Copatterns

Programming Infinite Structures by Observations

Andreas Abel
Department of Computer Science,
Ludwig-Maximilians-University Munich,
Germany
andreas.abel@ifi.lmu.de

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

Anton Setzer †
Computer Science,
Swansea University, Wales, UK
a.g.setzer@swan.ac.uk

Abstract

Inductive datatypes provide mechanisms to define finite data such as finite lists and trees via constructors and allow programmers to analyze and manipulate finite data via pattern matching. In this paper, we develop a dual approach for working with infinite data structures such as streams. Infinite data inhabits coinductive datatypes which denote greatest fixpoints. Unlike finite data which is defined by constructors we define infinite data by observations. Dual to pattern matching, a tool for analyzing finite data, we develop the concept of copattern matching, which allows us to synthesize infinite data. This leads to a symmetric language design where pattern matching on finite and infinite data can be mixed.

We present a core language for programming with infinite structures by observations together with its operational semantics based on (co)pattern matching and describe coverage of copatterns. Our language naturally supports both call-by-name and call-by-value interpretations and can be seamlessly integrated into existing languages like Haskell and ML. We prove type soundness for our language and sketch how copatterns open new directions for solving problems in the interaction of coinductive and dependent types.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Patterns, Recursion; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and recursion schemes, Type structure

General Terms Languages, Theory

Keywords Coinduction, Functional programming, Introduction vs. elimination, Message passing, Pattern matching

1. Introduction

Representing and reasoning about infinite computation plays a crucial role in our quest for designing and implementing safe software systems, since we often want to establish behavioral properties about our programs, reason about I/O interaction and processes, and establish liveness properties that eventually something good will happen. While finite structures such as natural numbers or finite lists are modelled by inductive types, infinite structures such as streams or processes are elegantly characterized by coinductive types. Inductive types are now very well understood and supported by functional languages and proof assistants, whereas the theoretical foundations and practical tools for coinductive types lag behind.

For example, in the Calculus of (Co)Inductive Constructions, the core theory underlying Coq [INRIA 2010], coinduction is broken, since computation does not preserve types [Giménez 1996; Oury 2008]. In Agda [2012], a dependently typed proof and programming environment based on Martin Löf type theory, inductive and coinductive types cannot be mixed in a compositional way. For instance, one can encode the property "infinitely often" from temporal logic, but not its dual "eventually forever" [Altenkirch and Danielsson 2010].

Over the past decade there has been growing consensus [Setzer 2012; McBride 2009; Granström 2009] that one should distinguish between finite, inductive data defined by constructors and infinite, coinductive data which is better described by *observations*. This view was pioneered by Hagino [1987] who modeled finite objects via initial algebras and infinite objects via final coalgebras in category theory. His development culminated in the design of symML, a version of ML where one can declare *codatatypes* via a list of their destructors [Hagino 1989]. For example, the codatatype of streams is defined via the destructors head and tail which describe the observations we can make about streams. Cockett and Fukushima [1992] took up his work and designed a language *Charity* where one programs directly with the morphisms of category theory. But while Charity was later extended with pattern matching on (initial) data types [Tuckey 1997], no corresponding dual concept was developed for codatatypes (called final data types in Charity).

In this paper, we take a first step towards building a type-theoretic foundation for programming with infinite structures via observations. Dual to pattern matching for analyzing finite data, we introduce *copattern* matching for manipulating infinite data and describe coverage for copatterns. In order to focus on the main concepts and avoid the additional complexities that come with dependent types, for instance, the need to guarantee termination or productivity, we confine ourselves to simple types in this article.

* David Thibodeau acknowledges financial support by the Graduiertenkolleg *Programm und Modellanalyse* (PUMA, 2008–) of the *Deutsche Forschungsgemeinschaft* (DFG) and by the *Undergraduate Student Research Award* (USRA) of the *Natural Sciences and Engineering Research Council of Canada* (NSERC).

† Anton Setzer has been supported by EPSRC grant EP/G033374/1, Theory and applications of induction-recursion. Part of the work was done while the author was a visiting fellow of the Newton Institute, Cambridge.

POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01. . . $10.00

Our theoretical treatment of patterns and copatterns takes inspiration from the growing body of work which relates classical and linear logic to programming language theory via the Curry-Howard-Isomorphism; more precisely, we build on the duality between (finite) values as right-hand side proof terms and continuations or evaluation contexts as left-hand side terms of sequent calculus [Curien and Herbelin 2000; Wadler 2005; Kimura and Tatsuta 2009]. Following Zeilberger's [2008b] deep analysis of focused proofs in linear sequent calculus [Andreoli 1992] and its relationship to pattern matching and evaluation order in programming languages, we distinguish between positive types which characterize finite data and negative types which describe infinite data. While values are matched against patterns, evaluation contexts are matched against copatterns. Our notion of copatterns extends previous work by Licata, Zeilberger, and Harper [2008] to treat least fixed-points as positive types and greatest fixed-point as negative types, and we recognize copatterns as a *definition scheme* for infinite objects.

More precisely, we regard copatterns as *experiments* on black-box infinite objects such as functions, streams and processes. Infinite objects can be defined by a finite, covering set of observations, which are pairs of experiment (copattern) together with its outcome (defining term). We take the distinction between the finite / initial / positive and the infinite / final / negative serious and give introduction rules and patterns for the former and elimination rules and copatterns for the latter. This leads to a core functional language based on natural deduction instead of the sequent calculus formulation explored in by Zeilberger [2008b] et al. [Licata et al. 2008].

Our contributions are the following:

1. We show how copatterns complement the syntax of existing functional languages and enable a style of programming with infinite objects by observations (Section 2.2). There is no need to move to a different programming paradigm such as classical logic [Curien and Herbelin 2000; Wadler 2003] or the morphisms of category theory (Charity).
2. We describe a non-deterministic algorithm for checking copattern coverage and prove that *well-typed* and complete *programs do not go wrong* (Section 5). The construction of a covering set of copatterns corresponds to the interactive construction of a program as in Agda and Epigram (Section 2.1).
3. We explain how copatterns can be a fruitful paradigm in rewriting and dependent type theory. In *rewriting*, definition by observations lead naturally to *strongly* normalizing rewrite rules and a semantics without infinitary rewriting [Kennaway and de Vries 2003] (Section 2.3). In dependent type theory, they overcome the subject reduction problem of the Calculus of (Co)inductive Constructions [Giménez 1996] (Section 2.4).

The remainder of this article is structured as follows: We informally describe copatterns in Section 2 and explain in detail their benefits in programming, rewriting and type theory. In Section 3, we define formally our core language with copatterns and its static semantics. We then describe its operational semantics together with copattern matching in Section 4. Coverage of copatterns together with the type safety proof is presented in Section 5. We briefly describe a prototype implementation of copatterns for Agda in Section 6, before we conclude with a discussion of related work (Section 7) and an outlook to further work (Section 8).

2. Copatterns and Their Applications

In this section we explain copatterns (Section 2.1) and flesh out the arguments started in the introduction. We illustrate how copatterns impact functional programming (Section 2.2), rewriting (Section 2.3), and dependent type theory (Section 2.4).

2.1 From function definition to copatterns

We introduce copatterns in the scenario of interactive program construction by refining the left hand side of a function definition step by step. As we will see, copatterns arise naturally as the generalization of definition by pattern matching.

As an example, we construct the infinite stream $N, N-1, \ldots, 1, 0, N, N-1, \ldots, 1, 0, \ldots$ of natural numbers via an auxiliary function $\mathsf{cycleNats} : \mathsf{Nat} \to \mathsf{Stream\ Nat}$ such that $\mathsf{cycleNats}\ n = n, n-1, \ldots, 1, 0, N, N-1, \ldots, 1, 0, \ldots$.

Before we begin, let us define Stream Nat as a recursive record type containing the possible observations we can make about streams. These observations are also referred to as destructors, since they allow us to take apart streams.

$$\begin{array}{ll} \mathsf{record\ Stream}\ A = \{ & \mathsf{head} : A, \\ & \mathsf{tail}\ \ : \mathsf{Stream}\ A\ \} \end{array}$$

Now we will construct cycleNats step-by-step, similar to the interactive editing in Agda [Norell 2007] and Epigram [McBride and McKinna 2004]. The starting point is the template:

$$\begin{array}{lcl} \mathsf{cycleNats} & : & \mathsf{Nat} \to \mathsf{Stream\ Nat} \\ \mathsf{cycleNats} & = & ? \end{array}$$

Since a function is an infinite object, we define it by *observation* rather than giving its value (a λ-abstraction) directly. However, we cannot give a value of $\mathsf{cycleNats}\ n$ for every natural number n, instead, we apply cycleNats to a *generic* natural number, the pattern variable x.

$$\mathsf{cycleNats}\ x \quad = \quad ?$$

Application to x, which we write as $\cdot\ x$, is our first instance of a copattern, an *applicative copattern*. It is, in this case, a generic *experiment* on cycleNats. The observation of its outcome determines cycleNats.

We are left with the task of constructing a Stream of natural numbers. We can make two principal experiments on a stream: we can observe its head and its tail, and the outcome of these two experiments determines the stream. These experiments give us two new copatterns: head $\cdot$ and tail $\cdot$, called *destructor copatterns*, and lead to the next refinement of our definition of cycleNats:

$$\begin{array}{lcl} \mathsf{head}\ (\mathsf{cycleNats}\ x) & = & ? \\ \mathsf{tail}\ \ (\mathsf{cycleNats}\ x) & = & ? \end{array}$$

The observed head of $\mathsf{cycleNats}\ x$ is just x. To determine the tail, we need to split on the pattern variable x:

$$\begin{array}{lcl} \mathsf{head}\ (\mathsf{cycleNats}\ x) & = & x \\ \mathsf{tail}\ \ (\mathsf{cycleNats}\ 0) & = & ? \\ \mathsf{tail}\ \ (\mathsf{cycleNats}\ (1+x')) & = & ? \end{array}$$

This generalizes the applicative copattern form $\cdot\ x$ to $\cdot\ p$ where p is a *pattern*, as usual, a term built from constructors, literals, and uniquely occurring variables only. We finally can fill the remaining two holes and complete our definition of cycleNats:

$$\begin{array}{lcl} \mathsf{head}\ (\mathsf{cycleNats}\ x) & = & x \\ \mathsf{tail}\ \ (\mathsf{cycleNats}\ 0) & = & \mathsf{cycleNats}\ N \\ \mathsf{tail}\ \ (\mathsf{cycleNats}\ (1+x')) & = & \mathsf{cycleNats}\ x' \end{array}$$

The infinite object cycleNats, a function yielding streams, is defined via the complete set of copatterns $\{\mathsf{head}\ (\cdot\ x), \mathsf{tail}\ (\cdot\ 0), \mathsf{tail}\ (\cdot\ (1+x'))\}$ where $\cdot$, the hole, is a placeholder for the function cycleNats. It is determined by the following set of observations, i. e., experiments mapped to their results:

$$\begin{array}{ll} \mathsf{head}\ (\cdot\ x) & \mapsto x \\ \mathsf{tail}\ \ (\cdot\ 0) & \mapsto \mathsf{cycleNats}\ N \\ \mathsf{tail}\ \ (\cdot\ (1+x')) & \mapsto \mathsf{cycleNats}\ x' \end{array}$$

2.2 Copatterns in functional programming: Restoring a missing symmetry

Destructor copatterns are a useful addition to functional languages even if no coinduction is involved. In the following we evolve an implementation of the state monad typical for Haskell to demonstrate why the absence of general copatterns breaks symmetry, and how copatterns restore it.

