpha](x).\langle v^\flat \| \alpha\rangle)$$

$$\{\mathsf{Up}^0 \Rightarrow v_0 | \, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}^\flat = \mu(\mathsf{Up}^0[\alpha].\langle v^\flat \| \alpha\rangle | \mathsf{Up}^{j+1}[\alpha](x).\langle v^\flat \| \alpha\rangle)$$

$$(\mathbf{loop}\, v\ \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 | \, \mathsf{Down}^{j+1}(x) \Rightarrow v_1)^\flat = \mu\alpha.\langle v^\flat \| \tilde{\mu}[\mathsf{Down}^0(x).\langle v_0^\flat \| \alpha\rangle | \mathsf{Down}^{j+1}(x)[\alpha].\langle v_1^\flat \| \alpha\rangle]\rangle$$

Figure 11. Type-preserving translation from a pure, natural deduction language to $\mu\tilde{\mu}_S$.

are shown in Figure 10. Intuitively, the objects of $\mathsf{Inflate}(A)$ are stepwise loops that can return any $A\ N$ by counting up from 0 and using the previous instances of itself, while we can write looping case expressions over values of $\mathsf{Deflate}(A)$ to count down from any $A\ N$ to 0. Similarly, values of $\mathsf{Ascend}(N, A)$ are self-referential objects that always behave the same no matter the number of recursive invocations. Curiously though, the recursive forms for $\mathsf{Descend}(N, A)$ are conspicuously missing from the functional calculus. In essence, the recursive form for $\mathsf{Descend}(N, A)$ is a case expression that introduces a continuation variable for the recursive path out of the expression in addition to the normal return path, effectively requiring a form of subtraction type $C - \mathsf{Descend}(M, A)$ for smaller indices M. So while $\mathsf{Descend}$ can still be used to hide indices, its recursive nature lies outside the pure functional paradigm. This follows the frequent situation where one of four classical principles gets lost in translation to intuitionistic or minimal settings. It occurs with De Morgan laws ($\neg(A \wedge B) \to (\neg A) \vee (\neg B)$ is not intuitionistically valid), the conjunctive and disjunctive connectives of linear logic ($\parr$ requires multiple conclusions so it does not fit the minimal mold), and here as well.

Intuitively, we can think of the values of $\mathsf{Inflate}(A)$ as a dependently typed version of the recursion operator for natural numbers in Gödel's System T [10]. Indeed, we can encode such an operator:

$$rec : \forall a : \mathsf{Ix} \to \star.$$
$$a\ 0 \to (\mathsf{Inflate}\, i : \mathsf{Ix}\, .a\ i \to a\ (i+1)) \to \mathsf{Inflate}\, i : \mathsf{Ix}\, .a\ i$$
$$rec = \lambda a\ x\ f.\{\mathsf{Up}^{0:\mathsf{Ix}} \Rightarrow x |\ \mathsf{Up}^{j+1:\mathsf{Ix}}(r) \Rightarrow f.\, \mathsf{Up}^{j}\ r\}$$

So essentially, we are using the natural number index to drive the

recursion upward to compute some value, where the type of that returned value can depend on the number of steps in the chosen index. In a call-by-name setting, where we choose a maximal set of values so that V can be any term, then the behavior of rec implements the recursor: given that $rec\ a\ x\ f \twoheadrightarrow rec_{a,x,f}$ we have

$$rec_{a,x,f}.\mathsf{Up}^0 \to x \quad rec_{a,x,f}.\mathsf{Up}^{M+1} \to f.\mathsf{Up}^M\ (rec_{a,x,f}.\mathsf{Up}^M)$$

