Indexed Codata Types

David Thibodeau [*]
School of Computer Science
McGill University
Montreal, Canada
david.thibodeau@mail.mcgill.ca

Andrew Cave [†]
School of Computer Science
McGill University
Montreal, Canada
andrew.cave@mail.mcgill.ca

Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

Abstract

Indexed data types allow us to specify and verify many interesting invariants about finite data in a general purpose programming language. In this paper we investigate the dual idea: indexed codata types, which allow us to describe data-dependencies about infinite data structures. Unlike finite data which is defined by constructors, we define infinite data by observations. Dual to pattern matching on indexed data which may refine the type indices, we define copattern matching on indexed codata where type indices guard observations we can make.

Our key technical contributions are three-fold: first, we extend Levy's call-by-push value language with support for indexed (co)data and deep (co)pattern matching; second, we provide a clean foundation for dependent (co)pattern matching using equality constraints; third, we describe a small-step semantics using a continuation-based abstract machine, define coverage for indexed (co)patterns, and prove type safety. This is an important step towards building a foundation where (co)data type definitions and dependent types can coexist.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory

Keywords Coinduction, Dependent types, Functional programming, Logical frameworks

1. Introduction

Over the past two decades we made significant progress in mechanically verifying inductive properties about finite data and computation using proof assistants. However, the situation is very different when it comes to specifying and mechanically verifying properties such as fairness or liveness about programs whose computation is infinite i. e. they continue to run and produce results. Such properties are elegantly stated and proven coinductively. Starting with Hagino's work (1987) in (co)algebras, there has been growing consensus that proof and programming environments should view infinite data dual to finite data. Under this view inductive data such as lists are modelled by constructors, while coinductive or infinite data such as streams are described by observations. An important step towards a sound type-theoretic foundation for inductive and coinductive definitions has been taken by Abel et al. (2013) where the authors present a simply-typed language using a rewriting semantics where finite data is defined using constructors and analyzed by pattern matching while infinite data is defined via copattern matching and analyzed by observations. Subsequently, it was shown that this language is normalizing using sized types (Abel and Pientka 2013). A prototype implementation of copatterns exists within Agda, a programming language based on Martin-Löf type theory (Norell 2007). However, a theoretical foundation that supports dependent types, (deep) (co)pattern matching and (co)recursion and at the same time allows inductive and coinductive definitions to be arbitrarily mixed remains elusive.

In this paper we take a substantial step towards such a general foundation concentrating on a flavor of dependent types, called indexed types (Zenger 1997; Xi and Pfenning 1999), where the language of indices is separate from the language of types and programs and describes a domain where equality is decidable. Specifically we present a core language for dependent (co)pattern matching that allows eager and lazy evaluation to be mixed by extending Levy's call-by-push value language (2001). Following Levy, our language is centered around the duality of positive types which we interpret as values and use to construct finite data, and negative types which we take as computations and use to describe the observations about infinite data. While indexed data types allow us to for example specify and statically enforce properties about finite lists and trees, indexed codata types allow us to specify and statically enforce properties about streams and traces. Throughout our development, we keep the index language abstract. Our main technical contributions are three-fold:

- We extend Levy's call-by-push value language (2001), with support for indexed (co)data and deep (co)pattern matching. To keep our design modular, we keep the index domain abstract and specify the key properties it must satisfy. In particular, our index domain must provide a decision procedure to reason about equalities and a unification procedure to compute the most general unifier of two index objects. We illustrate these properties by considering the domain of natural numbers.
- Our core language provides a clean foundation for dependent (co)pattern matching where we track and reason with depen-

[*] This author acknowledges funding from the Fonds Québécois de Recherche sur la Nature et les Technologies (FQRNT).

[†] This author acknowledges funding from the National Science and Engineering Research Council (NSERC).

dencies among indices using equality constraints that are accumulated in a context. Our equality context may contain both equality constraints that are satisfiable and equality constraints that are contradictory. This leads to an elegant foundation for (co)pattern matching in the dependently typed setting that may serve as an alternative to existing approaches (Brady et al. 2004; Goguen et al. 2006; Pientka and Dunfield 2008).

- We describe the operational semantics of our core language using a continuation-based abstract machine and prove type preservation. We also provide a sound non-deterministic algorithm to generate covering sets of copatterns. Finally, we show progress – in the presence of infinite data the key idea here is that every expression either returns a value or we can continue to evaluate it by supplying enough observations.

We see several applications of our work: it lays the foundation for extending languages such as DML (Xi and Pfenning 1999) and ATS (Xi 2004) to support indexed codata; choosing as an index language the language of types itself, it serves as a foundation for mixing eager and lazy evaluation in functional languages that support GADTs (Cheney and Hinze 2003; Xi et al. 2003); choosing as an index language LF (Harper et al. 1993; Cave and Pientka 2012), our work serves as a general foundation for writing both inductive and coinductive definitions and proofs about formal systems. Finally, we believe that the core language that we describe in this paper provides a stepping stone in developing a sound dependently typed foundation for Coq and Agda that supports deep (co)pattern matching and allows inductive and coinductive definitions to be mixed.

The remainder of this paper is organized as follows: We illustrate the main ideas of indexed (co)data types through several examples in Section 2. In Section 3 we introduce our language supporting both indexed data types and codata types together with pattern and copattern matching in a symmetric way. Section 4 describes the operational semantics, coverage, and type safety.

2. Main Idea

Indexed recursive types allow us to for example specify and program with lists that track their length thereby avoiding run-time checks for cases which cannot happen. We consider here a variation of this example: a recursive type `Msg` which describes a message consisting of bits and tracks its length by choosing as an index domain `nat`. Our pseudo-code follows closely the underlying foundation where we model data types using recursive types and disjoint sums together with equality constraints.

```
data Msg [N: nat] : type =
| Nil : N = z * 1
| Cons: ΣM:nat. N = s M * Bit * Msg [M]
```

We separate the index domain from the language of types and programs and embed index objects inside types and programs using `[ ]`. The distinction between the index domain and programs is also reflected in the syntactic convention we use for type-setting data-types and programs. Index variables are upper case letters, while index types and index constructors start with a lower case letter. However, term variables use a lower case letter while types and term constructors and observations start with an upper case letter.

The type `Msg [N]` defines messages inductively: either we have an empty message `Nil` where `N` must be zero, or we can construct a message using `Cons`, if there exists `M:nat` s.t. `N = s M` and we have a `Bit` together with a message of length `M`. In the latter case, we have built a message of length `s M`. As in ML-like languages, we require that constructors that correspond to the base case in our inductive definition take in formally an argument of type unit (denoted by `1`). When we pattern match on a message `m` of type `Msg [N]`, we need to consider the following two cases: if `m` stands for an empty message, written as `Nil (e, ())`, then we also obtain an equality proof `e` that `N = z`; if `m` stands for a message `Cons <M, (e, h, t)>` where `M` is the witness for the existential in the definition of `Cons` and `e` stands for the equality proof `N = s M`. In both cases, we can further pattern match on the equality proof `e`, writing $\wp$ as the witness which forces the type checker to solve the accumulated constraints setting in the base case `N` to zero and in the step case `N` to `s M`. As our index domain is restricted to a decidable domain, equality proofs can always be derived and reconstructed when elaborating a surface program into our core language.

Dually to model a stream of bits which keeps track of how many bits belong to one message we define three different observations:

```
codata Str [N:nat] : type =
| GetBit  : ΠM:nat. N = s M → Bit
| NextBits: ΠM:nat. N = s M → Str [M]
| Done    : N = z → NextMsg

and data NextMsg : type =
| NextMsg : ΣN:nat. Str [N]
```

Given a stream with index `N`, we can observe the next bits (`NextBits`) and get the current bit (`GetBit`), provided that we supply some number `M` and an equality proof that `N = s M`. We are done reading all bits belonging to our message, if `N = z`, i.e. we can get the next message, if we can provide a proof for `N = z`. This definition of a stream allows us to enforce that we read the correct number of bits belonging to a message.

While our indexed recursive type `Msg` is defined via positive types (equality, existentials, products), our coinductive definition of `Str` uses negative types (universals, functions). When we pattern match on a data type, we also learn about equality constraints that must hold. When we make observations on a codata type, we must supply an equality proof that satisfies the equality constraint that guards the observation. To our knowledge this dual role that equality plays in defining data and codata types has not been observed before, yet it seems central in understanding how to scale (co)data type definitions and (co)pattern matching to the dependently typed setting.

2.1 Message Processing Using Deep (Co)Pattern Matching

Interactions of a system with input/output devices or other systems are performed through a series of queries and responses which are represented using a stream of bits that can be read by the system. Processing requests over those streams can be error prone. If one reads too many or not enough bits, then there is a disconnect between the information a program reads and the one that was sent which potentially could be exploited by an attacker. To avoid such problem, we propose to use indexed codata types to parametrize a stream with a natural number indicating how many bits we are entitled to read until the next message starts. Thus, one can guarantee easily that a program will not leave parts of a message on top of the stream but that they consume all of it. We will use this example of message passing to highlight the role of indices in writing programs that use (co)pattern matching.

First, we want to read a message from the stream `Str [N]` and return the message together with the remaining stream. This is enforced in the type of the function `readMsg` below. The type can be read informally as: For all `N` given `Str [N]` we return a message together with `Str [z]` which indicates that we are done reading the entire message.

```
rec readMsg: Π N:nat. Str [N] → Msg [N] * Str [z] =
fn [z]   s ⇒ (Nil (℘, ()), s)
 | [s M] s ⇒
  let  c = s.GetBit [M] ℘ in
  let  (w, s') = readMsg [M] (s.NextBits [M] ℘) in
  (Cons <M, (℘, (c, w))>, s')
```

The program `readMsg` is written by pattern matching on the index object `N`. Using **fn**-abstractions we pattern match on multiple input

arguments simultaneously. We use a notation similar to ML-like languages, but we wrap index objects in `[ ]` to clearly distinguish them from computation-level data and terms. If `N` is zero, then we are finished reading all bits belonging to the message and we simply return the empty message together with the remaining stream `s`. If `N` is not zero but of the form `s M`, we observe the first element `c`, the bit at position `s M`, in the stream using the observation `.GetBit`. We then read the rest of the message `w` by making the recursive call `readMsg [M] (s.NextBits [M] ℘)` and then build the actual message by consing `c` to the front of `w`. Note that in order to make the observation `GetBit` or `NextBits` we must supply two arguments, namely `M` and a proof that `s M = s M`. Dually, when we construct a message `Nil` in the base case, we also must supply a proof that `z = z`; similarly in the step-case, we construct a message `Cons` by providing as a witness `M` together with a proof that `s M = s M`. It seem reasonable to assume that these arguments and equality proofs can be inferred in practice; however we make them explicit in our core language to emphasize their dual role in indexed (co)data types.