A first implementation defines the type $\mathsf{State}\ S\ A$ of a stateful computation with result A just as a synonym for $S \to (A \times S)$. The monadic operations return and $\gg=$ ("bind") are given in an applicative style.

$$\begin{array}{lcl} \mathsf{State}\ S\ A & = & S \to A \times S \\ \mathsf{return} & : & A \to \mathsf{State}\ S\ A \\ \mathsf{return}\ a\ s & = & (a,\ s) \\ \_{\gg=}\_ & : & \mathsf{State}\ S\ A \to (A \to \mathsf{State}\ S\ B) \to \mathsf{State}\ S\ B \\ (m \gg= k)\ s & = & \mathsf{let}\ (a,\ s') = m\ s\ \mathsf{in}\ k\ a\ s' \end{array}$$

Returning a in state s yields the pair (a, s) of result and unchanged state, and executing the sequence $m \gg= k$ in state s first executes $m : \mathsf{State}\ S\ A$ in state s, resulting in a value $a : A$ and a new state s', in which we run the continuation k applied to a. The code explains itself nicely if the application to s is read as *in state s*.

There are reasons to move away from type synonym $\mathsf{State}\ S\ A$ to a bijection between the monadic type $\mathsf{State}\ S\ A$ and its implementation as $S \to A \times S$. For instance, in Haskell, type synonyms interact badly with type-class based overloading, and instead, $\mathsf{State}\ S\ A$ is implemented as a single-field record type with projection $\mathsf{runState}$ and constructor state.

$$\begin{array}{lcl} \mathsf{record\ State}\ S\ A & = & \mathsf{state}\{\mathsf{runState} : S \to A \times S\} \\ \mathsf{runState} & : & \mathsf{State}\ S\ A \to S \to A \times S \\ \mathsf{state} & : & (S \to A \times S) \to \mathsf{State}\ S\ A \end{array}$$

As we update our implementation of the monad operations, we are in for an unpleasant surprise: We can only partially keep up the applicative style, more precisely, only on the right hand side of =, the expression side. Here, we only have to prefix the monadic values m and $k\ a$ with the projection $\mathsf{runState}$. But on the left hand side, the pattern side, we cannot do the same change, since projections are not allowed there. Instead we have to turn the l.h.s. application to s to a r.h.s. λ-abstraction over s, and prefix it with constructor state.

$$\begin{array}{lcl} \mathsf{return} & : & A \to \mathsf{State}\ S\ A \\ \mathsf{return}\ a & = & \mathsf{state}\ (\lambda s.\ (a,\ s)) \\ \_{\gg=}\_ & : & \mathsf{State}\ S\ A \to (A \to \mathsf{State}\ S\ B) \to \mathsf{State}\ S\ B \\ m \gg= k & = & \mathsf{state}\ (\lambda s.\ \mathsf{let}\ (a,\ s') = \mathsf{runState}\ m\ s \\ & & \qquad\qquad \mathsf{in\ runState}\ (k\ a)\ s') \end{array}$$

The projection bits are still nice to read, e. g., $\mathsf{runState}\ m\ s$ reads as *run m in state s*, however, the definition of $m \gg= k$ as *the stateful computation, that if you pass it state s, ...* is a bit bumpy. The symmetry is broken.

Copatterns, which allow projections also on the l.h.s., restore the symmetry and allow a smooth definition of the monad operations again.

$$\begin{array}{lcl} \mathsf{return} & : & A \to \mathsf{State}\ S\ A \\ \mathsf{runState}\ (\mathsf{return}\ a)\ s & = & (a,\ s) \\ \_{\gg=}\_ & : & \mathsf{State}\ S\ A \\ & \to & (A \to \mathsf{State}\ S\ B) \\ & \to & \mathsf{State}\ S\ B \\ \mathsf{runState}\ (m \gg= k)\ s & = & \mathsf{let}\ (a,\ s') = \mathsf{runState}\ m\ s \\ & & \mathsf{in\ runState}\ (k\ a)\ s' \end{array}$$

It reads *if you run* return *a in state s, you get* (a, s); and *to run $m \gg= k$ in state s, run m in s, obtaining a value a and a new state s' in which you run $k\ a$.*

2.3 Deep copatterns in rewriting: Strong normalization for corecursive definitions

In the following we argue that copatterns help to integrate infinite objects into term rewriting, without having to change the standard reduction semantics.

A popular example of programming with infinite objects is the creation of the stream of Fibonacci numbers. It is concisely defined as

$$\mathsf{fib} = \mathsf{cons}\ 0\ (\mathsf{cons}\ 1\ (\mathsf{zipWith}\ \_{+}\_\ \ \mathsf{fib}\ (\mathsf{tail\ fib})))$$

Herein, cons is the stream constructor, head and tail the projection, and $\mathsf{zipWith}\ \_{+}\_$ yields a stream by applying addition pointwise to a pair of streams (here: fib and $\mathsf{tail\ fib}$).

Clearly, reading this equation as a rewrite rule, the computation of fib does not terminate under the standard semantics, which is *rewrite when the left hand side matches*. Using infinitary rewriting [Kennaway and de Vries 2003], fib converges to the Fibonacci stream

$$\mathsf{cons}\ 0\ (\mathsf{cons}\ 1\ (\mathsf{cons}\ 1\ (\mathsf{cons}\ 2\ (\mathsf{cons}\ 3\ (\mathsf{cons}\ 5\ \ldots$$

However, we are interested in the *strong normalization* of a term rewriting system since this yields a decision procedure for equality (which then implies decidability for checking dependent types).

We can revert to a non-standard rewriting semantics: label the definition of fib as *corecursive* and only unfold it when its value is demanded, e. g., by a projection. This solution, for instance taken in Coq [Giménez 1996], does not help with our particular definition of fib either, since $\mathsf{tail\ fib}$ appears in its own unfolding, leading to infinite reduction. A workaround exists: we could introduce a mutually defined auxiliary stream fib' which denotes the tail of fib.

But copatterns provide a principled and scalable solution. Mechanically, transforming the definition of fib into copatterns yields

$$\begin{array}{rcl} \mathsf{head\ fib} & = & 0 \\ \mathsf{head}\ (\mathsf{tail\ fib}) & = & 1 \\ \mathsf{tail}\ (\mathsf{tail\ fib}) & = & \mathsf{zipWith}\ \_{+}\_\ \ \mathsf{fib}\ (\mathsf{tail\ fib}). \end{array}$$

These equations are actually fulfilled by our first equation for fib, but now we take them as *definition* of fib. The rewrite system is terminating; neither fib nor $\mathsf{tail\ fib}$ can be reduced by itself because none of the three defining copatterns matches.

Our syntax allows us to delay unfolding of corecursion until the *whole copattern* is matched. Copatterns, in particular, deep copatterns such as nested projections $\mathsf{tail}\ (\mathsf{tail}\ \cdot)$ give us greater flexibility for corecursive definitions than non-standard semantics.

2.4 Copatterns in dependent type theory: Reclaiming subject reduction

Following the lead of the functional programming language Haskell, the dependently typed language Coq introduces both finite and infinite trees via constructors. However, this leads to fundamental problems. For example, the Calculus of (Co)Inductive Constructions, the core theory underlying Coq [INRIA 2010], lacks subject reduction. This issue is already described in Section 3.4 of Giménez' thesis [1996]. Oury [2008] brought it back to the attention of the community. A detailed analysis has been given by McBride [2009].

Let us recapitulate Oury's counterexample to subject reduction, as reproduced in Fig. 1: Given a coinductive type U with constructor in that takes just an argument of type U, the (extensionally) sole inhabitant u of U can be constructed as the fixed point of in. The definitions force and eq seem "pointless" [Chlipala 2012, p. 91]

$$\begin{array}{lll}
\mathsf{U} & & \text{a codata type}\\
\mathsf{in} & :\ \mathsf{U}\to\mathsf{U} & \text{its only (co)constructor}\\
 & & \\
\mathsf{u} & :\ \mathsf{U} & \text{an inhabitant of } \mathsf{U}\\
\mathsf{u} & =\mathsf{cofix\ in} & \text{infinite succession of } \mathsf{in}\text{s}\\
 & & \\
\mathsf{force} & :\ \mathsf{U}\to\mathsf{U} & \text{extensionally, an identity}\\
\mathsf{force} & =\lambda x.\,\mathsf{case}\,x\,\mathsf{of} & \\
 & \qquad\mathsf{in}\ y\Rightarrow\mathsf{in}\ y & \\
 & & \\
\mathsf{eq} & :\ (x:\mathsf{U})\to x\equiv\mathsf{force}\ x & \\
\mathsf{eq} & =\lambda x.\,\mathsf{case}\,x\,\mathsf{of} & \\
 & \qquad\mathsf{in}\ y\Rightarrow\mathsf{refl} & \\
 & & \\
\mathsf{eq}_{\mathsf{u}} & :\ \mathsf{u}\equiv\mathsf{in\ u} & \\
\mathsf{eq}_{\mathsf{u}} & =\mathsf{eq\ u} &
\end{array}$$

Figure 1. Oury's counterexample.

but are in fact a major tool in Coq proofs about corecursive definitions such as u since they wrap a case-distinction around a cofix that triggers reduction: While unrestricted reduction of cofix f to f (cofix f) would diverge immediately, the following two rewrite equations for matching coinductive data maintain strong normalization:

$$\begin{array}{lcl}
\mathsf{case}\,(\mathsf{in}\ s)\quad\ \mathsf{of\ in}\ y\Rightarrow t & = & t[s/y]\\
\mathsf{case}\,(\mathsf{cofix}\ f)\,\mathsf{of\ in}\ y\Rightarrow t & = & \mathsf{case}\,(f\ (\mathsf{cofix}\ f))\,\mathsf{of\ in}\ y\Rightarrow t
\end{array}$$

Now, the closed term eq u simplifies via

$$\begin{array}{rcl}
\mathsf{eq\ u} & = & \mathsf{case\,u\,of\ in}\ y\Rightarrow\mathsf{refl}\\
 & = & \mathsf{case}\,(\mathsf{cofix\ in})\,\mathsf{of\ in}\ y\Rightarrow\mathsf{refl}\\
 & = & \mathsf{case}\,(\mathsf{in}\ (\mathsf{cofix\ in}))\,\mathsf{of\ in}\ y\Rightarrow\mathsf{refl}\\
 & = & \mathsf{refl}
\end{array}$$

to the single constructor refl of propositional equality _≡_ , which means that u and in u must be definitionally equal. Yet they are not, since u does not reduce to in u unless under a case distinction. Subject reduction is lost! As Giménez notes, subject reduction holds only modulo an undecidable equality on types that allows unrestricted fixed-point unfolding, but this is not what can be implemented in proof assistants for Intensional Type Theory with decidable type checking.

