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.

To illustrate the effectiveness of this idea, we describe several properties about infinite data such as the number of bits forming a message in a stream, fair streams, and streams of increasing numbers. Our key technical contributions are two-fold: first, we extend Levy's call-by-push value language with support for indexed (co)data and deep (co)pattern matching. Second, we describe a small-step semantics using a continuation-based abstract machine, define coverage for indexed (co)patterns, and prove type safety.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords Dependent Types; Coinduction; Functional Programming; Logical Frameworks

1. Introduction

Indexed data types as proposed by Zenger (1997) and Xi and Pfenning (1999) allow us to specify and verify many interesting invariants about finite data in a general purpose programming language. In particular, indexed types allow us to avoid run-time checks for cases which cannot happen (for example, we can never ask for the head of an empty list) and eliminate static array bounds checks (Xi and Pfenning 1998). Indexed types also integrate easily with effects (e.g. state, exceptions, non-termination). The idea is simple, yet powerful: we index types with objects from a decidable domain.

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

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

For example, we can define a recursive type `Msg` which describes a message encoded using bits and tracks its length by declaring its kind to be `[nat]` → **type**.

```
data Msg: [nat] → type =
| Nil : Msg [z]
| Cons: Bit → Msg [N] → Msg [s N]
;
```

We separate here the definitions of the index domain from recursive types defined on the computation level and embed index objects inside computation-level expressions using `[]`. Following Cave and Pientka (2012) we chose as an index domain contextual LF (Nanevski et al. 2008) which gives us the flexibility to not only describe simple domains such as natural numbers but also more powerful domains such as logics and proofs.

In this paper we extend the idea of indexing types to codata types. This allows us to describe interesting invariants about infinite data and track data-dependencies. Following Abel et al. (2013), we define infinite data via observations. As an example, consider reading messages which are sent via a network protocol. Network protocols often send along with the data stream the size `N` of the message to be read and we want to ensure we read exactly `N` bits from a stream. To model a stream of bits which keeps track of how many bits belong to one message we define three different observations as follows:

```
codata Str: [nat] → type =
| GetBit  : Str [s N] → Bit
| NextBits: Str [s N] → Str [N]
| Done    : Str [z] → NextMsg
and data NextMsg: type =
| NextMsg : {N : [nat]} Str [N] → NextMsg
;
```

Given `Str [s N]` we can either read a bit using the observation `GetBit` or obtain the remaining bits belonging to the message using the observation `NextBits`. If we encounter `Str [z]`, we know we have finished reading the message and begin reading the next one. Using the observation `Done`, we obtain the next message which consists of the size of the next message and the remaining input stream. Thus indexed codata types allow us to enforce that we read the correct number of bits belonging to a message. To illustrate the elegance and power of indexing codata types, we also describe streams of increasing numbers and implement a fair merge where we alternate selecting elements from two streams.

Fundamentally, our approach follows the tradition of indexed types and extends these ideas to indexed codata types building on work by Levy (2001). While our examples can also be modelled in more powerful dependently typed languages such as the Calculus of Construction (Paulin-Mohring 1993) or Martin Löf type theory using large eliminations, our approach is more light-weight and extends the duality of finite and infinite data already present in the

work by Levy (2001), Zeilberger (2008) or Licata, Zeilberger, and Harper (2008) to capturing dependencies. Proof-theoretically, our work extends this latter line of research to first-order logic.

From a more practical point of view, our computation language can easily be combined with imperative features, allows nonterminating computation, and requires fewer annotations to make type checking decidable. The main technical contributions are:

- We present a core ML-like language with indexed (co)data types which supports both programming with finite data via pattern matching and infinite data using copattern matching. One can think of this core language as the target of elaborating a surface language where we may omit implicit indices via type reconstruction. Following Levy (2001), our language exploits the duality of positive types which we interpret as values and negative types which we take as computations. We define observations about infinite data using function definitions using simultanous (co)patterns and make observations by function application. On the one hand our work is a novel reformulation of ideas first presented by Abel et al. (2013) based on Levy's work and on the other it significantly extends this previous work to support simultaneous deep (co)pattern matching and track data dependencies using indexed (co)data types. Proof-theoretically, our language provides a proof term assignment for first-order logic with (co)fix points over a user-defined domain.
- We model dependently typed (co)data types as (co)fixpoints with explicit equality constraints on objects from our domain following Cave and Pientka (2012). When defining an indexed recursive type, equality constraints impose further constraints on the index. In contrast, in indexed corecursive types equality constraints guard the observations we can make. We require that equality on domain objects must be decidable. Unlike the work of Cave and Pientka (2012) where equality was treated silently, equality has a first-class status in our core language making the core language more uniform. As a consequence equality proofs are explicit in our core language. Since we require that equality on our domain is decidable, we can expect such proofs to be reconstructed during elaboration of a source language to the given core.
- We describe a small-step semantics using a continuation-based abstract machine for our language. It accommodates both call-by-value and call-by-need evaluation strategies. This is in contrast to Abel et al. (2013) which gives a high-level declarative rewriting semantics. We describe coverage and prove both type preservation and progress for our language.

The proposed programming language extends the idea of indexed types to verify invariants about and dependencies in infinite data. Choosing as an index domain contextual objects makes the language a prime candidate for defining coinductive predicates such as bisimulation relations as codata types. It hence is a significant step towards mechanizing coinductive proofs about LF encodings where total programs about such codata type definitions then correspond to coinductive proofs.

The remainder of this paper is organized as follows. We illustrate the usefulness of indexed codata types through three examples (Section 2): reasoning about the size of messages in a bit streams, modelling fair merge of streams where we alternate between elements of two streams, and describing a stream of increasing numbers. Then, we introduce in Section 3 our language supporting both indexed data types and codata types, together with pattern and copattern matching in a symmetric way. We describe the typing rules together with a small-step semantics. We then define a sound coverage algorithm for indexed patterns and copatterns, and prove preservation and progress.