So far we have seen how to make observations about streams and use them. Next, we show how to build a stream which is aware of how many bits belong to a message effectively turning it into a stream of messages. This is accomplished via two mutually recursive functions mixing pattern and copattern matching: the first marshals the size of the message with the message stream and the second one continues to create the message stream. We assume that we have polymorphism here (which we do not treat in our foundation).

```
codata 'a Stream : type =
| Head : 'a
| Tail : 'a Stream

rec getMsg: Bit Stream → [nat] Stream → NextMsg =
fn s ns ⇒ let [N] = ns.Head in
NextMsg [N] (msgStr [N] s ns.Tail)

and msgStr: Π N:nat. Bit Stream → [nat] Stream → Str [N] =
fn  [z] s ns .Done ℘ ⇒ getMsg s ns.Tail
| [s N] s ns .GetBit  [M] ℘ ⇒ s.Head
| [s N] s ns .NextBits [M] ℘ ⇒ msgStr [N] s.Tail ns.Tail
```

The function `getMsg` takes in a stream `s` of bits and a stream of natural numbers that tells us the size of a message. It then returns a message of the required size by reading the appropriate number of bits from `s` using the function `msgStr` and creating a stream of type `Str [N]` where `N` is the size of the message. The function `msgStr` is defined by (co)pattern matching: the first branch says, we can only make the observation `Done` provided that `N` is zero; in this case we are done reading all bits belonging to the message. The second branch says: if the size of the message is `s N`, we can make the observation `GetBit` provided we have a proof $\wp$ showing that `s M` is equal to `s N`. Note that our (co)pattern remains linear - the fact that `M` is forced to be equal to `N` is guaranteed by the equality proof $\wp$ that solves the arising constraint `s N = s M`. The term $\wp$ is the canonical term for equality constraints. We exploit here the fact that equality and unification is decidable in our index domain. If we can solve the constraint, as is the case here, we keep track of the solution `N := M : nat` in our context of assumptions and continue to type check the body of the branch under this constraint; if we can disprove the arising equality constraint, we keep a contradiction in our context of assumption and continue to check the body. This allows for an elegant treatment of linear (co)patterns in the presence of dependent types.

Last, we show how to generate a bit stream where every message contains two random bits. This illustrates deep copattern matching.

```
rec genBitStr: Str [s (s z)] =
fn .GetBit   [s z] ℘ ⇒ RandomBitGen ()
 | .NextBits [s z] ℘ .GetBit [z] ℘ ⇒ RandomBitGen ()
 | .NextBits [s z] ℘ .NextBits [z] ℘ .Done ℘ ⇒
     NextMsg [s (s z)] genBitStr
```

Following the ideas described in this section, we can implement also fair merge of two streams **as** and `bs` where we consume a finite amount of `a`'s followed by a finite amount of `b`'s. We refer the interested reader to the extended technical report (Thibodeau et al. 2016) for more examples on streams.

2.2 Revisiting the Duality of (Co)Inductive Definitions

So far we have concentrated on two aspects: 1) how inductive data is constructed and analyzed by pattern matching while coinductive data is observed and analyzed by copattern matching; 2) the role of indices and equality constraints in (co)pattern matching. For data of type `Msg [M]`, we provided a way of constructing a message for each `M`. Dually, our codata type `Str [N]` provided observations for all possible `N`.

An important question to clarify is whether (co)data type definitions need or should be covering, i.e. provide a constructor or observation for each possible index. What does it mean to have no constructor for a possible index? And dually, what does it mean to have no observation for a possible index?

We discuss these questions by looking at how we define even numbers inductively and coinductively. Clearly, inductive definitions do not need to be covering. For example, our inductive definition of `Even [N]` states that we can construct a proof that `z` is even using `Ev_z` provided we have a proof that `N = z`. For clarity, we define the type of the constructor `Ev_z` as `N = z * 1` where `1` stands for unit (or top). Similarly, we can construct a proof that `N` is even, if there exists a number `M` s.t. `N = s (s M)` and `M` is even.

```
data Even [N:nat] : type =
| Ev_z : N = z * 1
| Ev_ss:ΣM:nat. N = s (s M) * Even [M]
```

The set of terms inhabiting this predicate is the least fixed point defining even numbers. Note that there is no way that we can construct a witness for `Even [s z]` and this type is empty. Modelling the empty type

```
data 0: type
```

by declaring no constructors, we could make this more explicit by adding a constructor `Ev_s` of type `N = (s z) * 0`. This explicitly states that `Even [s z]` cannot be constructed without any assumptions, since `0` has not elements. We typically omit such a case in the definition of our inductive types, but these impossible cases might arise when we pattern match on elements of the type `Even`.

Dually, we can define even numbers coinductively using a greatest fix point. By default the greatest fix point is inhabited by all natural numbers and in particular all even numbers. The observations we make describe those numbers that should not be in the set of even numbers! Specifically, we are stating that odd numbers, cannot be in the set of even numbers. This leaves us with the set of even numbers.

```
codata CoEven (N:nat) : type =
| Cev_sz : N = s z → 0
| Cev_ss : ΠM:nat. N = s (s M) → CoEven [M]
```

If `N = s z` then we return the empty type. If we make an observation `Cev_sz` and have a proof that `N = s z` then we have arrived at a contradiction. The observation `Cev_ss` extracts a proof of `CoEven [M]` from a proof of `CoEven [s (s M)]`.

This discussion highlights the difference between the definition of constructors and observations. If we omit a constructor for a given index, then the indexed data type is not inhabited and it is interpreted as false. Dually, if we omit an observation for a given index, then the indexed codata type is still inhabited and it corresponds to being trivially true.

We now prove that both interpretations give us the same set of terms. First we show that `Even [N]` implies `CoEven [N]`:

```
rec evToCoEv : ΠN:nat. Even [N] → CoEven [N] =
fn [z]   (Ev_z ℘ ())          .Cev_sz ℘
 | [z]   (Ev_z ℘ ())          .Cev_ss [M] ℘
 | [s N] (Ev_ss <M, (℘, e)>) .Cev_sz ℘
 | [s N] (Ev_ss <M, (℘, e)>) .Cev_ss [K] ℘ ⇒ evToCoEv e
```

We write this function by pattern matching on `Even [N]`. In the case where `Even [z]`, we want to return `CoEven [z]`. As elements of `CoEven [z]` are defined by the observations we can make about it, we consider two sub-cases. If we try to make the observation `Cev_sz`, we must provide a proof that `z = s z`. This will be refuted by our decision procedure in our index domain, i.e. our decision procedure will succeed, but add a contradiction to our context of assumptions, from which anything follows. Again, the pattern $\wp$ is the canonical pattern for equality constraints, even if they are unsatisfiable. In those cases, we can simply omit the right hand side of the inaccessible branch, since we cannot provide a term for an unsatisfiable equality constraint.

If we try to make the observation `Cev_ss`, then we must show `ΠM:nat. z = s (s M)→ CoEven [M]`. Again we have arrived at a contradiction, since there is no proof for `z = s (s M)`.

Finally, we consider the case where `Even [s (s N)]`. In this case, we can again make two possible observations, `Cev_sz` and `Cev_ss`. In the first case, we again arrive at a contradiction, since `s (s N) = s z` is always false. In the last case, we accumulate and solve two equality constraints while type checking the (co)pattern: `s (s N) = s (s M)` and `s (s N) = s (s K)`. Then we proceed to check the body of the branch in the context where `N := K: nat` and `M := N : nat`.

This example highlights the mix of pattern and copattern matching and the reasoning with equality constraints; it also highlight how impossible cases arise and how we treat them.

Can we also prove that `CoEven [N]` implies `Even [N]`? - In general this is not true since the coinductive interpretation may be strictly bigger than the inductive one. In our case however we can indeed show this property by induction on `N`. In the case where `N = s z` and we assume `CoEven [s z]`, we make the observation `(c. Cev_sz ℘)` which results in an object of type `0` – however, we know that this type is not inhabited and hence we abort. In our language **abort** is an abbreviation for a function without any branches.

```
rec coEvToEv : ΠN:nat. CoEven [N] → Even [N] =
fn [z]       c ⇒ Ev_zero (℘, ())
 | [s z]     c ⇒  abort (c.Cev_sz ℘)
 | [s (s N)] c ⇒
   Ev_ss <N, (℘ , coEvToEv [N] (c.Cev_ss [N] ℘))>
```

2.3 Final Remark

For simplicity all our previous examples use as index domain natural numbers. However, we want to emphasize that our theoretical foundation is parametric in the index domain. Choosing the logical framework LF as an index domain, we can model bisimulation of two automata and encode a type-preserving environment-based evaluator where values are defined coinductively following (Milner and Tofte 1991) (see (Thibodeau et al. 2016)). We note that we have not studied the use of coinductive index domains. While we believe our work to be compatible with such domains, defining an adequate notion of equality for coinductive terms is challenging, as explained in (McBride 2009).

3. Theory

We present in this section a general purpose programming language which supports defining finite data using indexed recursive types and infinite data using indexed corecursive types. To analyze and manipulate finite and infinite data, we support simultaneous pattern and copattern matching. We omit polymorphism which is largely an orthogonal issue.

3.1 Index Domain

Our programming language is parametric over the index domain which we describe abstractly with U. This index domain can be natural numbers, strings, types (Cheney and Hinze 2003; Xi et al. 2003), or (contextual) LF (Cave and Pientka 2012). Index objects are abstractly referred to as *index-term* C and have *index-type* U. As a running example we will use natural numbers to illustrate the requirements our index domain must satisfy. It can be defined as containing a single index-type nat and index-terms are simply built of zero, suc, and variables X.

$$
\begin{array}{ll}
\text{Index-Type } U & ::= \mathsf{nat} \\
\text{Index-Term } C & ::= X \mid \mathsf{zero} \mid \mathsf{suc}\ C \\
\text{Index-Context } \Delta & ::= \cdot \mid \Delta, X : U \mid \Delta, X := C : U \mid \Delta, \# \\
\text{Index-Substitution } \theta & ::= \cdot \mid \theta, C/X
\end{array}
$$

Variables that occur in index-terms must be declared in an index-context Δ. In our setting, the index-context also contains equality constraints. The constraint $X{:=}C{:}U$ says that the index-variable X is equal to the index-term C. Such constraints arise in typing (co)patterns (see the function `msgStr` from Sec. 2). The index-context also keeps track of contradictions, written $\#$, that may arise when we encounter in a (co)pattern a constraint that can never be satisfied.

Index-substitutions are built by supplying an index-term for an index-variable. We interpret $\cdot$ as the identity index-substitution. We define the lookup of the instantiation for a variable X as follows:

$\boxed{\theta(X) = C}$ Variable X is bound to term C in substitution θ

$$
\begin{array}{ll}
(\theta, C/X)(X) & = C \\
(\theta, C/Y)(X) & = \theta(X) \text{ if } X \neq Y \\
(\cdot)(X) & = X
\end{array}
$$

We use index-substitutions to model the run-time environment of index variables. Looking up X in the substitution θ returns the index-term C to which X is bound at run-time. The index-context Δ captures the information that is statically available and is used during type checking.

Typing of Index Domain We define the well-formedness of index-contexts and index-substitutions in Fig. 1. The definition of index-contexts is mostly straightforward noting that $\Delta, X{:=}C{:}U$ is a well-formed index-context if Δ is well-formed, the index-type U is well-formed in Δ, and the index-term C has index-type U. We make sure that there are no circularities in Δ. An index-substitution θ provides a mapping for declarations in the index-context Δ' and guarantees that all instantiations have the expected index-type and are compatible with existing constraints. Our judgment $\Delta \vdash \theta : \Delta'$ states that a given instantiation θ, computed via pattern matching at run-time, matches the assumptions in Δ' that were made statically during type checking. It is defined inductively on the domain Δ'. Although all instantiations computed by pattern matching are ground (i.e. Δ is empty), we state the relationship between θ and Δ more generally. If Δ' contained a contradiction, the contradiction must also be present in Δ. We still require in this case that θ provides consistent and well-typed instantiations for all the remaining declarations in Δ'.

