Structures for Structural Recursion

Paul Downen Philip Johnson-Freyd Zena M. Ariola
University of Oregon, USA
{pdownen,philipjf,ariola}@cs.oregon.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][13] 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 reasoning, 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 Hugino [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-style based 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 [3][5] 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
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 \(\text{length} \left(\text{Cons} \left(\text{error} \text{ “boom”} \right) \text{Nil} \right)\) 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 \((\text{Section } 6)\).

Next, we develop two forms of well-founded recursion which are reflected as pairs of dual data and co-data types, along with specific data and co-data types for performing well-founded recursion in programs \((\text{Section } 3)\). 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 \((\text{Section } 4)\). 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 \((\text{Section } 5)\).

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 logic 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 \(\text{length}\) function in a \(\text{Haskell-}\) or \(\text{Agda-like language by pattern-matching over the structure of the given List} \ a\) to produce a \(\text{Nat}\):

\[
\text{data Nat where } \quad \text{data List } a \ where \\
Z : \text{Nat} \quad \text{Nil : List } a \\
S : \text{Nat } \rightarrow \text{Nat} \quad \text{Cons} : a \rightarrow \text{List } a \rightarrow \text{List } a \\
\text{length Nil} = Z \\
\text{length } (\text{Cons } x \ \text{xs}) = \text{let } y = \text{length } x s \text{in } S \ y
\]

This definition of \(\text{length}\) describes its result for every possible call. Similarly, we can define \(\text{length} \) in the \(\mu\tilde{\mu}\)-calculus \([8]\), 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:

\[
\text{data Nat where } \quad \text{data List}(a) \ where \\
Z : \vdash \text{Nat} \quad \text{Nil : } \vdash \text{List}(a) \\
S : \vdash \text{Nat} \quad \text{Cons} : a, \text{List}(a) \vdash \text{List}(a)
\]

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 pre-defined function type, the shape of the constructors are described via \(\text{sequents} \), 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 \(\vdash \text{Nat}\) or \(\vdash \text{List}(a)\), 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 \(\text{Nil}\) or \(\text{Cons}(x, \text{xs})\), much like in functional languages. Next, we define \(\text{length}\) by specifying its behavior for every possible call:

\[
\begin{align*}
\langle \text{length} | \text{Nil} \cdot \alpha \rangle &= (Z|\alpha) \\
\langle \text{length} | \text{Cons}(x, \text{xs}) \cdot \alpha \rangle &= \langle \text{length} | \text{xs} \cdot \text{\mu}y. (S(y)|\alpha) \rangle
\end{align*}
\]

The main difference is that we consider more than just the argument to \(\text{length}\). Instead, we are describing the action of \(\text{length}\) with its entire context by showing the behavior of a \(\text{command}\) which connects together a producer and a consumer. For example, in the command \(\langle Z|\alpha\rangle\), \(Z\) is a term producing zero and \(\alpha\) 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 \(\text{Nil} \cdot \alpha\) consumes a function by supplying it with \(\text{Nil}\) as its argument and consuming its returned result with \(\alpha\). The input abstraction \(\text{\mu}y. (S(y)|\alpha)\) names its input \(y\) before running the command \(\langle S(y)|\alpha\rangle\), similarly to the context \(\text{let } y = \square \text{in } S(y)\) from the functional program.

\footnote{Note that symbols \(\mu\) 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 \( \text{length} \) as:

\[
\text{length} \text{ Nil} = Z \\
\text{length} \ (\text{Cons} \ x \ xs) = S(\text{length} \ xs)
\]

We can mimic this definition in the sequent calculus as:

\[
(\text{length} | \text{Nil} \cdot \alpha) = (Z | \alpha) \\
(\text{length} | \text{Cons} (x, xs) \cdot \alpha) = (S | (\mu \beta. (\text{length} \ xs \cdot \beta)) | \alpha)
\]

Note that to represent the functional call \( \text{length} \) inside the successor constructor \( S \), we need to make use of a new kind of term: the output abstraction \( \mu \beta. (\text{length} \ xs \cdot \beta) \) names its output channel \( \beta \) before running the command \( (\text{length} \ xs \cdot \beta) \), which calls \( \text{length} \) with \( xs \) as the argument and \( \beta \) as the return consumer. In the \( \mu \beta \)-calculus, output abstractions are exactly dual to input abstractions, and defining \( \text{length} \) in \( \mu \beta \) 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 \( \alpha \) 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 \( f \ y \ z \) 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.

End 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:

\[
\text{codata Stream}(a) \text{ where} \\
\text{Head} : \ | \text{Stream}(a) \vdash a \\
\text{Tail} : \ | \text{Stream}(a) \vdash \text{Stream}(a)
\]

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 \( \vdash \text{Stream}(a) \). The rest of the types describe the parameters that must be given to the constructor in order to properly consume this main input. For Streams, the observation \( \text{Head}[\alpha] \) requests the head value of a stream which should be given to \( \alpha \), and \( \text{Tail}[\beta] \) asks for the tail of the stream which should be given to \( \beta \). We can now define a function \( \text{countUp} \)—which turns an \( x \) of type \( \text{Nat} \) into the infinite stream \( x, S(x), S(S(x)), \ldots \)—by pattern-matching on the structure of observations on functions and streams:

\[
(\text{countUp} | x \cdot \text{Head}[\alpha]) = (x | \alpha) \\
(\text{countUp} | x \cdot \text{Tail}[\beta]) = (\text{countUp} | S(x) \cdot \beta)
\]

If we compare \( \text{countUp} \) with \( \text{length} \) 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 difference is that \( \text{length} \) happens to match on its input data structure in its call-stack, whereas \( \text{countUp} \) matches on its return co-data structure.

Abel et al. [2] have carried this intuition back into the functional paradigm. For example, we can still describe streams by their Head and Tail projections, and define \( \text{countUp} \) through co-patterns:

\[
\text{codata Stream a where} \\
\text{Head} : \text{Stream} a \rightarrow a \\
\text{Tail} : \text{Stream} a \rightarrow \text{Stream} a
\]

\[
(\text{countUp} x). \text{Head} = x \\
(\text{countUp} x). \text{Tail} = \text{countUp} (S x)
\]

This definition gives the functional program corresponding to the sequent version of \( \text{countUp} \). 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 \( \text{SkipStream}(a) = \text{Stream}(\text{Maybe}(a)) \), where \( \text{Maybe}(a) \) corresponds to the Haskell datatype with constructors Nothing and Just for \( a \) of type \( a \). Then we can define the empty skip stream which gives Nothing at every position, and the \( \text{countDown} \) function that transforms \( S^n(Z) \) into the stream \( S^n(Z), S^{n-1}(Z), \ldots, Z, \text{Nothing}, \ldots \):

\[
\begin{align*}
(\text{empty} | \text{Head}[\alpha]) &= (\text{Nothing} | \alpha) \\
(\text{empty} | \text{Tail}[\beta]) &= (\text{empty} | \beta) \\
(\text{countDown} | x \cdot \text{Head}[\alpha]) &= (\text{Just}(x) | \alpha) \\
(\text{countDown} | Z \cdot \text{Tail}[\beta]) &= (\text{empty} | \beta) \\
(\text{countDown} | S(x) \cdot \text{Tail}[\beta]) &= (\text{countDown} | x \cdot \beta)
\end{align*}
\]

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:

\[
\text{zeroes} = \text{Cons} Z \text{ zeroes}
\]

We could emulate this definition in sequent style as the expansion of \( \text{zero} \) when observed by any \( \alpha \):

\[
(\text{zeroes} | \alpha) = (\text{Cons}(Z, \text{ zeroes}) | \alpha)
\]

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

\[
\begin{align*}
(\text{cat} | \text{Nil} \cdot y \cdot \alpha) &= (y \cdot \alpha) \\
(\text{cat} | \text{Cons}(x, xs) \cdot y \cdot \alpha) &= (\text{Cons}(x, \mu \beta. (\text{cat} | xs \cdot y \cdot \beta)) | \alpha)
\end{align*}
\]

The intention is that, so long as we do not evaluate the sub-components of Cons eagerly, then \( \alpha \) 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 \( \mu \beta \)-calculus. The full syntax of this language is shown in Figure 1.

The different components of programs in the \( \mu \beta \)-calculus can be understood by their relationship between opposing forces of input and output. A term, \( \nu \), produces an output, a co-term, \( \epsilon \), consumes
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 $\alpha, \beta, \gamma$). Thus, besides variables and co-variables, the core $\mu\tilde{\mu}$-calculus includes the generic abstractions seen in Section 2.

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 $\Gamma$ and $\Delta$ respectively, and the entailing $(\vdash)$ separates inputs on the left from outputs on the right. Additionally, the context, $\Theta$, for type variables (written $a, b, c, d$), being neither input nor output, adorn the turnstile 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: $\Gamma \vdash^* v : A$ states that $v$ is a term producing an output of type $A$, $\Gamma[c : A] \vdash^\Theta$ states that $c$ 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 $\lambda$-calculus at the type level with $*$ as the base kind, $\Theta \vdash A : B : k$ states that $A$ is a type of kind $k$, and $\Theta \vdash A : B : k$ states that $A$ and $B$ are $\alpha, \beta, \gamma$-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 data terms as well as new case abstraction co-terms that perform case analysis to destruct its input before deciding which branch to take similar to the context case of... in functional languages. Co-data declarations are exactly symmetric, introducing new 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, List$(a)$, and 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:

\[
\text{data} (a : *) \otimes (b : *) \quad \text{where} \quad \langle\_, \_\rangle : a, b \vdash a \otimes b
\]

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 $\mu\tilde{\mu}(x, y, e, c)$, pattern-matches on its input pair before running the command $c$.

Furthermore, we can declare the type for functions as:

\[
\text{codata} (a : *) \rightarrow (b : *) \quad \text{where} \quad a[a \rightarrow b \vdash b]
\]

This co-data declaration says that building a function call-stack of type $a \rightarrow b$ 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 $\mu\tilde{\mu}(x \cdot a, c)$, pattern-matches on its output stack before running $e$. Note that this is an alternative representation of functions to $\lambda$-abstractions in functional languages, but an equivalent one. Indeed, the two views of functions are mutually definable:

\[
\lambda x. v = \mu(x \cdot a. (v[a])) \quad \mu(x \cdot a. c) = \lambda x. \mu \alpha. c
\]

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 $F(A)$ and $G(A)$ are well-formed, 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 $A$, we must substitute $\tilde{A}$ for all possible occurrences of the parameters $\vec{a}$ in the declaration. Furthermore, the chosen instances $\tilde{D}$ for the quantified type variables $\vec{d}$, which annotate the constructor, must also be substituted for their occurrences in other types. With this in mind, the rules for construction (the FRK, and GLH rules) check that the sub-(co)-terms and quantified types of a structure have the expected instantiated types, whereas the rules for deconstruction (FL and GR) extend the typing contexts with the appropriately typed (co-)variables and type variables.

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

\[
\text{data} \exists (a : *) \rightarrow \exists (b : *) \quad \text{where} \quad \text{codata} \forall (a : *) \rightarrow \forall (b : *) \quad \text{where}
\]

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

\[
\begin{aligned}
&\Gamma \vdash^* \alpha : A, \Delta \\
\Gamma &\vdash^\Theta \mu(\text{Spec}^{\tilde{\delta}}[\alpha].c) : \forall A \vdash^\Theta \Delta \\
\Gamma &\vdash^\Theta B : \tilde{\alpha} \vdash \forall A \vdash^\Theta \Delta \\
\Gamma &\vdash^\Theta \text{Spec}^{\tilde{\delta}}[\beta] : \forall A \vdash^\Theta \Delta \\
\Gamma &\vdash^\Theta \text{Pack}^{\tilde{\delta}}(v) : \exists A \vdash^\Theta \Delta \\
\Gamma &\vdash^\Theta \text{Pack}^{\tilde{\delta}}(v) : \exists A \vdash^\Theta \Delta
\end{aligned}
\]

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

\[
\text{length} = \mu(\text{Nil} \cdot \alpha. (\text{Z}[\alpha]))
\]

length $= \mu(x \cdot a. (x \cdot \mu(x, y, e). \langle S(x) | \alpha \rangle))$

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

\[
\begin{aligned}
&\text{length} = \mu(x \cdot a. (x \cdot \mu\tilde{\mu}(\text{Nil} \cdot (z \cdot \alpha))) \\
&\quad \langle \text{Cons}(x, x'), (\text{length} \cdot (x' \cdot \mu\tilde{\mu}(\text{Z}[\alpha]))) \rangle
\end{aligned}
\]

This case abstraction describes exactly the same specification as the definition for length in Example 1.
There is one fundamental difficulty in ensuring termination for assistants and dependently typed languages, which rely on the
well-founded recursion programs perform their structural recursion from within programs written in a sequent calculus style: even incredibly simple itself (see Section 5).

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
absence of infinite loops for their logical consistency. If we consider to be more complicated. Even worse, the “infinite data structures”
result. However, formulating this argument in general turns out
to be more complicated. Even worse, the “infinite data structures”
also specify which parts of a complex type are used as part of the
allowing us to abstract over termination-ensuring measures, we can
adopt a type-based approach to termination checking [1]. Besides
where
smaller” in a recursive program, pointing out
ever increasing result amidst our ever decreasing recursion.
However, in the sequent calculus, the actual recursive invocation of
does not return to its original caller, but to some place new.
length
is the
explicit part of the function call structure, necessary to remember
call-stack. This is because the recursive call to
length
is the
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 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.

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

\[
\begin{align*}
\Gamma, x : A &\vdash x : A & \text{Var} \\
\Theta \vdash A = B : * &\quad \Gamma \vdash v : A &\vdash B : \Delta & \text{Eq} \\
\Theta \vdash A = B : * &\quad \Gamma \vdash c : A &\vdash B &\vdash A &\Delta & \text{Act} \\
\Gamma \vdash c : A &\vdash B &\vdash A &\Delta & \text{Act} \\
\Gamma \vdash \mu x. c : A &\Delta & \text{CoAct} \\
\Gamma \vdash \alpha : A &\vdash \alpha : A &\Delta & \text{CoVar}
\end{align*}
\]

**Figure 2.** The type system for the core higher-order \( \mu \bar{\nu} \)-calculus.

<table>
<thead>
<tr>
<th>data ( F(a \rightarrow k) ) where</th>
<th>codata ( G(a \rightarrow k) ) where</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K_0 : B_1 \vdash_{d_1 \vdash_{d_1}} F(\overline{a})</td>
<td>C_1 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( K_n : \overline{B_n} \vdash_{d_n \vdash_{d_n}} F(\overline{a})</td>
<td>C_n )</td>
</tr>
</tbody>
</table>

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

| \( \Theta \vdash A : k \) | \( \Theta \vdash F(\overline{a}) : * \) |
| \( \Theta \vdash C_1 : A \vdash (\overline{a}/a) \vdash \Delta \), \( \alpha : C_1(\overline{a}/a) \) | \( \Gamma \vdash \mu [K_{d_1 \vdash_{d_1}}(\overline{a}, \overline{x}), C_1|\ldots|] : F(\overline{a}) \vdash \alpha \Delta \) |
| \( \Theta \vdash D : l_i(\overline{a}/a) \) | \( \Gamma \vdash \alpha : C_1(\overline{a}/a, D/d) \vdash \Delta \) |
| \( \Theta \vdash v : B_1(\overline{a}/a, D/d) \vdash \Delta \) | \( \Gamma \vdash \alpha : \mu [K_{d_1 \vdash_{d_1}}(\overline{x}, \overline{a}), C_1|\ldots] : G(\overline{a}) \vdash \Delta \) |

**Figure 4.** Typing rules for non-recursive, user-defined data and co-data types.
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 loop 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 case. On the one hand, depending on whether the vector is empty or not, we only have to handle the case for an empty vector of type Vec(N, A) to destruct an empty vector, we only have to handle the case for an empty vector, giving us

\[
\text{data } \text{Vec}(i : \text{Ix}, a : * ) \text{ by primitive recursion on } i \\
\text{where } i = 0 \quad \text{Nil} : \vdash \text{Vec}(0, a) \\
\text{where } i = j + 1 \quad \text{Cons} : a, \text{Vec}(j, a) \vdash \text{Vec}(j + 1, a)
\]

where Ix is the kind of type-level natural number indices. Nil builds an empty vector of type Vec(0, A), and Cons(a, Vec(j, a)) extends the vector with another element a : A, giving us a vector with one more element of type Vec(N + 1, A). Other than these restrictions on the instantiations of i : Ix for vectors constructed by Nil and Cons, the typing rules for terms of type Vec(N, A) follow the normal pattern for declared data types:\footnote{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 \( \mu x.c \) of type Vec(i, A).} 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 \( \mu [\text{Nil}].c \). On the other, destructing a non-empty vector requires us to handle the Cons case, as given by the co-term \( \mu [\text{Cons}(x, xs)].c \). These co-terms are typed by the two left rules for Vec—one for both its zero and successor cases:

\[
\Gamma \vdash_{\Theta} \Delta \\
\text{Vec}_0 \\
\Gamma \vdash_{\Theta} \Delta \\
\text{Vec}_{i+1}
\]