The culprit is that the rule for dependent matching is also available for coinductive data, i. e., in case of U we have the rule

$$\frac{\Gamma\vdash u:\mathsf{U}\qquad\Gamma,\ y:\mathsf{U}\vdash t:C(\mathsf{in}\ y)}{\Gamma\vdash\mathsf{case}\,u\,\mathsf{of\ in}\ y\Rightarrow t:C(u)}$$

The rule substitutes a constructed term in y for term u in type C, which may trigger reduction of case expressions in C that are not possible without the substitution. For instance, force (in y) reduces to in y, while force x does not reduce to x. This is exploited in the typing[1] of eq, which leads to the loss of subject reduction in the end.

The deeper reason for this dilemma is that coinduction is just understood as constructing infinite trees; coinductive structures are just non-well-founded data structures in Coq. However, even in dependent type theory, infinite objects are better understood through their elimination rules [Granström 2009; Setzer 2012], i. e., observations. Looking at Oury's example, it seems wrong to regard in as a *constructor*. Rather, it should be *defined* in terms of the only observation out : U → U (same for the infinite structure u : U):

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathsf{in} & : & \mathsf{U}\to\mathsf{U} & \mathsf{u} & : & \mathsf{U}\\
\mathsf{out}\ (\mathsf{in}\ y) & = & y & \mathsf{out\ u} & = & \mathsf{u}
\end{array}$$

The equations we have written are now exactly the reduction rules, no restricted unfolding of cofix has to be taken into consideration. This makes definitional equality and, thus, type checking more perspicuous, for the user it is a what-you-see-is-what-you-get equational theory. Dependent pattern matching on coinductive types U is no longer available and subject reduction is restored. The "pointless" tricks like force and eq, necessary to deal with edges of dependent pattern matching, are also obsolete.

While we do not develop a dependently typed language with copatterns in this article, we have illustrated the shortcomings of uniformly modeling finite and infinite data via constructors and highlighted the potential of copatterns in the dependently typed setting.

3. A Core Functional Language with Copatterns

In this section, we introduce a core language with recursive data types for modeling finite data and recursive record types[2] for modeling infinite data. The term language is redex-free, which allows for a *complete* bidirectional type checking algorithm.

We then proceed to define patterns and copatterns which allow us to define functions, records and programs—which can be type-checked by an extension of the algorithm. Function definitions will be modeled as sets of rewrite rules.

3.1 Types

We distinguish between *positive* types, 1, $A \times B$, and $\mu X D$, and *negative* types, $A \rightarrow B$ and $\nu X R$. This polarity assignment is inspired by focusing proofs in intuitionistic linear logic [Andreoli 1992; Benton et al. 1993]. Our types 1, ×, and μ correspond to the positive connectives 1, ⊗, and a combination of ⊕ and least fixed point, resp., and they classify finite data. The types → and ν correspond to the negative ⊸ and a combination of & and greatest fixed point, resp., and classify infinite data. Linearity will show up in the typing rules for patterns and copatterns, even though ordinary terms need not be linear.

In terms of categorical languages [Hagino 1987; Cockett and Fukushima 1992], positive types are *left objects* or *initial datatypes*, and negative types are *right objects* or *final datatypes*.

$$\begin{array}{lll}
A,B,C & ::= X & \text{Type variable}\\
 & \mid P & \text{Positive type}\\
 & \mid N & \text{Negative type}\\
 & & \\
P & ::= 1 & \text{Unit type}\\
 & \mid A\times B & \text{Cartesian product}\\
 & \mid \mu X D & \text{Data type}\\
 & & \\
N & ::= A\to B & \text{Function type}\\
 & \mid \nu X R & \text{Record type}\\
 & & \\
D & ::= \langle c_1\ A_1 \mid \cdots \mid c_n\ A_n\rangle & \text{Variant (labeled sum)}\\
R & ::= \{d_1 : A_1,\ldots,d_n : A_n\} & \text{Record (labeled product)}
\end{array}$$

Figure 2. Types.

Figure 2 introduces positive types P, negative types N, and types A which can be either positive or negative. Variants D serve

[1] Since $\mathsf{refl}\ :\ \mathsf{in}\ y\ \equiv\ \mathsf{force}\ (\mathsf{in}\ y)$, by dependent pattern matching $\mathsf{case}\,x\,\mathsf{of\ in}\ y\Rightarrow\mathsf{refl} : x\equiv\mathsf{force}\ x$, thus, $\mathsf{eq}\ x : x\equiv\mathsf{force}\ x$.

[2] Our terminology should not be confused with recursive records in object-oriented language foundations, e. g., by Abadi and Cardelli [1994].

to construct possibly recursive data types $\mu X D$, and records R list the fields d_i of a possibly recursive record type $\nu X R$.

In our non-polymorphic calculus, type variables X only serve to construct recursive data types and recursive record types. As usual, $\mu X D$ ($\nu X R$, resp.) binds type variable X in D (R, resp.). Capture-avoiding substitution of type C for variable X in type A is denoted by $A[C/X]$. The set $\mathsf{FTV}(A)$ of free type variables of a type expression A is defined in the standard way. A type is *well-formed* if it has no free type variables; in the following, we assume that all types are well-formed.

Datatypes Datatypes $C = \mu X D$ for $D = \langle c_1\ A_1 \mid \cdots \mid c_n\ A_n \rangle$ are recursive variant types. They could be called *algebraic* types. We write D_{c_i} for A_i. The constructor c_i of C takes an arguments of type $A_i[C/X]$, i. e., $D_{c_i}[C/X]$. Non-recursive data types can be represented by a void μ-abstraction μ_D. Like in SML, a constructor that requires no argument formally takes an argument of the unit type 1. Examples:

$$\begin{array}{lcll}
\mathsf{List}\ A & = & \mu X\ \langle \mathsf{nil}\ 1 \mid \mathsf{cons}\ (A \times X) \rangle & \\
\mathsf{Nat} & = & \mu X\ \langle \mathsf{zero}\ 1 \mid \mathsf{suc}\ X \rangle & \\
\mathsf{Maybe}\ A & = & \mu\_\ \langle \mathsf{nothing}\ 1 \mid \mathsf{just}\ A \rangle & \\
0 & = & \mu\_\ \langle\rangle & \text{(positive empty type)}
\end{array}$$

Record types Record types $C = \nu X R$ with $R = \{d_1 : A_1, \ldots, d_n : A_n\}$ are recursive labeled products. They could be called *coalgebraic* types. The destructor d_i, if applied to a record of type C, returns the ith field which has type $A_i[C/X]$, or, with a "R_d" notation, $R_{d_i}[C/X]$. The destructors are applied in postfix notation to a term t as $t.d_i$. As for data, non-recursive record types are encoded by a void ν-abstraction ν_R. Examples:

$$\begin{array}{lcll}
\mathsf{Stream}\ A & = & \nu X \{\mathsf{head} : A, \mathsf{tail} : X\} & \\
\mathsf{Colist}\ A & = & \nu X \{\mathsf{out} : \mu\_\ \langle \mathsf{nil}\ 1 \mid \mathsf{cons}\ (A \times X) \rangle\} & \\
\mathsf{Vector}\ A & = & \nu\_\{\mathsf{length} : \mathsf{Nat},\ \mathsf{elems} : \mathsf{List}\ A\} & \\
\top & = & \nu\_\{\} & \text{(negative unit type)}
\end{array}$$

Both D and R can be seen as finite maps from a set of labels (constructors and destructors, resp.) to types, with application written D_c and R_d. We write $c \in D$ and $d \in R$ to express that a label is in the domain of the corresponding finite map.

In this article, both $\mu X D$ and $\nu X R$ 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 $\mu X D$ to be a least fixed-point inhabited only by finite data, and $\nu X R$ 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.

3.2 Terms and typing

Next we describe terms which constitute the targets of our rewrite rules. Terms are given by the following grammar:

$$\begin{array}{lcll}
e, t, u & ::= & f & \text{Defined symbol (e.g. function)} \\
 & \mid & x & \text{Variable} \\
 & \mid & () & \text{Unit (empty tuple)} \\
 & \mid & (t_1, t_2) & \text{Pair} \\
 & \mid & c\ t & \text{Constructor application} \\
 & \mid & t_1\ t_2 & \text{Application} \\
 & \mid & t.d & \text{Destructor application}
\end{array}$$

Terms can be identifiers (variable x, defined symbol f), *introductions* (tuple $()$, (t_1, t_2), constructed term $c\ t$) of *positive* types (1, $A \times B$, $\mu X D$), or *eliminations* (application $t_1\ t_2$, projection $t.d$) of *negative* types ($A \to B$, $\nu X R$). Constructor applications choose a variant and fold the recursive type; destructor applications unfold the recursive type and select a component of the record. Missing, by intention, are eliminations for positive types like tuple projections and *case*; these are replaced by *pattern matching*. Dually, we omit introductions for negative types, such as λ-abstractions and record values; instead we have *definitions by copattern matching* (see 3.4).

$\boxed{\Delta \vdash t : A}$ In context Δ, term t can be assigned type A.

$$\frac{\Delta(x) = A}{\Delta \vdash x : A}\ \mathrm{T}_{\mathrm{Var}} \qquad \frac{}{\Delta \vdash f : \Sigma(f)}\ \mathrm{T}_{\mathrm{Fun}} \qquad \frac{}{\Delta \vdash () : 1}\ \mathrm{T}_{\mathrm{Unit}}$$

$$\frac{\Delta \vdash t : D_c[\mu X D/X]}{\Delta \vdash c\, t : \mu X D}\ \mathrm{T}_{\mathrm{Const}} \qquad \frac{\Delta \vdash t : \nu X R}{\Delta \vdash t.d : R_d[\nu X R/X]}\ \mathrm{T}_{\mathrm{Dest}}$$

$$\frac{\Delta \vdash t_1 : A_1 \to A_2 \quad \Delta \vdash t_2 : A_1}{\Delta \vdash t_1\ t_2 : A_2}\ \mathrm{T}_{\mathrm{App}}$$

$$\frac{\Delta \vdash t_1 : A_1 \quad \Delta \vdash t_2 : A_2}{\Delta \vdash (t_1, t_2) : A_1 \times A_2}\ \mathrm{T}_{\mathrm{Pair}}$$

Figure 3. Typing rules for terms.

Typing *Contexts* Γ and Δ denote finite maps from term variables to well-formed types. To ensure linearity in pattern typing, we write Δ, Δ' for the *disjoint union* of finite maps Δ and Δ', i. e., $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \emptyset$. We write $\cdot$ or simply nothing for a finite map with empty domain, and we usually drop the braces when giving a context explicitly as set of pairs $\{x_1 : A_1, \ldots, x_n : A_n\}$. We assume a global *signature* Σ which maps defined symbols f to their type.