While our rules for the well-formedness of index-contexts and index-substitutions are generic, typing of index-terms and index-types obviously depends on our choice of the index domain. We include here the rules for natural numbers.

Our index domain must satisfy several properties. The first one is the substitution property which we list here as a requirement[1].

[1] Proofs can be found in the supplementary material for this paper (Thibodeau et al. 2016).

$\boxed{\vdash \Delta\ \mathsf{ictx}}$ well-formed index-context Δ

$$\frac{}{\vdash \cdot\ \mathsf{ictx}} \qquad \frac{\vdash \Delta\ \mathsf{ictx} \quad \Delta \vdash U : \mathsf{Type} \quad \Delta \vdash C : U}{\vdash \Delta, X := C : U\ \mathsf{ictx}}$$

$$\frac{\vdash \Delta\ \mathsf{ictx}}{\vdash \Delta, \#\ \mathsf{ictx}} \qquad \frac{\Delta \vdash U : \mathsf{Type} \quad \vdash \Delta\ \mathsf{ictx}}{\vdash \Delta, X : U\ \mathsf{ictx}}$$

$\boxed{\Delta \vdash \theta : \Delta'}$ θ maps index variables from Δ' to Δ

$$\frac{\Delta \vdash \theta : \Delta' \quad \Delta \vdash \theta(X) : U[\theta]}{\Delta \vdash \theta : \Delta', X{:}U} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \# \in \Delta}{\Delta \vdash \theta : \Delta', \#}$$

$$\frac{\Delta \vdash \theta : \Delta' \quad \Delta \vdash \theta(X) : U[\theta] \quad \Delta \vdash \theta(X) = C[\theta]}{\Delta \vdash \theta : \Delta', X{:=}C{:}U} \qquad \frac{}{\Delta \vdash \theta : \cdot}$$

$\boxed{\Delta \vdash U : \mathsf{Type}}$ Index-type U is well-kinded in Δ

$$\frac{}{\Delta \vdash \mathsf{nat} : \mathsf{Type}}$$

$\boxed{\Delta \vdash C : U}$ Index-term C has type U

$$\frac{}{\Delta \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Delta \vdash C : \mathsf{nat}}{\Delta \vdash \mathsf{suc}\ C : \mathsf{nat}} \qquad \frac{\Delta(X) = U}{\Delta \vdash X : U}$$

Figure 1. Index-Contexts, Index-Substitution, Index-Types, and Index-Terms

Requirement 1 (Index-Substitution Lemma)**.**
If $\Delta \vdash \theta : \Delta'$ *and* $\Delta' \vdash C : U$ *then* $\Delta \vdash C[\theta] : U[\theta]$.

Proof. By induction on $\Delta' \vdash C : U$. □

In addition to the typing rules (see Fig. 1) we also require that equality on the index language is decidable and takes into account the equality constraints in Δ. To illustrate we give the definition of equality for natural numbers in Fig. 2.

$\boxed{\Delta \vdash C_1 = C_2}$ Term C_1 is equal to Term C_2 in Δ.

$$\frac{}{\Delta \vdash \mathsf{zero} = \mathsf{zero}} \qquad \frac{\Delta \vdash C_1 = C_2}{\Delta \vdash \mathsf{suc}\ C_1 = \mathsf{suc}\ C_2} \qquad \frac{}{\Delta \vdash X = X}$$

$$\frac{X{:=}C'{:}U \in \Delta \quad \Delta \vdash C' = C}{\Delta \vdash X = C} \qquad \frac{X{:=}C'{:}U \in \Delta \quad \Delta \vdash C = C'}{\Delta \vdash C = X}$$

$$\frac{\# \in \Delta}{\Delta \vdash C_1 = C_2}$$

Figure 2. Equality of Index Terms

Typing of Index-Terms in (Co)Patterns As index-terms occur within (co)patterns, we also require rules that extract the type of index-variables. As we process a list of (co)pattern we thread through a context Δ and accumulate index variables and constraints. As subsequent index patterns may depend on variables appearing earlier in the (co)pattern spine, we extend and refine Δ by imposing constraints on existing variable declarations. In fact, our typing rules for (co)patterns will also solve equality constraints using unification. Hence the resulting index-context Δ' is an extension and refinement of Δ, written as $\Delta \prec \Delta'$. We define synthesizing free index variables contained in an index-term C in Fig. 3. We assume that C only contains fresh variables, i.e. any variable in C does not already occur in Δ. We are threading through Δ which may contain variables introduced by a previous pattern match, since for richer and more expressive index domains U may depend on Δ and it is more uniform.

$\boxed{\Delta \vdash C : U \searrow \Delta'}$ Index-Pattern C of index-type U synthesizes an index-context Δ' s.t. $\Delta \prec \Delta'$

$$\frac{}{\Delta \vdash \mathsf{zero} : \mathsf{nat} \searrow \Delta} \qquad \frac{\Delta \vdash C : \mathsf{nat} \searrow \Delta'}{\Delta \vdash \mathsf{suc}\ C : \mathsf{nat} \searrow \Delta'}$$

$$\frac{X \notin \Delta}{\Delta \vdash X : U \searrow \Delta, X{:}U}$$

Figure 3. Meta-Pattern Checking Rules

Type checking of (co)patterns will also need to solve equations $C_1 = C_2$ using unification on our index domain, and thus introduce term assignments to variables in Δ, yielding Δ'. We define unification for the index domain of natural numbers in Fig. 4. Note that our unification always succeeds in producing an index-context Δ'. However, if C_1 and C_2 were unifiable, then the arising equality constraints are recorded in Δ'. If C_1 and C_2 were not unifiable, we return a Δ' that contains a contradiction $\#$. There are two possible sources of failure: either the two terms are syntactically different or the occurs check fails. As we keep track of constraints in Δ, checking whether X occurs in Y where $Y{:=}C{:}U \in \Delta$, must check whether X occurs in C. Our well-formedness of Δ guarantees that our context is not cyclic and hence the occurs check will terminate. Our definition of unification is then straightforward. As we must guarantee that Δ remains well-formed, we may permute it to an equivalent well-formed context (written as $\Delta \sim \Delta'$) when unifying X with an index-term C.

Properties about Unification As we alluded during our examples, typing of (co)patterns will rely on solving equality constraints. We therefore rely on the correctness of the unification algorithm. In particular, we require that unification will always succeed.

Requirement 2 (Unification of Index-Terms)**.** *For any* Δ*,* C_1*,* C_2 *and* U *such that* $\Delta \vdash C_1 : U$ *and* $\Delta \vdash C_2 : U$*, there is a* Δ' *such that* $\Delta \vdash C_1 = C_2 \searrow \Delta'$.

Further, we require that our unification algorithm produces the most general unifier.

Requirement 3. *If* $\Delta \vdash C_1 = C_2 \searrow \Delta'$*, then for all* Δ_0 *and* θ *such that* $\Delta_0 \vdash \theta : \Delta$*, we have that* $\Delta_0 \vdash C_1[\theta] = C_2[\theta]$ *if and only if* $\Delta_0 \vdash \theta : \Delta'$.

Proof. By induction on $\Delta \vdash C_1 = C_2 \searrow \Delta'$. □

As we mentioned, our operational semantics for our programs is environment based and our index-substitution θ provides instantiations for all the index variables. Proving type safety of our core language relies on a series of properties our index domain must satisfy. We prove these properties for our index domain nat. First, we rely on a substitution property of index-terms occurring in patterns.

Using these requirements, we can show that the unification algorithm is stable under substitution.

Lemma 1. *If* $\Delta \vdash C_1 = C_2 \searrow \Delta'$ *and* $\Delta_1 \vdash \theta : \Delta$ *then there is a* Δ_1' *such that* $\Delta_1 \vdash C_1[\theta] = C_2[\theta] \searrow \Delta_1'$ *and* $\Delta_1' \vdash \theta : \Delta'$.

Proof. By Req. 2 and Req. 3. □

$\boxed{\Delta \vdash C_1 = C_2 \searrow \Delta'}$ Given the index-terms C_1 and C_2 synthesize the most general index-context Δ' s.t. $\Delta \prec \Delta'$ and $\Delta' \vdash C_1 = C_2$.

$$\frac{}{\Delta \vdash \mathsf{zero} = \mathsf{zero} \searrow \Delta} \quad \frac{\Delta \vdash C_1 = C_2 \searrow \Delta'}{\Delta \vdash \mathsf{suc}\, C_1 = \mathsf{suc}\, C_2 \searrow \Delta'} \quad \frac{}{\Delta \vdash X = X \searrow \Delta}$$

$$\frac{}{\Delta \vdash \mathsf{zero} = \mathsf{suc}\, C \searrow \Delta, \#} \quad \frac{}{\Delta \vdash \mathsf{suc}\, C = \mathsf{zero} \searrow \Delta, \#}$$

$$\frac{\Delta \sim \Delta_0, X{:}U, \Delta_1 \quad \Delta_0 \vdash C : U}{\Delta \vdash X = C \searrow \Delta_0, X{:=}C{:}U, \Delta_1} \quad \frac{\Delta \sim \Delta_0, X{:}U, \Delta_1 \quad \Delta_0 \vdash C : U}{\Delta \vdash C = X \searrow \Delta_0, X{:=}C{:}U, \Delta_1}$$

$$\frac{\Delta \vdash \mathsf{occurs}^{n+1}(X, C)}{\Delta \vdash X = C \searrow \Delta, \#} \quad \frac{\Delta \vdash \mathsf{occurs}^{n+1}(X, C)}{\Delta \vdash C = X \searrow \Delta, \#}$$

$$\frac{X{:=}C'{:}U \in \Delta \quad \Delta \vdash C' = C \searrow \Delta'}{\Delta \vdash X = C \searrow \Delta'} \quad \frac{X{:=}C'{:}U \in \Delta \quad \Delta \vdash C = C' \searrow \Delta'}{\Delta \vdash C = X \searrow \Delta'}$$

$\boxed{\Delta \vdash \mathsf{occurs}^n(X, C)}$ X occurs in C under n constructors

$$\frac{}{\Delta \vdash \mathsf{occurs}^0(X, X)} \quad \frac{\Delta \vdash \mathsf{occurs}^n(X, C)}{\Delta \vdash \mathsf{occurs}^{n+1}(X, \mathsf{suc}\, C)} \quad \frac{Y{:=}C{:}U \in \Delta \quad \Delta \vdash \mathsf{occurs}^n(X, C)}{\Delta \vdash \mathsf{occurs}^n(X, Y)}$$

Figure 4. Unification of Index Terms

Pattern Matching on Index-Terms Our operational semantics for our language relies on (co)pattern matching. As index terms appear in (co)patterns, we rely on pattern matching on index terms. To prove preservation and progress, we rely on the fact that extracting the type of index-variables in (co)patterns is stable and does not depend on the current run-time environment. We state this property in general terms, although our instantiation θ will be ground during run-time. It can be intuitively understood as follows: When we process a (co)pattern we build up a context Δ before we extend Δ with the new variables that occur in an index term C obtaining Δ'. Given an instantiation θ for the variables in Δ, where $\Delta_1 \vdash \theta : \Delta$, we can also process C within the context Δ_1 and extract the new variables obtaining an extension Δ_1'. In fact, as variables occurring in C are not (yet) instantiated by θ, we also have that $\Delta_1' \vdash \theta : \Delta_1$.