Contraposed, $\mathsf{Deflate}(A)$ implements a dependently-typed, stepwise recursion going the other way. The looping form breaks down a value depending on an arbitrary index N until that index reaches 0, finally returning some value which does *not* depend on the index. For instance, we can sum the values in any vector of numbers, $v : \mathsf{Vec}(N, \mathsf{ANat})$, in accumulator style by looping over the recursive structure $\mathsf{Descend}\, i : \mathsf{Ix}\,.\, \mathsf{ANat} \otimes \mathsf{Vec}(i, \mathsf{ANat})$:[5]

$$\begin{aligned}&\mathbf{loop}\ \mathsf{Down}^N(\mathsf{Fall}^0(\mathsf{Z}), v)\ \mathbf{of}\\ &\quad \mathsf{Down}^0(acc, \mathsf{Nil}) \Rightarrow acc\\ &\quad \mid \mathsf{Down}^{i+1}(acc, \mathsf{Cons}(x, xs)) \Rightarrow (x + acc, xs)\end{aligned}$$

Instead, values of Ascend are useful for representing stronger induction that recurses on deeply nested sub-structures. For example, we can convert a list $x_1, x_2, \ldots, x_n$ into a list of its adjacency pairs $(x_1, x_2), (x_3, x_4), \ldots, (x_{n-1}, x_n)$ by

$$\begin{aligned}pairs\ \mathsf{Nil} &= \mathsf{Nil}\\ pairs\ \mathsf{Cons}(x, ys) &= \mathsf{Nil}\\ pairs\ \mathsf{Cons}(x, \mathsf{Cons}(y, zs)) &= \mathsf{Cons}((x, y), pairs\ zs)\end{aligned}$$

[5] Note, we assume an addition operator $+ : \mathsf{ANat} \to \mathsf{ANat} \to \mathsf{ANat}$.

where we silently drop the final element if the list is odd. The $pairs$ function can be straightforwardly encoded using Ascend as:

$$pairs : \forall a : \star.\, \mathsf{Ascend}\, i < \infty.\, \mathsf{List}(i, a) \to \mathsf{List}(i, a \otimes a)$$

$$\begin{aligned} pairs = \lambda a^{\star}.\{&\mathsf{Rise}^{i<\infty}(r) \Rightarrow \lambda x^{\mathsf{List}(i,a)}.\, \mathbf{case}\, xs\, \mathbf{of} \\ &\mathsf{Nil} \Rightarrow \mathsf{Nil} \\ &\mathsf{Cons}^{j<i}(x^a, ys^{\mathsf{List}(j,a)}) \Rightarrow \mathbf{case}\, ys\, \mathbf{of} \\ &\quad \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ &\quad \mathsf{Cons}^{k<j}(y^a, zs^{\mathsf{List}(k,a)}) \Rightarrow \mathsf{Cons}^{k}((x, y), r.\mathsf{Rise}^{k} zs)\} \end{aligned}$$

Note that the type of the recursive argument r is $\mathsf{Ascend}\, i' < i.\, \mathsf{List}(i', a) \to \mathsf{List}(i', a \otimes a)$. Thus, the recursive self-invocation $r.\mathsf{Rise}^{k} : \mathsf{List}(k, a) \to \mathsf{List}(k, a \otimes a)$ is well-typed, since we learn that $j < i$ and $k < j$ by analyzing the Cons structure of the list and learn that $k < i$ by transitivity.

Finally, note that we can translate this functional calculus into the minimal subset of the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, as shown in Figure 11. This translation is type-preserving, and each of the source reductions maps to at least one reduction in the call-by-name instance of $\mu\tilde{\mu}_{\mathcal{S}}$ [8], $\mu\tilde{\mu}_{\mathcal{N}}$, where the set of values is as large as possible and includes every term. So, because the $\mu\tilde{\mu}_{\mathcal{N}}$-calculus does not allow for well-typed infinite loops, neither does its functional counterpart.