The rules for the typing judgement $\Delta \vdash t : A$ are given in Figure 3. Note that, since we have no binder on the term level —no λ, in particular—, Δ remains fixed in all the rules. Assumptions in Δ describe the type of pattern variables occurring on the left hand side of a rule and are synthesized when analyzing copatterns. For each term constructor there is exactly one typing rule, so we trivially obtain the usual inversion lemmata.

In the following, whenever we have a judgement J (e.g. a typing judgement), we write $\boxed{\mathcal{D} :: J}$ to indicate that $\mathcal{D}$ is a derivation of J using the rules for J. Usually our proof proceeds by induction on a derivation of our judgement and we write in this case "by induction on $\mathcal{D}$". Some of our judgements are algorithmic, i. e., partial functional relations. Unless stated otherwise, all arguments to these relations are inputs.

Bidirectional Type Checking Our language naturally supports overloading of constructors and destructors, when employing a bidirectional type checking algorithm [Pierce and Turner 1998]. Supporting overloading is convenient in practice and leads to elegant, compact and readable code. With bidirectional checking, overloading comes for free since a constructor c gets its meaning in the context of a data type—to type check a constructed term $c\ t$ we push its type $\mu X D$ in. Dually, a destructor d only has a meaning in the context of a record type $\nu X R$, which is inferred from head t in the projection term $t.d$. Overloading projections is standard in object-oriented programming (here, projections correspond to method calls), and has contributed to the success of the OO paradigm. Constructor overloading is also emerging, e. g., in the dependently typed languages Agda [Norell 2007] and Epigram [McBride and McKinna 2004].

Given a typing context Δ, we infer the type A of identifiers and eliminations (judgement $\Delta \vdash t \Rightarrow A$), while we check introductions against a given type A (judgement $\Delta \vdash t \Leftarrow A$). The

$\boxed{\Delta \vdash t \Rightarrow A}$ In context Δ, the type of term t is inferred as A.

$$\frac{}{\Delta \vdash f \Rightarrow \Sigma(f)}\ \text{TC}_{\text{Fun}} \qquad \frac{\Delta(x) = A}{\Delta \vdash x \Rightarrow A}\ \text{TC}_{\text{Var}}$$

$$\frac{\Delta \vdash t_1 \Rightarrow A_1 \to A_2 \quad \Delta \vdash t_2 \Leftarrow A_1}{\Delta \vdash t_1\, t_2 \Rightarrow A_2}\ \text{TC}_{\text{App}}$$

$$\frac{\Delta \vdash t \Rightarrow \nu X R}{\Delta \vdash t.d \Rightarrow R_d[\nu X R/X]}\ \text{TC}_{\text{Dest}}$$

$\boxed{\Delta \vdash t \Leftarrow A}$ In context Δ, term t checks against type A.

$$\frac{\Delta \vdash t \Rightarrow A \quad A = C}{\Delta \vdash t \Leftarrow C}\ \text{TC}_{\text{Switch}} \quad \frac{\Delta \vdash t \Leftarrow D_c[\mu X D/X]}{\Delta \vdash c\, t \Leftarrow \mu X D}\ \text{TC}_{\text{Constr}}$$

$$\frac{}{\Delta \vdash () \Leftarrow 1}\ \text{TC}_{\text{Unit}} \quad \frac{\Delta \vdash t_1 \Leftarrow A_1 \quad \Delta \vdash t_2 \Leftarrow A_2}{\Delta \vdash (t_1, t_2) \Leftarrow A_1 \times A_2}\ \text{TC}_{\text{Pair}}$$

Figure 4. Type-checking rules for terms.

rules are given in Figure 4. Type checking is trivially sound, but it is also complete without the need for any additional type annotations.

THEOREM 1 (Soundness of type checking).

1. *If $\mathcal{D} :: \Delta \vdash t \Rightarrow A$ then $\Delta \vdash t : A$.*
2. *If $\mathcal{D} :: \Delta \vdash t \Leftarrow A$ then $\Delta \vdash t : A$.*

Proof. Simultaneously by induction on the derivation $\mathcal{D}$. □

For simply-typed lambda-calculus, bidirectional type checking is not complete and typically requires type annotations. It fails for redexes $(\lambda x t)\, u$, since the type of a λ is not inferred. In our case, since for a type we have either introduction or elimination, we lack the usual redexes, thus, bidirectional type checking is actually complete.

THEOREM 2 (Completeness of type checking). *If $\mathcal{D} :: \Delta \vdash t : A$ then $\Delta \vdash t \Leftarrow A$, and if A is a negative type, then $\Delta \vdash t \Rightarrow A$.*

Proof. By induction on $\mathcal{D}$. Note that a proof of $\Delta \vdash t \Rightarrow A$ is sufficient, since this trivially implies $\Delta \vdash t \Leftarrow A$ by $\text{TC}_{\text{Switch}}$. □

3.3 Patterns and copatterns

The driving force behind computation in our language is pattern and copattern matching. Pattern matching allows us to compensate for the missing eliminations for positive types, while copattern matching compensates for the missing introductions for negative types. In the following, we present (co)patterns and their typing.

Patterns

p	$::=$	x	Variable pattern
	$\mid$	$()$	Unit pattern
	$\mid$	(p_1, p_2)	Pair pattern
	$\mid$	$c\, p$	Constructor pattern

Copatterns

q	$::=$	$\cdot$	Hole
	$\mid$	$q\, p$	Application copattern
	$\mid$	$q.d$	Destructor copattern

The postfix application of a projection d in $q.d$ corresponds to the prefix application $d\, q$ we used in the introduction, to conform with Haskell and Agda syntax. Note that, in contrast to convention in the ML dialects, projection does *not* bind stronger than application, i. e., $f\ x\ .d$ is to be read $(f\ x).d$. Our style saves parentheses when writing nested copatterns.

Pattern typing is defined in Figure 5. It computes a context Δ containing all the variables in the pattern. A (co)pattern p (or q) must be linear, that is, each variable in Δ appears exactly once in p (or q, resp.). Again, there are two modes for pattern typing. The checking mode, denoted by $\Delta \vdash p \Leftarrow A$, works on patterns p and follows the checking mode for regular typing. The inference mode, denoted by $\Delta \mid A \vdash q \Rightarrow C$ works on copatterns q and additionally computes its type C from the given type A of the hole.

$\boxed{\Delta \vdash p \Leftarrow A}$ Pattern p checks against type A, yielding Δ.

$$\frac{}{x : A \vdash x \Leftarrow A}\ \text{PC}_{\text{Var}} \qquad \frac{\Delta \vdash p \Leftarrow D_c[\mu X D/X]}{\Delta \vdash c\, p \Leftarrow \mu X D}\ \text{PC}_{\text{Const}}$$

$$\frac{}{\vdash () \Leftarrow 1}\ \text{PC}_{\text{Unit}} \quad \frac{\Delta_1 \vdash p_1 \Leftarrow A_1 \quad \Delta_2 \vdash p_2 \Leftarrow A_2}{\Delta_1, \Delta_2 \vdash (p_1, p_2) \Leftarrow A_1 \times A_2}\ \text{PC}_{\text{Pair}}$$

$\boxed{\Delta \mid A \vdash q \Rightarrow C}$ Copattern q eliminates given type A into inferred type C, yielding context Δ.

$$\frac{}{\cdot \mid A \vdash \cdot \Rightarrow A}\ \text{PC}_{\text{Head}} \quad \frac{\Delta \mid A \vdash q \Rightarrow \nu X R}{\Delta \mid A \vdash q.d \Rightarrow R_d[\nu X R/X]}\ \text{PC}_{\text{Dest}}$$

$$\frac{\Delta_1 \mid A \vdash q \Rightarrow B \to C \quad \Delta_2 \vdash p \Leftarrow B}{\Delta_1, \Delta_2 \mid A \vdash q\, p \Rightarrow C}\ \text{PC}_{\text{App}}$$

Figure 5. Type checking for patterns and rewrite rules.

Again, there is a one-to-one connection between pattern constructors and pattern typing rules. A standard inversion lemma holds for all rules in Figure 5.

3.4 Programs

A program $\mathcal{P} ::= (\Sigma, \mathsf{Rules})$ consists of a signature Σ mapping defined symbols f to their types and a collection Rules of rewrite rules. For each symbol f defined in the signature, $\mathsf{Rules}(f)$ gives the rewrite rules for f as a set of pairs $(q \mapsto e)$, called *observations*, which define the behavior of f. We require a dedicated symbol $\mathsf{main} \in \Sigma$, called the *entry point*, that determines the value of a program. Execution of a program means rewriting main with the Rules until no more rewriting is possible.

The informal syntax used in the introduction can be mechanically transformed into programs of form $\mathcal{P}$. For instance, the definition fib of the stream of Fibonacci numbers corresponds to the following entries in Σ and Rules.

$$\Sigma(\mathsf{fib}) \quad = \nu X \{\mathsf{head} : \mu Y\ \langle \mathsf{zero}\ 1 \mid \mathsf{suc}\ Y \rangle\ ,\ \mathsf{tail} : X\}$$

$$\mathsf{Rules}(\mathsf{fib}) = \left\{ \begin{array}{ll} \cdot\ .\mathsf{head} & \mapsto \mathsf{zero}\ () \\ \cdot\ .\mathsf{tail}\ .\mathsf{head} & \mapsto \mathsf{suc}\ (\mathsf{zero}\ ()) \\ \cdot\ .\mathsf{tail}\ .\mathsf{tail} & \mapsto \mathsf{zipWith}\ \_{+}\_\ \ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail}) \end{array} \right\}$$

A complete program needs also entries for $\mathsf{zipWith}$ and $\_{+}\_$, and a symbol main. The type of main should be positive, otherwise the result of the program is an unprintable infinite object. Here, main could be a function listing the first 42 elements of the Fibonacci stream—we leave the details to the imagination of the reader.

A program $\mathcal{P}$ is well-typed if $\vdash \mathcal{P}$ as given in Figure 6, which in essence says that any rule $(q \mapsto u)$ for any defined symbol f must be well-typed. A first result, proven in the next section, is that during the execution of a well-typed program we never encounter a term which is ill-typed.

4. Evaluation and Type Preservation

In this section, we define program evaluation in terms of a small-step reduction relation. To decide whether a rewrite rule can fire,

$\boxed{\vdash q[f] \mapsto u}$ Check rewrite rule.

$$\frac{\Delta \mid \Sigma(f) \vdash q \Rightarrow C \quad \Delta \vdash u \Leftarrow C}{\vdash q[f] \mapsto u}\ \mathrm{TC}_{\mathrm{Rule}}$$

$\boxed{\vdash \mathcal{P}}$ Check program.

$$\frac{\mathsf{main} \in \Sigma \quad \forall f \in \Sigma,\ (q \mapsto u) \in \mathsf{Rules}(f).\ \vdash q[f] \mapsto u}{\vdash (\Sigma, \mathsf{Rules})}\ \mathrm{TC}_{\mathrm{Prg}}$$

Figure 6. Well-typed rules and programs.

we match evaluation contexts against copatterns. We prove that reduction preserves types.