Requirement 4 (Substitution Property of Index-Terms)**.**
If $\Delta \vdash C : U \searrow \Delta'$ *and* $\Delta_1 \vdash \theta : \Delta$ *then there is* Δ_1' *such that* $\Delta_1 \vdash C : U[\theta] \searrow \Delta_1'$ *and* $\Delta_1' \vdash \theta : \Delta'$.

Proof. The proof of this statement is by induction on the derivation $\Delta \vdash C : U \searrow \Delta'$. □

Requirement 5 (Adequacy of Pattern Matching for Index-Terms)**.**
Suppose $\cdot \vdash C : U$. *If* $\vdash C' : U \searrow \Delta$ *and* $C = C'[\theta]$. *Then* $\cdot \vdash \theta : \Delta$.

Proof. By induction on the derivation $\cdot \vdash C' : U \searrow \Delta$. □

Coverage of Index-Terms To discuss coverage of (co)patterns, we rely on coverage of index-terms. We therefore define a splitting algorithm that takes as input $\Delta \vdash X$ where either $X{:}U \in \Delta$ or $X{:=}C{:}U \in \Delta$. It generates a set containing $\Delta_i \vdash C_i$ where C_i are the possible refinements of X (see Fig. 5).

Requirement 6 (Coverage of Splitting for Index Objects)**.**
Suppose $\cdot \vdash \theta : \Delta$ *and* $\mathrm{split}(\Delta \vdash X) = (\Delta_i, C_i)_{\forall i \in I}$, *then there is an* i *and* θ_i *such that* $\vdash \theta_i : \Delta_i$ *and* $\theta(X) = C_i[\theta_i]$ *and* $\theta \prec \theta_i$.

The judgment $\theta \prec \theta_i$ means that θ_i is of the form $\theta, \vec{C}/\vec{X}$ for some terms $\vec{C} = C_1, \ldots, C_n$ and variables $\vec{X} = X_1, \ldots, X_n$ that are not already bound by θ.

Proof of Req. 6. By induction on the splitting judgment. □

We also require that the splitting algorithm preserves coverage under the application of a substitution.

Requirement 7 (Preservation of Splitting under Substitution)**.**
Suppose $\Delta' \vdash \theta : \Delta$ *and* $\mathrm{split}(\Delta \vdash X) = (\Delta_i \vdash C_i)_{\forall i \in I}$, *then there are* $\{\Delta_i'\}_{i \in I}$ *such that* $\mathrm{split}(\Delta' \vdash X) = (\Delta_i' \vdash C_i)_{\forall i \in I}$ *and for all* $i \in I$, $\Delta_i' \vdash \theta : \Delta_i$.

3.2 Indexed Types and Kinds

Following Levy (2001) we distinguish between positive types (1, $\Sigma X{:}U.P$, $P_1 \times P_2$) which characterize finite data and negative types ($P \to N$, $\Pi X{:}U.N$) which describes infinite data. We allow negative types to be embedded into positive types and vice versa using explicit coercions written as $\downarrow N$ and $\uparrow P$ respectively. Our language also supports indexed recursive and indexed corecursive types. The recursive type written as $\mu Y.\lambda\vec{X}.D$ is a positive type, as it allows us to construct finite data using labelled sums D (written as $\langle c\ \overrightarrow{P} \rangle$). While Y denotes a type variable, $\lambda\vec{X}.D$ describes a type-level function which expects index objects and returns a labelled sum D. Dually, in the corecursive type, written as $\nu Z.\lambda\vec{X}.R$, the type-level function $\lambda\vec{X}.R$ expects index objects and returns a record of indexed observations. Corecursive types are negative types, as they describe infinite data using records R (standing for $\{\overrightarrow{d{:}N}\}$).

Index objects from our index domain U can be embedded and returned by computations by returning an object of type $\Sigma X{:}U.1$. Our core language also includes equality constraints between index objects. They typically are used inside (co)recursive type definitions. As we have seen in the examples, we mostly use equalities in two forms: constrained products (written as $C_1 = C_2 \times P$) in defining indexed data types and constrained (or guarded) function (written as $C_1 = C_2 \to N$) in defining indexed codata types. As we require that our index domain comes with decidable equality, we believe that the equality proofs can always be reconstructed when elaborating source level programs into our core language.

Our computation-level types can directly refer to index types. In this article, both $\mu Y.\lambda\vec{X}.D$ and $\nu Z.\lambda\vec{X}.R$ are just *recursive types* rather than inductive and coinductive types resp. Since D and R

$\boxed{\text{split}(\Delta \vdash X) = \{\Delta_i \vdash C_i\}_{i\in I}}$ Splitting the index-variable X in Δ yields a complete and non-redundant set of refinements $\Delta_i \vdash C_i$

$$\frac{}{\text{split}(\Delta, X{:}\mathsf{nat}, \Delta' \vdash X) = \{(\Delta, Y{:}\mathsf{nat}, \Delta'[\mathsf{suc}\ Y/X] \vdash \mathsf{suc}\ Y)\ ,\ (\Delta, \Delta'[\mathsf{zero}/X] \vdash \mathsf{zero})\}}$$

$$\frac{\text{split}(\Delta, X{:}U \vdash X) = \{\Delta_i \vdash C_i\}_{i\in I} \quad \text{for each i, } \Delta_i \vdash C_i = C \searrow \hat{\Delta}_i}{\text{split}(\Delta, X{:=}C{:}U, \Delta' \vdash X) = \{\hat{\Delta}_i, \Delta'[C/X] \vdash C_i\}_{i\in I}}$$

Figure 5. Splitting of Index-Variable

$$\begin{array}{lll}
\text{Kinds} & K & ::= \ \mathsf{type} \mid \Pi X{:}U.K \\
\text{Positive Types} & P & ::= Y \mid 1 \mid P_1 \times P_2 \mid C_1 = C_2 \mid \downarrow N \\
& & \quad \mid \mu Y.\lambda\overrightarrow{X}.D \mid P\ \vec{C} \mid \Sigma X{:}U.P \\
\text{Negative Types} & N & ::= Z \mid P \to N \mid \uparrow P \\
& & \quad \mid \nu Z.\lambda\overrightarrow{X}.R \mid N\vec{C} \mid \Pi X{:}U.N \\
\text{Variants} & D & ::= \langle c_1\ P_1 \mid \cdots \mid c_n\ P_n\rangle \\
\text{Records} & R & ::= \{d_1 : N_1, \ldots, d_n : N_n\}
\end{array}$$

Figure 6. Types

are not checked for functoriality and programs are not checked for termination or productivity, resp., there are no conditions that ensure $\mu Y.\lambda\vec{X}.D$ to be a least fixed-point inhabited only by finite data, and $\nu Z.\lambda\vec{X}.R$ to be a greatest fixed-point that hosts infinite objects which are productive. However, we keep the notational distinction to allude to the intended interpretation as least and greatest fixed-points in a total setting.

Example 1: Indexed Recursive Types Data types $C = \mu Y.\lambda\vec{X}.D$ for $D = \langle c_1\ P_1 \mid \cdots \mid c_n\ P_n\rangle$ describe least fixed points. Choosing as index domain natural numbers, we can model our previous definition of `Msg` as follows in our core language.

$$\begin{array}{l}
\mu\text{Msg}.\lambda X.\langle \mathsf{Nil} \ \ : X = \mathsf{zero} \times 1\ , \\
\qquad\qquad \mathsf{Cons} : \Sigma Y{:}\mathsf{nat}.X = \mathsf{suc}\ Y \times (\mathsf{Bit} \times \text{Msg } Y)\rangle
\end{array}$$

Example 2: Indexed Corecursive Types Record types $C = \nu Z.\lambda\vec{X}.R$ with $R = \{d_1 : N_1, \ldots, d_n : N_n\}$ are recursive labeled products and describe infinite data. As for data, non-recursive record types are encoded by a void ν-abstraction $\nu\_.\lambda\vec{X}.R$. Consider our previous codata type definition for indexed streams, i.e. `Str`, with the three observations, `GetBit`, `NextBits`, and `Done`. Depending on the index `N` we choose the corresponding observation. It directly translates to the following:

$$\begin{array}{lll}
\nu\text{Str}.\lambda M.\ \{\ \mathsf{Done} & : & M = \mathsf{zero} \ \to \uparrow \mathsf{NextMsg}\ , \\
\quad \mathsf{NextBits} & : & \Pi N{:}\mathsf{nat}.M = \mathsf{suc}\ N \to \text{Str } N\ , \\
\quad \mathsf{GetBit} & : & \Pi N{:}\mathsf{nat}.M = \mathsf{suc}\ N \to \uparrow \mathsf{Bit} \qquad\ \}
\end{array}$$

$$\mu\text{NextMsg}.\ \langle\ \mathsf{NextMsg}\ :\ \Sigma N{:}\mathsf{nat}.\downarrow \mathsf{Str}\ N\ \rangle$$

Dually to data types where we employ Σ and product types, we use Π and simple function types when defining codata types.

3.3 Terms and Typing

In our core language, we distinguish between terms which have negative type and values which have positive type (see Fig. 7). Values include unit (written as $()$), pairs (written as (v_1, v_2)), dependent pairs (written as $\mathsf{pack}\,\langle C, v\rangle$). We also include data built using constructors (written as $c\ v$). Finally we can embed computation into values using $\mathsf{thunk}\ t$. A thunk represents a term which is suspended and may produce a value at a later stage. Last but not least, we include the witness for equality between two index objects, written as $\wp$, in our values.

$$\begin{array}{lll}
\text{Values} & v & ::= x \mid () \mid (v_1, v_2) \mid \wp \mid \mathsf{thunk}\ t \mid c\ v \mid \mathsf{pack}\,\langle C, v\rangle \\
\text{Terms} & t & ::= \mathsf{rec}\ f.t \mid \mathsf{fn}\ \vec{u} \mid t\ v \mid t\ C \mid \mathsf{produce}\ v \mid t.d \\
& & \quad \mid t_1\ \mathsf{to}\ x.t_2 \mid \mathsf{force}\ v \\
\text{Branches} & u & ::= q \mapsto t \mid q \\
\text{Patterns} & p & ::= x \mid () \mid (p_1, p_2) \mid \wp \mid c\ p \mid \mathsf{pack}\,\langle C, p\rangle \\
\text{Copatterns} & q & ::= \cdot \mid p\ q \mid C\ q \mid .d\ q
\end{array}$$

Figure 7. Values, Terms, (Co)Patterns

Computations (or terms) correspond to negative types. Computations include recursion (written as $\mathsf{rec}\ f.t$) and functions (written as $\mathsf{fn}\ \vec{u}$) which are defined by (co)pattern matching. In addition, we have application (written as $t\ v$), index domain application (written as $t\ C$) and destructor applications (written as $t.d$); given a term t describing infinite data we unfold its corresponding corecursive type to a record and select the component d of the record. Finally, we can force a suspended computation v using $\mathsf{force}\ v$ and produce a value (written as $\mathsf{produce}\ v$). We also include a sequencing term which is written as $(t_1\ \mathsf{to}\ x.t_2)$.