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

\[
\text{data } \text{bxList}(i : \text{Ix}, a : * ) \text{ by primitive recursion on } i \\
\text{where } i = 0 \quad \text{Nil} : \vdash \text{bxList}(0, a) \\
\text{where } i = j + 1 \quad \text{Cons} : a, \text{bxList}(j, a) \vdash \text{bxList}(j + 1, a)
\]

Now, destructing a non-zero bxList(N + 1, A) requires both cases, as given in the co-term \( \mu [\text{Nil}, c].\text{Cons}(x, xs), c' \). bxList 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 loop 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:

\[
P[0] \land (\forall j : N. P[j] \rightarrow P[j + 1]) \rightarrow (\forall i : N. P[i])
\]

and likewise the refutation of such a statement, as is given by any specific counter-example—a : N \( \vdash P[i] \rightarrow (\forall i : N. P[i]) \) into logical rules of the sequent calculi:\footnote{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 \textit{dual} to a proof that \( P \) is true, which is a refutation of \( P \) demonstrating that it is false.}

By the usual reading of sequents, proofs come to the right of entailment (\( \vdash \)) means “\( A \) is true”, whereas refutations come to the left (\( \vdash \) means “\( A \) is false”). Because we will have several recursion principles, we denote this particular one with a type named Inflate, so that the primitive recursive proposition \( \forall i : N. P[i] \) is written as the type Inflate(\( \lambda : \text{lx}, A \)) with the inference rules:

\[
\vdash A \quad j \vdash_{\text{lx}} A \quad j + 1 \quad \vdash \text{Inflate}(A) \\
\vdash M \quad \text{lx} \quad A \quad M \vdash \text{Inflate}(A)
\]

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 Inflate:

\[
\text{codata } \text{Inflate}(a : \text{lx} \rightarrow *) \text{ where } \\
\text{Up} : \vdash \text{Inflate}(a) \vdash_{\text{lx}, \text{a}, \overline{j}} a \quad j
\]

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 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 \( \mu [\text{Up}^{0: \text{Ix}}[\alpha], c_0].\text{Up}^{1: \text{Ix}}[\alpha](x), c_1] \) 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 \( \text{Up}^{1: \text{Ix}}[\alpha](x) \). This term has the typing rule:

\[
\Gamma \vdash_{\Theta} \mu[\text{Up}^{0: \text{Ix}}[\alpha], c_0].\text{Up}^{1: \text{Ix}}[\alpha](x), c_1] : \text{Inflate}(A)\overline{\Delta}
\]

Terms of type Inflate i : Ix.A (which is shorthand for the type Inflate(\( \lambda : \text{lx}.A \))) 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
We now consider the more complex of the two recursion schemes: a
which describe a process which is able to consume
primitive recursion as a data type. In particular, we get the data
instance of an ordered type.

Ord
may define natural numbers by noetherian recursion over ordered
principles into inference rules in the sequent calculus, where we
run out of possible indices below our current one as guaranteed by
bounded indices, we use an appropriate recursion scheme for
these two possible observations:

\[ \Gamma \vdash \alpha : A \rightarrow \Delta \]
\[ \Gamma \vdash \text{Head}[\alpha] : \text{Stream}(N, A) \rightarrow \Delta \]
\[ \Gamma \vdash \text{Tail}[\alpha] : \text{Stream}(N, A) \rightarrow \Delta \]

\[ \Theta \vdash M < N \rightarrow \Gamma \vdash \exists \alpha : \text{Nat}(\Delta) \]
\[ \Gamma \vdash \text{Head}[\alpha] : \text{Stream}(N, A) \rightarrow \Delta \]
\[ \Gamma \vdash \text{Tail}[\alpha] : \text{Stream}(N, A) \rightarrow \Delta \]

\[ \text{Stream}R \]

As before, to write loop 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:

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

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

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

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:

\[ \text{Ascend}(N, A) \vdash \text{Ascend}(N, A) \]

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

\[ c_0 : (\Gamma \vdash \Delta) \rightarrow (\Gamma \vdash \text{Nat}(j) \vdash \mu \nu. S^M(\nu) : \text{Nat}(\Delta) \vdash \Delta) \]

NatL

Like the case abstraction for tearing down an existentially con-
structured, 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
notated version of the definition from Example

\[ \text{data} \text{List}(i : \text{Ord}, a : :) \text{by noetherian recursion on } i \text{ where} \]
\[ \text{Nil} : \vdash \text{List}(\text{nil}, a) \]
\[ \text{Cons} : a, \text{List}(\text{cons}, a) \vdash \text{List}(j, a) \]

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

\[ \text{codata} \text{Stream}(i : \text{Ord}, a : :) \text{by noetherian recursion on } i \text{ where} \]
\[ \text{Head} : \vdash \text{Stream}(i, a) \rightarrow a \]
\[ \text{Tail} : \vdash \text{Stream}(i, a) \rightarrow (j < i) \vdash \text{Stream}(j, a) \]

Recursive co-data types follow the dual pattern as data types, with
finitely built observations and values given by case analysis on their
Recursive co-data types follow the dual pattern as data types, with
finitely built observations and values given by case analysis on their
Observations and values given by case analysis on their
We now flesh out the rest of the system for recursive types and we need to check that its declaration actually denotes a meaningful type. For the non-recursive (co-)data declarations, like those in Figure 5, 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 \((B_j \vdash \omega \rightarrow_\mu \mu \delta_i \cdot C_i)\#\) seq from Figure 5. When checking for well-formedness of (co-)data types defined by primitive induction on \(i : Ix\), as with the general form

\[
\text{data } F(i : Ix, a : k) \text{ by primitive recursion on } i
\]

\[
\begin{align*}
\text{where } i &= 0 & \Theta_i : K_i : & B_1 \vdash \delta_i \cdot F(0, \vec{a})|C_i \# & \ldots \\
\text{where } i &= j + 1 & \Theta_i : & B_i \vdash \delta_i \cdot F(j + 1, \vec{a})|C_i \# & \ldots 
\end{align*}
\]

the \(i = 0\) case proceeds by checking that the sequents are well-formed for each constructor \(K_1\ldots\) without referencing \(i\), \((B_1 \vdash \omega \rightarrow_\mu \mu \delta_1 \cdot C_1)\#\) seq, and in the \(i = j + 1\) case we check each \((B_j \vdash \omega \rightarrow_\mu \mu \delta_j \cdot C_j)\#\) seq with the extra rule

\[
\Theta, j : Ix, \Theta' \vdash A : \star \quad \Theta, j : Ix, \Theta' \vdash F(j, \vec{A}) : \star
\]

Intuitively, in the \(i = j + 1\) case the sequents for the constructors may additionally refer to smaller instances \(F(j, \vec{A})\) 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 \(F(0, \vec{A})\) and \(F(M + 1, \vec{A})\) with \(M\) substituted for \(j\), respectively. Additionally, there are two case abstractions: one of type \(F(0, \vec{A})\) that only handles constructors of the \(i = 0\) case, and one of type \(F(M + 1, \vec{A})\) that only handles constructors of the \(i = j + 1\) case. Similarly, when checking for well-formedness of (co-)data types \(F(i : Ix, a : k)\) defined by noetherian induction on \(i : Ord\), we get to assume the type is defined for smaller indices:

\[
\Theta, i : Ord, \Theta' \vdash M < i \\
\Theta, i : Ord, \Theta' \vdash A : \star
\]

Intuitively, the sequents for the constructors may refer to \(F(M, \vec{A})\), so long as they introduce quantified type variables \(d : I\) such that \(a : k, d : I \vdash M < i\). 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 \(\mu \delta_\Sigma\)-calculus in Figure 5 which is parameterized by the strategy \(S\). Since the classical sequent calculus inherently admits control effects, the result of a program can completely change depending on the strategy—\(\langle \text{length}\rangle \vdash \text{Rise}^\Sigma|\{\text{bounds}\}^{\langle 13 \rangle[\alpha], \text{Nil}, \alpha\} |\langle 1 \rangle[\alpha]\) under call-by-name evaluation and \(\langle 13 \rangle[\alpha]\) under call-by-value—so that the parametric \(\mu \delta_\Sigma\)-calculus is actually a family of related but different rewriting theories for reasoning about different abstractions. Thus, enabling strategy-independent reasoning. The choice of strategy is given as the syntactic notions of \textit{value} and \textit{co-value}; \(S\) is the subset of terms \(V \in Val\) and \(E \in CoV\) 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 \(\mu\)- and \(\mu\)-abstractions. For example, the strategies for call-by-value and call-by-name evaluation are shown in Figure 6 and a strategy representing call-by-need evaluation is representable this way as well.

The reduction rules are derived from the core theory of substitution in \(\mu \delta_\Sigma\) (the top rules of Figure 5), plus rules derived from generic \(\beta\) and \(\eta\) principles for every (co-)data type. Of note are the \(\xi\) 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.
“Don’t touch unreachable branches,” to ensure strong normalization
of the rewrite system, which must be performed independently.

Intuitively, this index-unaware loop unrolling is possible because the
recursive variable is inward—with a tighter upper bound—for the recursive variable.

As mentioned earlier, the reachability caveat prevents reduction inside a case abstraction which introduces type variables
that might be impossible to instantiate, like \(\text{case } i < 0 \text{ or } j < i \). The reductions following the reachability caveat are defined as:

\[
\begin{align*}
  c \to c' & : k \to (\langle N \rangle \in \Theta) \Rightarrow N = \infty \lor N = M + 1 \\
  \mu(\text{Up}^M[a]) & : \mu(\text{Up}^{M+1}[a]) \\
  k \to (\langle N \rangle \in \Theta) & \Rightarrow N = \infty \lor N = M + 1 \\
  \mu(\text{Up}^M[a]) & : \mu(\text{Up}^{M+1}[a]) \\
\end{align*}
\]

We also define the type erasure operation on programs, \(\text{Erase}(c)\), which removes all types from constructors and patterns in \(c\) with an erasable kind, while leaving intact the unerasable \(b\) types. The corresponding type-eraser \(\mu_G\) is the same, except that the reachability caveat is enhanced to never reduce inside case
abstractions. This means that every step of a type-eraser 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

...
Berardi’s symmetric candidates, a variant of Girard’s reducibility candidates, as well as Krivine’s classical realizability, an application of bi-orthogonality, strengthens 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 must be stable, meaning that (co)-values are closed under reduction and substitution, and non-(co)-values are closed under substitution and reduction. Second, $S$ 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—each criterion comes from an inference rule for typing a (co)-value in focus.

**Theorem 1.** For any stable and focalizing strategy $S$, if $f : \Gamma \vdash \Delta$ and $(\Gamma \vdash \Delta)^{seq}$, then $c$ is strongly normalizing in the $\mu_\Sigma$-calculus. Furthermore, $\text{Erase}(c)$ is strongly normalizing in the type-erased $\mu_\Sigma$-calculus.

Note that the call-by-name, call-by-value, and call-by-need strategies are all stable and focalizing, so that as a corollary, we achieve strong normalization for these particular instances of the parametric $\mu_\Sigma$-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 $\lambda$-calculus. 

### 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 4. 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 2 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 $\mu_\Sigma$-calculus, we can give a more precise and non-recursive definition for `length`:

$$
\text{length} : \forall a : *. \text{Ascend } i < \infty. \text{List}(i, a) \rightarrow \text{Nat}(i)
$$

$$
\text{length} = \mu(\text{Spec}[\text{Rise}^{\langle i \rangle}_{\langle \infty \rangle} \text{Nil} : [\gamma](r).([Z](\gamma)))
$$

$$
\text{Spec}[\text{Rise}^{\langle i \rangle}_{\langle \infty \rangle} \text{Cons}^{\langle x, a \rangle}_{\langle \infty \rangle} \cdot [\gamma](r)).
$$

$$
\langle r | \text{Rise}^{\langle x \cdot \mu_{\gamma} \cdot \text{Nat}(\text{Spec}(\langle y \rangle)) \rangle}_{\langle y \rangle})
$$

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:

$$
\text{length} = \mu(\text{Spec}[^a \text{Ascend } i < \infty. \text{List}(i, a) \rightarrow \text{Nat}(i)]).
$$

$$
(\mu(\text{Rise}^{\langle i \rangle}_{\langle \infty \rangle} \text{List}(i, a) \rightarrow \text{Nat}(j)).(\text{Ascend } j < i. \text{List}(j, a) \rightarrow \text{Nat}(j)),
$$

$$
(\mu([x] \text{List}(i, a) \rightarrow \text{Nat}(i)), ([x])
$$

$$
| \text{Nil}(\langle Z \rangle))
$$

$$
| \text{Cons}^{\langle x, a \rangle}_{\langle \infty \rangle} \cdot [\gamma](r). \langle r | \text{Rise}^{\langle x \cdot \mu_{\gamma} \cdot \text{Nat}(\text{Spec}(\langle y \rangle)) \rangle}_{\langle y \rangle})
$$

$$
| \langle \beta \rangle
$$

$$
| \langle \alpha \rangle
$$

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.

**Example 4.** Recall the `countUp` function from Example 2. When we attempt to encode this function into the $\mu_\Sigma$-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 $\text{ANat} = \text{Descend } i < \infty. \text{Nat}(i)$. We can then describe `countUp` as a function from $\text{ANat}$ to a $\text{Stream}(i, \text{ANat})$ by noetherian recursion on $i$:

$$
\text{countUp} : \text{Ascend } i < \infty. \text{ANat} \rightarrow \text{Stream}(i, \text{ANat})
$$

$$
\text{countUp} = \mu(\text{Rise}^{\langle \infty \rangle}_{\langle \infty \rangle} [x \cdot \text{Head}(\alpha)](r).([x \cdot \alpha])
$$

$$
| \text{Rise}^{\langle \infty \rangle}_{\langle \infty \rangle} [\text{Tail}^{\langle i \rangle}_{\langle \beta \rangle} (r). [\text{Rise}^{\langle k \cdot i \rangle}_{\langle \beta \rangle} (\text{Spec}(\langle S \rangle))]).
$$

$$
\langle r | \text{Rise}^{\langle k \cdot i \rangle}_{\langle \beta \rangle} (\text{Spec}(\langle S \rangle))
$$

**Example 5.**

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 `zeros` 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 $\text{InfList}(\alpha) = \text{InfList}(\text{List}(\alpha), a, a)$, we can encode `zeros` as:

$$
\text{zeros} : \forall i : \text{Nat}(i)\rightarrow \text{InfList}(\alpha)
$$

$$
\text{zeros} = \mu(\text{Up}^{\langle \infty \rangle}_{\langle i \rangle} \text{InfList}(\text{List}(\alpha), a, a)),
$$

$$
| \text{Up}^{\langle \infty \rangle}_{\langle i \rangle} \text{InfList}(\text{List}(\alpha), a, a)\rightarrow \text{InfList}(\alpha)
$$

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:

$$
\text{cat} : \forall a : *. \text{InfList}(\alpha) \rightarrow \text{InfList}(a) \rightarrow \text{InfList}(\alpha)
$$

$$
\text{cat} = \mu(\text{Spec}[^x \cdot y \cdot \text{Nat}(\text{Spec}(\langle y \rangle))]).
$$

$$
\langle x | \text{Up}^{\langle \infty \rangle}_{\langle i \rangle} \text{Nat}(\alpha)\rightarrow \text{InfList}(\alpha)
$$

$$
| \text{Up}^{\langle \infty \rangle}_{\langle i \rangle} \text{Nat}(\alpha)\rightarrow \text{InfList}(\alpha)
$$

$$
| \text{InfList}(\alpha)\rightarrow \text{InfList}(\alpha)
$$

$$
\langle x, y, \beta \rangle (r \cdot \text{Nat}(\alpha)\rightarrow \text{InfList}(\alpha))
$$

$$
\langle x, y, \beta \rangle (r \cdot \text{Nat}(\alpha)\rightarrow \text{InfList}(\alpha))
$$

$$
\langle x, y, \beta \rangle (r \cdot \text{Nat}(\alpha)\rightarrow \text{InfList}(\alpha))
$$

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:

$$
\text{codata StopStream}(i < \infty, a : *) \text{ where}
$$

$$
\text{Head} : \text{StopStream}(i, a) \rightarrow a
$$

$$
\text{Tail} : \text{StopStream}(i, a) \rightarrow \text{StopStream}(j, a)
$$

A `StopStream(i, a)` object is like a `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:

\[
\begin{align*}
\text{cat}(x \cdot s \cdot y \cdot s \cdot \text{Head}[a]) &= (x \cdot s \cdot \text{Head}[a]) \\
\text{cat}(x \cdot s \cdot y \cdot s \cdot \text{Tail}[\delta, \beta]) &= (\text{cat}(\mu \cdot \gamma, (x \cdot s \cdot \text{Tail}[\mu \cdot (y \cdot s \cdot \beta)]; \gamma)) \cdot y \cdot s \cdot \beta)
\end{align*}
\]