4.1 Evaluation contexts

Evaluation contexts E are elimination terms with a hole in head position. They generalize copatterns q in that they allow arbitrary terms e instead of just patterns p in argument positions.

$$\begin{array}{lcll} E & ::= & \cdot & \text{Hole} \\ & \mid & E\,e & \text{Application} \\ & \mid & E.d & \text{Projection} \end{array}$$

The hole $\cdot$ can be considered as a special variable. We write $E[t]$ as shorthand for $E[t/\cdot]$. Typing $\Delta \mid A \vdash E : C$ for evaluation context E is defined in Figure 7. This judgement holds iff $\Delta, x : A \vdash E[x] : C$ for a fresh variable $x \notin \Delta$. Well-typed evaluation contexts compose.

LEMMA 3 (Composition of contexts). *If* $\mathcal{D} :: \Gamma \mid A \vdash E_1 : B$ *and* $\mathcal{E} :: \Gamma \mid B \vdash E_2 : C$*, then* $\Gamma \mid A \vdash E_2[E_1[\cdot]] : C$.

$\boxed{\Delta \mid A \vdash E : C}$ In context Δ, evaluation context E eliminates type A into type C.

$$\frac{}{\Gamma \mid A \vdash \cdot : A}\ \mathrm{ET}_{\mathrm{Head}} \qquad \frac{\Gamma \mid A \vdash E : \nu X R}{\Gamma \mid A \vdash E.d : R_d[\nu X R/X]}\ \mathrm{ET}_{\mathrm{Dest}}$$

$$\frac{\Gamma \mid A \vdash E : B \to C \quad \Gamma \vdash e : B}{\Gamma \mid A \vdash E\,e : C}\ \mathrm{ET}_{\mathrm{App}}$$

Figure 7. Typing rules for evaluation context.

4.2 Pattern matching

Matching a term t against a pattern p, if successful, yields a substitution σ such that $p[\sigma] = t$. Pattern matching is defined in terms of a judgement $t =^? p \searrow \sigma$ whose rules appear in Figure 8. Herein, a substitution σ is a finite map from variables to terms; we write $\cdot$ for the empty map, t/x for the singleton mapping x to t and σ, σ' for the disjoint union of two maps σ and σ'. Substitution typing $\boxed{\Gamma \vdash \sigma : \Delta}$ simply means that for all $x \in \Delta$, we have $\Gamma \vdash \sigma(x) : \Delta(x)$.

While matching patterns p is standard, matching copatterns q is straightforward as well. The hole $\cdot$ serves as "anchor" and in an implementation it seems wise to match "inside-out", i. e., start at the hole and proceed outwards.

4.3 Reduction and type preservation

The only source of computation in our language is a defined function symbol f in an evaluation context E that matches the copattern q of one of the rules $(q \mapsto u) \in \mathsf{Rules}(f)$. Such an $e = E[f]$ is a redex which can be *contracted* to another expression e', written $\boxed{e \mapsto e'}$. The precise rule for contraction is:

$\boxed{t =^? p \searrow \sigma}$ Term t matches with pattern p under substitution σ.

$$\frac{}{t =^? x \searrow t/x}\ \mathrm{PM}_{\mathrm{Var}} \qquad \frac{t =^? p \searrow \sigma}{c\,t =^? c\,p \searrow \sigma}\ \mathrm{PM}_{\mathrm{Constr}}$$

$$\frac{}{() =^? () \searrow \cdot}\ \mathrm{PM}_{\mathrm{Unit}} \qquad \frac{t_1 =^? p_1 \searrow \sigma_1 \quad t_2 =^? p_2 \searrow \sigma_2}{(t_1, t_2) =^? (p_1, p_2) \searrow \sigma_1, \sigma_2}\ \mathrm{PM}_{\mathrm{Pair}}$$

$\boxed{E =^? q \searrow \sigma}$ Evaluation context E matches copattern q returning substitution σ.

$$\frac{}{\cdot =^? \cdot \searrow \cdot}\ \mathrm{PM}_{\mathrm{Head}} \qquad \frac{E =^? q \searrow \sigma}{E.d =^? q.d \searrow \sigma}\ \mathrm{PM}_{\mathrm{Dest}}$$

$$\frac{E =^? q \searrow \sigma \quad t =^? p \searrow \sigma'}{E\,t =^? q\,p \searrow \sigma, \sigma'}\ \mathrm{PM}_{\mathrm{App}}$$

Figure 8. Rules for pattern matching.

$$\frac{E =^? q \searrow \sigma}{E[f] \mapsto u[\sigma]}\ (q \mapsto u) \in \mathsf{Rules}(f)$$

One step reduction $\boxed{e \longrightarrow e'}$ is defined as the compatible closure of contraction, i. e., e *reduces* to e' if e' results from contraction of one redex in e. We omit the standard inductive definition of $e \longrightarrow e'$.

Our first major result is that reduction preserves types. We assume a well-typed program, i. e. all rewrite rules are well-typed.

THEOREM 4 (Subject reduction). *If* $\Gamma \vdash e : A$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : A$

Subject reduction is a consequence of the following statements: Substitution preserves types, (co)pattern matching yields a well-typed substitution, and contraction preserves types.

LEMMA 5 (Substitution). *If* $\mathcal{D} :: \Delta \vdash u : C$ *and* $\mathcal{E} :: \Gamma \vdash \sigma : \Delta$ *then* $\mathcal{F} :: \Gamma \vdash u[\sigma] : C$ *for some* $\mathcal{F}$.

Proof. By induction on $\mathcal{D}$. □

LEMMA 6 (Adequacy of pattern matching). *If* $\mathcal{D} :: \Delta \vdash p \Leftarrow A$ *and* $\mathcal{E} :: \Gamma \vdash e : A$ *and* $\mathcal{F} :: e =^? p \searrow \sigma$ *then* $\Gamma \vdash \sigma : \Delta$.

Proof. By induction on $\mathcal{F}$. □

LEMMA 7 (Adequacy of copattern matching). *If* $\mathcal{D} :: \Delta \mid A \vdash q \Rightarrow C$ *and* $\mathcal{E} :: \Gamma \mid A \vdash E : B$ *and* $\mathcal{F} :: E =^? q \searrow \sigma$ *then* $C = B$ *and* $\Gamma \vdash \sigma : \Delta$.

Proof. By induction on $\mathcal{F}$. □

LEMMA 8 (Correctness of contraction). *If* $\Gamma \mid \Sigma(f) \vdash E : C$ *and* $\vdash q[f] \mapsto u$ *and* $E =^? q \searrow \sigma$ *then* $\Gamma \vdash u[\sigma] : C$.

Proof. By assumption, we have

$$\mathcal{D} :: \frac{\overset{\mathcal{D}_1}{\Delta \mid \Sigma(f) \vdash q \Rightarrow B} \quad \overset{\mathcal{D}_2}{\Delta \vdash u \Leftarrow B}}{\vdash q[f] \mapsto u}$$

since it is the only rule that could have been used.

By Lemma 7, using $\mathcal{D}_1$ and both assumptions we have that $C = B$ and $\Gamma \vdash \sigma : \Delta$. Then, by Substitution (Lemma 5) and $\mathcal{D}_2$, we conclude that $\Gamma \vdash u[\sigma] : C$. □

Finally, the subject reduction theorem follows:

Proof of Theorem 4. By induction on the reduction relation, with contraction being the only interesting case:

$$\Gamma \vdash E[f] : B \quad \text{and} \quad \frac{E =^{?} q \searrow \sigma}{E[f] \mapsto u[\sigma]}\ (q \mapsto u) \in \mathsf{Rules}(f)$$

By well-typedness $\vdash q[f] \mapsto u$, we obtain from Lemma 8 that $\Gamma \vdash u[\sigma] : B$. □

5. Copattern Coverage and Progress

A fundamental property of strongly typed languages is *type soundness*; in the words of Milner [1978] "well-typed programs do not go wrong". This means that well-typed programs either produce a value or run forever, but never get *stuck* by encountering an invalid operation, like adding a function to a string or calling a number as one would call a function. For our language, there are three reasons why a program is stuck, i. e., no reduction step is possible yet we have not reached a printable value:

1. Missing rule. We might have defined a function $f : \mathsf{Nat} \to A$ but only given a rewrite rule $f\ \mathsf{zero} \mapsto \ldots$. In this case, $f\ (\mathsf{suc}\ n)$ is stuck. In this section, we give rules for *copattern coverage* that ensure no rewrite rules are forgotten.
2. Ill-typed term. The term $f\ \mathsf{nil}$ is stuck even if we have given a complete implementation of $f : \mathsf{Nat} \to A$. However, ill-typed terms like $f\ \mathsf{nil}$ are already excluded by type checking and the type preservation theorem.
3. Infinite object. The term f does not evaluate by itself; it is an underapplied function. However, just as the typical interpreter, we consider terms of negative types as values. As a consequence, our notion of value is not syntactic, but type-dependent.

As main technical result of this section and the article, we prove type soundness, syntactically [Wright and Felleisen 1994], by showing the progress theorem for a call-by-value strategy.

5.1 Values and evaluation contexts

Values are defined using a new judgment $\Delta \vdash_v e : A$ to mean that the expression e is a value of type A under the context Δ. We also use v to denote an expression which acts as a value. Whether an expression is considered a value or not depends also on its type, in particular, each expression of negative type N is considered a value—the rules are given in Figure 9.

$\boxed{\Delta \vdash_v e : A}$ In context Δ, e is a value of type A.

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash_v x : A}\ \mathrm{V}_{\text{Var}} \qquad \frac{\Gamma \vdash_v v : D_c[\mu X D / X]}{\Gamma \vdash_v c\, v : \mu X D}\ \mathrm{V}_{\text{Const}}$$

$$\frac{}{\Gamma \vdash_v () : 1}\ \mathrm{V}_{\text{Unit}} \qquad \frac{\Gamma \vdash_v v_1 : A_1 \quad \Gamma \vdash_v v_2 : A_2}{\Gamma \vdash_v (v_1, v_2) : A_1 \times A_2}\ \mathrm{V}_{\text{Pair}}$$

$$\frac{\Gamma \vdash e : N}{\Gamma \vdash_v e : N}\ \mathrm{V}_{\text{Neg}}$$

Figure 9. Rules for values.

LEMMA 9 (Inversion for values). *The following hold for $v \neq x$.*

1. *If $\Gamma \vdash_v v : 1$ then $v = ()$.*
2. *If $\Gamma \vdash_v v : A_1 \times A_2$ then $v = (v_1, v_2)$, $\Gamma \vdash_v v_1 : A_1$ and $\Gamma \vdash_v v_2 : A_2$.*
3. *If $\Gamma \vdash_v v : \mu X D$ then $v = c\ v'$ for some $c \in D$ and $\Gamma \vdash v' : D_c[\mu X D / X]$.*

We dualize the notion of value for terms to evaluation contexts, introducing a judgement $\Delta \mid A \vdash_v E : C$ (see Figure 10). It accepts those well-typed evaluation contexts E that have values in all argument positions. The idea is that if E is "long enough", i. e., if C is a positive type, then $E[f]$ is a redex because one of the defining copatterns for f has to match E. This would not necessary be the case if the arguments in E were not values.