We eliminate expressions of positive type such as recursive types via pattern matching; dually, we make observations about expressions of negative types such as corecursive types. Simultaneous (co)patterns are described using a spine that is built out of patterns (written as p) and observations (written as `.d`). Patterns themselves are derived from values and can be defined using pattern variables x, pairs (written as (p_1, p_2)), pattern instances (written as $\mathsf{pack}\,\langle C, p\rangle$) and patterns formed with a data constructor c.

Branches in case-expressions are modelled by $q \mapsto t$. We also allow branches with no body – they will only succeed if the copattern q is impossible, i.e. we arrived at some equality constraints that lead to a contradiction. Strictly speaking, it is not necessary as we could always write some arbitrary expression for the body which would be inaccessible and thus can never be reached.

The typing rules for terms and values are mostly straightforward (see Fig. 8). We highlight here a few. Typing of index object C refers to typing of index-terms as described in Section 3.1. A constructor takes a term of type $D_c[\mu Y.\lambda\vec{X}.D/Y, \vec{C}/\vec{X}]$, yielding a term of type $(\mu Y.\lambda\vec{X}.D)\ \vec{C}$. A thunk of a computation is well typed, if the computation itself is. The witness for an equality $C_1 = C_2$ is simply $\wp$ provided C_1 and C_2 are equal in our index domain using our rules from Fig. 2. As we have constraints in Δ, we also include type conversion rules (T_{PConv} and T_{NConv}). $\Delta \vdash P = P'$ (and resp. $\Delta \vdash N = N'$) is defined inductively on the structure of positive and negative types. When we compare $\Delta \vdash (P\ \vec{C}) = (P'\ \vec{C}')$, we simply compare $\Delta \vdash P = P'$ and for all i we have $\Delta \vdash C_i = C'_i$

$\boxed{\Delta;\Gamma \vdash v : P}$ Value typing: In index-context Δ and context Γ, value v has positive type P.

$$\frac{}{\Delta;\Gamma \vdash () : 1}\ \mathrm{T_{Unit}} \quad \frac{\Gamma(x) = P}{\Delta;\Gamma \vdash x : P}\ \mathrm{T_{Var}} \quad \frac{\Delta;\Gamma \vdash v_1 : P_1 \quad \Delta;\Gamma \vdash v_2 : P_2}{\Delta;\Gamma \vdash (v_1, v_2) : P_1 \times P_2}\ \mathrm{T_{Pair}} \quad \frac{\Delta \vdash C : U \quad \Delta;\Gamma \vdash v : P[C/X]}{\Delta;\Gamma \vdash \mathsf{pack}\,\langle C, v\rangle : \Sigma X{:}U.P}\ \mathrm{T_{Pack}}$$

$$\frac{\Delta;\Gamma \vdash v : D_c[\mu Y.\lambda\vec{X}.D/Y, \vec{C}/\vec{X}]}{\Delta;\Gamma \vdash c\,v : (\mu Y.\lambda\vec{X}.D)\ \vec{C}}\ \mathrm{T_{Const}} \quad \frac{\Delta;\Gamma \vdash v : P' \quad \Delta \vdash P = P'}{\Delta;\Gamma \vdash v : P}\ \mathrm{T_{PConv}}$$

$$\frac{\Delta \vdash C_1 = C_2}{\Delta;\Gamma \vdash \wp : C_1 = C_2}\ \mathrm{T_{CProd}} \quad \frac{\Delta;\Gamma \vdash t : N}{\Delta;\Gamma \vdash \mathsf{thunk}\ t : \downarrow N}\ \mathrm{T_{Thunk}}$$

$\boxed{\Delta;\Gamma \vdash t : N}$ Computation typing: In index-context Δ and context Γ, term t has negative type N.

$$\frac{\Delta;\Gamma, x : \downarrow N \vdash t : N}{\Delta;\Gamma \vdash \mathsf{rec}\ x.t : N}\ \mathrm{T_{Rec}} \quad \frac{\text{for each } i\ \Delta;\Gamma \vdash u_i : N}{\Delta;\Gamma \vdash \mathsf{fn}\ \vec{u} : N}\ \mathrm{T_{Fn}} \quad \frac{\Delta;\Gamma \vdash t : (\nu Z\lambda\vec{X}R)\ \vec{C}}{\Delta;\Gamma \vdash t.d : R_d[\nu Z.\lambda\vec{X}.R/Z, \vec{C}/\vec{X}]}\ \mathrm{T_{Dest}}$$

$$\frac{\Delta;\Gamma \vdash t : P \to N \quad \Delta;\Gamma \vdash v : P}{\Delta;\Gamma \vdash t\ v : N}\ \mathrm{T_{App}} \quad \frac{\Delta;\Gamma \vdash t : \Pi X{:}U.N \quad \Delta \vdash C : U}{\Delta;\Gamma \vdash t\ C : N[C/X]}\ \mathrm{T_{MApp}} \quad \frac{\Delta;\Gamma \vdash t : N' \quad \Delta \vdash N = N'}{\Delta;\Gamma \vdash t : N}\ \mathrm{T_{NConv}}$$

$$\frac{\Delta;\Gamma \vdash v : \downarrow N}{\Delta;\Gamma \vdash \mathsf{force}\ v : N}\ \mathrm{T_{Force}} \quad \frac{\Delta;\Gamma \vdash v : P}{\Delta;\Gamma \vdash \mathsf{produce}\ v : \uparrow P}\ \mathrm{T_{Produce}} \quad \frac{\Delta;\Gamma \vdash t_1 : \uparrow P \quad \Delta;\Gamma, x : P \vdash t_2 : N}{\Delta;\Gamma \vdash t_1\ \mathsf{to}\ x.t_2 : N}\ \mathrm{T_{To}}$$

$\boxed{\Delta;\Gamma \vdash u_i : N}$ In index-context Δ and context Γ, branch u_i has negative type N.

$$\frac{\Delta;\Gamma;N \vdash q \searrow \Delta';\Gamma';N' \quad \Delta';\Gamma' \vdash t : N'}{\Delta;\Gamma \vdash q \mapsto t : N} \quad \frac{\Delta;\Gamma;N \vdash q \searrow \Delta';\Gamma';N' \quad \# \in \Delta'}{\Delta;\Gamma \vdash q : N}$$

Figure 8. Typing Rules for Terms

falling back to the comparison on index terms. We proceed similarly when comparing negative types.

A rec-expression introduces a variable of type $\downarrow N$. Dual to a constructor, an observation $.d$ takes a term of type $(\nu Z.\lambda\vec{X}.R)\ \vec{C}$ yielding a term of type $R_d[\nu Z.\lambda\vec{X}.R/Z, \vec{C}/\vec{X}]$. For applications we ensure that we apply a term of function type to a value. The operational reading of t_1 to $x.t_2$ is that we first evaluate the computations of t_1 to $\mathsf{produce}\ v_1$ of type $\uparrow P$, and then evaluate the term t_2 where we replace x by the value v_1. This is captured in the typing rule for to-statements.

The function abstraction (written as $\mathsf{fn}\ \vec{u}$) introduces branches u of the form $q \mapsto t$. A branch is well typed if the copattern q checks against the overall type N of the function and synthesizes a new index-context Δ', a new context Γ', and the output type N', against which the term t is checked. The contexts Δ' and Γ' describe the types of the variables occurring in the pattern together with equality constraints. Note that Δ' not only accumulates equality constraints, but might also contain a contradiction, if some equality constraint is not satisfied. Our typing rules will then still guarantee that the body t is effectively simply typed, as all equalities that appear in the body and must be satisfied will be trivially true.

As mentioned earlier, we also allow branches that consist only of a (co)pattern but have no body. This allows programmers to write inaccessible (co)patterns. We check such branches by verifying that Δ' contains a contradiction. In this case, we know that the branch cannot be taken during run-time and is essentially dead-code.

The typing rules for (co)patterns (see Fig. 9) are defined using the following two judgments:

$\Delta;\Gamma$	$\vdash$	$p : P$	$\searrow \Delta';\Gamma'$	Typing for pattern p
$\Delta;\Gamma;N$	$\vdash$	q	$\searrow \Delta';\Gamma';N'$	Typing for copattern q

In both typing judgments, the index-context Δ and the context Γ contain variable declarations that were introduced at the outside. We assume that all variables occurring in the (co)pattern are fresh with respect to Δ and Γ and occur linearly, although this is not explicitly enforced in our rules. When we check a pattern p against a positive type P in the index-context Δ and context Γ, we synthesize an index-context Δ' such that Δ' is an extension of Δ (i.e. $\Delta \prec \Delta'$) and Γ' is an extension of Γ. We note that as we check the pattern p we may update and constrain some of the variables already present in Δ. This happens in the rule $\mathrm{P_{con}}$ where we fall back to type checking patterns in our domain and in the rule $\mathrm{P_{Eq}}$ where we unify two index objects C_1 and C_2, and return a new index-context Δ' such that $\Delta' \vdash C_1 = C_2$. For simplicity, we thread through both the index-context Δ and the context Γ, although only Δ may actually be refined.

The typing rules for patterns are straightforward except for equality. A pattern $\wp$ checks against $C_1 = C_2$ provided that C_1 and C_2 unify in our domain and Δ' contains the solution which makes C_1 and C_2 equal. It might also be the case that C_1 does not unify with C_2, i.e. there is no instantiation for the index-variables in C_1 and C_2 that makes C_1 and C_2 equal. In this case, we expect the judgment $\Delta \vdash C_1 = C_2 \searrow \Delta'$ to introduce in Δ' a contradiction $\#$ which will make typing of the expression in the branch trivial. This is necessary for the substitution lemma to hold.

Copattern spines allow us to make observations on a negative type N in the index-context Δ and context Γ. As we process the copattern spine from left to right, we synthesize a negative type N'. Intuitively, N' is the suffix of N. As copattern spines also contain patterns we also return a new index-context Δ' and context Γ'.

