Programming Infinite Structures using Copatterns

David Thibodeau
School of Computer Science
McGill University, Montreal

August 2015

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Abstract

Infinite structures are an integral part of computer science as they serve as representations for concepts such as constantly running devices and processes or data communication streams. Due to their importance, it is crucial that programming languages are equipped with adequate means to encode and reason about infinite structures.

This thesis investigates the recent idea of copatterns, a device to represent infinite structures in a fashion dual to usual definitions of finite data, by integrating it to Levy's Call-by-Push Value language. We define a coverage algorithm for copattern matching definitions. We prove that evaluation preserves types and that well-typed terms do not get stuck. We define a translation from our language to Levy's and prove that this translation preserves evaluation.

Résumé

Les structures infinies font partie intégrante de l'informatique puisqu'elles permettent la représentation de concepts tels que des processus ou appareils à exécution continue ou de flux de données servant à la communication entre différents appareils. En raison de leur importance, il est crucial que les langages de programmation soient capables d'adéquatement définir les structures infinies.

Ce mémoire étudie le concept de comotifs qui permettent de représenter les structures infinies d'une façon duale aux définitions usuelles de données finies. Ce concept est intégré dans une extension du langage d'appel par valeur empilée de Levy. Nous présentons un algorithme de couverture pour les définitions de filtrage par comotif. Nous faisons la démonstration que les séquences d'évaluation préservent les types et que les expressions adéquatement typées ne bloquent pas. Nous présentons aussi une traduction de notre langage vers celui originellement conçu par Levy et nous démontrons que cette traduction s'harmonise avec l'évaluation des deux langages.

Acknowledgements

I would like to thank my advisor Brigitte Pientka for her patience, her experience, her advice and her support, both financial and moral, which allowed me to complete this thesis and become a programming language researcher.

I also want to thank my coauthors Andreas Abel, Brigitte Pientka, and Anton Setzer for the development of the idea of copatterns [Abel et al., 2013] which is the basis of my thesis research. In particular, I thank Andreas Abel for hosting me in Munich and introducing and guiding my initial work on the subject.

I finally thank my colleagues Andrew Cave and Francisco Ferreira for their insights and their support throughout my masters' degree.

My masters degree has been funded by the Fonds Québécois de Recherche sur la Nature et les Technologies (FQRNT) and by McGill University.

Contents

List of Figures

Chapter 1

Introduction

At the heart of computer science lies the idea of data. It is part of the discussions of most branches of computer science, such as AI which interprets them to determine the best actions, algorithms which aims at their efficient computation, networks whose goal is to transmit them, logic which reasons about them, etc.

The problem of representing, computing and transmitting data is thus fundamental and, as programming language designers, we aim to design languages able to both adequately and efficiently solve those problems. This thesis develops a functional programming language adequate for representing, computing, and eventually reasoning about both finite and infinite structures and computations of data.

1.1 The finite and the infinite

Clearly finite structures are fundamental to computer programs as we define data such as numbers, strings, lists, etc. by smaller finite data. There is also a good understanding of finite data representations through the theory of recursive types. Data are made of constructors and smaller data assembled together. One can then split them apart by matching on the possible constructors and by analyzing recursively the smaller data. One of the easiest example of it is to define a list with the constructors nil and cons. For simplicity, we restrict

the lists to only include natural numbers. Using ML-like syntax, it would look like the following.

```
data List = nil | cons of Nat * List

let l = cons (0, cons (1, cons (2, nil)))

rec length. fun nil ⇒ 0
          | (cons (x, xs)) ⇒ 1 + (length xs)
```

The list l is a list containing $0, 1, 2$ created using constructors. The function length creates two patterns: either the list is nil, or it is the cons of a natural number x and of another list xs. For each case, there is a particular body. In the nil case, we return 0. In the other case, we return 1 + (length xs). Thus, we can easily create finite values and we use them to do some particular computation.

There is an important restriction that recursive data types have which is their need for wellfoundedness. For example, the above semantics would force any list to be of finite length. If l were an infinite list, then length l would run forever and thus prevent the program from outputting a result.

Computational entities such as servers, I/O devices, operating systems, or data streams, are not thought with respect to a starting and an end point but as a system of continuous interactions that we can always query for new computations or data. Hence, the requirement of wellfoundedness, even if those objects are finite by the physical restrictions of the world in which they exist, is an obstacle to their ideal design. On the other hand, infinite structures represent exactly the requirement of productivity that those entities have. By being infinite, we always have access to the next query.

Thus, adequate representations of infinite computation are crucial to faciliate error-free design, implementations of constantly running systems as well as correct reasoning of the results. It is therefore important to develop the proper programming constructs able to provide such representations.

1.2 Our approach

This thesis aims to investigate our recent joint work [Abel et al., 2013] which introduced the notion of copatterns, a way to mix the handling of both finite and infinite data in a symmetric and intuitive way by using the duality between the finite and the infinite.

Our investigation is done through the adaptation of this work to the language of Call-by-Push-Value by Levy [2001], a language which aims to subsume call-by-value and call-by-name semantics by making the evaluation structure explicit through the terms syntax.

1.2.1 Copatterns

Our approach is based on the duality that appears in category theory between induction, which defines finite structures, and coinduction, which encodes infinite computation. This view was pioneered by Hagino [1987a] who extended Standard ML [Hagino, 1989] with the new concept of codata types.

Let us recall that we defined the recursive datatype for list as

data List = nil | cons **of** Nat $*$ List

The constructor nil takes no argument while the constructor cons takes a pair of arguments of type Nat and List, respectively. In order, to make their presentation uniform in the number of arguments they are applied to, we introduce the type 1, called "unit type", which is inhabited by the single term (), called "unit". This type carry no information. Thus, we can say that nil takes a term of type 1 as argument and forms a term of type List. On the other hand, cons takes a term of pair type Nat $*$ List and forms a term of type List.

More generally, if we have a recursive datatype D, its constructors c_i are mappings from some types T_i to D which we can denote as $c_i : T_i \to D$. Duality demands that we reverse the direction of the arrow so a dual notion of data type, which is called a codata type, would have be series of mappings $d_i : D \to T_i$. Such mappings will be applied to our codata type to produce terms of particular types T_i. They can be viewed as observations done on the codata type. For example, a stream is defined as follows.

codata Stream = head : Nat & tail : Stream

Thus, head : Stream → Nat and tail : Stream → Stream. In order to define a term of type Stream, we need to provide a term for each observation. Hagino defines a construct merge for it. Such construct defines a term u of codata type which pairs each observation d_i with a term m_i. Applying an observation d_i to u will return m_i. In this sense, codata types are essentially recursive records. If we want to define the stream of natural numbers, we would create the following function.

```
rec nats. fun n = merge head <= n
                      & tail <= nats (n+1)
```

Our recent joint work [Abel et al., 2013] extends this idea into a system which mixes arbitrarily pattern matching and observation definition into copattern matching and allows arbitrary mixing of function abstraction and merge construct using the copattern abstraction denoted simply by **fun**.

Examples of copatterns

In order to make understand how copatterns can be used, we show several of examples of programs using copatterns adapted from Abel et al. [2013].