2. Motivation

To illustrate how and why we want to track data dependencies in infinite data structures, we present here several examples. We start by expanding upon indexed streams and their use. Our second example enforces a fairness property about streams. Our last example models a stream of increasing natural numbers.

2.1 Message processing

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. To illustrate how to program with indexed streams, we write several programs. 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 we are done reading the entire message.

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

We write curly braces for universally quantifying over `N` in the type of `readMsg`. In the program, we use anonymous functions which abstract and pattern match on multiple arguments using **fn**. 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` and then build the actual message by consing `c` to the front of `w`. Note that we omitted passing some implicit arguments which can be expected to be inferred.

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: 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 Stream → 'a
| Tail : 'a Stream → 'a Stream;

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

and MsgStr:{N:[nat]}Bit Stream → [nat] Stream → Str [N] =
fn  [z] s ns .Done ⇒ getMsg s ns.Tail
| [s N] s ns .GetBit ⇒ s.Head
| [s N] s ns .NextBits ⇒ MsgStr [N] s.Tail ns.Tail;
```

Unlike Abel et al. (2013), we write observations in postfix and we exploit here pattern matching on the index `N` as well as on the different observations. We hence mix pattern and copatterns.

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

```
rec genBitStr : Str [2] =
fn .GetBit ⇒ RandomBitGenerator ()
 | .NextBits .GetBit ⇒ RandomBitGenerator ()
 | .NextBits .NextBits .Done ⇒ NextMsg [2] genBitStr;
```

2.2 Fair scheduling

We define here the fair interleavings of two streams using a predicate `flip`. We define the predicate `flip` in LF, our domain language together with a type `Bit` which encodes `0` and `1`. Our syntax is taken from the BELUGA system (Pientka and Cave 2015).

```
LF bool: type = tt : bool | ff : bool ;

LF flip: bool → bool → type =
| flip0: flip ff tt
| flip1: flip tt ff;

data Bit: [bool] → type=
| 0: Bit [ff]
| 1: Bit [tt];
```

We want to implement a fair merge of two streams. Here we zip a stream of zeroes and a stream of ones together and return a stream that alternates between zeroes and ones. We describe the type of the alternating bit stream as follows:

```
codata Alt : [bool] → type =
| Zero: Alt [ff] → Bit [ff]
| One : Alt [tt] → Bit [tt]
| Next: Alt [B]  → {B':[bool]} [flip B B'] → Alt [B'];
```

We index the fair bit stream `Alt` with a boolean flag. If the flag is false (i.e. `ff`), we can observe a zero; if the flag is true (i.e. `tt`), we can observe a one. In both situations we can also always ask for the next bit using the observation `Next`, but we flip the boolean flag before proceeding to return more bits. Let us see how we zip a stream of zeroes and ones together to an alternating bit stream that starts with a zero. It essentially produces a stream

```
0 1 0 1 0 1 0 1 0 1 ...
```

```
rec merge:(Bit[ff])Stream → (Bit[tt])Stream → Alt[ff] =
fn s0 s1 .Zero ⇒ s0.Head
|  s0 s1 .Next [tt] [flip0] .One ⇒ s1.Head
|  s0 s1 .Next [tt] [flip0] .Next [ff] [flip1] ⇒
   merge s0.Tail s1.Tail;
```

`merge s0 s1` will produce a stream upon which we can make different observations. If we ask for `.Zero`, we simply return the head of the first stream `s0`. Note that we cannot ask for `.One` immediately, since we are making observations about `Alt [ff]`. We hence must first ask for the next element using the observation `.Next` and provide a witness that we can flip `ff` to `tt`. Providing `tt` and `flip0` as such a witness, we obtain a stream `Alt [tt]` upon which we can now ask for `.One`. We can however also ask for more bits, by now flipping `tt` to `ff` again via the witness `flip1`, and merging the `s0.Tail` and `s1.Tail`.

This example illustrates the power of mixing pattern and copattern matching. If we think of the streams `s0` and `s1` as streams of requests, zipping both streams together guarantees that we alternate between requests from stream `s0` and `s1`, hence guaranteeing fairness. We chose to define the `flip` predicate on the level of LF; this is not necessary, but it emphasizes that such definitions and proof obligations are strictly important when reasoning about the computation and are irrelevant during runtime.

Another scheduling example is the following: given three streams, the first one sends `a`'s, the second one `b`'s, and the third one `c`'s, we want to first serve `c`, followed by `b` followed by `a` and then restart with `c`.

```
c b a c b a c b a ...
```

This can be viewed as a variation of the alternating bit stream where we cycle between the three streams. We write `0` for `z` and `2` for `suc (suc z)`). First, we define values `a`, `b`, `c` of priority 1, 2, and 3 resp. The stream that cycles between elements of priority 1, priority 2, and priority 3 is defined as a codata type `CycStr`.

```
data Val: [nat] → type =
| a: Val 0
| b: Val 1
| c: Val 2;

codata CycStr : [nat] → type =
| GetVal : cycStr [N]     → Val [N]
| Reset  : cycStr [0]     → cycStr [2]
| Next   : cycStr [suc N] → cycStr N;

rec cycle:(Val [0]) Stream → (Val [1]) Stream
      → (Val [2]) Stream → CycStr [2] =
fn sA sB sC .GetVal ⇒ c
|  sA sB sC .Next .GetVal ⇒ b
|  sA sB sC .Next .Next .GetVal ⇒ a
|  sA sB sC .Next .Next .Reset ⇒
    cycle sA.Tail sB.Tail sC.Tail;
```

Finally, we merge three streams of different priority such that we cycle between them and treat each stream fairly.

2.3 Increasing streams

As a last example, we describe how to generate a stream of increasing numbers. We can define such a stream as follows:

```
LF lt: nat → nat → type =
| lt_z: lt z (suc N)
| lt_s: leq N M → lt N (suc M)

LF leq: nat → nat → type =
| eq   : eq N M → leq N M
| less : lt N M → leq N M;