$\boxed{\Delta \mid A \vdash_v E : C}$ E is an evaluation context with only values in application arguments.

$$\frac{}{\Gamma \mid A \vdash_v \cdot : A}\ \mathrm{EV}_{\text{Head}} \qquad \frac{\Gamma \mid A \vdash_v E : \nu X R}{\Gamma \mid A \vdash_v E.d : R_d[\nu X R / X]}\ \mathrm{EV}_{\text{Dest}}$$

$$\frac{\Gamma \mid A \vdash_v E : B \to C \quad \Gamma \vdash_v v : B}{\Gamma \mid A \vdash_v E\, v : C}\ \mathrm{EV}_{\text{App}}$$

Figure 10. Rules for value evaluation contexts.

The following two propositions enable us to analyze non-empty value evaluation contexts from the inside out; they will be used in Theorem 12.

LEMMA 10 (Splitting a function evaluation context). *If $\mathcal{D} :: \Gamma \mid B \to C \vdash_v E : A$ and $E \neq \cdot$ then $E = E'[\cdot\ v]$ with $\Gamma \vdash_v v : B$ and $\Gamma \mid C \vdash_v E' : A$.*

Proof. By induction on $\mathcal{D}$. □

LEMMA 11 (Splitting a record evaluation context). *If $\mathcal{D} :: \Gamma \mid \nu X R \vdash_v E : A$ and $E \neq \cdot$ then $E = E'[\cdot.d]$ with $\Gamma \mid R_d[\nu X R / X] \vdash_v E' : A$.*

Proof. By induction on $\mathcal{D}$. □

5.2 Coverage

Figure 11 defines a judgment to indicate that a list of copatterns covers all eliminations of a given type A. The judgment is $A \lhd \mid (\Delta \vdash q \Rightarrow C)$ or, more generally, $A \lhd \mid \vec{Q}$ where $\vec{Q} = (\Delta_i \vdash q_i \Rightarrow C_i)_{i=1,\ldots,n}$ is a set of non-overlapping copatterns q_i with their type C_i and context Δ_i, each satisfying $\Delta_i \mid A \vdash q_i \Rightarrow C_i$.

The rules to construct a covering set of copatterns are not syntax-directed. To check whether a given set of copatterns $\vec{Q}$ for a type A is complete, we non-deterministically guess the derivation of $A \lhd \mid \vec{Q}$, if it exists. Although this NP-algorithm is not the best we can do, we are confident that we can adopt existing efficient coverage algorithms [Norell 2007] for our language.

The initial covering is given by the axiom C_{Hole}. We can refine a covering $\vec{Q}$ by focusing on one copattern Q and either *split the result* of negative type or split one of its variables of positive type. Result splitting at function type $B \to C$ applies the copattern q to a fresh variable $x : B$, at record type $\nu X R$ we take all projections $(q.d)_{d \in R}$. Splitting a variable x replaces it by unit $()$, a pair (x_1, x_2) or all possible constructors $(c\ x')_{c \in D}$, in accordance with the type 1, $A_1 \times A_2$, or $\mu X D$ of the variable.

Let us revisit the example of the function cycleNats from the introduction and walk through the rules for coverage. With the following shorthands for types

$$\begin{array}{lcl} \mathsf{Nat} & = & \mu X \langle \mathsf{zero}\ 1 \mid \mathsf{suc}\ X \rangle \\ \mathsf{StreamNat} & = & \nu X \{\mathsf{head} : \mathsf{Nat}, \mathsf{tail} : X\}, \end{array}$$

$\boxed{A \lhd\!\mid \vec{Q}}$ Typed copatterns $\vec{Q}$ cover elimination of type A.

Result splitting:

$$\frac{}{A \lhd\!\mid (\cdot \vdash \cdot \Rightarrow A)}\ \mathrm{C}_{\mathrm{Hole}} \qquad \frac{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q \Rightarrow B \to C)}{A \lhd\!\mid \vec{Q}\,(\Delta, x : B \vdash q\,x \Rightarrow C)}\ \mathrm{C}_{\mathrm{App}}$$

$$\frac{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q \Rightarrow \nu X R)}{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q.d \Rightarrow R_d[\nu X R/X])_{d \in R}}\ \mathrm{C}_{\mathrm{Dest}}$$

Variable splitting:

$$\frac{A \lhd\!\mid \vec{Q}\,(\Delta, x : 1 \vdash q \Rightarrow C)}{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q[()/x] \Rightarrow C)}\ \mathrm{C}_{\mathrm{Unit}}$$

$$\frac{A \lhd\!\mid \vec{Q}\,(\Delta, x : A_1 \times A_2 \vdash q \Rightarrow C)}{A \lhd\!\mid \vec{Q}\,(\Delta, x_1 : A_1, x_2 : A_2 \vdash q[(x_1, x_2)/x] \Rightarrow C)}\ \mathrm{C}_{\mathrm{Pair}}$$

$$\frac{A \lhd\!\mid \vec{Q}\,(\Delta, x : \mu X D \vdash q \Rightarrow C)}{A \lhd\!\mid \vec{Q}\,(\Delta, x' : D_c[\mu X D/X] \vdash q[c\,x'/x] \Rightarrow C)_{c \in D}}\ \mathrm{C}_{\mathrm{Const}}$$

Figure 11. Rules for copattern coverage.

the signature entries for cycleNats are the following:

$$\begin{array}{lcl} \Sigma(\mathsf{cycleNats}) & = & \mathsf{Nat} \to \mathsf{StreamNat} \\ \mathsf{Rules}(\mathsf{cycleNats}) & = & \left\{ \begin{array}{lll} \cdot\, x & .\mathsf{head} & \mapsto x \\ \cdot\, (\mathsf{zero}\,()) & .\mathsf{tail} & \mapsto \mathsf{cycleNats}\ N \\ \cdot\, (\mathsf{suc}\ x) & .\mathsf{tail} & \mapsto \mathsf{cycleNats}\ x \end{array} \right\} \end{array}$$

To check coverage, we start with the trivial covering and successively apply the rules until we obtain the copatterns of cycleNats. Since $A = \mathsf{Nat} \to \mathsf{StreamNat}$ stays fixed throughout the derivation, we omit it and just write the copattern list $\vec{Q}$. We start with $\mathrm{C}_{\mathrm{Hole}}$.

$$(\cdot \vdash \cdot \Rightarrow \mathsf{Nat} \to \mathsf{StreamNat})$$

We apply x to the hole by $\mathrm{C}_{\mathrm{App}}$.

$$(x : \mathsf{Nat} \vdash \cdot\, x \Rightarrow \mathsf{StreamNat}).$$

Then we split the result by $\mathrm{C}_{\mathrm{Dest}}$.

$$\begin{array}{l} (x : \mathsf{Nat} \vdash \cdot\, x\ .\mathsf{head} \Rightarrow \mathsf{Nat}) \\ (x : \mathsf{Nat} \vdash \cdot\, x\ .\mathsf{tail}\ \ \Rightarrow \mathsf{StreamNat}) \end{array}$$

In the second copattern we split x via $\mathrm{C}_{\mathrm{Const}}$, reusing the variable name x.

$$\begin{array}{llll} (x : \mathsf{Nat} & \vdash \cdot\, x & .\mathsf{head} & \Rightarrow \mathsf{Nat}) \\ (x : 1 & \vdash \cdot\, (\mathsf{zero}\ x) & .\mathsf{tail} & \Rightarrow \mathsf{StreamNat}) \\ (x : \mathsf{Nat} & \vdash \cdot\, (\mathsf{suc}\ x) & .\mathsf{tail} & \Rightarrow \mathsf{StreamNat}). \end{array}$$

Finally, we apply $\mathrm{C}_{\mathrm{Unit}}$, replacing x by $()$.

$$\begin{array}{llll} (x : \mathsf{Nat} & \vdash \cdot\, x & .\mathsf{head} & \Rightarrow \mathsf{Nat}) \\ (\cdot & \vdash \cdot\, (\mathsf{zero}\ ()) & .\mathsf{tail} & \Rightarrow \mathsf{StreamNat}) \\ (x : \mathsf{Nat} & \vdash \cdot\, (\mathsf{suc}\ x) & .\mathsf{tail} & \Rightarrow \mathsf{StreamNat}) \end{array}$$

The lists of copatterns $\vec{q}$ for type A generated by the splitting rules is complete in the sense that every closed value context E eliminating A into a positive type P actually matches one of the copatterns q_i.

THEOREM 12 (Matching with a covering copattern). *If $\mathcal{D} :: \cdot \mid A \vdash_v E : P$ and $\mathcal{E} :: A \lhd\!\mid (\Delta_i \vdash q_i \Rightarrow C_i)_{i=1..n}$, then there are E_1, E_2 such that $E = E_1[E_2[\cdot]]$, $E_2 =^? q_i \searrow \sigma$ for some i, $\cdot \mid A \vdash_v E_2 : C_i$ and $\cdot \mid C_i \vdash_v E_1 : P$.*

To prove this theorem, we use the following statements.

LEMMA 13 (Splitting a pattern variable). *Let $\mathcal{D} :: \Delta, x : A \mid B \vdash q \Rightarrow C$ and $\mathcal{E} :: \mid B \vdash_v E : C$ and $\mathcal{F} :: E =^? q \searrow \sigma$.*

1. *Assume $A = A_1 \times A_2$ and let $q' = q[(x_1, x_2)/x]$. Then $\Delta, x_1 : A_1, x_2 : A_2 \mid B \vdash q' \Rightarrow C$ and $E =^? q' \searrow \sigma'$ with $\sigma = \sigma'[x \mapsto (\sigma'(x_1), \sigma'(x_2))]$.*
2. *Assume $A = \mu X D$ and let $q' = q[c\,x'/x]$ for some $c \in D$. Then $\Delta, x' : D_c[\mu X D/X] \vdash q' \Rightarrow C$ and $E =^? q' \searrow \sigma'$ with $\sigma = \sigma'[x \mapsto c\,\sigma'(x')]$.*
3. *Assume $A = 1$ and let $q' = q[()/x]$. Then $\Delta \vdash q' \Leftarrow C$ and $E =^? q' \searrow \sigma'$ with $\sigma = \sigma'[x \mapsto ()]$.*

Proof. First, prove an adaptation of these statements for patterns p and values v instead of copatterns q and evaluation contexts E. Then, prove this lemma by induction on $\mathcal{F}$. □

Proof of Theorem 12. The theorem is proved by induction on the coverage derivation $\mathcal{E}$. The variable splitting cases $\mathrm{C}_{\mathrm{Pair}}$, $\mathrm{C}_{\mathrm{Const}}$, and $\mathrm{C}_{\mathrm{Unit}}$ follow from Lemma 13. We consider the rules for result splitting.

$$\text{Case } \mathcal{E} :: \frac{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q \Rightarrow B \to C)}{A \lhd\!\mid \vec{Q}\,(\Delta, x : B \vdash q\,x \Rightarrow C)_{i=1,\ldots,n}}$$

By induction hypothesis, the statement holds for one of the patterns in $\vec{Q}\ (\Delta \vdash q \Rightarrow B \to C)$. If the pattern has been chosen from $\vec{Q}$, we are done. Thus, without loss of generality, $E = E_1[E_2[\cdot]]$ and $\cdot \mid A \vdash_v E_2 : B \to C$ and $\cdot \mid B \to C \vdash_v E_1 : P$ and $E_2 =^? q \searrow \sigma$.