Example 1 (Mixing matching on recursive and corecursive definitions). Suppose we want a function starting at some natural number x and creating a decreasing stream from x to 0. When it reaches 0, it goes back to some constant, say 5, and continues decreasing towards 0. If $x = 4$, the resulting stream would look like the following.

$$4\ 3\ 2\ 1\ 0\ 5\ 4\ 3\ 2\ 1\ 0\ 5\ 4\ 3\ 2\ 1\ 0\ \ldots$$

This function thus creates a corecursive object (a stream) based on a recursive object (a natural number). It thus requires us to be able to mix case analysis and copattern matching which can easily be done in our system. It is defined as follows.

```
rec cycleNats. fun x .head ⇒ x
             | 0      .tail ⇒ cycleNats 5
             | (s x) .tail ⇒ cycleNats x
```

Observations are record projections and are denoted in postfix notations as $.d$. In the case of streams, we have .head and .tail. If we look at the head of stream, we get the current value. If we look at the tail of the stream, then we simply have the stream starting at the next number. If the current number is 0, then the next number is 5. If the current number is the successor of some other number x, then the next number is x.

Example 2 (Deep copatterns). Copatterns allow us to require multiple observations to be applied before triggering a term rewriting rule. For example, a Haskell definition of the Fibonacci stream is the following.

```
fib = cons 0 (cons 1 (zipWith _+_ fib (tail fib)))
```

where zipWith _+_ perfoms a pointwise addition of the two streams. In such definition, an eager evaluation of the stream would result into unfolding tail fib into cons 1 (zipWith _+_ fib (tail fib)) which would again unfold tail fib and lead to an infinite unfolding and the lost of normalisation. For this reason, some systems will label the definition as corecursive and then have their semantics employ guardedness conditions on such definitions. (see Section 1.3.2 for some examples of guardedness conditions).

Using copatterns, we achieve normalisation simply from the semantics of copatterns without having to define such guardedness principles. The definition of the Fibonacci stream (and of the zipWith function) is as follows.

```
rec zipWith. fun f s1 s2 .head ⇒ f (s1.head) (s2.head)
               | f s1 s2 .tail  ⇒ zipWith f (s1.tail) (s2.tail)

rec fib. fun .head      ⇒ 0
         |  .tail .head ⇒ 1
         |  .tail .tail  ⇒ zipWith _+_ fib (fib.tail)
```

A copattern definition with deep copattern matching will only step when matched against all the observations. Hence, fib.tail does not trigger any reduction until we apply another observation (.head or .tail) to it, thus preserving normalisation.

Example 3 (Symmetrical behaviour of copatterns). Copatterns do not have to involve any corecursion for them to be useful. For example, the state monad as often used in Haskell

can be simply defined using type synonyms by setting State $S\ A = S \to (A \times S)$. The usual bind and return functions thus are the following.

rec return.**fun** a s ⇒ (a, s)
rec bind.**fun** m k s ⇒ **let** (a, s') = m s **in** k a s'

where return has type $A \to$ State $S\ A$ and Bind has type State $S\ A \to (A \to$ State $SB) \to$ State $S\ B$.

However, type synonyms interact badly with type-class based overloading so we usually would rather implement the state monad using a record type which has a projection runState and a constructor state.

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

This gives us the following return and bind functions.

rec return.**fun** a ⇒ state(λs. (a, s'))
rec bind.**fun** m k ⇒ state(λs. **let** (a, s') = m.runState s **in** (k a).runState s')

This view is inconvenient as since we enclose the state monad in a record, we need to use the constructor state and thus we push the binding of s inside which destroys the simplicity of the original representation. Here, copatterns can allow us to restore this missing piece. The copattern definition is thus the following. We note that while we define a polymorphic definition of State, we do not treat polymorphism as part of CBPV$^{\text{copat}}$.

codata State S A = runState : S → A ∗ S

rec return.**fun** a .runState s ⇒ (a, s)
rec bind.**fun** m k .runState s ⇒ **let** (a, s') = m.runState s **in** (k a).runState s'

What we notice is that by using copattern matching, we define directly what is the body of a record without having to use a constructor such as state which allows us to keep the binding for s to the left-hand side rather than to have to move it to the right-hand side.

Example 4 (Mixing data and codata types). Our previous examples were only defined using codata types. We can mix data and codata types to build other interesting examples. Let us define possibly infinite lists or colists:

```
codata Colist = out : Colist'
and data Colist' = clnil | clcons of Nat * Colist
```

In this definition, the observation out serves as a delay mechanism placed on the usual list definition. Upon unfolding, we are able to know whether the colist is empty or contains an element. We can extract a finite number of elements as follows.

```
rec take. fun zero c ⇒ nil
        | (succ n) c ⇒ case c.out of
                       | clnil ⇒ nil
                       | (clcons (x, xs)) ⇒ cons (x, take n xs)
```

Our function reads up to n elements of the colist and creates a list with those elements. At each step, we need to unfold our colist to be able to pattern match on the Colist' term. We can also write the dual function which creates a colist out of a list.

```
rec make. fun nil .out ⇒ clnil
          | (cons (x, xs)) .out ⇒ clcons (x, make xs)
```

Alternatively, we can make a more elaborate use of copatterns by defining Colist as follows

```
data Colist = clnil | clnext of Colist'
and codata Colist' = head : Nat & tail : Colist
```

The two definitions differ in some subtle aspects. The latter gives us right away the information whether the list is empty or not. The former only provides us with this information when we actually make the observation. This difference can change the actual semantics of the program when we deal with effects. However, this thesis doesn't take effects into consideration. We thus can consider those two definitions as equivalent.

1.2.2 Call-by-Push value (CBPV)

This language aims at subsuming both the call-by-value and call-by-name semantics by making explicit the evaluation order using a separation of terms between values and computations, which resembles focusing calculi (see Section 1.3.5). Levy insists that it is the treatment of what is bound to variables that makes the difference between both semantics.

He restricts the CBPV variables to be of positive, or value, types. The embedding of negative, or computation, types must be done through a thunking process that prevents the evaluation of the computation until it is forced. Programming in CBPV using call-by-value semantics will simply produce values out of computations before proceeding to the binding while call-by-name semantics will thunk the computation and bind it.

In his thesis, Levy also introduces recursive types which, due to the separation of values and computations, allows for a dual representation of both recursive and corecursive data types behaving in the same way copatterns do. Our extension to Levy's work thus extends this views with a notion of deep copattern matching rather than simple case splitting and unfolding.

1.2.3 Contributions

This thesis presents the CBPV$^{\text{copat}}$ language which extends Levy's CBPV with copatterns (Section 2.1). We define a continuation-based small-step operational semantics which allows for both call-by-value and call-by-name semantics (Section 2.2). We prove a substitution lemma, adequacy of (co)pattern matching and type preservation.

We describe a non-deterministic coverage algorithm which allows us to define covering copatterns through successive splitting in a way similar to Agda's interactive mode [Norell, 2007] (Section 2.2.6). We prove a progress theorem through a safety predicate defined recursively on the type of the terms (Section 2.2.7).

We describe Levy's CBPV and provide a translation from CBPV$^{\text{copat}}$ to Levy's CBPV (Chapter 3). Such translation can serve as a first step towards a compilation process for CBPV$^{\text{copat}}$ where CBPV is used as an intermediate language. We prove that evaluation is preserved by the translation (Section 3.3) and so well-typed programs behave the same whether they are translated or not.

1.3 Related work

In order to make more sense of the motivation to investigate and build upon this dual approach, we look at the other developments in terms of infinite structures and coinduction.

1.3.1 General purpose functional programming languages

ML languages [Milner et al., 1997] have a call-by-value operational semantics but they don't evaluate under function abstraction. The evaluation of an expression thus can be delayed by suspending it into a dummy function. Then, it can be forced by applying the function to a dummy expression.

Haskell [Peyton-Jones, 2003] is a call-by-need language which treats any data type lazily. It can force eagerness through a `seq` construct which stands for sequential computation.

While both languages allow for a way to handle both finite and infinite data, our language offers a principled way to handle both in a symmetric fashion. In particular, copatterns can easily be integrated to an ML language by simply extending the fun-construct to copatterns and introducing codata type definitions.