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

\[
\text{cat}' : \forall a : \ast. \text{Ascend } i < \infty.
\]

\[
\text{StopStream}(i, a) \rightarrow \text{InList}(a) \rightarrow \text{StopStream}(i, a)
\]

\[
\text{cat}' = \mu (\text{Spec} \cdot \text{Rise}^{<\infty}[x \cdot s \cdot y \cdot s \cdot \text{Head}[a]](r_1), (x \cdot s \cdot a))
\]

\[
\text{Spec} \cdot \text{Rise}^{<\infty}[x \cdot s \cdot y \cdot s \cdot \text{Tail}[\delta, \beta]](r_1)
\]

\[
(\forall r_1 [\text{Rise}^{<\infty}[x \cdot s \cdot y \cdot s \cdot \text{Tail}[\mu \cdot \gamma, (y \cdot s \cdot \beta)]; \gamma, \gamma) \cdot y \cdot s \cdot \beta])
\]

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 [13]. Intuitively, the root of the problem is that Coq’s co-inductive objects are non-extensional, since the interaction between case analysis and the 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 extensibility 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:

\[
\begin{align*}
(\text{ack}[Z \cdot y \cdot \alpha]) &= (S(y)[\alpha]) \\
(\text{ack}[S(x) \cdot Z \cdot \alpha]) &= (\text{ack}[x \cdot S(Z) \cdot \alpha]) \\
(\text{ack}[S(x) \cdot y \cdot \text{μz}(\text{ack}[x \cdot z \cdot \alpha])]
\end{align*}
\]

The fact that this function terminates follows by lexicographic induction on both arguments: to every recursive call of \(\text{ack}\), 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:

\[
\begin{align*}
\text{ack} : \text{Ascend } i < \infty. \text{Ascend } j < \infty. \text{Nat}(i) \rightarrow \text{Nat}(j) \rightarrow \text{ANat} \\
\text{ack} = \mu (\text{Rise}^{<\infty}[\text{Rise}^{<\infty}[Z \cdot y \cdot \alpha](r_1), (\text{Fall}^{+1}(S(y)[\alpha]))(r_2))(r_1).
\]

\[
(\text{Rise}^{<\infty}[\text{Rise}^{<\infty}[S'(x) \cdot Z \cdot \alpha](r_1), (\text{Fall}^{+1}(S(Z)[\alpha])(r_2))])(r_1).
\]

\[
(\text{Rise}^{<\infty}[\text{Rise}^{<\infty}[S'(x) \cdot S'(y) \cdot \alpha](r_1), (\text{Fall}^{+1}(S'(y)[\alpha])(r_2)])(r_2).
\]

\[
(\text{Rise}^{<\infty}[\text{Rise}^{<\infty}[x \cdot \text{μz}(\text{ack}[x \cdot z \cdot \alpha])](r_1), (\text{Fall}^{+1}(S(Z)[\alpha])(r_2)])(r_1).
\]

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 \(\mu\eta\)-calculus corresponding to minimal logic.

Essentially, the minimal restriction of the \(\mu\eta\)-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 \(\Gamma \vdash_{\circ} v : A\), where the active type on the right is no longer ambiguous and does not need to be distinguished with \(\iota\), 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

\[
\begin{align*}
data a \oplus b &\quad \text{codata } a \& b \where \\
\text{Left} : a \vdash a \oplus b &\quad \text{Fst} : \{a \& b \vdash a\} \\
\text{Right} : b \vdash a \& b &\quad \text{Snd} : \{a \& b \vdash b\}
\end{align*}
\]

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

\[
\begin{align*}
data a - b &\quad \text{codata } a \& b \where \\
\text{Pause} : a \vdash a - b \& b &\quad \text{Split} : \{a \& b \vdash a, b\}
\end{align*}
\]

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, \(v : \text{Fst}\), and we describe an object of this type by saying how it responds to all possible observations,

\[
\{\text{Fst} \Rightarrow v_1, \text{Snd} \Rightarrow v_2\}
\]

with the typing rules:

\[
\begin{align*}
\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B &\quad \Gamma \vdash v : A \& B \\
\Gamma \vdash \{\text{Fst} \Rightarrow v_1, \text{Snd} \Rightarrow v_2\} : A \& B &\quad \Gamma \vdash v : A \& B
\end{align*}
\]

Likewise, the traditional \(\lambda\)-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, Call : \(a[a] \rightarrow b \& b\) and Spec : \(\forall a \vdash b, a, b\), the abstractions can be given as syntactic sugar:

\[
\lambda a : \mu b. v = \{\text{Call}[a] \Rightarrow v\} \quad a : \mu b \vdash \{\text{Spec}^{b \Rightarrow} \Rightarrow v\}
\]

Thus, these objects also serve as “generalized \(\lambda\)-abstractions” defined by shallow case analysis rather than deep pattern-matching. The typing rules for recursive structures translated to functional style are shown in Figure 2 and the reduction rules for the calculus
are shown in Figure 10. Intuitively, the objects of 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 Deflate(A) to count down from any A N to 0. Similarly, values of 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 Descend(N, A) are conspicuously missing from the functional calculus. In essence, the recursive form for 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–Descend(M, A) for smaller indices M. So while 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 ¬(A ∧ B) → (¬A)∨(¬B) is not intuitionistically valid), the conjunctive and disjunctive connectives of linear logic (⊗ requires multiple conclusions so it does not fit the minimal mold), and here as well.

Intuitively, we can think of the values of 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:

\[
\begin{align*}
\text{rec} : & \mathbb{N} \times A \to A + A \ni \alpha \to \text{Inflate}_\alpha : \mathbb{N} \times A \\
\{ \alpha \} & = \lambda \alpha. x f. \{ \text{Up}^0 i x f \to x \} \{ \text{Up}^0 i x f \to x \} \to f. \text{Up}^0 i x f
\end{align*}
\]

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 recursion: given that rec a x f \to rec a x f, we have

\[
\text{rec a x f, Up}^0 i x f \to x \text{ rec a x f, Up}^0 i x f \to f. \text{Up}^0 i x f
\]

Contraposely, 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 : Vec(N,ANat), in accumulator style by looping over the recursive structure Descend i : lx. ANat ⊔ Vec(i,ANat)

\[
\text{loop Down}^N(\text{Fall}^0(Z), v) of \\
\text{Down}^0(\text{acc}, \text{Nil}) \to \text{acc} \\
\text{Down}^{i+1}(\text{acc}, \text{Cons}(x, zs)) \to (x + acc, zs)
\]

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 \((x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)\) by

\[
\text{pairs} \text{Nil} = \text{Nil} \\
\text{pairs} \text{Cons}(x, y) = \text{Nil} \\
\text{pairs} \text{Cons}(x, \text{Cons}(y, zs)) = \text{Cons}(x, y), \text{pairs} \text{zs}
\]

5 Note, we assume an addition operator + : ANat → ANat → ANat.
where we silently drop the final element if the list is odd. The **pairs** function can be straightforwardly encoded using Ascend as:

\[
pairs : \forall a : \ast, \text{Ascend} i < \infty. \text{List}(i, a) \rightarrow \text{List}(i, a \otimes a)
\]

\[
pairs = \lambda a \cdot \langle \text{Rise}^i_{<\infty} r \rangle \Rightarrow \lambda \text{List}(j, a), \text{case } x s \text{ of } \\
\text{Nil} \Rightarrow \text{Nil} \\
\text{Cons}^j_{<\infty}(x^a, y^a) \Rightarrow \text{case } y s \text{ of } \\
\text{Nil} \Rightarrow \text{Nil} \\
\text{Cons}^j_{<\infty}(y^a, z^a) \Rightarrow \text{Cons}^i_{<\infty}(\langle x, y, \text{Rise}^k z s \rangle)
\]

Note that the type of the recursive argument **r** is **Ascend** \(i' < i \). **List**(\(i', a\)) \(\rightarrow\) **List**(\(i', a \otimes a\)). Thus, the recursive self-invocation **r**.\(\text{Rise}^k : \text{List}(k, a) \rightarrow \text{List}(k, a \otimes a)\) 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 \(\mu\mu\Sigma\)-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 \(\mu\mu\Sigma\) [8], \(\mu\Sigma\), where the set of values is as large as possible and includes every term. So, because the \(\mu\mu\Sigma\)-calculus does not allow for well-typed infinite loops, neither does its functional counterpart.

**Theorem 2.** If \(\Gamma \vdash v : A\) and \((\Gamma \vdash \alpha : A)\) 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 noethern 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 duality 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


A. Strong Normalization

We now give a proof sketch for strong normalization of well-typed (co-)data structures are included. Indeed, head reductions are used to characterize these complex (co-)terms by their behavior instead of by their syntax in a general and extensible way.

We use the fixed point construction of the symmetric candidates technique [5] to simplify the process of ensuring the meaning of a type is a reducibility candidate: so long as we can fully describe a type by a pre-type of simple (co-)terms which can never take a positive or negative step, we can automatically generate a corresponding reducibility candidate for that type. The idea is to define the saturation of a pre-type that adds the missing (co-)terms. A strongly normalizing term is in Sat(A) when either \([v]E\) is strongly normalizing or \([v]E \mapsto_+ c\) and \(c\) is strongly normalizing for every co-value \(E\) in \(A\), and dually for co-terms in Sat(A). Sat preserves the sub-typing order of the lattice of pre-types, so we can take the fixed point (where \(\sqcup\) unions the (co-)terms of pre-types).

Lemma 1. For any \(C\), there is a solution to \(A = C \sqcup \text{Sat}(A)\).

Furthermore, as long as \(C\) is a forward closed, orthogonally sound pre-type of simple (co-values), then this fixed point is a reducibility candidate. Thus, this fixed point construction gives us an operation, \(R(\cdot)\), for generating saturated reducibility candidates from these simplified pre-types.

Types are interpreted by means of a generalized fold over the structure of their syntax. In general, we say \(\mathcal{F}(A)\) to simplify the process of ensuring the meaning of a type is a reducibility candidate: so long as we can fully describe a type by a pre-type of simple (co-)terms which can never take a positive or negative step, we can automatically generate a corresponding reducibility candidate for that type. The idea is to define the saturation of a pre-type that adds the missing (co-)terms. A strongly normalizing term is in Sat(A) when either \([v]E\) is strongly normalizing or \([v]E \mapsto_+ c\) and \(c\) is strongly normalizing for every co-value \(E\) in \(A\), and dually for co-terms in Sat(A). Sat preserves the sub-typing order of the lattice of pre-types, so we can take the fixed point (where \(\sqcup\) unions the (co-)terms of pre-types).
Lemma 2. For any well-formed \( F \), there exists a \( \mathcal{H} \in \mathcal{F} \).

Another source of complexity is that some syntactic types matter computationally, in the case of indices. Reduction rules (e.g. for Inflate) depend on the syntactic form of the index. We resolve this by interpreting kinds not just as sets of semantic objects, but also as logic relations between syntactic and semantic types. Because kinds can depend on types, the meaning of semantic types and kinds (as sets) are intertwined.

\[
\begin{align*}
[\alpha]_\mathcal{H} & \triangleq \mathcal{H}(\alpha) \\
[k]_\mathcal{H} & \triangleq \lambda A \in [k]_\mathcal{H}, [B]_\mathcal{H}(A/a) \\
[F(T)]_\mathcal{H} & \triangleq \mathcal{H}(F([A]_\mathcal{H})) \\
[A[B]]_\mathcal{H} & \triangleq [A]_\mathcal{H}([B]_\mathcal{H}) \\
[C]_\mathcal{H} & \triangleq \{ \mathcal{H}(0) \} \cup \{ \mathcal{H}(+1(M)) \mid M \in [\lambda] \}
\end{align*}
\]

\[(\mathcal{H}, \mathcal{M}) \triangleq \{ \lambda \in \mathcal{H} \mid \forall M \in \mathcal{M}, \mathcal{H}(M) \subseteq M \}
\]

Note that \([\mathcal{H}]\) contains all these semantic kinds. Furthermore, \([\mathcal{F}]\) adds the requirement that the relation between syntactic and semantic types is backward closed under syntactic \( \beta \)-reduction of the \( \lambda \)-calculus; if \( \kappa \in [\mathcal{F}] \), \( \Lambda \Rightarrow_\beta B \), and \( B \in A \), then \( A \in \kappa \).

We interpret typing environments as sets of substitutions.

\[
\begin{align*}
\gamma \in [\Gamma]_\mathcal{H} & \iff \forall A \in \mathcal{H}, x(\gamma) \in \text{Val}([A]_\mathcal{H}) \\
\delta \in [\Delta]_\mathcal{H} & \iff \forall A \in \mathcal{H}, \alpha(\delta) \in \text{Val}([A]_\mathcal{H})
\end{align*}
\]

Kinding environments are interpreted as relations between syntactic substitutions and semantic interpretations.

\[
\begin{align*}
\theta[\theta']_\mathcal{H} & \triangleq \forall \mathcal{H}, \mathcal{H}(h) = \mathcal{H}(h) \\
\theta[\alpha : k]_\mathcal{H} & \triangleq \forall \mathcal{H}, \forall A \in [k]_\mathcal{H}, a(\theta)[k]_\mathcal{H}(a) \\
& \land \forall b \neq b(\theta) \land \forall \mathcal{H}(h) \triangleq \mathcal{H}(h)
\end{align*}
\]

The main soundness property is that if \( c : \Gamma \Rightarrow_\Delta \mathcal{H} \) and \( (\Gamma \Rightarrow_\mathcal{H} \Delta) \Rightarrow \text{seq} \) are derivable, then for any \( \mathcal{H} \in \mathcal{F} \), where \( \mathcal{H} \) assigns a big enough meaning to 0, then \([c : \Gamma \Rightarrow_\Delta \mathcal{H}]_\mathcal{H}\) as given in Figure 1. Further, because \( \mathcal{H} \)-variables inhabit every reducibility candidate, \([c : \Gamma \Rightarrow_\Delta \mathcal{H}]_\mathcal{H}\) entails that \( c \) is strongly normalizing. Additionally, if \( \text{Erase}(c) \Rightarrow_\beta c' \) then there must be a command \( c'' \) such that \( c \Rightarrow_\beta c'' \) and \( c'' = \text{Erase}(c') \) since the type-erased reduction rules are more limited than the ones for non-erased commands. Thus, if a typed command \( c \) is strongly normalizing, then \( \text{Erase}(c) \) must be as well.

Theorem 1. If \( c : \Gamma \Rightarrow_\Delta \mathcal{H} \) and \( (\Gamma \Rightarrow_\mathcal{H} \Delta) \Rightarrow \text{seq} \), then \( c \) is strongly normalizing in the \( \mu \lambda \)-calculus. Furthermore, \( \text{Erase}(c) \) is strongly normalizing in the type-erased \( \mu \lambda \)-calculus.

B. Pre-types

A pre-type is a pair of a set of terms and a set of co-terms. If \( A \) is a pre-type, we write \( v \in A \) or \( e \in A \) to indicate that a given (co)-term is an element of that pre-type.

Since we are interested in strong normalization, we carve out the strongly normalizing commands and (co-)terms. The set \( \bot \) of commands consists of all those commands which are strongly normalizing. \( \bot \) is closed under reduction. The pre-type \( W \) of well-behaved (co-)terms is defined as containing all the strongly normalizing (co-)terms. Note that since all reducible of strongly normalizing commands and (co-)terms are themselves strongly normalizing, \( \bot \) and \( W \) are forward closed: if \( c \in \bot \) and \( c \Rightarrow_\beta c' \) then \( c' \in \bot \), if \( v \in W \) and \( v \Rightarrow v' \) then \( v' \in W \), and if \( e \in W \) and \( e \Rightarrow e' \) then \( e' \in W \).

There are two fundamental orderings of pre-types. Given two pre-types \( A \) and \( B \), \( A \) refines \( B \), written \( A \subseteq B \), if and only if

\[
\begin{align*}
v & \in A \iff v \in B \\
e & \in A \iff e \in B
\end{align*}
\]

By contrast, \( A \) is a subtype of \( B \), written \( A \subseteq B \), if and only if

\[
\begin{align*}
v & \in A \implies v \in B \\
e & \in A \implies e \in B
\end{align*}
\]

Note that these two orderings form a complete lattice on the set of strongly-normalizing pre-types. In terms of refinement, \((0,0)\) is the smallest element that refines every pre-type, and \(W\) serves as a largest element that every strongly-normalizing pre-type refines.

In terms of subtyping, \((0,\{v \in W\})\) is the smallest element that is a sub-type of every pre-type, and \((\{v \in W\},0)\) is the largest element that is a super-type of every pre-type. In general, we use square operations to refer to operations of the refinement lattice, and triangular operations to refer to the subtyping lattice. In particular given the semantic types \( \mathcal{A} = (A^+, A^-) \) and \( \mathcal{B} = (B^+, B^-) \), where \( A^+ \) and \( B^- \) contain the terms of \( A \) and \( B \) and dually for \( A^- \) and \( B^+ \), we have the joins and meets of both orders:

\[
\begin{align*}
(A^+, A^-) \cup (B^+, B^-) & \triangleq (A^+ \cup B^+, A^- \cup B^-) \\
(A^+, A^-) \cap (B^+, B^-) & \triangleq (A^+ \cap B^+, A^- \cap B^-) \\
(A^+, A^-) \setminus (B^+, B^-) & \triangleq (A^+ \setminus B^+, A^- \setminus B^-) \\
(A^+, A^-) \setminus (B^+, B^-) & \triangleq (A^+ \setminus B^+, A^- \setminus B^-)
\end{align*}
\]

Lemma 3 (Pre-type lattice). Each of the \( \leq \) and \( \subseteq \) orderings on the set of pre-types form a complete lattice: that is, they have all joins and meets.