Theorem 2. *If* $\Gamma \vdash_{\Theta} v : A$ *and* $(\Gamma \vdash_{\Theta} \alpha : A)\, \mathbf{seq}$ *are derivable then* v *is strongly normalizing.*

7. Conclusion

Co-induction need not be a second-class citizen compared to induction in programming languages. Dedication to duality provides the key for unlocking co-recursion from recursion as its equal and opposite force. We are able to freely mix inductive and co-inductive styles of programming along with computational effects (specifically, classical control effects) without losing properties like strong normalization or extensional reasoning. Additionally, we show how the lessons we learn can be translated back to the more familiar ground of effect-free functional programming, although its inherent lack of duality causes some symmetries of recursion schemes to be lost in translation. We can write pure functional programs with mixed induction and co-induction, but the asymmetry of the paradigm blocks the full expression of certain recursion principles.

In order to ensure that recursion is well-founded, we use type-level indices indicating the size of types as a tool. This is a pragmatic choice: the nature of computation in the sequent calculus makes it essential to track size arguments for well-foundedness "inside" larger structures. Allowing size information to flow into structures is a natural consequence of the co-data presentation of functions. Implementations of type theory typically check the arguments to a recursive function definition, but since functions are just another user-defined co-data structure containing these arguments, there is no inherent reason to limit this functionality to function types alone.

We have shown how both recursion and co-recursion in programs can be drawn from the mathematical principles of primitive and noetherian induction, and codified as programming structures for representing recursive processes. The style of primitive recursion with computationally sensitive type-level indices can be mixed with noetherian recursion that use computationally-irrelevant indices. We

see that the primitive and noetherian recursion principles, which are generally distinct mathematically, are also distinct computationally and have different uses. The general (co-)data mechanism helped us to understand these principles for recursion in programs, but the recursors were generated by hand. Can we find the general mechanism that encompasses recursion in programs, in the same way that we have encompassed recursion in (co-)data types?

A clear subject for future study is to enrich the existing dependencies in types to be closer to full-spectrum dependent types. We find that a modest amount of dependency in primitive recursion, in the form of numeric type indices admitting case analysis, helps us encode programs over Haskell-style infinite lists. Further exploring the nature of this dependency may show how to adapt this theory to be applicable to the use in proof assistants with dependent types. We also saw how the duality of classical logic is useful in the study of recursion. Can this classicality be rectified with more complex notions of dependency, so that dependent types can be given a computational view of classical reasoning principles?

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback on improving this paper. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-1423617.

References

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

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a

unified approach to termination and productivity. In *ICFP*, 2013.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *POPL*, 2013.

[4] F. Barbanera and S. Berardi. A symmetric lambda calculus for "classical" program extraction. In *TACS '94*, pages 495–515, 1994.

[5] T. Coquand and P. Dybjer. Inductive definitions and type theory an introduction. In *FSTTCS*, volume 880 of *LNCS*, 1994.

[6] P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming*, pages 233–243, 2000.

[7] P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. *Theoretical Computer Science*, pages 165–181, 2010.

[8] P. Downen and Z. M. Ariola. The duality of construction. In *European Symposium on Programming*, 2014.

[9] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.

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

[11] T. Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[12] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, 1987.

[13] J.-L. Krivine. Realizability in classical logic. In *Interactive models of computation and program behaviour*, volume 27, pages 197–229. Société Mathématique de France, 2009.

[14] S. Lengrand and A. Miquel. Classical Fω, orthogonality and symmetric candidates. *Annals of Pure and Applied Logic*, 153(1):3–20, 2008.

[15] P. Martin-Löf. A theory of types. Technical Report 71-3, University of

Stockholm, 1971.

[16] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.

[17] P. M. Nax. *Inductive Definition in Type Theory*. Ph.D. thesis, Cornell University, 1988.

[18] N. Oury. Coinductive types and type preservation. Message on the Coq-club mailing list, June 2008.

[19] S. Singh, S. P. Jones, U. Norell, F. Pottier, E. Meijer, and C. McBride. Sexy types—are we done yet? Software Summit, Apr. 2011.

[20] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of ICFP*, pages 189–201. ACM, 2003.

[21] N. Zeilberger. On the unity of duality. *Annals of Pure Applied Logic*, 153(1-3):66–96, 2008.