1.3.2 Dependently typed functional languages

While our language is restricted to a simply typed setting, our long term goal is to be able to extend copatterns to dependent types to provide a suitable foundation to proofs by coinduction. We thus compare our approach with two of the most popular dependently typed programming languages: Coq and Agda.

The proof assistant Coq [The Coq development team, 2004] defines infinite structures as coinductive definitions using constructors in a similar fashion as how they define the finite ones [Chlipala, 2011]. However, in order to prevent infinite terms from unfolding forever, they are only unfolded in a case statement (denoted as `match with` in Coq). Moreover, recursive calls on coinductive arguments are done directly under a coinductive constructor. This expression can only be nested inside a case statement, a function abstraction or other constructors.

Borrowing Chlipala's example, a stream is defined as

coinductive Stream = cons **of** Nat $*$ Stream

We then define a term as a cofixed point. The stream of zeroes is defined as

let zeroes = cofix (cons 0)

where cofix indicates that cons 0 is applied repetitively. The term cofix (cons 0) doesn't unfold by itself. The only reduction rules are

$$\begin{aligned} \mathsf{case}\ (\mathsf{cofix}\ (\mathsf{cons}\ 0))\ \mathsf{of}\ \mathsf{cons}\ x\ xs \Rightarrow t &= \mathsf{case}\ (\mathsf{cons}\ 0\ (\mathsf{cofix}\ (\mathsf{cons}\ 0)))\ \mathsf{of}\ \mathsf{cons}\ x\ xs \Rightarrow t \\ \mathsf{case}\ (\mathsf{cons}\ l\ ls)\ \mathsf{of}\ \mathsf{cons}\ x\ xs \Rightarrow t &= [l/x, ls/xs]t \end{aligned}$$

which requires a case statement to trigger.

This solution thus makes a clear distinction between the finite and infinite terms in easily usuable constructs. However, Giménez [1996] showed that this breaks subject reduction, that is, types are not preserved by the evaluation. This example has then been made popular by Oury [2008]. To see this, we define a type

coinductive U = **in of** U

The only inhabitant of U is $u = \mathsf{cofix\ in}$. It is possible to prove within Coq that $\mathsf{eq}_u : u \equiv \mathsf{in}\ u$, that is, u is definitionally equal to $\mathsf{in}\ u$. However, definitional equality only allows the rule $\mathsf{refl} : t \equiv t$ for any t. Since u only unfolds into $\mathsf{in}\ u$ when under a case statement, they are not equal and the proof eq_u breaks subject reduction.

Let us now construct the term eq u.

rec force. **fun** x $\Rightarrow$ **case** x **of in** y $\Rightarrow$ **in** y

rec eq. **fun** x = **case** x **of in** y $\Rightarrow$ refl

In order for those functions to make sense, we need to mention that the type of eq is $(x : U) \to x \equiv \mathsf{force}\ x$. Those two functions are necessary since their case-statements will force the unfolding, that is,

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

Then, $\mathsf{eq}_u = \mathsf{eq}\ u$.

The reason such behaviour occurs is that Coq uses the following dependent matching rule.

$$\frac{\Gamma \vdash t_1 : U \quad \Gamma, y : U \vdash t : C(\mathsf{in}\ y)}{\Gamma \vdash \mathsf{case}\ t_1\ \mathsf{of\ in}\ y \Rightarrow t : C(t_1)}$$

The term t has type $C(\mathsf{in}\ y)$ while the whole $\mathsf{case}\ t_1\ \mathsf{of\ in}\ y \Rightarrow t$ has type $C(t_1)$. Looking back at the program eq, we see that refl has type $\mathsf{in}\ y \equiv \mathsf{force}\ (\mathsf{in}\ y)$ which holds since $\mathsf{force}\ (\mathsf{in}\ y)$ reduces to $\mathsf{in}\ y$. Thus, we can construct the term $\mathsf{case}\ x\ \mathsf{of\ in}\ y \Rightarrow t$ of type $x \equiv \mathsf{force}\ x$ even if $\mathsf{force}\ x$ does not reduce to x. It follows that we have $u \equiv \mathsf{force}\ u$ and $\mathsf{force}\ u = \mathsf{in}\ u$.

Our solution, by defining infinite objects via observations rather than constructors, brings back subject reduction, as we prove it in Section 2.2.5. However, one could argue that our setting simply doesn't allow dependent pattern matching and so surely wouldn't fail where Coq does. Looking back at Oury's example from a copattern point of view would bring us to define a single observation $\mathsf{out} : U \to U$ such that $(\mathbf{fun}\ (\mathsf{out} \Rightarrow y)).\mathsf{out} = y$ and so $u.\mathsf{out} = u$. We don't need to consider restrictions on unfolding for cofix. Adding dependent pattern matching would not affect coinductive types and so a dependently typed version based on copatterns would also preserve subjection reduction.

The proof assistant Agda [Norell, 2007] uses explicit terms for delay, denoted $\sharp$, and force, denoted $\flat$, to introduce infinite computations, whose types are denoted by using the prefix ∞. They don't allow dependent pattern matching on delayed computations so they preserve subject reduction.

Their current approach is not completely general in the way induction and coinduction is mixed. It was noted by Altenkirch and Danielsson [2009] that we can only encode the property "infinitely often" but not its dual "eventually forever". To make that more explicit, we consider a tree which is infinitely branching, but each path starting from the root has to be finite. This tree is infinitely wide but finitely deep. It is defined in Agda as the following.

data Colist A = nil | cons **of** A $*$ ∞(Colist A)
and Tree = node **of** Colist Tree

Such data type can be represented in terms of fixed point notation. A recursive data type is a least fixed point, denoted by $\mu X.A$, while a corecursive data type (or delayed data type, in

terms of Agda) is a greatest fixed point, denoted by $\nu X.A$, where A is a type depending on the variable X which stands for a recursive occurence. The type A represents the type of the constructors. If there are several constructors, we have a sum of types ($A_1 + A_2$ in the case of two constructors). Each type in the sum is the type of the argument to the constructor. We recall that a constructor without an argument takes in fact an argument of type 1. Mutually recursive definitions are nested fixed points. Hence, the fixed point representation of the data type above is $\mu X.\nu Y.1 + X \times Y$. Since we consider trees, the least fixed point is on the outside. As least fixed points are wellfounded, a term of such type can only unfold the X finitely many times, leading to a finitely deep tree.

However, Agda admits the following term

```
rec bad. node (cons (node nil, (♯ bads))))
and bads. cons (bad, ♯bads)
```

This tree represents an infinitely wide and deep tree which the semantics for inductive data type should forbid and they represent a wellfounded structure. The problem lies in the fact that Agda's productivity checker interprets such definition as $\nu Y.\mu X.1 + X \times Y$. Now, since νY is on the outside, there can only be finitely many unfolding of X before an unfolding of Y. However, since νY is a greatest fixed point, it doesn't have a wellfoundedness requirement. Thus, there can be infinitely many unfolding of Y which leads to infinitely many possible unfoldings of X.

The above fixed point can be expressed in terms of Agda code as the following.

```
data List A = nil | cons of A ∗ (List A)
and Tree' = node of List (∞ Tree')
```

In Agda, both definitions behave the same.

The presentation in this thesis does not differentiate between those two definitions. However, Abel and Pientka [2013] defined a type based productivity checker for copatterns which uses size annotations in types to ensure well-foundedness of recursive types is preserved even under a copattern abstraction. Our language thus could easily be extended to such type based productivity checker and avoid such problem altogether.