LF eq: nat → nat → type =
| ref: eq N N;

codata IncStr : [nat] → type =
| Head : IncStr [N] → Val N
| Tail : IncStr [N] → {M:[nat]} [lt N M] → IncStr [M]
```

Here we guard the generation of the next elements with the fact that it must be bigger. We can then write a simple program that generates natural numbers using deep pattern and copattern matching as follows:

```
rec natStr: {N:[nat]} IncStr [N] =
fn natStr [K] .Head ⇒ gen_value [K]
|  natStr [K] .Tail [suc K] [lt_s (eq ref)] ⇒
   natStr [suc K];
```

Note that the index allows us to save the state, i.e. what natural number we have already computed. Similarly, we can construct a stream of ever doubling numbers by simply replacing `[lt N M ]` with the definition of `[double N M]`. More generally using this methodology we can construct streams where the next element is guarded by a condition.

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, or (contextual) LF which we used in the examples. Index objects are abstractly referred to by C.

We make a few assumptions about our index domain: first, we assume that equality is decidable; second, we assume that unification in the index domain, i.e. solving equations, is decidable; third, we assume that there is a notion of coverage, i.e. given a type U, we can always get a complete non-redundant set of covering patterns for U. In particular, we define below several judgements which characterize the desired properties of our index domain and which we will be using in defining the static and operational semantics of our programming language. These judgments refer to variables from the index domain; such variables are for example introduced during (co)pattern matching when we write (co)recursive functions.

As we want to allow nested function definitions, pattern matching on arguments that depend on variables introduced by the outer functions may refine and restrict index variables. We therefore allow a meta-context Δ of index variables to contain unrestricted index variables $X : U$ and index variables that are already instantiated and bound to a particular object, written as $X := C : U$. This allows us to track the refinement of index variables using constraints in the meta-context Δ. As we will see this has two main advantages: 1) we can elegantly enforce linearity in patterns and moreover pattern matching on a linear pattern reduces to assigning values to variables. 2) we can support nested function definitions where inner functions may refine indices that were introduced in an outer function.

$$\begin{array}{lll} \text{Meta-Context } \Delta & ::= & \cdot \mid \Delta, X : U \mid \Delta, X := C : U \\ \text{Meta-Substitution } \theta & ::= & \cdot \mid \theta, C/X \end{array}$$

Typing judgements for meta-contexts, meta-substitutions, terms, and equality We define here some preliminaries that characterize our index domain and play a key role in the typing rules of our core language (see Sec. 3.2 and Sec. 3.3). In particular, we assume that index types are well-kinded and index terms are well-typed.

$\vdash \Delta\ \mathsf{mctx}$	Meta-context Δ is well-formed
$\Delta \vdash U : \mathsf{Type}$	Index type U is well-kinded in Δ
$\Delta \vdash C : U$	Term C is of type U in meta-context Δ.
$\Delta \vdash C = C'$	Term C and C' are equal
$\Delta \vdash U = U'$	Type U and U' are equal
$\Delta' \vdash \theta : \Delta$	θ maps index variables from Δ to Δ'

Equality on index objects stands for structural equality of two index objects and is defined as the least congruence using congruence rules for all the constructors closed under

$$\frac{(X := C : U) \in \Delta \qquad \Delta \vdash C : U}{\Delta \vdash X = C}$$

We define in Fig. 1 in more detail when a meta-context Δ and meta-substitution θ are well-formed and meaningful, as their definition may not be obvious due to our use of constraints in the meta-context. We assume that our typing rules for index objects satisfy the substitution property.

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

Typing judgement for terms and equality in patterns Our typing rules for simultaneous (co)patterns synthesize the type of pattern variables and rely on being able to synthesize the type of index variables occurring in the index object C (see also Fig. 6) together with possible constraints. As we process the (co)pattern spine 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. Hence the resulting meta-context Δ' is an extension of Δ, written as $\Delta \prec \Delta'$.

$\boxed{\vdash \Delta\ \mathsf{mctx}}$ well-formed meta-context Δ

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

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

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

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

$$\frac{\Delta \vdash \theta : \Delta' \quad \Delta \vdash C : U[\theta] \quad \Delta \vdash C = C'[\theta]}{\Delta \vdash \theta, C/X : \Delta', X := C' : U}$$

Figure 1. Meta-Contexts and Meta-Substitution

$\boxed{\Delta \prec \Delta'}$ Δ' is an extension of Δ

$$\frac{}{\cdot \prec \Delta'} \qquad \frac{\Delta \prec \Delta' \quad X{:}U \in \Delta'}{\Delta, X{:}U \prec \Delta'} \qquad \frac{\Delta \prec \Delta' \quad X{:=}C{:}U \in \Delta'}{\Delta, X{:}U \prec \Delta'}$$

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

Figure 2. Extension of Meta-Context

We use the following judgments to synthesize the index variables occurring in C. We assume that C only contains fresh variables, i.e. any variable in C does not already occur in Δ. We carry Δ which may contain variables introduced by a previous pattern match for two reasons: first, U may depend on Δ and second, variables occurring in C may be added together with constraints setting them equal to a variable already in Δ.

$\Delta \vdash C : U \searrow \Delta'$	pattern C of type U in the meta-context Δ synthezises a meta-context Δ' s.t. $\Delta \prec \Delta'$ and $\Delta' \vdash C : U$
$\Delta \vdash C_1 = C_2 \searrow \Delta'$	Given two terms C_1 and C_2 synthesize a meta-context Δ' s.t. $\Delta \prec \Delta'$ and $\Delta' \vdash C_1 = C_2$.

As pattern matching on dependently typed objects may refine the indices, the meta-context Δ may be updated, when type checking a pattern, to reflect these constraints. A typical example is the following: we pattern match on a vector, `[vec N]` after having introduced `N`, i.e. in the context Δ =`N:nat`. When considering the case `nil : vec 0`, we learn that `N` must be `0` and we update Δ to `N:=0:nat` which is the new Δ' that is returned. We come back to this issue in Sec. 3.2.

Type checking of (co)pattern spines will 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'. This will become clear in Sec. 3.2.

Pattern matching for index objects 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. Our pattern matching judgment for index terms corresponds closely to typing equalities appearing in patterns.

$\theta \vdash C =^{?} C' \searrow \theta'$ Term C matches against C' s.t $C = C'[\theta']$ and $\theta \prec \theta'$ (θ' extends θ).

We note that $\cdot \vdash \theta : \Delta$. Further, C is closed (i.e. $\cdot \vdash C : [\theta]U$) and C' describes a well-typed pattern (i.e. $\Delta \vdash C' : U \searrow \Delta'$). Adequacy of pattern matching then guarantees that $\cdot \vdash \theta' : \Delta'$ and $\Delta' \vdash C = C'[\theta]$.

Requirement 2 (Adequacy of pattern matching for index objects)**.** *Suppose* $\cdot \vdash \theta : \Delta$ *and* $\cdot \vdash C : U[\theta]$. *If* $\Delta \vdash C' : U \searrow \Delta'$ *and* $\theta \vdash C =^{?} C' \searrow \theta'$ *then* $\cdot \vdash \theta' : \Delta'$.

Splitting algorithm $\text{split}(\Delta \vdash U) = (\Delta_i, C_i)_{\forall i \in I}$ splits on a type U yielding a complete non-redundant set of covering patterns $\Delta \vdash C_i : U \searrow \Delta_i$. The set I denote simply a subset of the natural numbers and thus i ranges from 0 to some $n \in \mathbb{N}$.

Requirement 3 (Coverage of splitting for index objects)**.**
Suppose $\cdot \vdash \theta : \Delta$ *and* $\vdash C : U[\theta]$ *and* $\Delta \vdash X : U \searrow \Delta'$.
If $\theta \vdash C =^{?} X \searrow \theta'$ *and* $\text{split}(\Delta \vdash U) = (\Delta_i, C_i)_{\forall i \in I}$, *then there is an* i *such that* $\theta \vdash C =^{?} C_i \searrow \theta_i$.

3.2 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 construct finite data using labelled sums D (written as $\langle \overrightarrow{c\ P} \rangle$). 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 $\{\overrightarrow{d{:}N}\}$).

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

Figure 3. Types