Proof. The set of subsets of a set is a complete lattice ordered by \( \subseteq \) with the usual \( \cup \) and \( \cap \) operations. Further, the dual of a complete lattice is itself a complete lattice, and the product of two complete lattices is a complete lattice. The case of \( \subseteq \) is the product of the two subset lattices; the case of \( \leq \) is the product of the two subset lattices where one is dualized.

\( \square \)

Lemma 4. \( \cup \) is monotonic (in both arguments) with respect to \( \leq \) and is commutative and associative.

Proof. That \( \cup \) is commutative and associative follows immediately from its definition. The interesting fact is that, for any pre-types \( A = (A^+, A^-) \), \( B = (B^+, B^-) \), and \( C = (C^+, C^-) \) if \( A \subseteq B \) then we have

\[
\begin{align*}
A \cup C & \triangleq (A^+ \cup C^+, A^- \cup C^-) \\
& \leq (B^+ \cup C^+, B^- \cup C^-) \\
& = B \cup C
\end{align*}
\]

since \( A^+ \subseteq B^+ \) means \( A^+ \cup C^+ \subseteq B^+ \cup C^+ \) and \( B^- \subseteq A^- \) means \( B^- \cup C^- \subseteq A^- \cup C^- \).

The most basic operation on pre-types is the orthogonal \( A^\perp \):

\[
\begin{align*}
v & \in A^\perp \iff v \in W \land \forall e \in A, \langle v,e \rangle \in \bot \\
e & \in A^\perp \iff e \in W \land \forall v \in A, \langle v,e \rangle \in \bot
\end{align*}
\]

We say that a pre-type \( A \) is orthogonally sound if and only if \( A \subseteq A^\perp \). In other words, for all \( A \subseteq W \) and for all \( v, e \in A \), \( \langle v,e \rangle \in \bot \). Furthermore, a pre-type \( A \) is orthogonally complete
Lemma 6. Therefore, \( A \subseteq B \) implies \( \text{Opp}(A) \subseteq \text{Opp}(B) \).

Proof. Suppose \( v \in \text{Opp}(A) \), so we know that \( v \in D \) and for all \( e \in A, P(v, e) \). Then, given any \( e \in B \), we know that \( e \in A \) because \( A \subseteq B \), and so \( P(v, e) \). Therefore, \( v \in \text{Opp}(B) \) as well.

Suppose \( e \in \text{Opp}(B) \), so we know that \( e \in D \) and for all \( B, Q(v, e) \). Then, given any \( v \in A \), we know that \( v \in B \) because \( A \subseteq B \), and so \( Q(v, e) \). Therefore, \( e \in \text{Opp}(A) \) as well.

Lemma 7 (Double Negation Introduction). For any symmetric negation operation \( \text{Opp} \) inside \( D, A \subseteq D \) implies \( A \subseteq \text{Opp}(A) \).

Proof. For any \( v \in A \) and \( e \in \text{Opp}(A) \), \( Q(v, e) \) by definition of \( \text{Opp}(A) \), so \( P(v, e) \) as well since \( \text{Opp} \) is symmetric. Therefore, \( A \subseteq D \) implies that \( v \in D \), so that \( v \in \text{Opp}(A) \). The case of \( e \in A \) implies \( e \in \text{Opp}(\text{Opp}(A)) \) is dual.

Lemma 8 (Triple Negation Elimination). For any symmetric negation operation \( \text{Opp} \) inside \( D, A \subseteq D \) implies \( \text{Opp}(\text{Opp}(A)) = \text{Opp}(A) \).

Proof. Note that \( \text{Opp}(A) \subseteq D \) because \( \text{Opp} \) is a negation operation inside \( D \). Therefore, by double negation introduction (Lemma 7), \( \text{Opp}(A) \subseteq \text{Opp}(\text{Opp}(A)) \). Additionally, because \( A \subseteq D \), we know that \( A \subseteq \text{Opp}(\text{Opp}(A)) \) by double negation introduction (Lemma 7), and so \( \text{Opp}(\text{Opp}(A)) \subseteq \text{Opp}(A) \) by contrapositive (Lemma 6). Therefore, \( \text{Opp}(\text{Opp}(A)) = \text{Opp}(A) \).

We can rephrase the pre-type \( W \) of well-behaved (co-)terms solely in terms of orthogonality and (co-)variables. This ensures to us that any other pre-type \( A \subseteq W \) which is orthogonally complete must contain (co-)variables.

Lemma 9. 1. \( v \) is strongly normalizing iff \( \langle v \alpha \rangle \) is.

2. \( e \) is strongly normalizing iff \( \langle x \rangle e \) is.

Proof. 1. Since \( v \) is a sub-term of \( \langle v \alpha \rangle \), strong normalization of \( \langle v \alpha \rangle \) implies strong normalization of \( v \).

Going the other way, we show that every reduction of \( \langle v \alpha \rangle \), except for possibly one top-level \( \mu E \) reduction, can be traced by \( v \) as well. We proceed to show that \( \langle v \alpha \rangle \) is strongly normalizing because all of its reducts are by well-founded induction on \( |\alpha| \):

- Suppose \( v = \mu \beta.c, \) so that we have the top-level \( \mu E \) reduction:

\[
(\mu \beta.c.e) \rightarrow_{\mu E} c(\alpha/\beta)
\]

Furthermore, we know \( \mu \alpha.c(\alpha/\beta) \) is strongly normalizing since it is \( \alpha \)-equivalent to the strongly normalizing \( \mu \beta.c, \) which means that \( c(\alpha/\beta) \) is also strongly normalizing since it is a sub-command of \( \mu \alpha.c(\alpha/\beta) \).

- Suppose we have some other reduction internal to \( v \), so that:

\[
\langle v \alpha \rangle \rightarrow \langle v' \alpha \rangle
\]

Then we know that \( v \rightarrow v' \) so \( |v'| < |v| \). Therefore, by the inductive hypothesis, we get that \( \langle v' \alpha \rangle \) is strongly normalizing.

Since every reduct of \( \langle v \alpha \rangle \) is strongly normalizing, then \( \langle v \alpha \rangle \) is also strongly normalizing.

2. Analogous to the above by duality.

Corollary 1. \( W = V a r^{-}\).

Proof. Note that \( V a r^{-} \) is the semantic type:

\[
v \in V a r^{-} \iff v \in W \land \forall \alpha \in V a r.\langle v \alpha \rangle \in I
\]

And so \( v, e \in W \) if and only if \( v, e \in V a r^{-} \) by the above Lemma 2.

Corollary 2. If \( A^{-} \subseteq A \subseteq W \) then \( V a r \subseteq A \).

Proof. Using the above Corollary 1 we conclude by double negation introduction (Lemma 7) and contrapositive (Lemma 8): \( V a r \subseteq V a r^{-} = W^{-} \subseteq A^{-} \subseteq A \).

C. Head Reduction

There are two important properties for the chosen strategy \( S \) needed for the proof of strong normalization. First, we say a strategy stable if and only if:

1. (co-)values are closed under reduction and substitution, and
2. non-(co-)values are closed under substitution and \( \varsigma \) reduction.

Second, we say a strategy is focalizing if and only if all:
1. (co-)variables,  
2. structures built from (co-)values, and  
3. case abstractions (recursive or non-recursive) are considered (co-)values. The focalizing property of strategies corresponds to focalization in logic $\beta$—each criterion for a focalizing strategy comes from an inference rule for typing a (co-)value in focus. For the remainder of this proof, we assume that the chosen strategy $S$ is both stable and focalizing.  

The head reduction relation describes only those reductions that happen at the top of a command and is given in Figure 2. These reductions are charged. Neutrally charged reductions $\rightarrow_\sigma$ require cooperation of the term and co-term, like in the $\beta$ rules. Positively charged reductions $\rightarrow_\pi$ allow the term to take over the command in order to simplify itself. Negatively charged reductions $\rightarrow_\rho$ are allowed the co-term to take over the command. There are several useful facts that are immediately apparent about this definition of head reduction.  

1. Every step of head reduction is simulated by several steps of the general reduction theory: $c \rightarrow_+ \rho, - c \rightarrow'$.  
2. Head reduction only occurs when one side of the command is a (co-)value: if $(v|\langle c \rangle) \rightarrow_\pi$ then $c$ is a co-value, if $(v|c) \rightarrow_\rho$ then $v$ is a value, and if $(v|\langle c \rangle) \rightarrow_\pi$ then both $v$ and $c$ are (co)-values. This last point about neutral head reductions implying both sides of the command is a (co-)value follows from the assumption that the chosen strategy $S$ is focalizing.  
3. When taken on their own, each of the charged head reduction relations, $\rightarrow_\pi$, $\rightarrow_\pi$, and $\rightarrow_\rho$ are deterministic regardless of the chosen strategy, although the combined $\rightarrow_+\rho / \rightarrow_+\pi$ head reduction relation may be non-deterministic. Additionally, the combined $\rightarrow_+\rho$ and $\rightarrow_+\pi$ reduction relations are deterministic as well.  

Furthermore, head reduction steps commute with the internal reductions inside either side of the command.

**Lemma 10.**  
1. If $v \rightarrow v'$ and $(v|c) \rightarrow_\pi c$ then there is a $c'$ such that $(v'|c') \rightarrow_\pi c' \cdots c \rightarrow' c'$.  
2. If $c \rightarrow e'$ and $(v|c) \rightarrow_\pi c$ then there is a $c'$ such that $(v|e') \rightarrow_\pi c'$ and $c \rightarrow' \cdots c'$.  
3. If $v \rightarrow v'$ and $(v|c) \rightarrow_\pi c$ then there is a $c'$ such that $(v'|c) \rightarrow_\pi c'$ and $c \rightarrow' \cdots c'$.  
4. If $c \rightarrow e'$ and $(v|c) \rightarrow_\pi c$ then there is a $c'$ such that $(v|e') \rightarrow_\pi c'$ and $c \rightarrow' \cdots c'$.  
5. If $v \rightarrow v'$ then either  
   (a) $\forall c$, such that $(v|c) \rightarrow_\pi c$ we have $c \rightarrow (v'|c)$, or  
   (b) $\forall c$, such that $(v|c) \rightarrow_\pi c$ there exists a $c'$ such that $(v|c') \rightarrow_\pi c'$ and $c \rightarrow c'$.  
6. If $c \rightarrow e'$ then either  
   (a) $\forall c$, such that $(v|c) \rightarrow_\pi c$ we have $c \rightarrow (v'|c)$ or  
   (b) $\forall c$, such that $(v|c) \rightarrow_\pi c$ there exists a $c'$ such that $(v|e') \rightarrow_\pi c'$ and $c \rightarrow c'$.  

**Proof.** By cases on the possible reductions. Note that for statements 1-4, there are no critical pairs between neutral head reductions and general reductions, because (co-)values are closed under reduction by the stability of $S$. Additionally, for the other statements, the fact that (co-)values are closed under reduction cuts out many critical pairs. For statement 5, we illustrate the remaining interesting critical pairs:

- $\mu\alpha.(v|\alpha) \rightarrow v$. Note that this is equivalent in the only possible head reduction in the command $(\mu\alpha.(v|\alpha)|E) \rightarrow_+ (v|E)$, so we have case (a).

- $C^S[v] \rightarrow \mu\alpha.(v|\mu\alpha.(\langle C^S[v]|E\rangle))$. Given that $(\langle C^S[v]|E\rangle) \rightarrow_+ (v|\mu\alpha.(\langle C^S[v]|E\rangle)$ we have case (b) where $c'$ is equal to the same $(v|\mu\alpha.(\langle C^S[v]|E\rangle)$ and  
  $(\mu\alpha.(v|\mu\alpha.(\langle C^S[v]|E\rangle))|E) \rightarrow_+ (v|\mu\alpha.(\langle C^S[v]|E\rangle)$

- $C^S[e] \rightarrow \mu\alpha.(\mu\beta.(\langle C^S[e]|\alpha|E\rangle))$. Analogous to a previous case by duality.

- $C^S[v] \rightarrow C^S[V]$. Given that $(\langle C^S[v]|E\rangle) \rightarrow_+ (v|\mu\alpha.(\langle C^S[v]|E\rangle)$ we have case (a) where $(v|\mu\alpha.(\langle C^S[v]|E\rangle) \rightarrow (v|\mu\alpha.(\langle C^S[v]|E\rangle) \rightarrow (\langle C^S[v]|E\rangle)$

- $C^S[e] \rightarrow C^S[E]$. Analogous to a previous case by duality.

- $V[N/i] \rightarrow \mu\alpha.(Rise^{\leq N}[\alpha]|x|c(M/j, V/x))$ where $V = \mu\alpha.(Rise^{\leq N}[\alpha]|x|c(x)). Note that this is equivalent to the only possible head reduction in the command $(v[N/i]|E)$, so we have case (a).

For statement 6, commutation follows analogously by duality.  

A term is simple if it never causes a positive head reduction. A co-term is simple if it never causes a negative head reduction. A pre-type is simple if all its (co-)items are. Of note, observe that because our chosen strategy is assumed to be focalizing, all simple (co-)terms are (co-)values: a simple (co-)term can either take a neutral head reduction step, in which case it must be a (co-)value, or it is a variable and doesn’t actively participate in any reduction, so it must be a (co-)value by the assumption that the strategy is focalizing. Furthermore, the set of simple (co-)terms is closed under reduction because (co-)values are closed under reduction (since the strategy is stable). Also of note is the fact that all non-simple (co-)terms (that is, the ones which can cause a positive or negative head reduction) can never be part of a neutral reduction.

We define the operation $\text{Simp}(A)$ as containing all the (co-)terms of $A$ which are simple. There is a simplified version of the orthogonality operation, $(-)^{\perp\perp}$, that operates in the simplified version of well-behaved (co-)terms $W$. More specifically, the $(-)^{\perp\perp}$ is defined as:

$$A^{\perp\perp} \triangleq \text{Simp}(A^{\perp})$$

Note that $(-)^{\perp\perp}$ is a symmetric negation operation inside $\text{Simp}(W)$, so all the basic properties of monotonicity, contrapositive, double negation introduction, and triple negation elimination apply.

**D. Reducibility Candidates**

To define the the set of reducibility candidates, we have to determine the pre-types are sufficiently saturated so that they contain enough (co-)terms such that the general typing rules are sound, without invalidating the $\text{Cut}$ rule. For example, we need to be sure that reducibility candidates contain the necessary $\mu$- and $\mu$-abstractions according to $\text{Act}$ and $\text{CoAct}$. We characterize this saturation in terms of head reduction, using the $\text{Head}(-)$ operation on pre-types defined as:

$$v \in \text{Head}(A) \iff v \in W \land \forall E \in A. (v|E) \rightarrow_+ c \in \bot$$

$$e \in \text{Head}(A) \iff e \in W \land \forall V \in A. (V|e) \rightarrow_+ c \in \bot$$

Note that $\text{Head}(-)$ is a (non-symmetric) negation operation inside $W$, so the monotonicity and contrapositive properties apply.

We now define reducibility candidates as all orthogonally sound and complete pre-types that contain their own $\text{Head}$:
Lemma 12 of its (co-)values. This follows from the fact that the \( \mu \)-reducibility candidates is written reducibility candidate. Proof.

Lemma 11. 2. If \( A \) is a reducibility candidate and for all \( V \in A, c[V/x] \in \perp \), then \( \mu A.c \in A \).

Proof. • Since \( A \) is a reducibility candidate, we know that \( Head(A) \subseteq A \). Observe that for all \( E \in A \),

\[ \{\mu A.c\} E \rightarrow c\{E/\alpha\} \subseteq \perp \]

by assumption. Therefore, \( \mu A.c \in Head(A) \subseteq A \).

• Analogous to the previous statement by duality. \( \square \)

Additionally, a reducibility candidate contains all the structures built from (co-)values of other reducibility candidates, then it must contain the general constructs built from (co-)terms as well. This follows from the fact that all the lifting rules are implemented as non-neutrally charged head reductions.

Lemma 12 (Unfocalization). 1. If \( A, B, C \) are reducibility candidates and for all \( \overrightarrow{V} \in B, E \in C \), \( K^B(\overrightarrow{V}, \overrightarrow{E}) \in A \), then for all \( \overrightarrow{v} \in B, e \in C \), \( K^B(\overrightarrow{v}, \overrightarrow{e}) \in A \).

2. If \( A, B, C \) are reducibility candidates and for all \( \overrightarrow{V} \in B, E \in C \), \( H^B[\overrightarrow{v}, \overrightarrow{e}] \in A \), then for all \( \overrightarrow{v} \in B, e \in C \), \( H^B[\overrightarrow{v}, \overrightarrow{e}] \in A \).