1.3.3 Object-oriented programming

Cook [1991] compares abstract data types *à la* ML with object-oriented programming by noting that objects form a data abstraction through the idea of procedural abstraction. He explains that objects are records of attributes and methods form the observations on the data. He adds that "a procedural data value is simply defined by the combination of all possible observations upon it" which is a also a very adequate description of our codata type definitions.

In addition, Jacobs [1995] shows that objects identify with terminal coalgebras in categorical models. Hagino [1989] proved that his codata types also identify with coalgebras. Since our work extends Hagino's, there is a clear link between copatterns and objects and our language is thus a step towards bridging the gap between languages with recursive data type definitions and the ones following the object-oriented paradigm.

1.3.4 Categorical programming languages

Copatterns are built upon the idea that finite data types correspond to initial algebras and infinite data types correspond to terminal coalgebras which was first brought to light by Hagino [1987a] and was used to define categorical programming languages such as in Hagino [1987b] or Charity [Cockett and Fukushima, 1992]. Such languages program directly with the morphisms of category theory which allows defining codata types. They typically do not support general recursion or pattern matching. The latter has been added to Charity later on [Tuckey, 1997] but Charity still does not support a form of copattern matching.

1.3.5 Focusing

Levy's type separation between values and computations bears resemblance with focusing calculi which has been used for proofs [Andreoli, 1992], pattern matching [Zeilberger, 2008, Krishnaswami, 2009], and evaluation order [Zeilberger, 2009, Curien and Herbelin, 2000]. However, Krishnaswami [2014] notes that focusing and CBPV differ in the treatment of their contexts. In the case of focusing, negative assumptions inhabit contexts while CBPV

admits positive ones instead. To our knowledge, no research has been done to explicitely compare how this distinction impacts the languages.

Licata et al. [2008] present a sequent calculus style language mixing LF [Harper et al., 1993] style encoding with computational-level types. They define a concept of copatterns (named destructor patterns) on the meta-level which restricts induction to ω-rule style definitions. They introduce corecursive definitions in their technical report version. Our work presents copatterns in a natural deduction style calculus which can easily be integrated into existing languages. We also defined a coverage algorithm for copatterns which is not addressed in their development.

Kimura and Tatsuta [2009] define a dual calculus in sequent calculus form with inductive and coinductive definitions. Their system is split between terms and continuations in a fashion similar to our type separation. Inductive definitions are term constructors while coinductive definitions are continuation destructors. However, they only proceed by (co)iteration rather than having (co)pattern matching. Their evaluation is done on statements made from pairs of terms and continuations which recall our evaluation on configurations. However, their continuations are built on the outside and are part of the syntax rather than being accumulations from the original term which only serve to store branching information.

Chapter 2

The language: CBPV$^{\text{copat}}$

Call-by-push-value as described by Levy [2001] is a language making explicit the evaluation sequence through the structure of the language. The power this structure brings allows us to define programs that obey call-by-value or call-by-name semantics (or a hybrid of the two) without having to modify the rules. This gives us a way to define both eager and lazy evaluation which is usually linked with recursive and corecursive definitions, respectively (as it is observed in ML languages versus Haskell).

This chapter presents a reformulation of Levy's call-by-push-value language using copatterns. While Levy's CBPV defines a different term construct for each of recursive computation, record and function definitions, case analysis of pairs, recursive values and sums, our language, CBPV$^{\text{copat}}$, mixes all those constructs into one: the copattern abstraction which makes use of deep copattern matching.

Moreover, our treatment of CBPV, both in this chapter and in the next one includes a **rec** construct which allows us to define term recursion which was not addressed in Levy's work outside of recursive types. We also choose to go from an indexed definition of sums and records to a definition using a list of labels. This distinction is, however, mostly syntactic.

We present the syntax of CBPV$^{\text{copat}}$ in Section 2.1 and the operational semantics in Section 2.2. We prove that the operational semantics preserves types in Section 2.2.5 and that well-typed terms do not get stuck in Section 2.2.7.

2.1 Syntax

2.1.1 Types

The types are separated between positive and negative ones. They are listed in Figure 2.1. Positive types P denote types of values including products $P_1 \times P_2$, the unit type 1, data types $\mu X.D$, and embeddings of negative types $\downarrow N$. Positive types are always in normal form and represent the information produced by the program.

Negative types N denote functions $P \to N$, codata types $\nu Y.R$ and embeddings of positive types $\uparrow P$. They represent computations. The evaluation of a program will affect terms of negative types.

P	$::=$	X	Positive type variable
	$\mid$	$P_1 \times P_2$	Product type
	$\mid$	$\mu X.D$	Data type
	$\mid$	1	Unit type
	$\mid$	$\downarrow N$	Embedding negative type
N	$::=$	Y	Negative type variable
	$\mid$	$P \to N$	Function type
	$\mid$	$\nu Y.R$	Codata type
	$\mid$	$\uparrow P$	Embedding positive type
D	$::=$	$\langle c_1\ P_1 \mid \cdots \mid c_n\ P_n \rangle$	Variant (labeled sum)
R	$::=$	$\{d_1 : N_1\ \&\ \ldots\ \&\ d_n : N_n\}$	Record (labeled product)

Figure 2.1: Types

Type variables X and Y are used only in recursive and corecursive types as our system does not have polymorphism. Variants are labelled recursive sums of positive types identified

using constructors. Records are labelled recursive lazy products of negative types identified using observations. The use of variants and records is restricted to data and codata types. However, if the variable does not appear in the body of the data or the codata type, then they become regular sums and (lazy) records.

For example, data types allow us to define natural numbers and lists but also non recursive sum types such as booleans or option types.

$$\begin{array}{lcl} \text{Nat} & = & \mu X.\, \langle \text{zero } 1 \mid \text{succ } X \rangle \\ \text{List} & = & \mu X.\, \langle \text{nil } 1 \mid \text{cons Nat} \times X \rangle \\ \text{Bool} & = & \mu \_.\, \langle \text{true } 1 \mid \text{false } 1 \rangle \\ \text{Option} & = & \mu \_.\, \langle \text{none } 1 \mid \text{some Nat} \rangle \end{array}$$

On the other hand, codata types allow us to define corecursive types such as streams, but also define record types such as vectors. It is also possible to mix them arbitrarily as the lazy list example shows.

$$\begin{array}{lcl} \text{Stream} & = & \nu Y.\, \{\text{head} \ : \uparrow \text{Nat} \ \& \ \text{tail} \ : \text{Stream}\} \\ \text{Vector} & = & \nu Y.\, \{\text{list} \ : \uparrow \text{List} \ \& \ \text{length} \ : \uparrow \text{Nat}\} \\ \text{LazyList} & = & \nu Y.\, \{\text{out} : \uparrow \mu \_.\, \langle \text{nil } 1 \mid \text{cons Nat} \times \downarrow Y \rangle\} \end{array}$$

It is important to note that Vector does not provide any guarantee that the length parameter is the size of the list given by the list observation as the system does not have any form of dependent types. This is merely a pair of a list and a natural number that carries a convention of usage.

2.1.2 Terms

Following the separation between positive and negative types, terms are separated between values and computations. The grammar is defined in Figure 2.2. Values are made of variables, denoted x, unit, represented as $()$, pairs of values, written (v_1, v_2), constructors applied to values, denoted $c\ v$, and thunks of computations, written **thunk** m. They are associated to positive types and thus form the data a program carries.

Computations are made of produced values, written **produce** v, forcing of values, written **force** v, applications of values, denoted $m\ v$, destructor applications, denoted $m.d$, chaining of computations, written m_1 **to** $x.m_2$, recursive computations, defined as **rec** $x.m$, and copattern abstractions, written **fun** $\vec{u}$. They encode the flow of the program and determine how the evaluation is performed.