Index objects from our index domain U can be embedded into computations via the box modality written as $[U]$; this allows us in general to work directly with domain-specific index objects which is particularly convenient if our index domain not only defines for example natural numbers but also predicates about them. As a consequence, index-objects can be directly analyzed and manipulated by our computation language. This is convenient in our setting, however, it also prevents a naive erasure of all the index objects. We note that we only allow dependencies on objects from our decidable domain not on arbitrary computation-level expressions.

Our language also supports equality constraints following previous work by Cave and Pientka (2012). They typically are used inside (co)recursive type definitions. When defining a recursive type, the equality constraints impose additional constraints on the type index, while in a corecursive type, the equality constraint guards the observations we can make (see examples). Unlike Cave and Pientka (2012), we give equality a first-class status which leads to a more uniform and general foundation. In practice, we mostly use equalities in two forms: constrained products (written as $C_1 = C_2 \times P$) and constrained (or guarded) function (written as $C_1 = C_2 \to N$). As we require that our index domain comes with decidable equality, we believe 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 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.

Examples 1: Indexed recursive type Datatypes $C = \mu Y.\lambda\vec{X}.D$ for $D = \langle c_1\ P_1 \mid \cdots \mid c_n\ P_n \rangle$ describe least fixed point. As in SML, a constructor that requires no argument formally takes an argument of the unit type 1. Choosing as index domain natural numbers, we can model our previous definition of `Msg` as follows in our core language.

$$\mu\text{Msg}.\lambda X.\langle \begin{array}{ll} \mathsf{Nil} & : X = 0 \times \mathsf{Unit}\ , \\ \mathsf{Cons} & : \Sigma Y{:}\mathsf{nat}.X = \mathsf{s}\ Y \times (\mathsf{Bit} \times \text{Msg}\ Y) \end{array} \rangle$$

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.

We can make explicit the fact that `GetBit`, `NextBit` and `NextMsg` all define ways of observing objects of type `Str [M]` by using equality constraints. We emphasize that the index `M` is shared among all records. Our source level syntax is logically equivalent to this reformulated definition of the stream codata type. To make more explicit the relation between positive and negative types, we define here the core representation of both `Str` and `NextMsg`.

$$\nu\text{Str}.\lambda M.\ \{ \begin{array}{lll} \mathsf{Done} & : & M = 0 \quad \to \uparrow \mathsf{NextMsg}\ , \\ \mathsf{NextBits} & : & \Pi N{:}\mathsf{nat}.M = \mathsf{s}\ N \to \text{Str}\ N\ , \\ \mathsf{GetBit} & : & \Pi N{:}\mathsf{nat}.M = \mathsf{s}\ N \to \uparrow \mathsf{Bit} \end{array} \}$$

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

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. 4). Val-

ues include unit (written as ()), pairs (written as (e_1, e_2)), dependent pairs (written as pack $\langle C, e\rangle$). We also include data built using constructors (written as $c\ v$) and allow index objects (written as $[C]$) as first-class values. Finally we can embed computation into values using 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 refl, in our values.

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

Figure 4. Values, Terms, (Co)patterns

Computations (or terms) correspond to negative types. Computations include recursion (written as rec $f.e$) and functions (written as 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 force v and produce a value (written as produce v). We also include a sequencing term which is written as $(t_1 \text{ to } x.t_2)$.

We eliminate expressions of positive type such as recursive types via pattern matching; dually, we make observations about expression of negative types such as corecursive types by making observations. Unlike the language described by Abel et al. (2013) which presented programs as rewrite rules, we choose to describe our programs in a more traditional functional programming style supporting simultaneous (co)pattern matching. We might view this language as a core language into which we can compile programs given as rewrite rules to. It also illustrates how to extend more traditional ML-like languages with copattern matching.

Simultaneous (co)patterns are described using a spine that is built out of patterns (written as p), an index object pattern (also called copattern instance and written as C), or observations (written as `.d`). Patterns themselves are derived from values and can be defined using pattern variables x, embedded index object (written as $[C]$), pairs (written as (p_1, p_2), pattern instances (written as pack $\langle C, e\rangle$) and patterns formed with a data constructor c.

Branches in case-expressions are modelled by $q \mapsto t$.

The typing rules for terms and values are mostly straightforward (see Fig. 5). We highlight here a few. Typing of embedded index objects (written as $[C]$) 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 refl (reflexivity) provided C_1 and C_2 are equal in our index domain. 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 $\Delta \vdash C_i = C'_i$ falling back to the comparison on index terms. We proceed similarly when comparing negative types.

A rec-expression introduces a variable of type $\downarrow N$ in the type t. Dual to 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 applyg 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 a value 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 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 synthesizing a new meta-context Δ' together with a new context Γ' describing the types of the variables occurring in the pattern together with an output type N' against which the term t is checked.

It might also be the case that the copattern spine q does not eliminate elements of type N; in particular, N might contain some equality constraints that q does satisfy. This is modelled by returning $\perp$ when checking the copattern spine q against N. In this case, we know that the branch $q \mapsto t$ cannot be taken during runtime; it is essentially dead-code and we simply succeed.

The typing rules for (co)patterns (see Fig. 6) are defined using the following two judgments:

$$
\begin{array}{ll}
\Delta; \Gamma \vdash p : P \searrow \Delta'; \Gamma' & \text{Typing for pattern } p \\
\Delta; \Gamma; N \vdash q \searrow \Delta'; \Gamma'; N' & \text{Typing for copattern } q
\end{array}
$$

In both typing judgments, the meta-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 meta-context Δ and context Γ, we synthesize a meta-context Δ' s.t. 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 P_{con} where we fall back to type checking patterns in our domain and in the rule P_{Eq} where we unify two index objects C_1 and C_2, and return a new meta-context Δ' s.t. $\Delta' \vdash C_1 = C_2$. For simplicity, we thread through both the meta-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 refl 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 meta-variables in C_1 and C_2 that makes C_1 and C_2 equal. In this case unification of C_1 and C_2 fails. This is described as $\Delta \vdash C_1 = C_2 \searrow \perp$. We omit here the rules that propagate $\perp$ due to space. Our treatment of equality is inspired by work of Dunfield and Krishnaswami (2015).

Copattern spines allow us to make observations on a negative type N in the meta-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 meta-context Δ' and context Γ'.

To illustrate we give the typing derivation for the copattern spine `[s N] s ns .GetBit` that arises from the program `MsgStr` given in Sec. 2.1. This copattern spine is represented in our core language as $(\text{s}\ N)\ s\ ns$ `.GetBit` M refl. We now show that it checks against ΠN :`nat. Bit Stream →[nat] Stream →Str [`N`]`. After inferring the type of N and introducing declarations for s and ns, we need to show that `.GetBit` M refl has type `Str [`N`]` in the context N : `nat` and the context $\Gamma = s$: `Bit Stream`, ns : `[nat] Stream`. We show partial typing derivation in Fig. 6.

Example 3 Recall our previous program `genBitStr` which generated `Str [2]`. Abbreviating `RandomBitGenerator` simply by `RBG`, this program can be elaborated into our core language straightforwardly to a program of type `Str` 2.

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

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

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

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

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

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

Figure 5. Typing rules for terms.