Proof. • We proceed by right-to-left induction on the immediate non-(co-)value sub-(co-)terms \( \overrightarrow{v}, \overrightarrow{e} \). Note that because \( A \) is a reducibility candidate, \( Head(A) \subseteq A \), so it suffices to show that \( K^B(\overrightarrow{v}, \overrightarrow{e}) \in Head(A) \).

• All of \( \overrightarrow{v}, \overrightarrow{e} \) are (co-)values; \( K^B(\overrightarrow{v}, \overrightarrow{e}) \in A \) by assumption.

• All of \( \overrightarrow{v}, \overrightarrow{e} \) are co-values, and \( v_i \) is the right-most non-value in \( \overrightarrow{v} = v_i, \overrightarrow{v}, \overrightarrow{V} \); Observe that for any \( E \in A \), we have the head lifting reduction

\[ \langle K^B(\overrightarrow{v}, \overrightarrow{e}) \rangle E \rightarrow \langle v_i, \mu A.\langle K^B(\overrightarrow{v}, \overrightarrow{e}, x, \overrightarrow{V}) \rangle \rangle \subseteq \perp \]

because \( \mu A \cdot \langle K^B(\overrightarrow{v}, \overrightarrow{e}, x, \overrightarrow{V}) \rangle \in B \), by strong activation (Lemma 1[1]) and the inductive hypothesis. Therefore, \( K^B(\overrightarrow{v}, \overrightarrow{e}) \in Head(A) \).

Definition 1 (Reducibility candidates). A semantic type, \( A \), is a reducibility candidate iff \( A = A^\perp \) and \( Head(A) \subseteq A \). The set of reducibility candidates is written \( CR \).

We can show that a reducibility candidate must contain all the \( \mu \)- and \( \mu^\perp \)- abstractions that are well-behaved when paired with any of its (co-)values. This follows from the fact that the \( \mu E \) and \( \mu^\perp V \) reductions are non-neutrally charged head reductions.

Lemma 11 (Strong activation). 1. If \( A \) is a reducibility candidate and for all \( \alpha \subseteq A, c[V/x] \in \perp \), then \( \mu A.c \in A \).

Proof. • Since \( A \) is a reducibility candidate, we know that \( Head(A) \subseteq A \). Observe that for all \( E \in A \),

\[ \{\mu A.c\} E \rightarrow c\{E/\alpha\} \subseteq \perp \]

by assumption. Therefore, \( \mu A.c \in Head(A) \subseteq A \).

• Analogous to the previous statement by duality. \( \square \)

Fig. 2. Head Reductions.

\[ e_i \text{ is the right-most non-co-value in } \overrightarrow{v} = \overrightarrow{v}, v_i, \overrightarrow{V} : \text{Observe that for any } E \in A, \text{ we have the head lifting reduction } \]

\[ \langle K^B(\overrightarrow{v}, \overrightarrow{e}) \rangle E \rightarrow \langle v_i, \mu A.\langle K^B(\overrightarrow{v}, \overrightarrow{e}, x, \overrightarrow{V}) \rangle \rangle \subseteq \perp \]

because \( \mu A \cdot \langle K^B(\overrightarrow{v}, \overrightarrow{e}, x, \overrightarrow{V}) \rangle \in B \), by strong activation (Lemma 1[1]) and the inductive hypothesis. Therefore, \( K^B(\overrightarrow{v}, \overrightarrow{e}) \in Head(A) \).

We now show how to generate a full-fledged reducibility candidate from any simple core definition for a type using a variant of the symmetric candidates technique of Barbanera and Berardi [1].

We define the saturation function \( Sat \), which determines all the (co-)terms that are strongly normalizing with all the (co)-values of \( A \) either now, or one step by head reduction in the future:

\[ v \in Sat(A) \iff v \in W \]

\[ \land \forall E \in A, \langle v \rangle E \in \perp \land \langle v \rangle E \rightarrow c \in \perp \]

\[ e \in Sat(A) \iff e \in W \]

\[ \land \forall V \in A, \langle V \rangle e \in \perp \land \langle V \rangle e \rightarrow c \in \perp \]

So starting with a seed \( C \), we can use \( Sat(-) \), to grow a full-fledged reducibility candidate by iteratively saturating it, one \( Step \) at a time:

\[ Step_C(A) = C \cup Sat(A) \]

This process eventually finishes, due to the fact that preserves the subtyping order of the lattice of pre-types (Lemma 3).

Corollary 3.

\[ A \leq B \implies Step_C(A) \leq Step_C(B) \]

Proof. By monotonicity (Lemma 5) because \( Step_C \) is a negation operation inside \( W \), for any pre-type \( C \). \( \square \)

Corollary 4. For any \( C \), there is a fixed point \( A = Step_C(A) \).

The fact that the fixed point to the \( Step \) function exists lets us define a function that saturates any pre-type \( C \).

Corollary 5. There exists a function on pre-types \( R(-) \) such that \( R(C) = Step_C(R(C)) \).

Proof. By Corollary 8, there is a fixed to the function \( Step_C \), so we take \( R(C) \) to be the least one. \( \square \)

Note that this \( R(-) \) operation gives us something that is almost a reducibility candidate. For any \( Sat(A) \notin A \subseteq W \) we have that:

\[ Head(A) \subseteq Sat(A) \subseteq A \]

\[ A^\perp \subseteq Val(A)^\perp \subseteq Sat(A) \subseteq A \]
So all that is remaining is to show that, for certain well-behaved choices for $C$, $R(C) \subseteq R(C)^+$. In particular, we will find that $R(C)$ is guaranteed to be orthogonally sound, and thus a reducibility candidate, whenever $C$ is in $\text{TypeCore}$.

**Definition 2.** The set $\text{TypeCore}$ consists of pre-types $C$ such that $C \subseteq C^{+}$ and $C$ is forward closed.

Note all pre-types $C$ in $\text{TypeCore}$ are orthogonally sound because $C \subseteq C^{+} \subseteq C$. Furthermore, since $(-)^+$ is itself a negation operation in $\text{Simp}(W)$, $C \subseteq C^{+} \subseteq \text{Simp}(W)$ so it contains only simple (co-)terms, which must be (co-)values because the chosen strategy $S$ is assumed to be focalizing.

**Lemma 13** (Head orthogonality). Suppose that $A \subseteq W$ is forward closed and for all $v, e \in A$, $(v|\epsilon) \rightarrow_{+, o, \neg} e$ implies $e \in \perp$. Then $A \subseteq A^+$. 

**Proof.** Note that a command $c$ is in $\perp$, meaning it is strongly normalizing, if and only if all reducts of $c$ are in $\perp$. Since $A \subseteq W$, every (co-)term in $A$ is strongly normalizing, so let $|v|$ and $|e|$ be the lengths of the longest reduction sequence from any $v, e \in A$, respectively. We now proceed to show that for any $v, e \in A$, $(v|\epsilon) \in \perp$ because all of its reducts are in $\perp$, by induction on $|v| + |e|$.

- $(v|\epsilon) \rightarrow_{+, o, \neg} c$: we know $c \in \perp$ by assumption.
- $(v|\epsilon) \rightarrow (v'|\epsilon)$ because $v \rightarrow v'$: then $v' \in A$ because $A$ is forward closed, and $|v'| < |v|$. Furthermore, we have that all head reducts of $(v'|\epsilon)$ are in $\perp$ by Lemma $[10]$ and the fact that $\perp$ is closed under reduction. Therefore, $(v'|\epsilon) \in \perp$ by the inductive hypothesis.
- $(v|\epsilon) \rightarrow (v|\epsilon')$ because $e \rightarrow e'$: analogous to the previous case by duality.

**Lemma 14.** If $v, e \in W$ and $(v|\epsilon) \rightarrow_0 c \in \perp$, then $(v|\epsilon) \in \perp$.

**Proof.** Consider the pre-type $A$ defined as

$$v' \in A \iff v \rightarrow v'$$

$$e' \in A \iff e \rightarrow e'$$

Note that $v$ and $e$ are in $A$. Further, given any $v', e' \in A$. $(v'|\epsilon') \rightarrow c'$ such that $c \rightarrow c'$ by Lemma $[10]$ and induction on their reduction paths, so that $c' \in \perp$ because $\perp$ is closed under reduction. Observe that $A \subseteq W$ and is forward closed by definition. Finally, because $v$ and $e$ are simple, and the set of simple (co-)terms is forward closed, $A$ is simple. From this, we know that for all $v', e' \in A$, $(v'|\epsilon') \rightarrow_{0, o, \neg} c'$ implies that $c' \in \perp$. Thus, $A \subseteq A^+$ by Lemma $[13]$ and so $(v|\epsilon) \in \perp$.

**Lemma 15.** If $C \in \text{TypeCore}$ and $A = \text{Stepc}(A)$, then for all $v, e \in A$, $(v|\epsilon) \rightarrow_{+, o, \neg} c$ implies $c \in \perp$.

**Proof.** Let $v, e \in A = C \sqcup \text{Sat}(A)$. By cases, we have:

- $v, e \in C$: $v \in C$ by the assumption that $C \in \text{TypeCore}$, so $C$ is orthogonally sound.
- $v \in C$, $e \in \text{Sat}(A)$: by $e \in \text{Sat}(A)$ and the fact that $V$ is a value, we know that either $(V|\epsilon) \in \perp$ or $(V|\epsilon) \rightarrow c \in \perp$ for some $c$. In the first case, every reduct of $(V|\epsilon)$ is in $\perp$, including any head reductions, since $\perp$ is forward closed. In the second case, we know in simple $A$ that $C \in \text{TypeCore}$ and $e$ is non-simple so it cannot take part of a neutral head reduction. This means $(V|\epsilon) \rightarrow_{o, \neg} a$ and $(V|\epsilon) \rightarrow e'$ implies $e' \in \perp$ because $\rightarrow_o$ is deterministic so $e' = c \in \perp$. Therefore, $(V|\epsilon) \rightarrow_{+, o, \neg} c$ implies $c \in \perp$.
- $v \in \text{Sat}(A)$, $e \in C$: analogous to the previous case by duality.

**Lemma 16.** $Sat(A)$ is forward closed.

**Proof.** Suppose $v \in Sat(A)$ and $v \rightarrow v'$. Let $E \in A$, so that $v \in Sat(A)$ implies that either $v \in E$ or $(v|\epsilon) \rightarrow c$. In the first case we know that $(v|\epsilon) \in \perp$, so $(v|\epsilon) \rightarrow (v'|\epsilon)$. By induction on the commutation of internal and head reduction (Lemma $[10]$), and the fact that $\perp$ is forward closed, we have one of two cases:

- $(v'|\epsilon) \rightarrow c' \in \perp$, or
- $(v'|\epsilon) \rightarrow_{+, o, \neg} c' \in \perp$

Therefore, since in any case $(v'|\epsilon) \in \perp$ or $(v'|\epsilon) \rightarrow c' \in \perp$, then $v \in Sat(A)$.

Suppose $v \in Sat(A)$ and $e \rightarrow e'$. Analogous to the previous case by duality.

**Corollary 6.** Given $A = \text{Stepc}(A)$, $A$ is forward closed when $C \in \text{TypeCore}$.

**Proof.** Any $v, e \in A$ is in either $C$ or $Sat(A)$, so it follows by Lemma $[16]$ and definition of $\text{TypeCore}$.

**Corollary 7.** For any $C \in \text{TypeCore}$, $R(C)$ is a reducibility candidate such that $C \subseteq R(C)$.

**Proof.** By Lemma $[5]$ $R(C) = \text{Stepc}(R(C))$, so by definition we know that $C \subseteq R(C)$. Furthermore, we know that both $\text{Head}(R(C))$ and $R(C)^+ \cong \text{Refine}(R(C))$, which in turn refines $R(C)$. Furthermore, by definition of $\text{TypeCore}$ and $Sat$, we know that $R(C) \subseteq W$. Finally, by Corollary $[6]$ and Lemma $[13]$ we know that $R(C)$ is forward closed and all head reducts are in $\perp$, so $R(C) \subseteq R(C)^+$ by Lemma $[13]$. Therefore, $R(C)$ is a reducibility candidate.

**E. The Model**

We use the notation $\mathbb{O}$ to denote a set of ordinals that is sufficiently large for our purposes: $\mathbb{O}$ contains all ordinals less than or equal to $\omega \times 2$. However, any larger set of ordinals would serve just as well.

**Definition 3.** $PK$, or pre-kinds is the smallest set such that

1. $\mathbb{O} \in PK$.
2. $CR \in PK$, and
3. $\forall a, b \in PK, \{ f : a \rightarrow b \} \in PK$.

The universe $U$ is defined as the union of $PK$

$$U = \bigcup_{a \in \mathbb{O}} a$$

**Definition 4.** A semantic kind $\kappa \in \text{SKind}$ is a pair $(S, R)$ where $S \in \mathcal{P}(U)$ and $R \in \mathcal{P}(\text{Type} \times S)$. We say that $S$ is the domain of the semantic kind and $R$ is the syntactic-semantic relation.
In order to define the meaning of types as a generic fold parameterized by an interpretation operation, we define an intermediate stage between syntactic and semantic types called hybrid types.

**Definition 5.** The set of hybrid types (HType) consists of

1. type variables,
2. $F(\vec{A})$ where $\vec{A} \in \mathcal{U}$ and $F \in \mathcal{F}$, 
3. the constants $0$ and $\infty$, 
4. $+1(M)$ where $M \in \mathcal{U}$.

We define the interpretation functions on types and kinds using the same notation $[\cdot]$, which is disambiguated by context. Intuitively, this interpretation is parameterized by an operation, $\mathcal{H}$, which does the real hard work of assigning a semantic meaning to syntactic types. The purpose of abstracting out $\mathcal{H}$ is to break up the many recursions involved with interpreting recursive (co-)data types, so that the outermost interpretation is defined structurally over just the syntax of types, without concern on how the different types are related to one another.

$[k \in \text{Kind}] : (\text{HType} \to \mathcal{U}) \to \text{SKind}$

$\llbracket A \in \text{Type} \rrbracket : (\text{HType} \to \mathcal{U}) \to \mathcal{U}$

The interpretation of types and kinds is given by structural induction over the syntax. Technically, we consider the interpretation giving the two components of semantic kinds, the domain and syntactic-semantic relation, as two separate functions. The interpretation of types and the kind of domains are defined by mutual induction over the structure of syntactic types and kinds, and the interpretation of the syntactic-semantic relation of kinds is defined by structural induction on kinds, and using the previous definition.

$[a]_{\mathcal{H}} \triangleq \mathcal{H}(a)$

$[0]_{\mathcal{H}} \triangleq \mathcal{H}(0)$

$[\infty]_{\mathcal{H}} \triangleq \mathcal{H}(\infty)$

$[N + 1]_{\mathcal{H}} \triangleq \mathcal{H}(+1([N]_{\mathcal{H}}))$

$[F(\vec{A})]_{\mathcal{H}} \triangleq \mathcal{H}(F([\vec{A}]_{\mathcal{H}}))$

$[\lambda a : k. B]_{\mathcal{H}} \triangleq \lambda \lambda_{\mathcal{H}} \in \pi_{2}([k]_{\mathcal{H}}).B_{\mathcal{H}}(A/a)$

$[A B]_{\mathcal{H}} \triangleq [A]_{\mathcal{H}}([B]_{\mathcal{H}})$

$\pi_{1}([k^+]_{\mathcal{H}}) \triangleq CR$

$\pi_{1}([k_1 \rightarrow k_2]_{\mathcal{H}}) \triangleq \pi_{1}([k_2]_{\mathcal{H}})^{+} \pi_{1}([k_1]_{\mathcal{H}})^{-}$

$\pi_{1}([k]_{\mathcal{H}}) \triangleq \{N \in \mathcal{O} \| \exists N \in \text{Type}, N \sim_{\mathcal{H}} N' \}$

$\pi_{1}([\text{Ord}]_{\mathcal{H}}) \triangleq \mathcal{O}$

$\pi_{1}([< N]_{\mathcal{H}}) \triangleq \{M \in \mathcal{O} \| \exists N' \in \text{Type}.N \Rightarrow_{\beta} N' \land M \subset N' \}$

$\pi_{2}([k]_{\mathcal{H}}) \triangleq \text{Type} \times CR$

$\pi_{2}([k_1 \rightarrow k_2]_{\mathcal{H}}) \triangleq \{(A, A) \| [\forall (B, B) \in \pi_{2}([k_2]_{\mathcal{H}}).A, B, (A, B)) \in \pi_{2}([k_1]_{\mathcal{H}})\}$

$\pi_{2}([k]_{\mathcal{H}}) \triangleq \{(M, M) \| [-, M] \sim_{\mathcal{H}} M\}$

$\pi_{2}([\text{Ord}]_{\mathcal{H}}) \triangleq \{(M, M) \| [-, M] \sim_{\mathcal{H}} M\}$

$\pi_{2}([< N]_{\mathcal{H}}) \triangleq \{(M, M) \| [-, M] \sim_{\mathcal{H}} M\}$

Note that the relation $\sim_{\mathcal{H}}$ used in $[[x]]_{\mathcal{H}}$ is defined as the smallest subset of $\text{Type} \times \mathcal{O}$ such that

1. $0 \sim_{\mathcal{H}} \mathcal{H}(0)$ when $\mathcal{H}(0) \in \mathcal{O}$, 
2. for all $N \sim_{\mathcal{H}} N', N + 1 \sim_{\mathcal{H}} \mathcal{H}(+1(N))$ when $\mathcal{H}(+1(N)) \in \mathcal{O}$.