There is an important difference in produce and force. Produce embeds a value into a computation, which can then be used in a to-statement. Force is applied to a thunk of a computation, forcing the computation to evaluate. To-statements evaluate the left-hand side computation into a produced value, then bind the value to the bound variable and continue evaluating the right-hand side computation under the substitution for the variable.

Copattern abstractions are lists of pairs of copatterns and computational terms ($q \mapsto m$). The copattern q binds new variables used in the term m. Copatterns are simply lists of patterns ($p\ q$) and destructors ($.d\ q$). A pattern is made of variables (x), pairs (p_1, p_2) and constructors applications ($c\ p$).

Example 5. The ML program

```
let l = [0, 1] in
   let fun length nil = 0
         | length x :: xs = 1 + (length xs)
   in
      length l
   end
end
```

would be translated as

```
produce
  (cons (zero (), cons (suc (zero ()), nil ())))
to l. produce
   (thunk (rec length.
             fun (nil ()) ⇒ produce (zero ())
               | (cons (x, xs)) ⇒ ((force length) xs) to y. produce (suc y)))
to length.
  (force length) l
```

Values	v	::=	x	Variable
		|	$()$	Unit
		|	(v_1, v_2)	Pair
		|	$c\ v$	Constructor application
		|	**thunk** m	Thunk of a computation
Computations	m	::=	**produce** v	Production of a value
		|	**force** v	Forcing of a thunk of a computation
		|	$m\ v$	Application of a value to a computation
		|	$m.d$	Destructor application
		|	m_1 **to** $x.m_2$	Chaining of computations
		|	**rec** $x.m$	Recursive computation
		|	**fun** $\vec{u}$	Copattern abstraction
Copattern definition	u	::=	$q \mapsto m$	
Copattern	q	::=	$\cdot$	Empty copattern
		|	$.d\ q$	Destructor copattern
		|	$p\ q$	Application copattern
Pattern	p	::=	x	Variable pattern
		|	(p_1, p_2)	Pair pattern
		|	$c\ p$	Constructor pattern

Figure 2.2: Terms

We note that unlike Levy's who divides let-bindings between **let** x be v **in** m and m **to** x.n to separate when we are binding a value to a variable and when we bind the value resulting from a computation in the variable, respectively, we merge both statements into the **to**-statement. To bind a value v to a variable x in the term m, it suffices to write the expression **produce** v **to** $x.m$.

Other translations of the above program are possible if we interpret it in a semantics other than call-by-value as our system, just like Levy's subsumes both semantics. The let-in-statements are translated into to-statements. The term that is bound to the variable is a produced value. If it is translated to a value term, it is written directly. Otherwise, it is wrapped in a thunk. A function is split into a rec-statement handling the recursion and a copattern abstraction doing the matching. The cons case lifts the recursive call into a to-statement as if the above program would have had **let** y = length xs **in** 1 + y. Since a variable is a value, we need to force the computation thunked inside it. Thus, we **force** length. We will make those choices more precise when defining the typing rules and the operational semantics.

Example 6. The cycleNats function is translated as follows.

```
rec cycleNats. fun n    .head ⇒ produce n
               | (zero ()) .tail ⇒ (force cycleNats) 5
               | (suc m) .tail ⇒ (force cycleNats) m
```

As we pointed out above, a copattern abstraction takes pairs of copatterns and computations. The copatterns here have a pattern together with a destructor. When matching (which is presented in Section 2.2), we will consume from left to right thus we have a function taking a natural number and outputting a stream. This stream is then defined through its head and tail. We note that the tail case is split based on the natural number argument. The splitting can be done in arbitrary order and is discussed in Section 2.2.6 when introducing coverage.

$\boxed{\Gamma \vdash v : P}$ Value v has type P in context Γ.

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P} \quad \frac{}{\Gamma \vdash () : 1} \quad \frac{\Gamma \vdash v_1 : P_1 \quad \Gamma \vdash v_2 : P_2}{\Gamma \vdash (v_1, v_2) : P_1 \times P_2} \quad \frac{\Gamma \vdash m : N}{\Gamma \vdash \textsf{thunk}\ m : \downarrow N} \quad \frac{\Gamma \vdash v : [\mu X.D/X]D_c}{\Gamma \vdash c\ v : \mu X.D}$$

$\boxed{\Gamma \vdash m : N}$ Computation m has type N in context Γ.

$$\frac{\Gamma \vdash v : P}{\Gamma \vdash \textsf{produce}\ v : \uparrow P} \quad \frac{\Gamma \vdash v : \downarrow N}{\Gamma \vdash \textsf{force}\ v : N} \quad \frac{\Gamma \vdash m : P \to N \quad \Gamma \vdash v : P}{\Gamma \vdash m\ v : N} \quad \frac{\Gamma, f : \downarrow N \vdash m : N}{\Gamma \vdash \textsf{rec}\ f.m : N}$$

$$\frac{\Gamma \vdash m : \nu Y.R}{\Gamma \vdash m.d : [\nu Y.R/Y]R_d} \quad \frac{\Gamma \vdash m_1 : \uparrow P \quad \Gamma, x : P \vdash m_2 : N}{\Gamma \vdash m_1\ \textsf{to}\ x.m_2 : N} \quad \frac{\text{for each } i\ \Gamma \vdash u_i : N}{\Gamma \vdash \textsf{fun}\ \vec{u} : N}$$

$\boxed{\Gamma \vdash u_i : N}$ Branch u_i has type N in context Γ.

$$\frac{\Gamma_i \vdash_{N_i} q_i : N \quad \Gamma, \Gamma_i \vdash m_i : N_i}{\Gamma \vdash (q_i \mapsto m_i) : N}$$

Figure 2.3: Typing rules

2.1.3 Typing rules

The typing rules for terms appear in Figure 2.3 while the ones for (co)patterns are presented in Figure 2.4.

In terms of values, all the constructs are in one to one correspondance with the positive types, to which we add variables. We note that D_c means the type P in D associated with constructor c. Thus, if $D = \langle c_1\ P_1 \mid \cdots \mid c_n\ P_n \rangle$, then $D_{c_i} = P_i$. The same reasoning applies for R_d.

The types make clearer the distinction between produce and force. The former creates a term of type $\uparrow P$ while the second eliminates a term of type $\downarrow N$ into a term of type N. Left-hand sides of to-statements will always evaluate into a produce statement whose value will be passed to the bound variable.

Typing rules for patterns are the same as the typing rules for the corresponding terms. The judgement for typing of copatterns is the following $\Gamma \vdash_N q : N'$ and means that q eliminates type N' into type N and returns both N and the context Γ. Because copatterns are in elimination form, their typing rules appear dual to the ones for terms. For example, the application copattern $v\ q$ is of type $P \to N$ while the regular application $m\ v$ is of type N where its left-hand side m is of type $P \to N$.

Copattern abstractions then introduce pairs of a copattern q_i and body m_i. A copattern abstraction has a type which is eliminated by each of the copatterns q_i into a new type N_i, creating a context Γ_i and we check that the body m_i has type N_i in the context extended with Γ_i.