If $E_1 = \cdot$ then $P = B \to C$, which is a contradiction since P is a positive type. If $E_1 \neq \cdot$, then $E_1 = E_1'[\cdot\ v]$ with $\cdot \vdash_v v : B$ and $\cdot \mid C \vdash_v E_1' : P$ by Lemma 10.

Thus, let $E_2' = E_2\ v$ and $\cdot \mid A \vdash_v E_2' : C$ by $\mathrm{EV}_{\mathrm{App}}$, and $E_2' =^? q\,x \searrow \sigma, v/x$ by $\mathrm{PM}_{\mathrm{App}}$.

$$\text{Case } \mathcal{E} :: \frac{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q \Rightarrow \nu X R)}{A \lhd\!\mid \vec{Q}\,(\Delta \vdash q.d \Rightarrow R_d[\nu X R/X])_{d \in R}}$$

Analogously, using Lemma 11. □

5.3 Progress

We are ready to show that evaluation of a well-typed program does not get stuck, provided that all definitions come with a complete set of observations. First we note that closed terms are either values or eliminations of a defined symbols. Such an elimination is either a value evaluation context or contains a closed non-value.

LEMMA 14 (Decomposition). *If $\cdot \vdash e : A$ then either*

1. *$e = (), A = 1$,*
2. *$e = (e_1, e_2), A = A_1 \times A_2$,*
3. *$e = c\,e', A = \mu X D$,*
4. *$e = E_2[E_1[f]\ e']$ where $\cdot \mid \Sigma(f) \vdash_v E_1 : B \to C$ and $\cdot \mid C \vdash E_2 : A$ and $\not\vdash_v e' : B$ for some f, some evaluation contexts E_1, E_2, some term e' and some types B, C.*
5. *$e = E[f]$ for some f, E with $\cdot \mid \Sigma(f) \vdash_v E : A$.*

Proof. By induction on e. We only show the cases $e = e_1\ e_2$ and $e = e'.d$. The other cases are trivial.

Case $\vdash e_1\ e_2 : A$. Then by inversion $\vdash e_1 : B \to A$ and $\vdash e_2 : B$. By induction hypothesis $e_1 = E[f]$ with $\cdot \mid \Sigma(f) \vdash_v E : B \to A$ for some f, E or $e_1 = E_2[E_1[f]\ e']$ for some f, E_1, E_2, and e' where $\cdot \mid \Sigma(f) \vdash_v E_1 : B' \to C', \cdot \mid C' \vdash E_2 : B \to A$ and $\not\vdash_v e' : B'$, as the 3 other cases are impossible. In the former case, if $\not\vdash_v e_2 : B$, we can obtain case 4 by letting $E_2 = \cdot, E = E_1$ and $e_2 = e'$. This gives us $e_1\ e_2 = \cdot[E[f]\ e_2]$. If $\vdash_v e_2 : B$, then, by EV$_{\text{App}}$, $\cdot \mid \Sigma(f) \vdash_v E[f]\ e_2 : A$ and $E' = E\ e_2$. In the latter case, we have $E_2[E_1[f]\ e']\ e_2 = E_2'[E_1[f]\ e']$ by setting $E_2' = E_2\ e_2$.

Case $\vdash e.d : A$. Then by inversion, $\vdash e : \nu X R$ for some R. By induction hypothesis, $e = E[f]$ and $\cdot \mid \Sigma(f) \vdash_v E : \nu X R$, or $e = E_2[E_1[f]\ e']$ where e' is not a value. In the former case, $e.d = E[f].d = E'[f]$ and $\cdot \mid \Sigma(f) \vdash_v E.d : R_d[\nu X R / X]$ by EV$_{\text{Dest}}$. Otherwise, $e.d = E_2[E_1[f]\ e'].d = E_2'[E_1[f]\ e']$. □

Finally we prove progress under the assumption that every definition f is complete, written $\Sigma(f) \lhd\mid \mathsf{Rules}(f)$.

THEOREM 15 (Progress). *If* $\mathcal{D} :: \vdash e : A$ *then either* $\vdash_v e : A$ *or* $e \longrightarrow e'$ *for some* e'.

Proof. The proof is done by induction on e. By Lemma 14, we have five possible cases. Since the four first cases follow by a simple induction argument, we only present the last case.

Here $e = E[f]$ and $\cdot \mid \Sigma(f) \vdash_v E : A$. If A is negative, then e is already considered a value and we are done. Otherwise, since by assumption $\Sigma(f) \lhd\mid \mathsf{Rules}(f)$, we can apply Theorem 12 and obtain E_1, E_2 such that $E = E_1[E_2[\cdot]]$, $E_2 \overset{?}{=} q_i \searrow \sigma$ for some $q_i \in \mathsf{Rules}(f)$, plus $\cdot \mid \Sigma(f) \vdash E_2 : C_i$ and $\cdot \mid C_i \vdash_v E_1 : A$. Thus, by our reduction rules $E_2[f] \mapsto u_i[\sigma]$ where $(q_i, u_i) \in \mathsf{Rules}(f)$ and so $E_2[f] \to u_i[\sigma]$. We conclude that $E_1[E_2[f]] \to E_1[u_i[\sigma]]$. □

6. Extensions and Implementations

Our core language misses introduction rules for functions and objects, thus, we do not have lambda abstractions or record expressions. However, we can embed sets of behaviors $\{\vec{q} \mapsto \vec{u}\}$ into the expression syntax and obtain *anonymous objects* that subsume λ abstractions, SML's anonymous functions defined by pattern matching, and record expressions:

$$\begin{array}{lcl}
\lambda x t & = & \{\cdot\, x \mapsto t\} \\
\begin{array}{l}\mathsf{fn}\ \mathsf{nil} \qquad\quad\ \Rightarrow \mathsf{false} \\ \quad \mid \mathsf{cons}\ x\ xs \Rightarrow \mathsf{true}\end{array} & = & \left\{\begin{array}{ll}\cdot\ \mathsf{nil} & \mapsto \mathsf{false} \\ \cdot\ (\mathsf{cons}\ x\ xs) & \mapsto \mathsf{true}\end{array}\right\} \\
\mathsf{record}\{\mathsf{fst} = t_1; \mathsf{snd} = t_2\} & = & \left\{\begin{array}{l}\cdot.\mathsf{fst} \mapsto t_1 \\ \cdot.\mathsf{snd} \mapsto t_2\end{array}\right\}
\end{array}$$

Of course, bidirectional type checking is no longer complete since anonymous objects can only be checked against a given type, but can appear in elimination position.

Copatterns have been added to the development version[3] of Agda [Agda team 2012]. Currently, projection copatterns are not part of the core of Agda, they are parsed but then translated into record expressions. This does not give the full flexibility of copatterns, but allows us to experiment with them. Full copatterns in the core would allow us to exploit the benefits of deep projection copatterns and mixed projection/application copatterns, but for that, Agda's coverage checker has to be extended to copatterns. To accomplish this, further research is required, because dependent pattern matching is a far from trivial enterprise [Coquand 1992; Schürmann and Pfenning 2003; Goguen et al. 2006; Norell 2007; Dunfield and Pientka 2009].

Another prototypical implementation of copatterns exists in MiniAgda [Abel 2012]. In MiniAgda, one can certify termination and productivity using sized types [Hughes et al. 1996; Barthe et al. 2004; Abel 2007]. Copatterns provide the right syntax to decorate corecursive definitions with size variables that witness productivity.

7. Related Work

Our work builds on the insight that finite datatypes correspond to initial algebras and infinite datatypes correspond to final coalgebras. This was first observed by Hagino [1989] and was the basis of categorical programming languages such as symML [Hagino 1987] and Charity [Cockett and Fukushima 1992]. Categorical programming languages typically support programming with the morphisms of category theory; while they do provide iteration, they do not support general recursion and pattern matching. In Charity, support for pattern matching on data types was added [Tuckey 1997], but it lacks support for copattern matching.

Our type theoretic development of copatterns exploits the duality of positive and negative types which is well known in focused proofs [Andreoli 1992]. Previously, focusing has been applied to pattern matching [Zeilberger 2008a; Krishnaswami 2009] and evaluation order [Zeilberger 2009; Curien and Herbelin 2000]. Closest to our work from a theoretical point of view is the work by Licata, Zeilberger and Harper [2008] where a language based on the sequent calculus is described which supports mixing LF types with computation-level types. The weak representational function space of LF is classified as a positive connective and admits pattern matching using constructor patterns; the strong computation level function space is classified as negative connective which is defined by destructor patterns. The accompanying technical report also describes briefly how to add ν-formulas to the proposed system. However, in their work, (co)pattern matching happens at the *meta level*; this is like replacing induction by an ω-rule. Our work provides an *object-level* syntax for copatterns and an *algorithm* for copattern coverage.

Kimura and Tatsuta [2009] extend Wadler's [2003] Dual Calculus to inductive and coinductive types, treating the constructor for inductive data as value constructor and the destructor for coinductive data as continuation constructor. However, they do not introduce recursive values or recursive continuations nor pattern and copattern matching, but allow only iteration over finite data and coiteration into infinite data.

Agda, in its currently released version 2.3.0, already avoids Coq's subject reduction problem. Infinite objects are created via *delay* ♯ and analyzed via *force* ♭, the corresponding operation on types is *lifting* ∞. In spirit, this approach mimics the standard trick in call-by-value languages such as ML and Scheme to encode lazy values by *suspensions*, i. e., functions over the unit type. The two-edged dependent pattern matching on infinite objects is ruled out, since one cannot match on functions.

Agda's coinduction is informally described by Danielsson and Altenkirch [2010], but it lacks solid theoretical backing. Indeed, compositionality is lost, because any data type that uses lifting is coinductive [Altenkirch and Danielsson 2010]. For instance, a data type of trees with infinite branching realized via streams host automatically infinitely *deep* trees, even if that is not expressed by the data type definition. Our work reinterpretes lifting as the generation of a mutual recursive record type that contributes the coinductive part to the data type. Forcing is interpreted as destructor and delaying as a mutual definition by destructor pattern. This way, we provide a standard semantics for coinduction in Agda and recover compositional construction of data types.