The application $[A]_{\mathcal{H}}([B]_{\mathcal{H}})$ is defined whenever there exists $k_1, k_2 \in PK$ such that $[A]_{\mathcal{H}} \in k_1 \rightarrow k_2$ and $[B]_{\mathcal{H}} \in k_1$, otherwise it is undefined. We will use the shorthand notation $A \in [k]_{\mathcal{H}}$ when $k$ is a kind to indicate that $A$ is an element of $\pi_{1}([k]_{\mathcal{H}})$, and $A \llbracket k \rrbracket \mathcal{H} \mathcal{A}$ when $(A, A)$ is an element of $\pi_{2}([k]_{\mathcal{H}})$.

Also note that we give an interpretation of sorts, as well. The sort of non-erasable kinds, $\mathbb{I}$, is interpreted as the whole set of all semantic kinds, and the sort of erasable kinds, $\Box$, adds the restriction that the syntactic-semantic relation is backwards closed under $\beta$ reduction.

$[\mathbb{I}] \triangleq \text{SKind}$

$[\Box] \triangleq \{\kappa \in \text{SKind} \| \forall (A, A) \in \pi_{2}(\kappa).A' \Rightarrow_{\beta} A \implies (A', A) \in \pi_{2}(\kappa)\}$

The main work of defining types is in giving the core interpretation of type constructors in $\mathcal{F}$:

$\langle\langle A \in \text{HType} \rangle\rangle : (\text{HType} \to \mathcal{U}) \to \text{TypeCore}$

First, we specify when an interpretation operation $\mathcal{H}$ assigns a plausible meaning to the numeric measures used in types.

**Definition 6.** A map $\mathcal{H} : \text{HType} \to \mathcal{U}$ is plausible if

1. $\mathcal{H}(0) \in \mathcal{O}$, 
2. $\mathcal{H}(\infty) \in \mathcal{O}$, 
3. for all $N \in \mathcal{O}, \mathcal{H}(+1(N)) \in \mathcal{O}$, 
4. $\mathcal{H}(0) < \mathcal{H}(\infty)$.
5. for all $N \in \mathcal{O}, N < \mathcal{H}(\infty)$ implies $\mathcal{H}(+1(N)) < \mathcal{H}(\infty)$, and
6. for all $M, N \in \mathcal{O}, M < N$ implies $M \subset N$.

We assume that $\mathcal{F}$ comes with some ordering among hybrid types that is both well-founded but also enough that the core interpretation of types is well-defined with respect to $\langle\langle - \rangle\rangle$. Note that we say two maps $\mathcal{H}, \mathcal{I} : \text{HType} \to \mathcal{U}$ are equivalent up to a hybrid type $h$ (written $\mathcal{H} \equiv_{h} \mathcal{I}$) if and only if for all $g < h \in \text{HType}, \mathcal{H}(g) = \mathcal{I}(g)$ and are defined.

**Definition 7.** The set of type constructors $\mathcal{F}$ is well-founded if there is a partial well order $\llbracket \cdot \rrbracket$ on the set of hybrid types formable from $\mathcal{F}$ such that for any plausible $\mathcal{H}$:

1. Given any $F(\bar{a} : k_1) : k_2 \in \mathcal{F}$ and $\bar{A} \in \llbracket k_1 \rrbracket_{\mathcal{H}}, \langle\langle F(\bar{A}) \rangle\rangle_{\mathcal{H}}$ is defined and in $\llbracket k_2 \rrbracket_{\mathcal{H}}$ whenever for all $G(b : k_3) : k_4 \in \mathcal{F}$ and $\bar{B} \in \llbracket k_3 \rrbracket_{\mathcal{H}}$, $\mathcal{H}(\mathcal{F}(\bar{A})) \subseteq \mathcal{H}(\mathcal{F}(\bar{B}))$ is defined and in $\llbracket k_4 \rrbracket_{\mathcal{H}}$.
2. For any other plausible $\mathcal{I}$ and for any $h \in \text{HType}$ such that $\mathcal{H} \equiv_{h} \mathcal{I}, \langle\langle h \rangle\rangle_{\mathcal{I}} = \langle\langle h \rangle\rangle_{\mathcal{H}}$.

**Definition 8.** The set $\llbracket \mathcal{F} \rrbracket$ consists of partial functions $\text{HType} \to \mathcal{U}$ such that if $\mathcal{H} \equiv_{h} \mathcal{I}$ then

1. for any $F(\bar{a} : k) : k' \in \mathcal{F}$ and any $\bar{A} \in \llbracket k \rrbracket_{\mathcal{H}}, \langle\langle F(\bar{A}) \rangle\rangle_{\mathcal{H}} \subseteq \mathcal{H}(\mathcal{F}(\bar{A})) \subseteq \llbracket k' \rrbracket_{\mathcal{H}}$, and
2. $\mathcal{H}$ is plausible.

**Lemma 17.** Given any well-founded set of type constructors $\mathcal{F}$ and $\mathcal{I} : \text{HType} \to \mathcal{U}$ there exists a $\mathcal{H} \in \llbracket \mathcal{F} \rrbracket$ such that for any $h = 0, \infty, +1(M), a$, we have $\mathcal{H}(h) = \mathcal{I}(h)$. 

We use the notation \( F \) next \( h \) and kind \( F \) for any kind \( F \). This follows from the fact that if \( c \rightarrow c' \), then \( c(\theta) \rightarrow c'(\theta) \), so strong normalization of \( c(\theta) \) implies strong normalization of \( c \). The main point is that filling in types for type variables can only allow for more reductions, either due to filling in the index of an Inflated or Deflate structure or by specifying that the upper bound for a quantified type variable in a case abstraction is inhabited.

By contrast, we interpret \([\Gamma]\) and \([\Delta]\) as just sets of substitutions.

\[\gamma \in [\Gamma]_H \iff \forall x : A \in \Gamma, \tilde{x}(\gamma) \in VP([A]_H)\]
\[\delta \in [\Delta]_H \iff \forall x : A \in \Delta, \tilde{\alpha}(\delta) \in VP([A]_H)\]

Note that \( VP \) is the function on pre-types that keeps only their (co-)values.

**Lemma 21.** 1. If \([\Gamma]\) is defined then it is inhabited by the identity substitution.

2. If \([\Delta]\) is defined then it is inhabited by the identity substitution.

**Proof.** If \([\Gamma]\) is defined then \( \forall x : A \in \Gamma, [A]_H \) is defined and must yield a reducibility candidate since \( VP([A]_H) \) is defined. Since variables inhabit every reducibility candidate (Lemma 2], and are (co-)values (because we assume the chosen strategy is focalizing), the identity substitution inhabits \([\Gamma]\) and (by duality \([\Delta]\)).

Note that we denote the extension of an interpretation operation \( H \) with the interpretation \( \beta \) for a type variable \( b \) as \( H(\beta/b) \), where

\[H(\beta/b)(h) \triangleq \begin{cases} B & \text{if } h = b \\ H(h) & \text{otherwise} \end{cases}\]

**Lemma 22.** If \([B]\_H\) is defined and \([A]\_H([B/H])\) is defined, then \([A]\_H([B/H]) = [A(B/b)]_H\).

**Proof.** By induction on the structure of \( A \).

**F. Interpreting Specific Types**

To define the core meaning following an orthogonality-based methodology, except that instead of attempting to build a reducibility candidate through the double-orthogonal closure, we build a simplified interpretation in TypeCore using the \( \perp \) s negation operation inside \( \text{Simp}(\mathcal{V}) \).

On the one hand, for the hybrid types of data type constructors, \( h \), we build this core definition through some set of focalized constructions, \( Cons(h) \), from the \( \perp \) s s closure such that

\[\forall v \in [h]_H \iff v \in (Cons(h)(0), 0 \topp)\]
[9]
\[\forall e \in [g]_H \iff e \in (Cons(h)(0), 0 \topp)\]

On the other hand, for the hybrid types of co-data type constructors, \( g \), we build this core definition through some set of focalized observations, \( Obs(h) \), from the \( \perp \) s closure such that

\[\forall v \in [g]_H \iff v \in (0, Obs(h)(g))\]
[10]
\[\forall e \in [g]_H \iff e \in (0, Obs(h)(g))\]

Note that in both cases, \([h]\_H = [h]_H\topp\). It follows that this definition is always in TypeCore. Furthermore by double negation introduction (Lemma 2], in the case of hybrid types for data type constructors \( h \), we are ensured that \( \neg Cons(h)(0) \subseteq [h]_H \) whenever \( Cons(h) \) is a set of strongly-normalizing simple values, and in the case of hybrid types for co-data type constructors \( g \), \( \neg (0, Obs(h)(g)) \subseteq [g]_H \) whenever \( Obs(h)(g) \) is a set of strongly-normalizing simple co-values.
We require that for all constructors, the sequent for each of its constructors, replace the (co-)variables

Proof. Because the declaration of a type constructor is given in a linear manner, so that declarations can only refer to the type constructors introduced by previous declarations, which is ensured by their checks for well-formedness. If a (co-)data type declaration refers to a type constructor that hasn’t yet been declared, then the sequents for its constructors can’t be well-formed by the rules we have available. This gives us the main basis for the well-formedness ordering for F: type constructors can only be greater than or equal to other type constructors earlier in the line. The rest of the relations among hybrid types of F are given by the possible recursive nature of the declaration itself.

\section{Non-recursive (Co-)data Declarations}

For any declared data type of the form

\[
\text{data } F(a:k) \text{ where } \\
K : B \vdash \text{data } F(a:k) \text{ such that } \\
\forall K : B \vdash \text{data } F(a) C \in K : B \vdash \text{data } F(a) C \text{.}
\]

We require that for all \(G(d:k') : * \in F, G(\tilde{B}) \prec F(N, \tilde{A})\). This allows us to give a Construction-oriented definition for \(\llangle F(N, \tilde{A}) \rrangle_H\) such that the order of hybrid types satisfies Definition 2. In particular, \(\llangle F(N, \tilde{A}) \rrangle_H\) is defined as:

\[
\llangle F(N, \tilde{A}) \rrangle_H = \{k \text{.} \tilde{d} : \alpha \} \llangle F(\tilde{A}) \rrangle_H \}
\]

\[
\exists K : B \vdash \text{data } F(a) C \in K : B \vdash \text{data } F(a) C \text{.}
\]

\[
\exists \theta [d : k] \llangle F(N, \tilde{A}) \rrangle_H \llangle F(A) \rrangle_H \}
\]

Lemma 23. If \(F\) is well-formed, \(H \in \mathcal{F}\) and the declaration of \(F\) is well-formed with respect to \(F\), then \(\llangle F(N, \tilde{A}) \rrangle_H\) is a defined pre-type for all \(N \in [k]'\).

\[\llangle F(N, \tilde{A}) \rrangle_H \]

Proof. Because the declaration of \(F\) is well-formed, we know that the sequent for each of its constructors, \(K : B \vdash \text{data } F(a) C \text{.}\), is also well-formed, \(\llangle F(\tilde{A}) \rrangle_H\). By the soundness of kinding rules (shown later in Theorem 35), we therefore know that each sequent is soundly represented in the model, \([x : \tilde{B}] \llangle \theta, \gamma, \delta \rrangle \llangle F(A) \rrangle_H\}. Therefore, \(\llangle F(N, \tilde{A}) \rrangle_H\) is a defined pre-type. \(\square\)

As a corollary, we have that if \(\llangle F(N, \tilde{A}) \rrangle_H\) is defined then \(\llangle F(\tilde{A}) \rrangle_H\) is in \(\text{TypeCore}\).

\[\llangle F(N, \tilde{A}) \rrangle_H\]

Lemma 24. If \(\llangle F(N, \tilde{A}) \rrangle_H\) is defined, then it is a set of simple values in \(\mathcal{W}\).

\[\llangle F(N, \tilde{A}) \rrangle_H\]

Proof. Given any constructor \(v = \tilde{K} \llangle \tilde{\sigma}, \tilde{x} \rrangle (\theta, \gamma, \delta) \in \llangle F(N, \tilde{A}) \rrangle_H\), we know that the substitutions \(\gamma\) and \(\delta\) only ever replace the (co-)variables \(\tilde{\sigma}\) and \(\tilde{x}\) with strongly-normalizing (co-)values, since the substitutions only range over the (co-)values in reducibility candidates when defined (and all reducibility candidates refine \(\mathcal{W}\)). This means that \(v\) is simple, since it cannot take part in a positive head reduction (lifting never applies), and is therefore also a value because we assume our chosen strategy is focalizing. Furthermore, \(v \in \mathcal{W}\) because all of its sub-(co-)terms are strongly normalizing, and no other reductions apply to \(v\). \(\square\)

As a corollary, we have that if \(\llangle F(N, \tilde{A}) \rrangle_H\) is defined and contains a term \(v\), then \(v \in \llangle F(\tilde{A}) \rrangle_H\).

\[\llangle F(N, \tilde{A}) \rrangle_H\]

\section{(Co-)data Declarations by Noetherian Recursion}

Declarations by noetherian recursion is largely analogous to the non-recursive declarations shown previously, and we highlight the difference here. For any declared data type of the form

\[
data F(i : \text{Ord}, a : k) \text{ by noetherian recursion on } i \text{ where } \\
K : B \vdash \text{data } F(i, a) C \text{.}
\]

In addition to the normal ordering of hybrid types for non-recursive declarations, we also have \(F(N, \tilde{A}) \prec F(M, \tilde{B})\) whenever \(N \prec M \in \mathcal{O}\). This allows us to give a Construction-oriented definition for \(\llangle F(N, \tilde{A}) \rrangle_H\) such that \(F\) can refer to itself for smaller choices of the ordinal index. In particular, \(\llangle F(N, \tilde{A}) \rrangle_H\) is defined as:

\[
\llangle F(N, \tilde{A}) \rrangle_H = \{k \text{.} \tilde{d} : \alpha \} \llangle F(\tilde{A}) \rrangle_H \}
\]

\[
\exists K : B \vdash \text{data } F(i, a) C \in K : B \vdash \text{data } F(i, a) C \text{.}
\]

\[
\exists \theta [d : k] \llangle F(N, \tilde{A}) \rrangle_H \llangle F(A) \rrangle_H \}
\]

Lemma 25. If \(\llangle F(\tilde{A}) \rrangle_H\) is defined and \(E\) is the case abstraction \(\mu K[d : \tilde{d}] (\tilde{\sigma}, \tilde{x} : c)\) such that

\[
\forall K : B \vdash \text{data } F(a) C \in K : B \vdash \text{data } F(a) C \text{.}
\]

\[
\exists K[d : \tilde{d}] (\tilde{\sigma}, \tilde{x} : c) \in K[d : \tilde{d}] (\tilde{\sigma}, \tilde{x} : c) \text{.}
\]

\[
\forall \bar{\delta} [d : k] \llangle F(N, \tilde{A}) \rrangle_H \llangle F(A) \rrangle_H \}
\]

then \(E \in \llangle F(\tilde{A}) \rrangle_H\).

Proof. We must show that \(E \in \mathcal{W}\) and \(\langle V \rangle E \in \mathcal{W}\) for every \(V \in \llangle F(\tilde{A}) \rrangle_H\).

- If \(E \in \mathcal{W}\), for each sub-command \(c \in E\) we have two cases, depending on whether or not any kind \(k\) of the quantified type variables \(d : k\) introduced by the pattern has the form \(k' \rightarrow 0 < k'' \rightarrow i < k\):

  - There is a \(k\) such that \(k' \rightarrow 0 < k'' \rightarrow i < k\); then by the caveat on reduction inside a case abstraction, \(c \neq \bot\), so \(c \in \mathcal{W}\).

  - There is no \(k\) such that \(k' \rightarrow 0 < k'' \rightarrow i \leq k\); then by Lemma 21 there is a \(\theta [d : k'] \llangle F(N, \tilde{A}) \rrangle_H \llangle F(A) \rrangle_H \}

  - The identity substitutions inhabit \([x : \tilde{B}]_Z\) and \([\alpha : C]_Z\).

Therefore, we know that \(c(\theta) \in \mathcal{W}\), and so \(c \in \mathcal{W}\).

In either case, all sub-commands of \(E\) are in \(\mathcal{W}\), so \(E \in \mathcal{W}\).

\[\langle V \rangle E \in \mathcal{W}\] for every \(V \in \llangle F(\tilde{A}) \rrangle_H\): note that for every such command, \(\langle V \rangle E \Rightarrow \theta_0 c \in \mathcal{W}\).