$\boxed{\Gamma \vdash p : P}$ Pattern p of type P generates context Γ.

$$\frac{}{x : P \vdash x : P} \quad \frac{\Gamma_1 \vdash p_1 : P_1 \quad \Gamma_2 \vdash p_2 : P_2}{\Gamma_1, \Gamma_2 \vdash (p_1, p_2) : P_1 \times P_2} \quad \frac{\Gamma \vdash p : [\mu X.D/X]D_c}{\Gamma \vdash c\ p : \mu X.D}$$

$\boxed{\Gamma \vdash_N q : N'}$ Copattern q eliminates type N' into type N and creates context Γ.

$$\frac{\Gamma \vdash_N q : [\nu Y.R/Y]R_d}{\Gamma \vdash_N .d\ q : \nu Y.R} \quad \frac{\Gamma_1 \vdash_N q : N' \quad \Gamma_2 \vdash p : P}{\Gamma_1, \Gamma_2 \vdash_N p\ q : P \to N'} \quad \frac{}{\vdash_N \cdot : N}$$

Figure 2.4: Typing rules for (co)patterns

Example 7. Let us go back to our length example. Since the derivation tree is very large, we will not develop all of it. The most interesting part is the copattern abstraction

```
fun (nil ()) ⇒ produce (zero ())
  | (cons (x, xs)) ⇒ (force length xs) to y. produce (suc y)))
```

Each branch needs to eliminate the type List → ↑Nat. The first branch is

```
(nil x) ⇒ produce (zero ())
```

in the empty context. The outer context is $l : \text{List}, \text{length} : \downarrow(\text{List} \to \uparrow\text{Nat})$ but we denote it as Γ for space reasons. The derivation is the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : 1 \vdash x : 1}}{x : 1 \vdash \text{nil}\ x : \text{List}} \quad \dfrac{}{\cdot \vdash_{\uparrow\text{Nat}} \cdot : \uparrow\text{Nat}}}{x : 1 \vdash_{\uparrow\text{Nat}} (\text{nil}\ x) \cdot : \text{List} \to \uparrow\text{Nat}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma, x : 1 \vdash () : 1}}{\Gamma, x : 1 \vdash \text{zero}\ () : \text{Nat}}}{\Gamma, x : 1 \vdash \textbf{produce}\ (\text{zero}\ ()) : \uparrow\text{Nat}}}{\Gamma \vdash x : 1.(\text{nil}\ x) \cdot \mapsto \textbf{produce}\ \ (\text{zero}\ ()) : \text{List} \to \uparrow\text{Nat}}$$

The second branch is

(cons (x, xs)) ⇒ (**force** length xs) **to** y. **produce** (suc y)))

The derivation for the copattern side is the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \text{Nat} \vdash x : \text{Nat}} \quad \dfrac{}{xs : \text{List} \vdash xs : \text{List}}}{x : \text{Nat}, xs : \text{List} \vdash (x, xs) : \text{Nat} \times \text{List}}}{x : \text{Nat}, xs : \text{List} \vdash \text{cons}\ (x, xs) : \text{List}} \quad \dfrac{}{\cdot \vdash_{\uparrow\text{Nat}} \cdot : \uparrow\text{Nat}}}{x : \text{Nat}, xs : \text{List} \vdash_{\uparrow\text{Nat}} (\text{cons}\ (x, xs)) \cdot : \text{List} \to \uparrow\text{Nat}}$$

On the term side, we use Γ' to denote the context "$l : \text{List}, \text{length} : \downarrow(\text{List} \to \uparrow\text{Nat})$, $x : \text{Nat}, xs : \text{List}$". The derivation is described below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'(\text{length}) = \downarrow(\text{List} \to \uparrow\text{Nat})}{\Gamma' \vdash \text{length} : \downarrow(\text{List} \to \uparrow\text{Nat})}}{\Gamma' \vdash \textbf{force}\ \text{length} : \text{List} \to \uparrow\text{Nat}} \quad \dfrac{\Gamma'(xs) = \text{List}}{\Gamma' \vdash xs : \text{List}}}{\Gamma' \vdash \textbf{force}\ \text{length}\ xs : \uparrow\text{Nat}} \quad \dfrac{\dfrac{\dfrac{\Gamma'(y) = \text{Nat}}{\Gamma', y : \text{Nat} \vdash y : \text{Nat}}}{\Gamma', y : \text{Nat} \vdash \text{suc}\ y : \text{Nat}}}{\Gamma', y : \text{Nat} \vdash \textbf{produce}\ (\text{suc}\ y) : \uparrow\text{Nat}}}{\Gamma' \vdash \textbf{force}\ \text{length}\ xs\ \textbf{to}\ y.\textbf{produce}\ (\text{suc}\ y) : \uparrow\text{Nat}}$$

This concludes the typing derivation.

2.2 Operational Semantics

This section defines a small-steps operational semantics which uses evaluation contexts as continuations to accumulate branching informations. We also define (co)pattern matching and substitutions and prove subject reduction. The second part of this section introduces a non deterministic coverage algorithm in order to prove progress.

$$\frac{}{\vdash_N nil : N} \quad \frac{\vdash_N K : [\nu Y.R/Y]R_d}{\vdash_N .d\ K : \nu Y.R} \quad \frac{\vdash v : P \quad \vdash_N K : N'}{\vdash_N v\ K : P \to N'} \quad \frac{x : P \vdash n : N' \quad \vdash_N K : N'}{\vdash_N ([]\ \textbf{to}\ x.n)\ K : \uparrow P}$$

Figure 2.5: Evaluation Contexts

2.2.1 Evaluation contexts

The evaluation contexts are defined as follows.

$K ::= ([]\ \textbf{to}\ x.n)\ K \mid .d\ K \mid v\ K \mid nil$

When we evaluate a to-statement, we first evaluate the left hand side, then evaluate the right hand side under the substitution for x. Thus, we delay the evaluation of the right hand side by adding it to the evaluation context. When dealing with an application or a destructor application, we need to evaluate the left side term until we obtain a copattern abstraction. Then, we consume as many observations or values needed to have a matching copattern and we proceed to the evaluation of the body. Thus, we need to accumulate all the right hand terms of the applications.

The typing for evaluation contexts is defined through the judgement $\vdash_N K : N'$ that we can read as "evaluation context K eliminates the type N' into N in the empty context". The rules appear in Figure 2.5. Evaluation contexts always live in the empty context since they represent concrete terms and we do not traverse variable bindings during evaluation, but rather simply substitute concrete terms for those variables. Their typing rules follow the ones of copatterns because we are matching copatterns and evaluation contexts.

Our operational semantics acts on configurations $m; K$ which are typed using the rule

$$\frac{\vdash m : N' \quad \vdash_N K : N'}{\vdash m; K : N}$$

A configuration thus has type N if m has type N' and K eliminates the type N' into N. As evaluation is done on configurations, subject reduction requires us to type configurations rather than terms.

2.2.2 Substitutions

A substitution σ is a mapping of variables in a context Γ to values.

$$\sigma ::= \cdot \mid \sigma, v/x$$

If values live in a context Δ, we say that σ is a mapping from context Γ to Δ, denoted $\Delta \vdash \sigma : \Gamma$. It obeys the following typing rules.

$$\frac{\Delta \vdash v : P \quad \Delta \vdash \sigma : \Gamma}{\Delta \vdash \sigma, v/x : \Gamma, x : P} \qquad \frac{}{\Delta \vdash \cdot : \cdot}$$

We consider the single variable substitution v/x as the special case $\cdot, v/x$. While our presentation gives the idea of a particular ordering on both contexts Γ and substitutions σ, we treat them as unordered sets and thus admit any implicit reordering of Γ and σ. Since there are no dependencies between typing assumptions in Γ, the order does not actually matter. We will commonly use the notation $\Gamma, x : P$ (or $\sigma, v/x$) to expose a particular variable x in the context Γ' (or substitution σ') formed of the union of Γ (or σ) and $\{x : P\}$ (or $\{v/x\}$) without requiring x to be the last variable added to Γ'. We will use the notation Γ_1, Γ_2 to mean the disjoint union of both contexts (and, equivalently, σ_1, σ_2 for joining substitutions).