To illustrate we show the partial typing derivation (see Fig. 9) for the copattern spine `[s N] s ns .GetBit` that arises from the program `MsgStr` from Sec. 2.1. This copattern spine is represented in our core language as $(\mathtt{s}\ N)\ s\ ns\ \mathtt{.GetBit}\ M\ \wp$ and has type $\Pi N{:}\mathtt{nat.Bit\ Stream} \to \mathtt{[nat]\ Stream} \to \mathtt{Str}\ N$. After inferring the type of N and introducing declarations for s and ns,

$\boxed{\Delta;\Gamma \vdash p : P \searrow \Delta';\Gamma'}$ Pattern p of positive type P extends contexts $\Delta;\Gamma$ into $\Delta';\Gamma'$.

$$\frac{}{\Delta;\Gamma \vdash x : P \searrow \Delta;\Gamma, x{:}P}\ \mathrm{P_{Var}} \qquad \frac{\Delta;\Gamma \vdash p : D_c[\mu Y.\lambda\vec{X}.D/Y, \vec{C}/\vec{X}] \searrow \Delta';\Gamma'}{\Delta;\Gamma \vdash c\ p : (\mu Y.\lambda\vec{X}.D)\ \vec{C} \searrow \Delta';\Gamma'}\ \mathrm{P_{Const}}$$

$$\frac{}{\Delta;\Gamma \vdash () : 1 \searrow \Delta;\Gamma}\ \mathrm{P_{Unit}} \qquad \frac{\Delta;\Gamma \vdash p_1 : P_1 \searrow \Delta';\Gamma' \quad \Delta';\Gamma' \vdash p_2 : P_2 \searrow \Delta'';\Gamma''}{\Delta;\Gamma \vdash (p_1, p_2) : P_1 \times P_2 \searrow \Delta'';\Gamma''}\ \mathrm{P_{Pair}}$$

$$\frac{\Delta \vdash C : U \searrow \Delta' \quad \Delta';\Gamma \vdash p : P[C/X] \searrow \Delta'';\Gamma'}{\Delta;\Gamma \vdash \mathsf{pack}\,\langle C, p\rangle : \Sigma X{:}U.P \searrow \Delta'';\Gamma'}\ \mathrm{P_{Pack}} \qquad \frac{\Delta \vdash C_1 = C_2 \searrow \Delta'}{\Delta;\Gamma \vdash \wp : C_1 = C_2 \searrow \Delta';\Gamma}\ \mathrm{P_{Eq}}$$

$\boxed{\Delta;\Gamma;N \vdash q \searrow \Delta';\Gamma';N'}$ Copattern q eliminates negative type N into type N' and extending contexts $\Delta;\Gamma$ into $\Delta';\Gamma'$.

$$\frac{}{\Delta;\Gamma;N \vdash \cdot \searrow \Delta;\Gamma;N}\ \mathrm{CP_{Base}} \qquad \frac{\Delta;\Gamma \vdash p : P \searrow \Delta';\Gamma' \quad \Delta';\Gamma';N \vdash q \searrow \Delta'';\Gamma'';N'}{\Delta;\Gamma;P \to N \vdash p\ q \searrow \Delta'';\Gamma'';N'}\ \mathrm{CP_{App}}$$

$$\frac{\Delta \vdash C : U \searrow \Delta' \quad \Delta';\Gamma;N[C/X] \vdash q \searrow \Delta'';\Gamma';N'}{\Delta;\Gamma;\Pi X{:}U.N \vdash C\ q \searrow \Delta'';\Gamma';N'}\ \mathrm{CP_{MApp}} \qquad \frac{\Delta;\Gamma;R_d[(\nu Z.\lambda\vec{X}.R)/Z, \vec{C}/\vec{X}] \vdash q \searrow \Delta';\Gamma';N'}{\Delta;\Gamma;(\nu Z.\lambda\vec{X}.R)\ \vec{C} \vdash .d\ q \searrow \Delta';\Gamma';N'}\ \mathrm{CP_{Dest}}$$

Example:

$$\dfrac{\dfrac{\vdots}{N : \mathtt{nat} \vdash M : \mathtt{nat} \searrow \Delta_0} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Delta_0 \vdash \mathrm{s}\ M = \mathrm{s}\ N \searrow \Delta_1} \qquad \Delta_1 = N : \mathtt{nat}, M := N : \mathtt{nat}}{\Delta_0 \vdash \wp : \mathrm{s}\ M = \mathrm{s}\ N \searrow \Delta_1} \qquad \dfrac{}{\Delta_1;\Gamma;\mathtt{Str}\ M \vdash \cdot \searrow \Delta_1;\Gamma;\mathtt{Str}\ M}}{\Delta_0;\Gamma;\mathrm{s}\ M = \mathrm{s}\ N \to \mathtt{Str}\ M \vdash \wp \cdot \searrow \Delta_1;\Gamma;\mathtt{Str}\ M}}{N : \mathtt{nat};\Gamma;\Pi M{:}\mathtt{nat}.\mathrm{s}\ M = \mathrm{s}\ N \to \mathtt{Str}\ M \vdash M\ \wp \cdot \searrow \Delta_1;\Gamma;\mathtt{Str}\ M}}{N : \mathtt{nat};\Gamma;\mathtt{Str}\ N \vdash .\mathtt{GetBit}\ M\ \wp \cdot \searrow \Delta_1;\Gamma;\mathtt{Str}\ M}$$

where $\Gamma = s : \mathtt{Bit\ Stream}, ns : \mathtt{[nat]\ Stream}$ and $\Delta_0 = N : \mathtt{nat}, M : \mathtt{nat}$

Figure 9. Type Checking for Patterns

we synthesize $\mathtt{Str}\ M$ for $.\mathtt{GetBit}\ M\ \wp \cdot$ in Δ_1 and the context $\Gamma = s{:}\mathtt{Bit\ Stream}, ns{:}\mathtt{[nat]\ Stream}$. Recall, we write $\mathtt{[nat]}$ as a notation for $\Sigma X{:}\mathtt{nat}.1$.

Example 3 Recall our previous program `genBitStr` which generated a stream where every message consisted of two bits. This program can be elaborated into our core language straightforwardly to a program of type $\mathtt{Str}\ 2$.

```
rec genBitStr.fn
  | .GetBit (suc zero) ℘                          ↦ RandomBitGen ()
  | .NextBits (suc zero) ℘ .GetBit z ℘ ↦ RandomBitGen ()
  |  .NextBits (suc zero) ℘ .NextBits (suc zero) ℘ .Done ℘
                  ↦ NextMsg (pack ⟨(suc zero), genBitStr⟩)
```

Example 4 Next, we consider the translation of `readMsg`.

```
rec readMsg.fn
  | zero        s  ↦ produce (Nil (℘, () ), s)
  | (suc M) s      ↦
  (force s).GetBit to c.
    (force readMsg) M (thunk (force s).NextBits) to x.
      (fn (w, s') ↦
         produce (Cons (pack ⟨M, (℘, (c, w))⟩ , s'))) x
```

This function deserves some explanation. The type of `readMsg` is translated to $\Pi N{:}\mathtt{nat}.\downarrow(\mathtt{Str}\ N) \to \uparrow((\mathtt{Msg}\ N) \times \downarrow(\mathtt{Str\ z}))$. Since $\mathtt{Str}\ N$ is in negative position, it needs to have positive type (thus the $\downarrow$) and so the input s of the function is in fact a thunk that needs to be forced before we can use the observations `GetBit` and `NextBits`. The recursive call needs also to be forced because the variable `readMsg` needs to be positive to live in the context. Let-statements are defined as to-statements whose left-hand side produces a value, that is then bound to the variable c and x, respectively. Moreover, the second let-statement in the original program also used pattern matching. Our language does not have case-expression. Hence, we use a function to pattern match on x. The output needs to be of negative type but we want to return a product which is positive. It is thus embedded using a produce-statement. This also allows the recursive call to be put on the left-hand side of a to-statement.

4. Evaluation and Type Preservation

In this section, we present a small step operational semantics using evaluation contexts (continuations) following Levy (2001). We also define a non-deterministic coverage algorithm and prove that our operational semantics satisfies subject reduction and progress.

4.1 Evaluation Contexts

Evaluation contexts are defined inductively. We start from a hole $\cdot$ and we accumulate values, index objects, observations, and suspended to-bindings.

$$\text{Evaluation Contexts } E ::= \cdot \mid v\ E \mid C\ E \mid .d\ E \mid ([]\ \mathsf{to}\ x.t)\ E$$

We note that we only collect closed values, index objects, etc. in the evaluation context and hence the typing judgment for them does not carry any contexts. We use the following judgment to define well-typed evaluation contexts:

$$N \vdash E \searrow N' \quad \text{Evaluation context } E \text{ transforms } N \text{ to } N'$$

The negative type N describes some computation t which when used in the evaluation context E returns a computation of type N'.

Intuitively, t stands for a function $\mathsf{fn}\ (\overrightarrow{q_i \mapsto t_i})$ and we match the evaluation context E against the copattern spine q_i and consume part of E to take a step. As evaluation contexts closely correspond to copattern spines, their typing rules follow the ones for (co)patterns.

When the evaluation context is empty (rule $\mathrm{E}_{\mathrm{Base}}$), we simply return N. Intuitively, nothing is applied to the computation of type N. If we have a computation of type $P \to N$ and our evaluation context provides a value v of type P, then we check that, given a computation of type N, applying the remaining evaluation context takes us to N' (see $\mathrm{E}_{\mathrm{App}}$). If we have a computation of type $\Pi X{:}U.N$ and the evaluation context supplies an index object C, then we verify that, given a computation of type $N[C/X]$, applying the remaining evaluation context takes us to N'. Similarly, given a term of type $(\nu Z.\lambda\vec{X}.R)\vec{C}$ and an evaluation context that supplies an observation $.d$, we verify that, given a computation of type $R_d[(\nu Z.\lambda\vec{X}.R)/Z, \vec{C}/\vec{X}]$, applying the remaining evaluation context takes us to N'.

Finally, given a computation of type $\uparrow P$ and an evaluation context $([]\ \mathsf{to}\ x.t)\ E$, we check that once we are done evaluating t and return a computation of type N, passing to it the remaining evaluation context E yields a computation of type N'.

$$\frac{}{N \vdash \cdot \searrow N}\ \mathrm{E}_{\mathrm{Base}} \qquad \frac{R_d[(\nu Z.\lambda\vec{X}.R)/Z, \vec{C}/\vec{X}] \vdash E \searrow N}{(\nu Z.\lambda\vec{X}.R)\vec{C} \vdash .d\ E \searrow N}\ \mathrm{E}_{\mathrm{Dest}}$$

$$\frac{\vdash v : P \quad N \vdash E \searrow N'}{P \to N \vdash v\ E \searrow N'}\ \mathrm{E}_{\mathrm{App}} \qquad \frac{x : P \vdash t : N \quad N \vdash E \searrow N'}{\uparrow P \vdash ([]\ \mathsf{to}\ x.t)\ E \searrow N'}\ \mathrm{E}_{\mathrm{to}}$$

$$\frac{N[C/X] \vdash E \searrow N' \quad \vdash C : U}{\Pi X{:}U.N \vdash C\ E \searrow N'}\ \mathrm{E}_{\mathrm{MApp}}$$

4.2 Small Step Operational Semantics

Unlike the language described by Abel et al. (2013) which presented programs as rewrite rules, we give here the operational semantics in a more traditional functional programming style using a continuation-based abstract machine semantics. We might view our language as a core language into which we can compile programs given as rewrite rules to. More importantly it directly gives rise to an implementation and illustrates how to extend more traditional ML-like languages with copattern matching.

Our operational semantics is defined on configurations $t; E$ which contain a term and an evaluation context. Such pair is said to have type N' (written $\vdash t; E : N'$) if $\vdash t : N$ and $N \vdash E \searrow N'$. The rules for the operational semantics on configurations are defined in Fig. 10. To evaluate an expression $t_1\ \mathsf{to}\ x.t_2$, we evaluate t_1 in the evaluation context extended with $[]\ \mathsf{to}\ x.t_2$. Once we have a value v for t_1 we pop off $[]\ \mathsf{to}\ x.t_2$ and continue evaluating $t_2[v/x]$. Forcing thunks continues the evaluation. When processing applications (i.e. applications to a value, an index object or an observation), we simply extend our evaluation context accordingly until we step a configuration $\mathsf{fn}\ (\overrightarrow{q_i \mapsto t_i}); E$. In this case, we match the evaluation context E against the copattern spine q_i yielding $(\theta; \sigma)$ and the tail E', and then step to $t_i[\theta; \sigma]$. The condition $E = q_i[\theta; \sigma]@E'$ denotes that appending q_i under the substitutions with E' results in the original evaluation context E.