Proof. \(\langle V \rangle E \in \mathcal{W}\) for every \(V \in \llangle F(\tilde{A}) \rrangle_H\), and the declaration of \(F\) is well-formed with respect to \(F\), then \(\llangle F(N, \tilde{A}) \rrangle_H\) is a defined pre-type for all \(N \in [k]'\).

The case for non-recursive co-data type declarations follows analogously by duality.
Proof. Because the well-formedness check for \( F \) can assume that \( F \) is already well-formed at smaller indices:

\[
\Theta, i : \text{Ord}, \Theta' \vdash M < i \quad \Theta, i : \text{Ord}, \Theta' \vdash A : k \quad \Theta, i : \text{Ord}, \Theta' \vdash F(M, \vec{A}) : \star
\]

we need to proceed by noetherian induction on the ordinal \( N \).

Besides this difference, the proof follows analogously to Lemma\textsuperscript{25}.

**Lemma 27.** If \( Cons_H(F(N, \vec{A})) \) is defined, then it is a set of simple values in \( \mathcal{W} \).

**Proof.** Analogous to the proof for Lemma\textsuperscript{24}.

**Lemma 28.** If \( Cons_H(F(N, \vec{A})) \) is defined and \( E \) is the case abstraction \( \mu[K^{d,k'}(\vec{\alpha}, \vec{\beta})].c \) such that

\[
\forall K : B \vdash_{d,k'} F(i, \vec{\alpha})[\vec{C}] \in K : B \vdash_{d,k'} F(i, \vec{\alpha})[\vec{C}]'.
\]

\[
\exists \mu K^{d,k'}(\vec{\alpha}, \vec{\beta}).c \in K^{d,k'}(\vec{\alpha}, \vec{\beta}).c.
\]

\[
\forall \theta [d : k'] H(N/\mu, A/\alpha) \quad \forall \gamma \in [x : B]_2, \delta \in [\alpha : C]_2.
\]

\[
c(\theta, \gamma, \delta) \in \bot
\]

then \( E \in \langle \langle F(N, \vec{A}) \rangle \rangle_H \).

**Proof.** Analogous to the proof for Lemma\textsuperscript{25}.

The case for a co-data type defined by noetherian recursion follows analogously by duality.

F3  (Co-)data Declarations by Primitive Recursion

The constructions for a (co-)data type defined by primitive recursion are themselves defined by primitive recursion on the specified index: if the index is \( \mathcal{H}(0) \) then the set of constructors used are those for the 0 case; if the index equals \( \mathcal{H}(+1(N)) \) for some \( N \in [Ix]_H \) then the successor case is chosen. For any declared data type of the form

\[
\text{data } F(i : b, a : k)	ext{ by primitive recursion on } i
\]

where \( i = 0 \)

\[
K : B \vdash_{d,k'} F(0, a)[\vec{C}]'
\]

where \( i = j + 1 \)

\[
K' : B' \vdash_{d,k'} F(0, a)[\vec{C}]
\]

In addition to the normal ordering of hybrid types for non-recurisive declarations, we also have the additional orderings:

\[
F(N, \vec{A}) < F(H(+1(N)), \vec{B}) \text{ for all } N \in [Ix]_H.
\]

We now define the constructions for \( F \) by primitive induction on the index \( N \):

\[
Cons_H(F(H(0), \vec{A})) \triangleq \{ K^{d,k'}(\vec{\alpha}, \vec{\beta})[\gamma, \delta, \theta] | \exists K : B \vdash_{d,k'} F(0, a)[\vec{C}]'.
\]

\[
\exists \theta [d : k'] H(N/\mu, A/\alpha) \quad \exists \gamma \in [x : B]_2, \delta \in [\alpha : C]_2.
\]

where \( H' = H(N/\alpha) \).

\[
Cons_H(F(H(+1(N)), \vec{A})) \triangleq \{ K^{d,k'}(\vec{\alpha}, \vec{\beta})[\gamma, \delta, \theta] | \exists K' : B' \vdash_{d,k'} F(j + 1, a)[\vec{C}]
\]

\[
\exists \theta [d : k'] H(N/\mu, A/\alpha) \quad \exists \gamma \in [x : B]_2, \delta \in [\alpha : C]_2.
\]

where \( H' = H(N/j, A/\alpha) \).

**Lemma 29.** If \( F \) is well-founded, \( H \in [F] \) and the declaration of \( F \) is well-formed with respect to \( F \), then \( Cons_H(F(N, \vec{A})) \) is a defined pre-type for all \( N \in \mathcal{G} \) and \( \vec{A} \in [k]_H \).

**Proof.** Because the well-formedness checks for the successor constructors of \( F \) can assume that \( F \) is already well-formed the previous index \( j \):

\[
\Theta, j : b, \Theta' \vdash A : k
\]

\[
\Theta, j : b, \Theta' \vdash F(j, \vec{A}) : \star
\]

we need to proceed by primitive induction on the ordinal \( N \). Besides this difference, the proof follows analogously to Lemma\textsuperscript{25}.

**Lemma 30.** If \( Cons_H(F(N, \vec{A})) \) is defined, then it is a set of simple values in \( \mathcal{W} \).

**Proof.** Analogous to the proof for Lemma\textsuperscript{24}.

**Lemma 31.** 1. If \( Cons_H(F(H(0), \vec{A})) \) is defined and \( E \) is the case abstraction \( \mu[K^{d,k'}(\vec{\alpha}, \vec{\beta})].c \) such that

\[
\forall K : B \vdash_{d,k'} F(0, a)[\vec{C}]'.
\]

\[
\exists K^{d,k'}(\vec{\alpha}, \vec{\beta}).c \in K^{d,k'}(\vec{\alpha}, \vec{\beta}).c.
\]

\[
\forall \theta [d : k'] H(\mu, A/\alpha) \quad \forall \gamma \in [x : B]_2, \delta \in [\alpha : C]_2.
\]

\[
c(\theta, \gamma, \delta) \in \bot
\]

then \( E \in \langle \langle F(H(0), \vec{A}) \rangle \rangle_H \).

2. If \( Cons_H(F(H(N), \vec{A})) \) is defined and \( E \) is the case abstraction \( \mu[K^{d,k'}(\vec{\alpha}, \vec{\beta})].c \) such that

\[
\forall K' : B' \vdash_{d,k'} F(j + 1, a)[\vec{C}]
\]

\[
\exists K^{d,k'}(\vec{\alpha}, \vec{\beta}).c \in K^{d,k'}(\vec{\alpha}, \vec{\beta}).c.
\]

\[
\forall \theta [d : k'] H(N/\mu, A/\alpha) \quad \forall \gamma \in [x : B']_2, \delta \in [\alpha : C']_2.
\]

\[
c(\theta, \gamma, \delta) \in \bot
\]

then \( E \in \langle \langle F(H(N), \vec{A}) \rangle \rangle_H \).
Proof. Analogous to the proof for Lemma \ref{lem:25} for both the zero and successor cases.

\subsection{F.4 Ascend and Descend}

Interestingly, we do not need to include Ascend and Descend as part of the ordering relation on hybrid types. Taking the view that Descend and Ascend are just user defined types, we can compute their definitions as special cases of the general pattern:

\[ \text{Cons}_H(\text{Ascend}(\mathcal{N}, \mathcal{A})) \]
\[ \triangleq \{ \text{Fall}^M(V) \mid M \not< \mathcal{N} \}_{H} M, V \in \mathcal{A}(\mathcal{N}) \} \]

By Lemma \ref{lem:26} we know that for all well-founded $\mathcal{F}$ and $\mathcal{H} \in \mathcal{F}$, then $\text{Cons}_H(\text{Descend}(\mathcal{N}, \mathcal{A}))$ is defined and is $\langle \text{TypeCore} \rangle$ for any $\mathcal{N} \in \mathcal{O}$ and $\mathcal{A} \in \langle \text{Ord} \rightarrow s \rangle_H$. By Lemma \ref{lem:27} we know that any defined $\text{Cons}_H(\text{Ascend}(\mathcal{N}, \mathcal{A}))$ is included in $\langle \text{Descend}(\mathcal{N}, \mathcal{A}) \rangle_H$. And by Lemma \ref{lem:28} we know that $E = \bar{\mu}[\text{Fall}^\leq N(x).c] \in \langle \text{Descend}(\mathcal{N}, \mathcal{A}) \rangle_H$ whenever

$$ \forall M \not< \mathcal{N} H, M, V \in \mathcal{A}(M).c[V/x, M/j] \in \bot $$

For the special recursive form of case abstraction for Descend, we show that it is included in any family of reducibility candidates indexed by $\mathcal{O}$ which all include the non-recursive form.

**Lemma 30 (Folding). Suppose that $\mathcal{A}, \mathcal{B} : \mathcal{O} \rightarrow CR$ and $\mathcal{N} \langle \text{Ord} \rangle_{H} \mathcal{N}$ such that

\[ \forall M \not< \mathcal{N} H, M, E \in \mathcal{B}(M).c[M/j, E/\alpha] \in \bot \implies \]

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c] \in \mathcal{A}(\mathcal{N})$

Then

\[ \forall M \not< \mathcal{N} H, M, E \in \mathcal{B}(M), V \in \mathcal{A}(M).c[M/j, E/\alpha, V/x] \in \bot \implies \]

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c] \in \mathcal{A}(\mathcal{N})$

Proof. By noetherian induction on $\mathcal{O} \subseteq \mathcal{N}$.

Note that the inductive hypothesis is that given any $\mathcal{M} \not< \mathcal{N}$ and $\mathcal{M} \not< \mathcal{N} H \mathcal{M}$, $\forall O \not< \mathcal{M} H, O, E \in \mathcal{B}(O), V \in \mathcal{A}(O).c(O/j, E/\alpha, V/x) \in \bot$ implies that

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c] \in \mathcal{A}(\mathcal{M})$

We will now use the fact that $\mathcal{A}(\mathcal{N}) \in \text{CR}$ to prove that it contains $\bar{\mu}[\text{Fall}^\leq N[\alpha].c] \in \mathcal{A}(\mathcal{N})$. Observe that for any $E \in \mathcal{A}(\mathcal{N})$, we have the positive head reduction

$$ (\bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] E) \rightarrow (+ \bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] \bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] E) \]

And since $\text{Head}(\mathcal{A}(\mathcal{N})) \subseteq \mathcal{A}(\mathcal{N}) = \mathcal{A}(\mathcal{N})$+, it suffices to show that

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] \bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] \in \mathcal{A}(\mathcal{N})$

This follows from our assumption about $\mathcal{A}(\mathcal{N})$ so long as

\[ \forall M \not< \mathcal{N} H, M, E \in \mathcal{B}(M).c[M/j, E/\alpha, \bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] \in \bot \]

which follows from our assumption about $c$ so long as for any $M \not< \mathcal{N} H M$,

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c/i] \in \mathcal{A}(\mathcal{M})$

which we show by the inductive hypothesis.

Now, suppose that $O \not< \mathcal{M} H, O, E \in \mathcal{B}(O)$, and $V \in \mathcal{A}(O)$. By unfolding the definitions of $\not< \mathcal{M} H$ and $\not< \mathcal{N} H$, we have:

- $M \not< \mathcal{N} H, M \not< \mathcal{M} H \mathcal{M}$
- $O \not< \mathcal{M} H, O \not< \mathcal{N} H \mathcal{O}$

Note that forward reduction of syntactic types cannot change the ordinal value of their interpretation.

**Lemma 33. If $\mathcal{M} \not< \mathcal{N} H, \mathcal{M} \not< \mathcal{M} H$ then $\mathcal{M} \not< \mathcal{N} H$.**

Proof. By induction on the reductions $\mathcal{M} \not< \mathcal{M} H$ and induction on the source term to find the redux.

Therefore, $\not< \mathcal{N} H \mathcal{O} \not< \mathcal{O} \not< \mathcal{M} H \not< \mathcal{N} H$. We can now use the assumption on the command $E$ again to obtain $\mathcal{O}/j, E/\alpha, V/x \in \bot$. Thus it follows that

$\forall O \not< \mathcal{N} H, O, E \in \mathcal{B}(O), V \in \mathcal{A}(O).c(O/j, E/\alpha, V/x) \in \bot$

Applying the inductive hypothesis now grants us that

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c] \in \mathcal{A}(\mathcal{M})$

thus completing the proof.

As a corollary, note that so long as $H(\text{Descend}(\mathcal{N}, B))$ is a reducibility candidate for all $\mathcal{N} \in \mathcal{O}$ and $\mathcal{B} : \mathcal{O} \rightarrow CR$ where $\langle \text{Descend}(\mathcal{N}, B) \rangle_H \subseteq H(\text{Descend}(\mathcal{N}, B))$, then the function $\lambda \mathcal{N} \in \mathcal{O}.H(\text{Descend}(\mathcal{N}, B))$ meets the criteria of Lemma 32 so it contains all the matching recursive case abstractions

$\bar{\mu}[\text{Fall}^\leq N[\alpha].c] \in \text{Descend}(\mathcal{N}, B)$

meeting the specified criteria for every $\mathcal{N} \langle \text{Ord} \rangle_{H} \mathcal{N}$.

The co-data type constructor Ascend follows analogously by duality.

\subsection{F.5 Deflate and Inflate}

Similarly, we can compute the definition of Deflate.

$\text{Cons}_H(\text{Deflate}(\mathcal{A})) \triangleq \{ \text{Down}^{\leq N}(V) \mid M \not< \mathcal{N} H M, V \in \mathcal{A}(M) \}$

All the usual properties for a data type like Deflate hold, and in addition we have the recursive form of case abstraction.

**Lemma 34 (Looping). Suppose $\mathcal{H}$ is plausible and $\nu_1 : \mu \mathcal{N} \mathcal{H} \rightarrow CR$. Given $c_0, c_1$ such that

\[ \forall V \in \mathcal{H}(\mathcal{H}(0)).c_0(V/x) \in \bot \]
\[ \forall N \not< \mathcal{N} H, V \in \mathcal{H}(\mathcal{H}(0)).c_1(N/j, V/x, E/\alpha) \in \bot \]

then $E = \bar{\mu}[\text{Down}^{\leq N}(x).c_1] \bar{\mu}[\text{Down}^{\leq N}(x).c_1] \in \langle \text{Deflate}(\mathcal{A}) \rangle_H$.**

Proof. Note that $\langle \text{Deflate}(\mathcal{A}) \rangle_H = \langle \text{Deflate}(\mathcal{A}) \rangle_H \sqcap \mathcal{S}$. Since $E$ is simple, it remains to show that $E \in W$ and for all $V \in \text{Cons}_H(\text{Deflate}(\mathcal{A})), (V/E) \in \bot$.

- $E \in W$: Note that since $\mathcal{H}$ is plausible, $\mathcal{H}(0), \mathcal{H}(0) \sqcap \mathcal{S}$. Additionally, $0 \sim H(0)$ and $0 + 1 \sim H(1(H(0)))$ by definition. Therefore, $\mathcal{A}(H(0), \mathcal{H}(0) \sqcap \mathcal{S}) \in CR$, and so both contain the (co)-variables by Lemma 2 Therefore, $c_0 \in \bot$ and $c_1 \in \bot$ by Lemma 20. Thus, $E$ is strongly normalizing so $E \in W$.

- For all $V \in \text{Cons}_H(\text{Deflate}(\mathcal{A})), (V/E) \in \bot$: Note that $V = \text{Down}^{\leq N}(V')$ for some $N \sim H(0)$ and $V' \in \mathcal{A}(N)$. We proceed by induction on $N \sim H(0)$.
We give the following meaning to the sequents

\[ \{ \text{\textbf{The co-data type constructor Inflate follows analogously by duality.}} \}

### G. Soundness

We give the following meaning to the sequents

\[ [\Theta \vdash A : k]_H ⇔ (\forall \alpha[\Theta], A(\alpha) \in [k]_I) \wedge \forall Z \in [\Theta]_H, [A]_Z \in [k]_I \]

\[ [\Theta \vdash k : s]_H \Leftarrow \forall Z \in [\Theta]_H, [A]_Z \in [k]_I \]

\[ [\Theta \vdash A : k]_H \Leftarrow \forall Z \in [\Theta]_H, [A]_Z \in [k]_I \]

\[ [\Theta \vdash A : k \& \text{(\textit{\textbf{The co-data type constructor Inflate follows analogously by duality.}})}]_H \Leftarrow \forall Z \in [\Theta]_H, [A]_Z \in [k]_I \]

\[ [\Theta \vdash A = B : k]_H \Leftarrow \forall Z \in [\Theta]_H, [A]_Z = [B]_Z \in [k]_I \]

\[ [\Theta \vdash c : (\Gamma \vdash \Delta)]_H \Leftarrow \forall \Theta_\emptyset \in [\Gamma]_I, \forall \gamma_\emptyset \in [\Delta]_Z, \forall \gamma_\emptyset \in [\Delta]_Z, c(\gamma_\emptyset, \theta_\emptyset) \in [\Gamma]_I \]

\[ [\Theta \vdash v : T \& \Delta \_H \Leftarrow \forall \Theta_\emptyset \in [\Gamma]_I, \forall \gamma_\emptyset \in [\Delta]_Z, v(\gamma_\emptyset, \theta_\emptyset) \in [\Gamma]_I \]

\[ [\Theta \vdash e : T \& \Delta \_H \Leftarrow \forall \Theta_\emptyset \in [\Gamma]_I, \forall \gamma_\emptyset \in [\Delta]_Z, e(\gamma_\emptyset, \theta_\emptyset) \in [\Gamma]_I \]

***Lemma 35.*** *Given any \( H \in [F] \):*

1. *if \( \Theta \vdash k : s \) then \( \Theta \vdash k : s \) in \( k_\emptyset \).
2. *if \( \Theta \vdash A : k \) and \( \text{(\textit{\textbf{The co-data type constructor Inflate follows analogously by duality.}})} \) in \( k_\emptyset \).
3. *if \( \Theta \vdash A = B : k \) and \( \text{(\textit{\textbf{The co-data type constructor Inflate follows analogously by duality.}})} \) in \( k_\emptyset \).
4. *if \( (\Gamma \vdash \Delta) \) in \( \mu \_\emptyset \) then \( (\Gamma \vdash \Delta) \) in \( \text{(\textit{\textbf{The co-data type constructor Inflate follows analogously by duality.}})} \).