The operation is defined on terms as presented in Figure 2.6. $\sigma(x)$ denotes the value v assigned to the variable x in σ, ie $\sigma = \sigma', v/x$. **rec**-statements and **to**-statements and copattern abstractions introduce new variables. In order to maintain proper substitution typing $\Delta \vdash \sigma : \Gamma$, we introduce a constant id_Γ to indicate the identity substitution on Γ. In the case where we only touch a single variable x, we use the notation id_x as a shortcut for x/x. In addition, Γ_{q_i} denotes the context obtained from the variables defined in q_i, as per the copattern typing judgment.

We now want to prove the standard substitution lemma. We however need to introduce some intermediate lemma and the notion of context merging, denoted $\Delta_1 \cup \Delta_2$. It acts as a simple union operation on sets since we don't require a particular ordering on the contexts. If a variable x occurs in both Δ_1 and Δ_2, then we require both contexts to agree on the type of x. Otherwise, the merging is ill-defined. Thus, every use of the notation $\Delta_1 \cup \Delta_2$ will assume that both contexts agree on the type of their shared variables.

$$
\begin{array}{lcllcl}
[\sigma]x & = & \sigma(x) & [\sigma](\textsf{produce}\ v) & = & \textsf{produce}\ [\sigma]v \\
[\sigma]() & = & () & [\sigma](\textsf{force}\ v) & = & \textsf{force}\ [\sigma]v \\
[\sigma](v_1, v_2) & = & ([\sigma]v_1, [\sigma]v_2) & [\sigma](m\ v) & = & [\sigma]m\ [\sigma]v \\
[\sigma](\textsf{thunk}\ m) & = & \textsf{thunk}\ [\sigma]m & [\sigma](\textsf{rec}\ f.m) & = & \textsf{rec}\ f.[\sigma, \mathrm{id}_f]m \\
[\sigma](c\ v) & = & c\ [\sigma]v & [\sigma](m.d) & = & ([\sigma]m).d \\
 & & & [\sigma](m_1\ \textsf{to}\ x.m_2) & = & [\sigma]m_1\ \textsf{to}\ x.[\sigma, \mathrm{id}_x]m_2
\end{array}
$$

$$
[\sigma]\textsf{fun}\ (q_i \mapsto m_i)_{\forall i} = \textsf{fun}\ (q_i \mapsto [\sigma, \mathrm{id}_{\Gamma_{q_i}}]m_i)_{\forall i}
$$

Figure 2.6: Substitution

Lemma 2.1. *If* $\Delta_i \vdash \sigma_i : \Gamma_i$ *for* $i = 1, 2$*, then* $\Delta_1 \cup \Delta_2 \vdash \sigma_1, \sigma_2 : \Gamma_1, \Gamma_2$.

Proof. By lexicographic induction on the typing derivations $\Delta_1 \vdash \sigma_1 : \Gamma_1$ and $\Delta_2 \vdash \sigma_2 : \Gamma_2$.

Case $\dfrac{}{\Delta_1 \vdash \cdot : \cdot}$ and $\dfrac{}{\Delta_2 \vdash \cdot : \cdot}$

Trivially, $\dfrac{}{\Delta_1, \Delta_2 \vdash \cdot : \cdot}$.

Case $\dfrac{}{\Delta_1 \vdash \cdot : \cdot}$ and $\dfrac{\Delta_2 \vdash v : P \quad \Delta_2 \vdash \sigma_2 : \Gamma_2}{\Delta_2 \vdash \sigma_2, v/x : (\Gamma_2, x : P)}$

We can weaken the context of any term. Thus, we have $\Delta_1 \cup \Delta_2 \vdash v : P$. By induction on $\Delta_2 \vdash \sigma_2 : \Gamma_2$ we have $\Delta_1 \cup \Delta_2 \vdash \sigma_2 : \Gamma_2$. Thus $\Delta_1 \cup \Delta_2 \vdash \sigma_2, v/x : (\Gamma_2, x : P)$.

Case $\dfrac{\Delta_1 \vdash v : P \quad \Delta_1 \vdash \sigma_1 : \Gamma_1}{\Delta_1 \vdash \sigma_1, v/x : (\Gamma_1, x : P)}$ and $\dfrac{}{\Delta_2 \vdash \cdot : \cdot}$.

By term weakening, we have $\Delta_1 \cup \Delta_2 \vdash v : P$. By induction on $\Delta_1 \vdash \sigma_1 : \Gamma_1$ we have $\Delta_1 \cup \Delta_2 \vdash \sigma_1 : \Gamma_1$. Thus $\Delta_1 \cup \Delta_2 \vdash \sigma_1, v/x : (\Gamma_1, x : P)$.

Case $\dfrac{\Delta_1 \vdash v_1 : P_1 \quad \Delta_1 \vdash \sigma_1 : \Gamma_1}{\Delta_1 \vdash \sigma_1, v_1/x_1 : (\Gamma_1, x_1 : P_1)}$ and $\dfrac{\Delta_2 \vdash v_2 : P_2 \quad \Delta_2 \vdash \sigma_2 : \Gamma_2}{\Delta_2 \vdash \sigma_2, v_2/x_2 : (\Gamma_2, x_2 : P_2)}$.

By induction on $\Delta_1 \vdash \sigma_1, v_1/x_1 : (\Gamma_1, x_1 : P_1)$ and $\Delta_2 \vdash \sigma_2 : \Gamma_2$, we have

$$\Delta_1 \cup \Delta_2 \vdash \sigma_1, v_1/x_1, \sigma_2 : (\Gamma_1, x_1 : P_1, \Gamma_2).$$

We can then use term weakening v_2 to get $\Delta_1 \cup \Delta_2 \vdash v_2 : P_2$. We conclude that

$$\Delta_1 \cup \Delta_2 \vdash \sigma_1, v_1/x_1, \sigma_2, v_2/x_2 : (\Gamma_1, x_1 : P_1, \Gamma_2, x_2 : P_2).$$

□

Lemma 2.2 (Substitution lemma)**.** *The following holds.*

1. *If* $\Gamma \vdash v : P$ *and* $\Delta \vdash \sigma : \Gamma$*, then* $\Delta \vdash [\sigma]v : P$*.*
2. *If* $\Gamma \vdash m : N$ *and* $\Delta \vdash \sigma : \Gamma$*, then* $\Delta \vdash [\sigma]m : N$*.*

Proof. The proof is done by mutual induction on the typing derivations of v and m.

We will only show the cases for pairs copattern abstractions. The other ones are similar.

Case $\dfrac{\Gamma \vdash v_1 : P_1 \quad \Gamma \vdash v_2 : P_2}{\Gamma \vdash (v_1, v_2) : P_1 \times P_2}$

We make two appeals to the induction hypothesis to get $\Delta \vdash [\sigma]v_1 : P_1$ and $\Delta \vdash [\sigma]v_2 : P_2$. We thus have $\Delta \vdash [\sigma](v_1, v_2) : P_1 \times P_2$.