8. Conclusion

In this paper, we have presented a type-safe foundation for programming with infinite structures via observations. We model finite data using variant types and infinite data via record types. Pattern matching of finite data is extended with its dual notion of copattern matching on infinite data. While we do not consider termination

[3] Available from the darcs repository `http://code.haskell.org/Agda`.

and productivity in this paper, we guarantee that the functions are covering, i.e., they are defined on all possible inputs.

Copatterns lay a foundation for finitary rewriting with infinite objects. They are also an excellent candidate for representing corecursive definitions in type-theoretic proof assistants such as Coq and dependently typed languages like Agda.

In the future, we plan to extend the presented work to full dependent types. There are two main theoretical issues we need to tackle: first, extension of copattern coverage to dependent types, and secondly, checking termination and productivity of functions to guarantee strong normalization. A candidate for the latter task are sized types [Hughes et al. 1996; Barthe et al. 2004; Abel 2007] as already implemented in MiniAgda [Abel 2012]. Further, we aim at developing a denotational model for languages with copatterns. It seems that semantics based on orthogonality [Parigot 1997; Vouillon and Melliès 2004] provides a good starting point for this investigation.

From a practical point of view, we plan to fully integrate copatterns into Agda for a perspicuous and robust foundation of coinduction.

Acknowledgments

The first author thanks Thierry Coquand and Nils Anders Danielsson for email exchange about copatterns and the regular participants of the Agda Implementor's Meetings from 2008 on for lively discussions on copatterns and the best way to integrate coinduction into Agda. During an invitation to McGill University by the second author, much of the theory of copatterns was already worked out. He is also grateful towards Tarmo Uustalu and James Chapman for an invitation to the Institute for Cybernetics in Tallinn in November 2011. During that visit, the implementation of copatterns for Agda began. We also thank Jacques Carette and the anonymous referees for suggestions to improve this paper.

References

M. Abadi and L. Cardelli. A theory of primitive objects. Untyped and first-order systems. In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lect. Notes in Comput. Sci.*, pages 296–320. Springer, 1994.

A. Abel. Mixed inductive/coinductive types and strong normalization. In Z. Shao, editor, *Proc. of the 5th Asian Symp. on Programming Languages and Systems, APLAS 2007*, volume 4807 of *Lect. Notes in Comput. Sci.*, pages 286–301. Springer, 2007. ISBN 978-3-540-76636-0.

A. Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. *Electr. Proc. in Theor. Comp. Sci.*, 77:1–11, 2012. Proceedings of FICS 2012.

Agda team. The Agda Wiki, 2012.

T. Altenkirch and N. A. Danielsson. Termination checking in the presence of nested inductive and coinductive types. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers, FLoC 2010, 2010.

J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comput. Sci.*, 14 (1):97–141, 2004.

N. Benton, G. M. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proc. of the 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA '93*, volume 664 of *Lect. Notes in Comput. Sci.*, pages 75–90. Springer, 1993. ISBN 3-540-56517-5.

A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, June 2012. Unpublished draft.

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.

T. Coquand. Pattern matching with dependent types. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Types for Proofs and Programs (TYPES'92), Båstad, Sweden*, pages 71–83, 1992.

T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lect. Notes in Comput. Sci.*, pages 62–78. Springer, 1993.

P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of the 5th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2000)*, SIGPLAN Notices 35(9), pages 233–243. ACM Press, 2000. ISBN 1-58113-202-6.

N. A. Danielsson and T. Altenkirch. Subtyping, declaratively. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Proc. of the 10th Int. Conf. on Mathematics of Program Construction, MPC 2010*, volume 6120 of *Lect. Notes in Comput. Sci.*, pages 100–118. Springer, 2010. ISBN 978-3-642-13320-6.

J. Dunfield and B. Pientka. Case analysis of higher-order data. *Electr. Notes in Theor. Comp. Sci.*, 228:69–84, 2009.

E. Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, Dec. 1996. Thèse d'université.

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*, volume 4060 of *Lect. Notes in Comput. Sci.*, pages 521–540. Springer, 2006. ISBN 3-540-35462-X.

J. Granström. *Reference and Computation in Intuitionistic Type Theory*. PhD thesis, Mathematical Logic, Uppsala University, 2009.

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 *Lect. Notes in Comput. Sci.*, pages 140–157. Springer, 1987.

T. Hagino. Codatatypes in ML. *J. Symb. Logic*, 8(6):629–650, 1989.

J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL'96*, pages 410–423, 1996.

INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.3 edition, 2010.

J. Kennaway and F. de Vries. *Infinitary Rewriting*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*, chapter Chapter 12 in Term Rewriting Systems, pages 668–711. Cambridge University Press, 2003.

D. Kimura and M. Tatsuta. Dual calculus with inductive and coinductive types. In R. Treinen, editor, *Rewriting Techniques and Applications (RTA 2009), Brasília, Brazil*, volume 5595 of *Lect. Notes in Comput. Sci.*, pages 224–238. Springer, 2009. ISBN 978-3-642-02347-7.

N. R. Krishnaswami. Focusing on pattern matching. In Z. Shao and B. C. Pierce, editors, *Proc. of the 36th ACM Symp. on Principles of Programming Languages, POPL 2009*, pages 366–378. ACM Press, 2009. ISBN 978-1-60558-379-2.

D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In F. Pfenning, editor, *Proc. of the 23nd IEEE Symp. on Logic in Computer Science (LICS 2008)*, pages 241–252. IEEE Computer Soc. Press, 2008. ISBN 978-0-7695-3183-0. Long version available as Technical Report CMU-CS-08-101.

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 2009*, volume 5728 of *Lect. Notes in Comput. Sci.*, pages 113–126. Springer, 2009. ISBN 978-3-642-03740-5.

C. McBride and J. McKinna. The view from the left. *J. Func. Program.*, 14 (1):69–111, 2004.

R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, Aug. 1978.

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.

N. Oury. Coinductive types and type preservation. Message on the coq-club mailing list, June 2008.

M. Parigot. Proofs of strong normalization for second order classical natural deduction. *J. Symb. Logic*, 62(4):1461–1479, 1997.

B. C. Pierce and D. N. Turner. Local type inference. In *Proc. of the 25th ACM Symp. on Principles of Programming Languages, POPL'98*, San Diego, California, 1998.

C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lect. Notes in Comput. Sci.*, pages 120–135, Rome, Italy, September 2003. Springer.

A. Setzer. Coalgebras as types determined by their elimination rules. In P. Dybjer, S. Lindström, E. Palmgren, and G. Sundholm, editors, *Epistemology versus ontology: Essays on the foundations of mathematics in honour of Per Martin-Löf*. Springer, 2012. To appear.

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

J. Vouillon and P.-A. Melliès. Semantic types: A fresh look at the ideal model for types. In N. D. Jones and X. Leroy, editors, *Proc. of the 31st ACM Symp. on Principles of Programming Languages, POPL 2004*, pages 52–63. ACM Press, 2004. ISBN 1-58113-729-X.

P. Wadler. Call-by-value is dual to call-by-name. In C. Runciman and O. Shivers, editors, *Proc. of the 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 189–201. ACM Press, 2003. ISBN 1-58113-756-7.

P. Wadler. Call-by-value is dual to call-by-name - reloaded. In J. Giesl, editor, *Rewriting Techniques and Applications (RTA 2005), Nara, Japan*, volume 3467 of *Lect. Notes in Comput. Sci.*, pages 185–203. Springer, 2005. ISBN 3-540-25596-6.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

N. Zeilberger. Focusing and higher-order abstract syntax. In G. C. Necula and P. Wadler, editors, *Proc. of the 35th ACM Symp. on Principles of Programming Languages, POPL 2008*, pages 359–369. ACM Press, 2008a. ISBN 978-1-59593-689-9.

N. Zeilberger. On the unity of duality. *Ann. Pure Appl. Logic*, 153(1-3): 66–96, 2008b.

N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A. Agda Examples

The development version of Agda has experimental support for copatterns which can be turned on by option `--copatterns`. In the following we present a few examples for copatterns in Agda syntax.

Colists Colists have a coinductive type with an embedded variant type. In Agda this is represented as mutual recursion between a coinductive record type and a data type.

```
mutual
  data μColist (A : Set) : Set where
    []    : μColist A
    _::_  : (x : A) (xs : νColist A) → μColist A
  record νColist (A : Set) : Set where
    coinductive
    field out : μColist A
  open νColist
```

Our first function lets us append a νColist to a List. It is defined by recursion on the list.

```
open import Data.List using (List; []; _::_; map; concatMap)
append : {A : Set} → List A → νColist A → νColist A
out (append []        ys) = out ys
out (append (x :: xs) ys) = x :: append xs ys
```

Note the overloading of constructor _::_ for lists and colists.

We can also define a zipWith function for colists defined as a pair of mutually recursive functions. One is acting on νColist, the other acting on μColist.

```
mutual
  zipWith : {A B C : Set} → (A → B → C) →
            νColist A → νColist B → νColist C
  out (zipWith f xs ys) = zipWithμ f (out xs) (out ys)
  zipWithμ : {A B C : Set} → (A → B → C) →
             μColist A → μColist B → μColist C
  zipWithμ f []        ys        = []
  zipWithμ f (x :: xs) []        = []
  zipWithμ f (x :: xs) (y :: ys) = (f x y) :: (zipWith f xs ys)
```

Another example is an unfold function. Suppose we have a set of states S and a set of values A corresponding to the observation we do at some particular state. Then, given a function taking a current state and outputting a new state and its value, or no state at all if it is a terminal state, and given an initial state, we can build a colist of values of all states visited.

```
open import Data.Maybe using (Maybe; nothing; just)
open import Data.Product using (_×_; _,_)
mutual
  unfold : {A S : Set} → (S → Maybe (A × S)) → S →
           νColist A
  out (unfold f s) = unfoldμ f (f s)
  unfoldμ : {A S : Set} → (S → Maybe (A × S)) →
            Maybe (A × S) → μColist A
  unfoldμ f (just (a, s)) = a :: unfold f s
  unfoldμ f nothing       = []
```

Breadth-first traversal of non-wellfounded tree Finitely branching but potentially infinite deep trees can be represented by a coinductive record with two fields, a label and a list subs of subtrees.

```
record νTree (A : Set) : Set where
  coinductive
  field label : A
        subs  : List (νTree A)
open νTree
```

If we have a forest List (νTree A), we can extract the labels in a breadth-first manner by first taking all the roots, then concatenating all the subtrees and recurse. To ensure productivity, we distinguish the empty forest from the non-empty forest.

```
bf : {A : Set} → List (νTree A) → νColist A
out (bf [])        = []
out (bf (t :: ts)) = label t ::
                       append (map label ts)
                              (bf (concatMap subs (t :: ts)))
```

bf is productive since it is guarded-by-constructors [Coquand 1993]: it directly outputs either the empty colist or the non-empty colist, and since append xs ys only adds elements in front of ys. The latter is not yet tracked by Agda's termination and productivity checker, thus, the termination checker rejects this code. Productivity checking using sized types, as realized in MiniAgda, does work for bf, and it is our goal to bring coinduction in Agda to the same level of expressiveness as in MiniAgda.