**Proof.** By mutual induction on the typing derivation. In the case where we have two assumption derivations, it is the first one which we take to be decreasing.

- \( \Theta \vdash 0 : k \)

If \( I \in [\Theta]_H \) then \( I \in [F] \) by Lemma 8 and since by the inductive hypothesis \( [M]_I \in [k]_I \), we know that \( [M + 1]_I = I + 1([M]_I) \in [k]_I \). Further, if \( \theta \in [\Theta]_H \) then \( \Theta \vdash [M \theta]_I \) so \( M \_I \) \( M \_I \) \( M \_I \) \( M \_I \) \( M \_I \).
then by inductive hypothesis, $M\{\theta\} \llbracket N \rrbracket \llbracket M \rrbracket\llbracket M \rrbracket$ meaning that there is a $M\{\theta\} \llbracket N \rrbracket \llbracket M \rrbracket$ such that $[M'] < [N] < [M']$ and $M\{\theta\} \llbracket \Ord \rrbracket \llbracket M \rrbracket$. Note that the use of the inductive hypothesis here is slightly non-trivial and so requires explanation. We know that $(\llbracket \cdot \cdot\cdot \cdot \cdot\cdot \cdot\cdot\cdot \cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·
• CoAct for type $A$: Analogous to the previous case by duality.
• Eq for types $A = B$: hold the soundness of type-level equality from Lemma[15] so that $[A]_x = [B]_x$.
• CoEq for types $A = B$: Analogous to the previous case by duality.

Next, we have the type-specific left and right introduction rules. Remember that for the active types $A$ in each case, because $\mathcal{H} \in [\mathcal{F}]$ we know that $\langle A \rangle_\mathcal{X} \subseteq [A]_\mathcal{X}$.

Suppose we have a data type
dataT \overset{\alpha}{\to} k where
\[ \overline{K} : \overline{B} \vdash \overline{F}(\overline{X}) \overset{\beta}{\to} \overline{C} \]
Then if $\overline{K} : \overline{B} \vdash \overline{F}(\overline{X}) \overset{\beta}{\to} \overline{C}$ is in $\overline{K} : \overline{B} \vdash \overline{F}(\overline{X}) \overset{\beta}{\to} \overline{C}$. We have the rule
\[
\Theta \vdash \Theta': \overline{K}'(\overline{A}/\alpha) \\
\Gamma \vdash \Gamma': \overline{C}_1(\overline{A}/\alpha, D/d_1) \vdash c_1 \alpha \\
\Gamma \vdash \Theta': \overline{C}_2(\overline{A}/\alpha, D/d_2) \vdash c_2 \alpha \\

\Gamma \vdash \overline{K}'(\overline{A}/\alpha) : \overline{F}(\overline{X}) \Delta
\]
Now supposing that each of $\overline{X}$ and $\overline{Y}$ are all (co)-values $\overline{E}$ and $\overline{V}$, by the inductive hypothesis, we know that
\[
K^{\overline{F}}(\overline{E}, \overline{V})(\theta, \gamma, \delta) \in \llbracket F(\overline{A}) \rrbracket_\mathcal{X} \subseteq \llbracket F(\overline{A}) \rrbracket_\mathcal{X}
\]
Furthermore, since
\[
[\llbracket B \rrbracket]_\mathcal{X}[(\overline{A}/\alpha, D/d_1)] = [\llbracket B \rrbracket]_\mathcal{X}[(\overline{A}/\alpha, D/d_1)] \\
[\llbracket C \rrbracket]_\mathcal{X}[(\overline{A}/\alpha, D/d_2)] = [\llbracket C \rrbracket]_\mathcal{X}[(\overline{A}/\alpha, D/d_2)]
\]
are all reducibility candidates, we must have that
\[
K^{\overline{F}}(\overline{E}, \overline{V})(\theta, \gamma, \delta) \in \llbracket F(\overline{A}) \rrbracket_\mathcal{X}
\]
in general by unfocalization (Lemma[12]).

For the left rule
\[
c_1 \alpha : \Gamma \vdash \overline{B}_1 \Gamma : b_1, d_1, \overline{K}_1(\overline{A}/\alpha) \ldots \\
\Gamma \vdash \mu(\overline{K}_1(\overline{A}/\alpha)) \overline{F}(\overline{X}) \Delta
\]
For each $c_1$, we have by inductive hypothesis that for any of $\theta'((\overline{A}/\alpha, D/d_1), \overline{V}, \overline{X} : [\Gamma, x : B_1]_\mathcal{X}$ and $\gamma' : \Delta, \alpha : \overline{C}_1$ such that $c_1(\theta', \gamma', \delta') \not\in \mu$. Observe that given any choice of types $\overline{D} : \overline{K}_1 \overline{D}$, values $\overline{V} \in [\overline{B}_1]_{\overline{X}(\overline{D}/d_1)}$, and co-values $\overline{E} \in [\overline{C}_1]_{\overline{X}(\overline{D}/d_1)}$ and $\alpha : \overline{C}_1$ then
\[
\theta(\overline{D}/d_1), \gamma'(\overline{V}/\overline{X}), \delta(\overline{E}/\alpha) \in \mu.\llbracket F(\overline{X}) \rrbracket_\mathcal{X} \subseteq \llbracket F(\overline{X}) \rrbracket_\mathcal{X}
\]
Thus,
\[
c_1(\theta(\overline{D}/d_1), \gamma(\overline{V}/\overline{X}), \delta(\overline{E}/\alpha)) = c_1(\theta, \gamma, \delta)(\overline{V}/\overline{X}, \overline{E}/\alpha, D/d_1) \not\in \mu.
\]
From this we can see that
\[
\mu(\overline{K}_1(\overline{A}/\alpha, \overline{X}, c_1(\theta, \gamma, \delta) \ldots) \in \llbracket F(\overline{X}) \rrbracket_\mathcal{X} \subseteq \llbracket F(\overline{X}) \rrbracket_\mathcal{X}
\]
by Lemma[25].

The rules for non-recursive co-data declarations are analogous by duality. Additionally, (co-)data type declarations defined by neitherian and primitive recursion also follow analogously.

The ordinary rules for Ascend and Descend are handled by viewing them as user defined co-data types. The remaining rule we need to look at is the recursive case abstraction.
\[
c : \Gamma, x : A \vdash c : \text{Descend}(A, \Delta) \\
\Gamma \downarrow \text{Fall}^{\llbracket \mu \rrbracket(\overline{X})}([\alpha][c] \cdot \text{Descend}(N, A) \vdash \Delta
\]
So that the inductive hypothesis fits the requirements of Lemma[32].

Note that the recursive case abstraction for Ascend follows dually.

The recursive case abstraction rule for Deflate is
\[
c_1 \alpha : \Gamma, x : A \vdash c_1 : \text{Deflate}(A) \vdash \Delta \\
\Gamma \downarrow \text{Deflate}^{\llbracket \mu \rrbracket(\overline{X})}([\alpha]c_1 \cdot \text{Deflate}(A) \vdash \Delta
\]
and its soundness follows by Lemma[31] and the inductive hypothesis. Note that the recursive case abstraction for Inflate follows dually.

Definition 9. The set $\{\Theta\}$ is the subset of the maps $HType \to U$ which are “big enough” with respect to $\Theta$.

Proof. Let $\mathcal{H}(0)$ be to be the length of $\Theta$, which covers the worst case where we have $\Theta = \iota_n - 1 < 0, \iota_n - 2 < \iota_n - 1, \ldots, \iota_n < \iota_n$ by assigning $0$ to $i_1, \ldots, n - 2$ to $i_{n - 2}$, and $n - 1$ to $i_{n - 1}$. More specifically, whenever we see a $i < M$ in $\Theta$, we can assign $a$ a value based on its position in $\Theta$.

Lemma 37. For all $\Theta$, $\{\Theta\}$ is inhabited.

Proof. From composition of the previous lemma with Lemma[17].

Lemma 39. If $\mathcal{H} \in \{\Theta\}$ and $\mathcal{H} \in [\mathcal{F}]$ then there exists $\Theta \llbracket \mathcal{H} \rrbracket_\mathcal{X}$

Proof. From induction on $\Theta$.

Corollary 9. If $\mathcal{H} \in \{\Theta\}$ and $\mathcal{H} \in \{\mathcal{F}\}$ then $\llbracket c : (\Gamma \vdash \Delta) \rrbracket_\mathcal{H}$ implies $c \in \mathcal{H}$. Similarly, $\llbracket \Gamma \vdash v : A \rrbracket_\mathcal{H} \subseteq \mathcal{W}$ and $\llbracket \Gamma \vdash c : A \rrbracket_\mathcal{H} \subseteq \mathcal{W}$.

Theorem 3. If $\mathcal{F}$ is well-founded, then

1. If $\llbracket c : (\Gamma \vdash \Delta) \rrbracket_\mathcal{H}$ then $c$ is strongly normalizing in $\mu \mathcal{F}$.
2. If $\llbracket \Gamma \vdash v : A \rrbracket_\mathcal{H}$ then $v$ is strongly normalizing in $\mu \mathcal{F}$.
3. If $\llbracket c : A \rrbracket_\mathcal{H}$ then $c$ is strongly normalizing in $\mu \mathcal{F}$.

Proof. From there exists $\mathcal{H} \in \{\Theta\}$ and $\mathcal{H} \in \{\mathcal{F}\}$ we have $\llbracket c : (\Gamma \vdash \Delta) \rrbracket_\mathcal{H}$ by Theorem[2] and we know that $c : (\Gamma \vdash \Delta)$ implies $c \in \mathcal{H}$ by the previous corollary. The cases for (co-terms) follows analogously.

Note that type-erasures preserves strong normalization precisely because the type-eraser rewriting theory is strictly weaker than rewriting the pre-eraser programs.

Lemma 40. If $c$ is strongly normalizing in $\mu \mathcal{F}$, then $Erase(c)$ is strongly normalizing in the type-eraser $\mu \mathcal{F}$, and similarly for (co-terms).
Proof. The Erase operation removes all erasable types, with kinds inhabiting \( \Box \), from commands and (co-)terms. In particular, we remove the type-level content of constructors \( K^B (\epsilon, \nu) \) and patterns \( K^{B^{\prime \prime}} (\overline{\alpha}, \overline{x}) \), leaving only those types with the non-erasable kind \( \Box \). Furthermore, because the caveat for reducing inside the branches of a case abstraction are strictly more limiting in the type-erased \( \mu B_{E} \)-calculus, we know that

- If \( \text{Erase}(c) \rightarrow e' \) in the type-erased \( \mu B_{E} \)-calculus, there is a \( e'' \) such that \( \text{Erase}(e'') = e' \) and \( c \rightarrow e'' \) in the \( \mu B_{E} \)-calculus.
- If \( \text{Erase}(v) \rightarrow v' \) in the type-erased \( \mu B_{E} \)-calculus, there is a \( v'' \) such that \( \text{Erase}(v'') = v' \) and \( v \rightarrow v'' \) in the \( \mu B_{E} \)-calculus.
- If \( \text{Erase}(e) \rightarrow e' \) in the type-erased \( \mu B_{E} \)-calculus, there is a \( e'' \) such that \( \text{Erase}(e'') = e' \) and \( e \rightarrow e'' \) in the \( \mu B_{E} \)-calculus.

Therefore, because the chosen \( c, v, e \) are strongly normalizing in the \( \mu B_{E} \)-calculus, their erasure \( \text{Erase}(c), \text{Erase}(v), \text{Erase}(e) \) must also be strongly normalizing in the type-erased \( \mu B_{E} \)-calculus.

H. Natural Deduction Embedding

To show that the natural deduction calculus for effect-free functional programs is strongly normalizing, we demonstrate that:

1. the translation into the call-by-name \( \mu B_{N} \)-calculus is type-preserving for well-typed terms,
2. each reduction of a natural deduction term corresponds to at least one reduction in \( \mu B_{N} \).

Lemma 41. If \( \Gamma \vdash v : A \) is derivable then \( \Gamma \vdash v^\downarrow : A \) is derivable.

Proof. By induction on the structure of the derivations for \( \Gamma \vdash v : A \).

Note that \( \rightarrow^+ \) denotes the transitive, but not reflexive, closure of \( \rightarrow \).

Lemma 42. If \( v \rightarrow v' \) in the natural deduction calculus then \( v^\downarrow \rightarrow^+ v'^\downarrow \).

Proof. By cases on the reductions in the natural deduction calculus.

- if \( \{ H^{E}[\overline{x}] \Rightarrow v' \ldots \} \), \( H^{E}[\overline{v}] \Rightarrow v' \overline{B/b}, \overline{v/x} \)

\[
\begin{align*}
\{ H^{E}[\overline{x}] & \Rightarrow v' \ldots \}, H^{E}[\overline{v}] \\
& \Rightarrow \mu x.\langle H^{E}[\overline{x}, \beta], \overline{v^\downarrow/\beta} \ldots \rangle[H^{E}[v', \alpha]] \\
& \Rightarrow \mu x.\langle \overline{B/b}, \overline{v^\downarrow/x} \rangle[\alpha] \\
& \Rightarrow \overline{v^\downarrow/\beta} \overline{B/b}, \overline{v^\downarrow/x} \\
& \Rightarrow \langle \overline{v/B/b}, \overline{v/x} \rangle \\
& \Rightarrow \langle \overline{v/\beta}, \overline{v/x} \rangle^\downarrow \\
\end{align*}
\]

- case \( K^{E} (\overline{v}) \) of \( K^{E} (\overline{x}) \Rightarrow v' \ldots \Rightarrow v' \overline{B/b}, \overline{v/x} \)

\[
\begin{align*}
\{ K^{E} (\overline{v}) & \Rightarrow v' \ldots \} \Rightarrow \mu x.\langle K^{E} (\overline{x}), \overline{v^\downarrow/\beta} \ldots \rangle \\
& \Rightarrow \mu x.\langle \overline{B/b}, \overline{v^\downarrow/x} \rangle[\alpha] \\
& \Rightarrow \overline{v^\downarrow/\beta} \overline{B/b}, \overline{v^\downarrow/x} \\
& \Rightarrow \langle \overline{v/\beta}, \overline{v/x} \rangle^\downarrow \\
\end{align*}
\]
in the following:

\[
\text{(loop } \text{Down}^{M+1}(v) \text{ of } \text{Down}^0(x) \Rightarrow v_0 | \text{Down}^{j+1}(x) \Rightarrow v_1)\]

\[
= \mu \alpha. (\text{Down}^{M+1}(v) )\| E_\alpha \]

\[
\rightarrow \mu \alpha. (\mu \beta. (v^\flat \{M/j, v^\flat/x\})\| \tilde{\mu} y. (\text{Down}^{M}(x) )\| E_\alpha) \]

\[
\rightarrow \mu \alpha. (\text{Down}^{M}(v^\flat \{M/j, v^\flat/x\}) )\| E_\alpha \]

\[
\rightarrow (\text{case } \text{Down}^{M}(v_1 \{M/j, v^\flat/x\}) \text{ of } \text{Down}^{0}(x) \Rightarrow v_0 \]

\[
| \text{Down}^{j+1}(x) \Rightarrow v_1)\]

Cases where reduction occurs inside of a larger context follow from compositionality of the translation \((-)^\flat\). □

**Theorem 4.** If \(\Gamma \vdash v : A\) and \((\Gamma \vdash A)\) seq are derivable then \(v\) is strongly normalizing.

**Proof.** By Lemma 411 we know that \(\Gamma \vdash v^\flat : A\) seq, and so by Theorem 424 is a strongly normalizing term of the \(\mu \tilde{\mu} N\)-calculus. Now, suppose that there is an infinite reduction path from \(v\). By applying Lemma 421 over the steps of this infinite reduction path from \(v\) and composing them together, we obtain an infinite reduction path from \(v^\flat\), which is a contradiction. Therefore, there is no infinite reduction path from \(v\), so \(v\) is also strongly normalizing.

**References**