Next, we prove that types are preserved during evaluation (see Theorem 4). This relies on substitution lemmas for values and computations and adequacy of copattern matching. For convenience, we describe below well-typed environments (θ, σ) and generalize the relationship between the computation of the type N and an evaluation contexts E that transforms N into N'.

$$\frac{\Delta' \vdash \theta : \Delta \quad \Delta'; \Gamma' \vdash \sigma : \Gamma[\theta]}{\Delta'; \Gamma' \vdash (\theta; \sigma) : (\Delta; \Gamma)} \qquad \frac{\vdash (\theta; \sigma) : (\Delta; \Gamma) \quad N[\theta] \vdash E \searrow N'}{\vdash (\theta; \sigma; E) : (\Delta; \Gamma; N) \searrow N'}$$

$\boxed{t_1; E_1 \longrightarrow t_2; E_2}$ $t_1; E_1$ evaluates to $t_2; E_2$ in one step.

$$\begin{array}{lcl}
t_1\ \mathsf{to}\ x.t_2; E & \longrightarrow & t_1; ([]\ \mathsf{to}\ x.t_2)\ E \\
\mathsf{produce}\ v; ([]\ \mathsf{to}\ x.t)\ E & \longrightarrow & t[v/x]; E \\
\mathsf{force}\ (\mathsf{thunk}\ t); E & \longrightarrow & t; E \\
t.d; E & \longrightarrow & t; .d\ E \\
t\ v; E & \longrightarrow & t; v\ E \\
t\ C; E & \longrightarrow & t; C\ E \\
\mathsf{rec}\ x.t; E & \longrightarrow & t[\mathsf{thunk}\ (\mathsf{rec}\ x.t)/x]; E
\end{array}$$

$$\frac{E = q_i[\theta; \sigma]@E'}{\mathsf{fn}\ (\overrightarrow{q_i \mapsto t_i}); E \longrightarrow t_i[\theta; \sigma]; E'}$$

Figure 10. Operational Semantics

Lemma 2 (Substitution Lemmas)**.** *The following hold*

1. *If* $\Delta; \Gamma \vdash v : P$ *and* $\Delta'; \Gamma' \vdash (\theta; \sigma) : (\Delta; \Gamma)$ *then* $\Delta'; \Gamma' \vdash v[\theta; \sigma] : P[\theta]$.
2. *If* $\Delta; \Gamma \vdash t : N$ *and* $\Delta'; \Gamma' \vdash (\theta; \sigma) : (\Delta; \Gamma)$ *then* $\Delta'; \Gamma' \vdash t[\theta; \sigma] : N[\theta]$.

Proof. The proof of both statements is done by mutual induction on the derivations of $\Delta; \Gamma \vdash v : P$ and $\Delta; \Gamma \vdash t : N$, respectively □

Lemma 3 (Adequacy of Copattern Matching)**.**

1. *Suppose* $\vdash v : P$. *If* $\vdash p : P \searrow \Delta; \Gamma$ *and* $v = p[\theta; \sigma]$ *then* $\vdash (\theta; \sigma) : (\Delta; \Gamma)$.
2. *Suppose* $N \vdash E \searrow N''$ *If* $N \vdash q \searrow \Delta; \Gamma; N'$ *and* $E = q[\theta; \sigma]@E'$, *then* $\vdash (\theta; \sigma; E') : (\Delta; \Gamma; N') \searrow N''$.

Proof. The proof is done by induction on $\vdash p : P \searrow \Delta; \Gamma$ and $N \vdash q \searrow \Delta; \Gamma; N'$. □

Theorem 4 (Type Preservation)**.**
If $\cdot; \cdot \vdash t; E : N$ *and* $t; E \longrightarrow t'; E'$, *then* $\vdash t'; E' : N$.

Proof. The proof is done by case analysis on the stepping rule. The only interesting case is when dealing with function abstraction.

$$\frac{E = q_i[\theta; \sigma]@E'}{\mathsf{fn}\ (\overrightarrow{q_i \mapsto t_i}); E \longrightarrow t_i[\theta; \sigma]; E'}$$

$\vdash \mathsf{fn}\ (\overrightarrow{q_i \mapsto t_i}); E : N$	by assumption
$\vdash \mathsf{fn}\ (\overrightarrow{q_i \mapsto t_i}) : N'$ and $N' \vdash E \searrow N$	by inversion
$\cdot; \cdot; N' \vdash q_i \searrow \Delta_i; \Gamma_i; N_i$ and $\Delta_i; \Gamma_i \vdash t_i : N_i$	by inversion
$\vdash \theta; \sigma; E' : \Delta_i; \Gamma_i; N_i \searrow N$	by lemma 3
$\vdash \theta; \sigma : \Delta_i; \Gamma_i$ and $N_i[\theta] \vdash E' \searrow N$	by inversion
$\vdash t_i[\theta; \sigma] : N_i[\theta]$	by substitution lemma
$\vdash t_i[\theta; \sigma]; E' : N$	by definition

□

4.3 Coverage

In this section, we define a notion of coverage for copatterns, which allows us to prove a type safety result.

To define coverage, we need to take into account that a function abstraction can be underapplied, i.e., it will not trigger a reduction step unless we add more to the evaluation context. To take into account such possibility, we need to introduce some notation. We define the append operation of evaluation contexts, denoted $E@k$, where $k = .d \mid v \mid []\ \mathsf{to}\ x.n \mid C$ which adds to the end of an evaluation context. We also use this operation on copatterns.

We now define coverage. The main judgment $\Delta; \Gamma; N \lhd| Q$ defined in Figure 11 means that the (finite) set Q of copatterns covers the type N in context $\Delta; \Gamma$. It is established by iteratively refining a covering set, beginning with the trivial copattern. It is easiest to read the rules from the top to the bottom. A covering set Q is refined by choosing a particular copattern $q \searrow \Delta'; \Gamma'; N'$ in Q and refining it further into a (finite) set of copatterns. This is accomplished using the auxiliary judgment $(q \searrow \Delta'; \Gamma'; N') \Longrightarrow Q'$, which states that the copattern q refines into the set of copatterns Q'.

There are two different types of refinement which can be done. The first one is introducing the result type. We look at the type of a particular rule and we introduce it. If we have an arrow type $P \to N$, we introduce a variable of that type, yielding the copattern $q@x$. If we have a corecursive type, for each observation $d \in R$, we create a new copattern $q@.d$ for each $d \in R$.

The second type of refinement is the splitting on a variable. We expose a variable occurring in q, and its type in Δ or Γ. We write $q[x]$ for a copattern q with a single distinguished position in which the variable x occurs. We consider in this judgment the contexts to be unordered, so the notation $\Gamma, x : P$ (or $\Delta, X : U$) is simply to expose any variable $x \in \Gamma$ ($X \in \Delta$, respectively), no matter its actual position in the context. The splitting is done by examining the type of the exposed variable. If $x : P_1 \times P_2$, we introduce two new variables $x_1 : P_1$ and $x_2 : P_2$ and perform the instantiation $q[(x_1, x_2)]$. If the variable is of recursive type $(\mu Y.\lambda \vec{X}.D)\vec{C}$, we introduce a new copattern for each constructor $c \in D$ with the variable replaced by $c\ x'$ where $x' : D_c[\mu Y.\lambda \vec{X}.D/Y, \vec{C}/\vec{X}]$. If we have an equality constraint $C_1 = C_2$, we attempt to unify them. If they cannot be unified, we record this copattern as unreachable, marking it with $\bot$. Again we omit for space reasons rules which perform further refinements on unreachable copatterns.

When splitting on an index variable in Δ, we use the splitting mechanism from the index domain, as discussed in Section 3.1 which produces a set of refined patterns $\{(\Delta_i, C_i) \mid i \in I\}$. We then return the refined set of copatterns $q[C_i/X]$ for each $i \in I$.

Our coverage algorithm generates a covering set Q. However, it does not account for writing overlapping and fall-through patterns. In this sense, our notion of coverage is not complete: there are sets Q of copatterns which a programmer might write in a program and one would consider covering, but for which one cannot derive $\Delta; \Gamma; N \lhd| Q$. However, it would be possible to check that for all copattern spines q in the generated covering set Q, there exists a copattern spines q' in a given program s.t. q is an instance of q'. For simplicity, we omit this generalization.

With copattern refinement and coverage of evaluation contexts defined, we are able to prove some technical results which justify the soundness of the copattern refinement rules. The first of these states that if a copattern q matches an evaluation context E, and q refines in one step into the set Q of copatterns, then eventually E will match one of the copatterns in Q.

Lemma 5. *Soundness of Copattern Refinement*
If $(q \searrow \Delta; \Gamma; N) \Longrightarrow Q$ *and* $\vdash \theta; \sigma; E : \Delta; \Gamma; N \searrow \uparrow P$ *then there exists* $q' \searrow \Delta'; \Gamma'; N' \in Q$ *and* $\vdash \theta'; \sigma'; E' : \Delta'; \Gamma'; N' \searrow \uparrow P$ *such that* $q[\theta; \sigma]@E = q'[\theta'; \sigma']@E'$.

Proof. The proof is done by case analysis on $(q \searrow \Delta; \Gamma; N) \Longrightarrow Q$ and inversion on the typing $\vdash \theta; \sigma; E : \Delta; \Gamma; N \searrow \uparrow P$. □

The soundness of our notion of coverage now follows easily. It states that if E is an evaluation context consuming type N, and Q covers N, then eventually E will match one of the copatterns in Q.

Corollary 6. *Soundness of Coverage*
If $N \vdash E \searrow \uparrow P$ *and* $\cdot; \cdot; N \lhd| Q$, *then there exists* $(q \searrow \Delta; \Gamma; N') \in Q$, *and* $\theta; \sigma; E' : \Delta; \Gamma; N' \searrow \uparrow P$ *such that* $E = q[\theta; \sigma]@E'$.

Proof. By induction on the derivation of $\cdot; \cdot; N \lhd| Q$. □

In order for progress to hold for whole programs, we need the operational semantics to preserve coverage so that the program cannot become stuck under an incomplete copattern set. The only concern lies when a covering set is under a substitution.

Lemma 7 (Preservation of Coverage under Substitution)**.**
If $\Delta'; \Gamma' \vdash (\theta; \sigma) : \Delta; \Gamma$ *and* $\Delta; \Gamma; N \lhd| \{q_i \searrow \Delta_i; \Gamma_i; N_i\}_{i \in I}$ *then there are* $\{\Delta'_i; \Gamma'_i\}_{i \in I}$ *such that*

$$\Delta'; \Gamma'; N[\theta] \lhd| \{q_i \searrow \Delta'_i; \Gamma'_i; N_i[\theta]\}_{i \in I}$$

and for all $i \in I$, $\Delta'_i; \Gamma'_i \vdash (\theta; \sigma) : \Delta_i; \Gamma_i$

Proof. By induction on the derivation of $\Delta; \Gamma; N \lhd| \{q_i \searrow \Delta_i; \Gamma_i; N_i\}_{i \in I}$. □

We are now able to prove progress, assuming each copattern set Q used in a function abstraction is covering.

Theorem 8 (Progress Theorem)**.** *If* $\vdash t; E : \uparrow P$, *then either* $t; E \longrightarrow t'; E'$ *or* $t; E = \mathsf{produce}\ v; \cdot$.

Proof. Proof by case analysis on t.