```
rec genBitStr.fn
  | .GetBit 1 refl                                  ↦ RBG ()
  | .NextBits 1 refl .GetBit 0 refl                 ↦ RBG ()
  | .NextBits 1 refl .NextBits 1 refl .Done refl ↦
                          NextMsg (pack ⟨2, genBitStr⟩)
```

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

```
rec readMsg.fn
  | [z]    s      ↦ produce (Nil (refl, () ), s)
  | [s M] s       ↦
  (force s).GetBit to c.
    (force readMsg) [M] (thunk (force s).NextBits) to x.
      (fn(w, s′) ↦ produce (Cons (pack ⟨M, (refl, (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 `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 orignal program also used pattern matching. We do not directly support nested (co)pattern matching. Hence, we define 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.

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

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:

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

The negative type N describes some computation t which when used in the evaluation context K returns a computation of type N'. Intuitively, t stands for a function fn $(\overrightarrow{q_i \mapsto t_i})$ and we match the evaluation context K against the copattern spine q_i and consume part of K 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 $\mathrm{K_{Base}}$), 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 $\mathrm{K_{App}}$). If we have a computation of type $\Pi X{:}U.N$ and the evaluation context supplies an index object C,

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

$$\frac{}{\Delta;\Gamma \vdash x : P \searrow \Delta;\Gamma, x{:}P}\ \mathrm{P_{Var}} \qquad \frac{}{\Delta;\Gamma \vdash () : 1 \searrow \Delta;\Gamma}\ \mathrm{P_{Unit}} \qquad \frac{\Delta;\Gamma \vdash p : D_c[\mu Y.\lambda\vec{X}.D/Y, \vec{C}/\vec{X}] \searrow \Delta';\Gamma'}{\Delta;\Gamma \vdash c\ p : (\mu Y.\lambda\vec{X}.D)\vec{C} \searrow \Delta';\Gamma'}\ \mathrm{P_{Const}}$$

$$\frac{\Delta \vdash C : U \searrow \Delta'}{\Delta;\Gamma \vdash [C] : [U] \searrow \Delta';\Gamma}\ \mathrm{P_{Con}} \qquad \frac{\Delta;\Gamma \vdash p_1 : P_1 \searrow \Delta';\Gamma' \quad \Delta';\Gamma' \vdash p_2 : P_2 \searrow \Delta'';\Gamma''}{\Delta;\Gamma \vdash (p_1, p_2) : P_1 \times P_2 \searrow \Delta'';\Gamma''}\ \mathrm{P_{Pair}}$$

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

$\boxed{\Delta;\Gamma;N \vdash q \searrow \Delta';\Gamma';N'}$ Copattern q elimininates 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:

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

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

Figure 6. Type checking for patterns.