If t is of the form $t'.d$, or $t'\ v$, or $\mathsf{force}\ \mathsf{thunk}\ t'$, or $t'\ \mathsf{to}\ x.t''$, or $\mathsf{rec}\ f.t'$, then there is a stepping rule. If t is $\mathsf{produce}\ v$ we have two cases: If $E = \cdot$ we are done. If $E = ([]\ \mathsf{to}\ x.t'')\ E'$, then it steps to $[v/x]t''; E'$.

The last case is $t = \mathsf{fn}\ \vec{u}$. By assumption, if $\vdash \mathsf{fn}\ \vec{u} : N$ and $u = q_i \mapsto t_i$, then we have $N \lhd| Q$ where Q is the set of copatterns in $\vec{u}$. By Corollary 6, there exists $q \searrow \Delta; \Gamma; N' \in Q$ and $\theta; \sigma; E' : \Delta; \Gamma; N' \searrow \uparrow P$ such that $E = q_i[\theta; \sigma]@E'$. Hence $\mathsf{fn}\ \vec{u}; E$ steps. □

5. Related Work

Our work builds and extends directly the work by Levy (2001) to track data-dependencies in finite and infinite data. We model finite data using dependent sums and infinite data using dependent records where fields share a given index. This is in contrast to dependent records that allow a particular field to depend on previous ones (Betarte 1998).

Most closely related to our development is the work on DML (Xi and Pfenning 1999) where the authors also accumulate equality constraints during type checking to reason about indices. However, in DML all indices are erased before running the program while we reason about indices and their instantiation during run-time. As indices are also computationally relevant in fully dependently typed languages, we believe our work lays the ground of understanding the interaction of indices and (co)pattern matching in these languages. Finally, our work may be seen as extending DML to support both lazy and eager evaluation using (co)pattern matching.

Dependent type theories provide in principle support to track data dependencies on infinite data, although this has not received much attention in practice. Agda (Norell 2007), a dependently typed proof and programming environment based on Martin-Löf's type theory, has support for copatterns since version 2.3.4 (Agda team 2014). We can directly define equality guards and using large eliminations we can match on index arguments.

Agda uses inaccessible patterns (also called dot-patterns) (Brady et al. 2004; Goguen et al. 2006) to maintain linear pattern matching in a dependently typed setting. Inaccessible patterns mark patterns that are fully determined by their type. They do not bind additional variables not already occurring in the rest of the pattern, which is then linear. Our approach offers an alternative view which is mostly notational where relationships between arguments in a pattern are

$\boxed{(q \searrow \Delta'; \Gamma'; N') \Longrightarrow Q}$ Copattern q refines into copatterns Q

Impossible (Co)Pattern

$$(q \searrow \Delta'; \Gamma'; N') \quad \Longrightarrow \quad \{\} \qquad \text{if } \# \in \Delta'$$

(Co)Pattern Introduction

$$(q \searrow \Delta'; \Gamma'; \Pi X{:}U.N') \quad \Longrightarrow \quad \{q@X \searrow \Delta', X{:}U; \Gamma'; N'\}$$

$$(q \searrow \Delta'; \Gamma'; P \to N') \quad \Longrightarrow \quad \{q@x \searrow \Delta'; \Gamma', x{:}P; N'\}$$

$$(q \searrow \Delta'; \Gamma'; (\nu Z.\lambda \vec{X}.R)\vec{C}) \quad \Longrightarrow \quad \{q@.d \searrow \Delta'; \Gamma'; R_d[^{\nu Z \lambda \vec{X}.R}/_Z, \vec{C}/\vec{X}] \mid d \in R\}$$

Pattern Refinement

$$(q[x] \searrow \Delta'; \Gamma', x : C_1{=}C_2; N') \quad \Longrightarrow \quad \{q[\wp] \searrow \Delta''; \Gamma'; N'\} \qquad \text{provided } \Delta' \vdash C_1 = C_2 \searrow \Delta''$$

$$(q[x] \searrow \Delta'; \Gamma', x : P_1 \times P_2; N') \quad \Longrightarrow \quad \{q[(x_1, x_2)] \searrow \Delta'; \Gamma', x_1{:}P_1, x_2{:}P_2; N'\}$$

$$(q[x] \searrow \Delta'; \Gamma', x : \Sigma X{:}U.P; N') \quad \Longrightarrow \quad \{q[\mathsf{pack}\ \langle X, x'\rangle] \searrow \Delta', X{:}U; \Gamma', x'{:}P; N'\}$$

$$(q[x] \searrow \Delta'; \Gamma', x : (\mu Y.\lambda \vec{X}.D)\vec{C}; N') \quad \Longrightarrow \quad \{q[c\ x'] \searrow \Delta'; \Gamma', x'{:}D_c[^{\mu Y.\lambda \vec{X}.D}/_Y, \vec{C}/\vec{X}]; N' \mid c \in D\}$$

$$(q[X] \searrow \Delta'; \Gamma'; N') \quad \Longrightarrow \quad \{q[C_i] \searrow \Delta_i; (\Gamma'; N')[^{C_i}/_X]\}_{i \in I} \qquad \text{if split}(\Delta' \vdash X) = \{(\Delta_i \vdash C_i)\}_{i \in I}$$

$\boxed{\Delta; \Gamma; N \lhd\mid Q}$ Copatterns Q cover type N in context $\Delta; \Gamma$

$$\frac{}{\Delta; \Gamma; N \lhd\mid \{\cdot \searrow \Delta; \Gamma; N\}} \qquad \frac{\Delta; \Gamma; N \lhd\mid (Q \uplus \{q \searrow \Delta'; \Gamma'; N'\}) \quad (q \searrow \Delta'; \Gamma'; N') \Longrightarrow Q'}{\Delta; \Gamma; N \lhd\mid Q \cup Q'}$$

Figure 11. Coverage

kept in the index context while the pattern is fully linear. The equality checks are done during type checking and the constraints are irrelevant at run-time since a matching branch will always satisfy all of its constraints.

Our work draws on the distinction between finite data defined by constructors and infinite data described by observations which was pioneered by Hagino (1987). Hagino models finite objects via initial algebras and infinite objects via final coalgebras in category theory. This work, as others in this tradition such as Cockett and Fukushima (1992) and Tuckey (1997), concentrates on the simply typed setting. Extensions to dependent types with weakly final coalgebra structures have been explored Hancock and Setzer (2005). However in this line of work one programs directly with coiterators and corecursors instead of using general recursion and deep copattern matching. Further, equality is not treated first-class in their system – however, we believe understanding the role of equality constraints is central to arriving at a practical sound foundation for dependently typed programming.

Our development of indexed patterns and copatterns builds on the growing body of work (Zeilberger 2008a; Licata et al. 2008) which relates focusing and linear logic to programming language theory via the Curry-Howard isomorphism. Zeilberger (2008b) and Krishnaswami (2009) have argued that focusing calculi for propositional logic provide a proof-theoretic foundation for pattern matching in the simply-typed setting. Our work extends and continues this line of work to first-order logic (= indexed types) with (co)recursive types and equality. Our work also takes inspiration from the proof theory described in Baelde (2012) and Baelde et al. (2010) and the realization of this work in the Abella system (Baelde et al. 2014). While Baelde's proof theory supports coinductive definitions and equality, coinduction is defined by a non-wellfounded unfolding of a coinductive definition. Proofs in this work would correspond to programs written by (co)iteration. This is in contrast to our work, which is centered around the duality of (co)data types and supports simultaneous deep (co)pattern matching.

Finally, our approach of defining infinite data using records bears close similarity to the treatment and definition of objects and methods in foundations for object-oriented languages. To specify invariants about objects and methods and check them statically, DeLine and Fähndrich (2004) propose typestates. While this work focuses on the integration of typestates with object-oriented features such as effects, subclasses, etc., we believe many of the same examples can be modelled in our framework.

6. Conclusion

In this paper, we have presented an extension of a general purpose programming language with support for indexed (co)data type to allow the static specification and verification of invariants of infinite data such as streams or bisimulation properties. In our development we keep the index domain abstract and clearly state structural requirements our index domain must satisfy. Our language extends Levy (2001)'s call-by-push value with indexed (co)data types and deep (co)pattern matching. We use equality constraints to reason about dependencies between index arguments providing a clean foundation for dependent (co)pattern matching. We describe the operational semantics using a continuation-based abstract machine and prove that our language's operational semantics preserves types. We also provide a non deterministic algorithm to generate covering sets of copatterns, ensuring that terms do not get stuck during evaluation.

In the future, we plan to address two main directions: first, we aim to prove normalization of our language restricting our programs to total functions. This then justifies the use of our core language as a proof language for developing coinductive proofs; second, we will extend this work to full dependent types providing a foundation for Agda and Coq.

References

A. Abel and B. Pientka. Well-founded recursion with copatterns:a unified approach to termination and productivity. In *18th ACM International Conference on Functional Programming (ICFP '13)*, pages 185–196. ACM Press, 2013.

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *40th ACM Symp. on Principles of Programming Languages (POPL'13)*, pages 27–38. ACM Press, 2013.

Agda team. The Agda Wiki, 2014.

D. Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1):2:1–2:44, 2012.

D. Baelde, Z. Snow, and D. Miller. Focused inductive theorem proving. In J. Giesl and R. Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 278–292. Springer, 2010.

D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

G. Betarte. *Dependent Record Types and Formal Abstract Reasoning: Theory and practice*. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 1998.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs (TYPES'03), Revised Selected Papers*, Lecture Notes in Computer Science (LNCS 3085), pages 115–129, 2004.

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *39th ACM Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

R. Cockett and T. Fukushima. About charity. Technical report, Department of Computer Science, The University of Calgary, June 1992. Yellow Series Report No. 92/480/18.

R. DeLine and M. Fähndrich. Typestates for objects. In *18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Lecture Notes in Computer Science (LNCS 3086), pages 465–490. Springer, 2004.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science (LNCS 4060), pages 521–540. Springer, 2006.

T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, Lecture Notes in Computer Science (LNCS 283), pages 140–157. Springer, 1987.

P. Hancock and A. Setzer. Interactive programs and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pages 115 – 134, Oxford, 2005. Clarendon Press.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

N. R. Krishnaswami. Focusing on pattern matching. In *36th Annual ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 366–378. ACM Press, 2009.

P. B. Levy. *Call-by-push-value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*, pages 241–252. IEEE Computer Society Press, 2008.

C. McBride. Let's see how things unfold: Reconciling the infinite with the intensional. In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *3rd Int. Conf. on Algebra and Coalgebra in Computer Science (CALCO'09)*, Lecture Notes in Computer Science (LNCS 5728), pages 113–126. Springer, 2009.

R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209 – 220, 1991.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007. Technical Report 33D.

B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *ACM Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, 2008.

D. Thibodeau, A. Cave, and B. Pientka. Indexed codata (extended version). Technical report, School of Computer Science, McGill University, July 2016. Techincal Report.

C. Tuckey. Pattern matching in Charity. Master's thesis, The University of Calgary, July 1997.

H. Xi. Applied type system. In *Types for Proofs and Programs (TYPES'03), Revised Selected Papers*, Lecture Notes in Computer Science (LNCS 3085), pages 394–408. Springer, 2004.

H. Xi and F. Pfenning. Dependent types in practical programming. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM Press, 2003. .

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

N. Zeilberger. Focusing and higher-order abstract syntax. In *35th Annual ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 359–369. ACM Press, 2008b.

C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997.