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 $([]\ \mathrm{to}\ x.t)\ K$, we check that once we are done evaluating t and return a computation of type N, passing to it the remaining evaluation context K yields a computation of type N'.

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

$$\frac{\vdash v : P \quad N \vdash K \searrow N'}{P \to N \vdash v\ K \searrow N'}\ \mathrm{K_{App}} \qquad \frac{N[C/X] \vdash K \searrow N' \quad \vdash C : U}{\Pi X{:}U.N \vdash C\ K \searrow N'}\ \mathrm{K_{MApp}}$$

$$\frac{x : P \vdash t : N \quad N \vdash K \searrow N'}{\uparrow P \vdash ([]\ \mathrm{to}\ x.t)\ K \searrow N'}\ \mathrm{K_{to}}$$

4.2 (Co)Pattern matching

We describe in Fig. 7 pattern and copattern matching using two judgments:

$\theta;\sigma \vdash v =^{?} p \searrow \theta';\sigma'$	Pattern matching
$\theta;\sigma \vdash K =^{?} q \searrow \theta';\sigma';K'$	Copattern matching

Pattern matching the value v against the pattern p synthesizes instantiations θ' for index objects and instantiations σ' for ordinary pattern variables in p. Both, θ' and σ' are extensions of θ and σ respectively, i.e. $\theta \prec \theta'$ and $\sigma \prec \sigma'$.

Dually, copattern matching matches an evaluation context K against a copattern spine q extending θ to θ' and σ to σ'. As the evaluation context may be bigger than the copattern spine, copattern matching may not consume all of K and return the remaining evaluation context K' (where K' is a suffix of K).

For copattern matching to succeed, the evaluation context must supply at least as many observations as required by the copattern spine otherwise it will not succeed.

4.3 Small step operational semantics

The operational semantics is defined on configurations $t;K$ which contain a term and an evaluation context. Such pair is said to have type N' (written $\vdash t;K : N'$) if $\vdash t : N$ and $N \vdash K \searrow N'$. The rules for the operational semantics on configurations are defined in Fig. 8. To evaluate an expression t_1 to $x.t_2$, we evaluate t_1 in the evaluation context extended with $[]$ to $x.t_2$. Once we have a value v for t_1 we pop off $[]$ 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 $\mathrm{fn}\ (\overrightarrow{q_i \mapsto t_i});K$. In this case, we match the evaluation context K against the copattern spine q_i yielding $(\theta;\sigma)$ and then step $t_i[\theta;\sigma]$.

Next, we prove that types are preserved during evaluation (see Theorem 3). 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

$\boxed{\theta;\sigma \vdash v =^? p \searrow \theta';\sigma'}$ Value v matches against pattern p extending environments $\theta;\sigma$ to $\theta';\sigma'$.

$$\frac{}{\theta;\sigma \vdash v =^? x \searrow \theta;\sigma,v/x}\ \text{PM}_{\text{Var}} \qquad \frac{\theta;\sigma \vdash v =^? p \searrow \theta';\sigma'}{\theta;\sigma \vdash c\ v =^? c\ p \searrow \theta';\sigma'}\ \text{PM}_{\text{Constr}} \qquad \frac{}{\theta;\sigma \vdash () =^? () \searrow \theta;\sigma}\ \text{PM}_{\text{Unit}}$$

$$\frac{\theta \vdash C =^? C' \searrow \theta'}{\theta;\sigma \vdash [C] =^? [C'] \searrow \theta';\sigma}\ \text{PM}_{\text{IndexObj}} \qquad \frac{\theta \vdash C =^? C' \searrow \theta' \quad \theta';\sigma \vdash v =^? p \searrow \theta'';\sigma'}{\theta;\sigma \vdash \text{pack}\ \langle C, v\rangle =^? \text{pack}\ \langle C', p\rangle \searrow \theta'';\sigma'}\ \text{PM}_{\text{Pack}}$$

$$\frac{\theta;\sigma \vdash v_1 =^? p_1 \searrow \theta';\sigma' \quad \theta';\sigma' \vdash v_2 =^? p_2 \searrow \theta'';\sigma''}{\theta;\sigma \vdash (v_1, v_2) =^? (p_1, p_2) \searrow \theta'';\sigma''}\ \text{PM}_{\text{Pair}} \qquad \frac{}{\theta;\sigma \vdash \text{refl} =^? \text{refl} \searrow \theta;\sigma}\ \text{PM}_{\text{Eq}}$$

$\boxed{\theta;\sigma \vdash K =^? q \searrow \theta';\sigma';K'}$ K matches copattern q returning environment $\theta';\sigma'$ and evaluation context K'.

$$\frac{\theta;\sigma \vdash K =^? q \searrow \theta';\sigma';K'}{\theta;\sigma \vdash .d\ K =^? .d\ q \searrow \theta';\sigma';K'}\ \text{CPM}_{\text{Dest}} \qquad \frac{\theta \vdash C =^? C' \searrow \theta' \quad \theta';\sigma' \vdash K =^? q \searrow \theta'';\sigma';K'}{\theta;\sigma \vdash C\ K =^? C'\ q \searrow \theta'';\sigma';K'}\ \text{CPM}_{\text{MApp}}$$

$$\frac{}{\theta;\sigma \vdash K =^? \cdot \searrow \theta;\sigma;K}\ \text{CPM}_{\text{Obs}} \qquad \frac{\theta;\sigma \vdash v =^? p \searrow \theta';\sigma' \quad \theta';\sigma' \vdash K =^? q \searrow \theta'';\sigma'';K'}{\theta;\sigma \vdash v\ K =^? p\ q \searrow \theta'';\sigma'';K'}\ \text{CPM}_{\text{App}}$$

Figure 7. Rules for pattern matching.

$\boxed{t_1;K_1 \longrightarrow t_2;K_2}$ $t_1;K_1$ evaluates to $t_2;K_2$ in one step.

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

$$\frac{\cdot;\cdot \vdash K =^? q_i \searrow \theta;\sigma;K'}{\text{fn}\ (\overrightarrow{q_i \mapsto t_i});K \longrightarrow t_i[\theta;\sigma];K'}$$

Figure 8. Operational Semantics

relationship between the computation of the type N and an evaluation contexts K that transforms N into N'.

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

Lemma 1 (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. Both statements are proved by induction on the typing derivation. The case on index terms uses Req. 1. □

Lemma 2 (Adequacy of copattern matching)**.**

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

Proof. Both statements are proved by induction on the copattern matching derivation. The cases for matching on index objects make use of Req. 2. □

Theorem 3 (Type preservation)**.**
If $\cdot;\cdot \vdash t;K : N$ *and* $t;K \longrightarrow t';K'$, *then* $\vdash t';K' : N$.

Proof of theorem 3. The proof is done by case analysis on the stepping rule. The only interesting case is when dealing with function abstraction.

$$\frac{\cdot;\cdot \vdash K =^? q_i \searrow \theta;\sigma;K'}{\text{fn}\ (\overrightarrow{q_i \mapsto t_i});K \longrightarrow t_i[\theta;\sigma];K'}$$

By inversion on the derivation for $\vdash \text{fn}\ (\overrightarrow{q_i \mapsto t_i});K : N$, we have that $\vdash \text{fn}\ (\overrightarrow{q_i \mapsto t_i}) : N'$ and $N' \vdash K \searrow N$.

By inversion on the derivation for $\text{fn}\ (\overrightarrow{q_i \mapsto t_i})$, we have $\cdot;\cdot;N' \vdash q_i \searrow \Delta_i;\Gamma_i;N_i$ and $\Delta_i;\Gamma_i \vdash t_i : N_i$.

By lemma 2, copattern matching yields substitutions $\cdot;\vdash \theta;\sigma : \Delta_i;\Gamma_i$ together with continuation $N_i \vdash K' \searrow N$. Thus, by substitution lemma, we have $\vdash t_i[\theta;\sigma] : N_i$

Thus, $\vdash t_i[\theta;\sigma];K' : N$. This concludes the proof. □

4.4 Coverage

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

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 $K@k$, where $k = .d \mid v \mid [] \text{ to } x.n \mid C$ which adds to the end 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 9 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, begining 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 occuring 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.

As an intermediate technical device, we introduce a notion of coverage of evaluation contexts. We write $K \lhd Q : N'$ to mean that, eventually (i.e. if K is extended with sufficiently many appropriately typed observations), it will match one of the copatterns in Q. This is necessary because the (deep) copatterns in Q may require several observations before they are able to fire. This notion is defined in Figure 10.

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 K, and q refines in one step into the set Q of copatterns, then eventually K will match one of the copatterns in Q.

Lemma 4. *Soundness of copattern refinement*
If $\Delta; \Gamma; N \vdash q \searrow \Delta'; \Gamma'; N'$ *and* $\vdash \theta; \sigma; K : \Delta; \Gamma; N \searrow N''$ *and* $(q \searrow \Delta'; \Gamma'; N') \Longrightarrow Q$ *and* $\theta; \sigma \vdash K =^{?} q \searrow \theta'; \sigma'; K'$ *then* $K \lhd Q : N''$

Proof. By cases on $(q \searrow \Delta'; \Gamma'; N') \Longrightarrow Q$ and induction on $\theta; \sigma \vdash K =^{?} q \searrow \theta'; \sigma'; K'$. The case splitting on a variable in Δ' requires us to use Req. 3. □

Corollary 5. *Soundness of copattern refinement*
If $\Delta; \Gamma; N \vdash q \searrow \Delta'; \Gamma'; N'$ *and* $\vdash \theta; \sigma; K : \Delta; \Gamma; N \searrow N''$ *and* $(q \searrow \Delta'; \Gamma'; N') \Longrightarrow Q'$ *and* $K \lhd (Q \uplus \{q \searrow \Delta'; \Gamma'; N'\}) : N''$ *then* $K \lhd (Q \cup Q') : N''$

Proof. By induction on $K \lhd (Q \uplus \{q \searrow \Delta'; \Gamma'; N'\}) : N''$, appealing to Lemma 4 in the base case. □

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

Corollary 6. *Soundness of coverage*
If $N \vdash K \searrow N'$ *and* $\cdot; \cdot; N \lhd Q$, *then* $K \lhd Q : N'$.

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

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

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

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

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

$$\frac{\Delta' \vdash C_1 = C_2 \searrow \bot}{(q[x] \searrow \Delta'; \Gamma', x : C_1{=}C_2; N') \Longrightarrow \{q[\text{refl}] \searrow \bot\}}$$

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

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

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

$$\frac{\text{split}(\Delta \vdash U) = \{(\Delta_i, C_i)\}_{i \in I}}{(q[X] \searrow \Delta', X{:}U; \Gamma'; N') \Longrightarrow \{q[C_i] \searrow \Delta_i; (\Gamma'; N')[{}^{C_i}/_{X}]\}_{i \in I}}$$

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

$$\frac{}{\Delta; \Gamma; N \lhd \{\cdot \searrow \Delta; \Gamma; N\}}$$

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

Figure 9. Coverage

$\boxed{K \lhd Q : N'}$ Evaluation context K is covered by the copattern set Q at N'

$$\frac{q \in Q \quad \vdash K =^{?} q \searrow \theta; \sigma; K'}{K \lhd Q : N'}$$

$$\frac{\forall v : P \quad K@v \lhd Q : N'}{K \lhd Q : P \to N'} \qquad \frac{\forall C : U \quad K@C \lhd Q : N'[C/X]}{K \lhd Q : \Pi X : U.N'}$$

$$\frac{\forall d \in R \quad K@.d \lhd Q : R_d[(\nu Z.\lambda \vec{X}.R)/Z, \vec{C}/\vec{X}]}{K \lhd Q : (\nu Z.\lambda \vec{X}.R)\vec{C}}$$

Figure 10. Coverage of evaluation contexts

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

4.5 Progress

In this section, we assume that any copattern set Q used in a function abstraction is covering, according to the notion of coverage defined in the previous section.

We define progress through a notion of safety denoted by the judgment

$$t; K \textbf{ safe at } N'$$

This judgment means that either $t; K$ is a terminal configuration which exposes a value, or eventually (i.e. if K is extended with sufficiently many appropriately typed observations), the configuration $t; K$ will step. The definition is in Figure 11.

$\boxed{t; K \textbf{ safe at } N'}$ Configuration $t; K$ is safe at type N'.

$$\frac{}{\text{produce } v; \cdot \textbf{ safe at } \uparrow P} \qquad \frac{t; K \longrightarrow t'; K'}{t; K \textbf{ safe at } N'}$$

$$\frac{\forall d \in R \quad t; K@.d \textbf{ safe at } R_d[\nu Z.\lambda \vec{X}.R/Z, \vec{C}/\vec{X}]}{t; K \textbf{ safe at } (\nu Z.\lambda \vec{X}.R)\vec{C}}$$

$$\frac{\forall v \in P \quad t; K@v \textbf{ safe at } N'}{t; K \textbf{ safe at } P \to N'}$$

$$\frac{\forall C \in U \quad t; K@C \textbf{ safe at } N'[C/X]}{t; K \textbf{ safe at } \Pi X{:}U.N'}$$

Figure 11. Progress judgement

Produce terms with empty evaluation contexts are terminal configurations and thus are safe. Stepping configurations are also safe. Configurations are safe at types $(\nu Z.\lambda \vec{X}.R)\ \vec{C}$, and $P \to N$, and $\Pi X{:}U.N$ if extensions of the evaluation context adequate for the corresponding type are also safe. This captures the idea that partial applications can occur if providing more to it eventually will make it step.

In order to prove that every well typed term is safe, we need to first introduce a lemma to handle the copattern abstraction case.

Lemma 7. *If Q is the set of copatterns in $\vec{u}$ and $K \lhd Q : N'$, then fn $\vec{u}; K$ **safe at** N'.*

Proof. Proof by induction on the derivation $K \lhd Q : N'$. □

Theorem 8 (Progress theorem)**.** *If $\vdash t; K : N'$, then $t; K$ **safe at** N'.*

Proof. Proof by case analysis on t.

If t is of the form $t'.d$, or $t'\ v$, or force thunk t', or t' to xt'', or rec $f..t'$, then there is a stepping rule. If t is produce v. Then we do a nested case analysis on K. If $K = \cdot$, then it progresses by assumption. If $K = ([]$ to $x.t'')\ K'$, then it steps to $[v/x]t''; K'$. Since $\vdash$ produce $v : \uparrow P$, we cannot have $K = .d\ K'$, $K = v\ K'$, or $K = C\ K'$.

The last case is $t =$ fn $\vec{u}$. By assumption, if $\vdash$ fn $\vec{u} : N'$, then we have $N' \lhd| Q$ where Q is the set of copatterns in $\vec{u}$. By Lemma 6, since $N \vdash K : N'$, it follows that $K \lhd Q : N$. Hence, by Lemma 7, fn $\vec{u}; K$ **safe at** N. □

5. Related Work

As mentioned 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. Our language does not support full dependent types, but only supports indexed types where indices are drawn from a user-defined domain with a decidable equality theory. This simplifies the development and allows the integration with effects.

Dependent type theories provide in principle similar 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.

Extensions of Martin Löf's type theory with dependent records were investigated by Betarte (1998). In his work, a dependent record is viewed as a sequence of fields where the type of a particular field may depend on previous ones. For example, a record representing a vector could have fields `list` and `length` where the type of `list` is indexed by the value of `length`. We do not allow such dependencies between the fields but we rather distinguish between fields based on a shared index.

Our work builds 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). In this line of work one programs directly with coiterators and corecursors instead of using general recursion and deep copattern matching.

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 oriented object features such as effects, subclasses, etc., we believe many of the same examples can be modelled in our framework.

Our development of indexed patterns and copatterns builds on the growing body of work (Zeilberger 2008; Licata et al. 2008) which relates focusing and linear logic to programming language theory via the Curry-Howard isomorphism. In particular, our work takes some inspiration from the proof theory in Baelde (2012) and Baelde et al. (2010) and the realization of this work in the Abella system (Baelde et al. 2014). While this work supports coinductive definitions and coinduction is defined by a non-wellfounded unfolding of a coinductive definition, it does not describe programs corresponding to proofs and does not support simultaneous deep (co)pattern matching.

6. Conclusion

In this paper, we have presented an extension of a general purpose programming language with support for indexed (co)datatype to allow the static specification and verification of invariants of infinite data such as streams. 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)datatypes and deep (co)pattern matching. We prove that our language's operational semantics preserves types and 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 extend our language to a proof language allowing inductive and coinductive definitions. Choosing contextual LF as the index domain, this can then serve as a foundation for developing coinductive proofs about LF specifications. Key to this step is a notion of totality or productivity which we plan to investigate building on work by Abel and Pientka (2013).

References

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

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium 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), 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.

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

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)*, volume 3086, pages 465–490, June 2004.

J. Dunfield and N. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism and indexed types. Draft, 2015.

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*, volume 283 of *Lecture Notes in Computer Science*, 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. ISBN 9780198566519.

P. B. Levy. *Call-by-push-value*. PhD thesis, Queen Mary and Westeld 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.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden, Sept. 2007.

C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications(TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

B. Pientka and A. Cave. Inductive Beluga: Programming Proofs (System Description). In A. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 2015.

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

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

H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257. ACM press, 1998.

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

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