Case $\dfrac{\text{for all } i \quad \Gamma_i \vdash_{N_i} q_i : N \quad \Gamma, \Gamma_i \vdash m_i : N_i}{\Gamma \vdash \textsf{fun}\ (q_i \mapsto m_i) : N}$

$\Delta, \Gamma_i \vdash (\sigma, \mathrm{id}_{\Gamma_i}) : \Gamma, \Gamma_i$ for all i. by Lemma 2.1. Our induction hypothesis thus is $\Delta, \Gamma_i \vdash [\sigma, \mathrm{id}_{\Gamma_i}]m_i : N_i$ for all i. Thus, $\Delta \vdash [\sigma]\textsf{fun}\ (q_i \mapsto m_i) : N$. □

2.2.3 Evaluation

The evaluation rules appear in Figure 2.7. As we described above, when dealing with an application, a destructor application, or a to-statement, we simply extend the evaluation context then continue evaluating the left hand side. If we have a produce statement and

$$
\begin{array}{rcl}
m \ \textbf{to}\ x.n; K & \longrightarrow & m; ([]\ \textbf{to}\ x.n)\ K \\
\textbf{produce}\ v; ([]\ \textbf{to}\ x.n)\ K & \longrightarrow & [v/x]n; K \\
m.d; K & \longrightarrow & m; .d\ K \\
m\ v; K & \longrightarrow & m; v\ K \\
\textbf{force}\ (\textbf{thunk}\ m); K & \longrightarrow & m; K \\
\textbf{rec}\ f.m; K & \longrightarrow & [\textbf{thunk}\ (\textbf{rec}\ f.m)/f]m; K
\end{array}
$$

$$
\frac{q_i \doteq K \searrow (\sigma; K')}{\textbf{fun}\ (q_i \mapsto m_i); K \longrightarrow [\sigma]m_i; K'}
$$

Figure 2.7: Operational semantics

the top of the evaluation context is a to-statement, we substitute v for x in n and continue evaluating the resulting term with the original K. If we have a forcing of a thunk of m, we simply continue evaluating m. The rule that requires more attention is the one for copattern abstractions. In this case, we match a copattern q_i against the evaluation context K which results in a substitution σ together with a new evaluation context K'. We apply the substitution to the term m_i and proceed with K'.

The rules for copattern matching are defined in Figure 2.8. Any evaluation context will match against an empty copattern, outputting an empty substitution and the original evaluation context. If both have the same observation in head position, we simply continue matching the body of each. If we have a pattern together with a value, we pattern match on both and get a substitution σ_1. We append to it the substitution σ_2 from the matching of the tails.

In terms of pattern matching, a variable x will match any value v, outputting the substitution v/x. When matching constructors $c\ p$ against $c\ v$, we simply continue to match p against v. In the case of pairs, we match each side separately and append the two resulting substitutions together. The typing rules for (co)patterns enforce linearity and so if a variable

$\boxed{p \doteq v \searrow \sigma}$ Pattern p matches against value v outputting substitution σ.

$$\frac{p_1 \doteq v_1 \searrow \sigma_1 \quad p_2 \doteq v_2 \searrow \sigma_2}{(p_1, p_2) \doteq (v_1, v_2) \searrow \sigma_1, \sigma_2} \quad \frac{p \doteq v \searrow \sigma}{c\ p \doteq c\ v \searrow \sigma} \quad \frac{}{x \doteq v \searrow v/x}$$

$\boxed{q \doteq K \searrow (\sigma; K')}$ Copattern q matches against evaluation context K outputting substitution σ and new evaluation context K'.

$$\frac{}{\cdot \doteq K \searrow (\cdot; K)} \quad \frac{q \doteq K \searrow (\sigma; K')}{.d\ q \doteq .d\ K \searrow (\sigma; K')} \quad \frac{p \doteq v \searrow \sigma_1 \quad q \doteq K \searrow (\sigma_2; K')}{p\ q \doteq v\ K \searrow (\sigma_1, \sigma_2; K')}$$

Figure 2.8: Copattern matching

x matches with a value v, we are certain that adding the assignment v/x in the substitution will not create multiple assignments for the same variable.

Lemma 2.3 (Adequacy of (co)pattern matching)**.** *The following holds.*

1. *If* $\Gamma \vdash p : P$ *and* $\vdash v : P$ *and* $p \doteq v \searrow \sigma$*, then* $\vdash \sigma : \Gamma$.
2. *If* $\Gamma \vdash_N q : N'$ *and* $\vdash_{N_1} K : N'$ *and* $q \doteq K \searrow (\sigma; K')$*, then* $\vdash_{N_1} K' : N$ *and* $\vdash \sigma : \Gamma$.

Proof. Each statement is proved by induction on the (co)pattern matching derivation. Statement 2 uses Statement 1. We only show the following case of Statement 2 and leave the rest to the reader.

$$\frac{p \doteq v \searrow \sigma_1 \quad q \doteq K \searrow (\sigma_2; K')}{p\ q \doteq v\ K \searrow (\sigma_1, \sigma_2; K')}$$

By inversion on the typing derivation for q and K, respectively, we have $\Gamma_1 \vdash_N q : N'$, and $\Gamma_2 \vdash p : P$, and $\vdash v : P$, and $\vdash_{N_1} K : N'$ where $\Gamma = \Gamma_1, \Gamma_2$. By Statement 1, $\vdash \sigma_1 : \Gamma_1$. We can appeal to our induction hypothesis and get that $\vdash_{N_1} K' : N$ and $\vdash \sigma_2 : \Gamma_2$. We thus have $\vdash (\sigma_1, \sigma_2) : \Gamma_1, \Gamma_2$ by Lemma 2.1.

This concludes the proof. □

2.2.4 Evaluation of the length example

To illustrate how evaluation proceeds we re-consider our length example. We start with an empty evaluation context.

Program		Stack

produce
  (cons (zero (), cons (suc (zero (), nil ()))))
to l. **produce**
  (**thunk** (**rec** length.
  fun (nil ()) ⇒ **produce** (zero ())
  | (cons (x, xs)) ⇒ ((**force** length) xs) **to** y.
  produce (suc y)))
to length.
  (**force** length) l
	;	nil

On the outside, we have a **to**-statement. We put it on top of the stack and continue evaluating its left-hand side.

Program		Stack

produce
  (cons (zero (),
  cons (suc (zero ()), nil ())))
	;	([] **to** l. (**produce**
		  (**thunk** (**rec** length.
		  fun (nil ()) ⇒ **produce** (zero ())
		  | (cons (x, xs)) ⇒
		  ((**force** length) xs) **to** y. **produce** (suc y)))
		  to length. (**force** length) l))
		nil

Now, we have a **produce** statement on the outside and a **to**-statement on top of the stack. We can now do the substitution and continue with the right-hand side of the **to**-statement.

Program		Stack
produce (**thunk** (**rec** length. **fun** (nil ()) ⇒ **produce** (zero ()) | (cons (x, xs)) ⇒ ((**force** length) xs) **to** y. **produce** (suc y))) **to** length. (**force** length) (cons (zero (), cons (suc (zero ()), nil ()))))	;	nil

Two other points of interest for the evaluation are the recursion and the copattern abstraction. We thus skip ahead several steps to the following part.

Program		Stack
rec length. **fun** (nil ()) ⇒ **produce** (zero ()) | (cons (x, xs)) ⇒ ((**force** length) xs) **to** y. **produce** (suc y)	;	(cons (zero (), cons (suc (zero ()), nil ()))) nil

The recursion is handled by substituting every occurence of length by a **thunk** of the body of the **rec** statement. The type restriction explains why we had to precede length by a **force**.

Program		Stack
fun (nil ()) ⇒ **produce** (zero ()) | (cons (x, xs)) ⇒ (**force** (**thunk rec** length. **fun** (nil ()) ⇒ **produce** (zero ()) | (cons (x, xs)) ⇒ ((**force** length) xs) **to** y. suc y) xs) **to** y. **produce** (suc y)	;	(cons (zero (), cons (suc (zero ()), nil ()))) nil

We then have a copattern abstraction that needs to be matched against the evaluation context. The copatterns are made each of a single pattern. We match on the second copattern as our argument is a cons and obtain the following.

Program		Stack
force (**thunk** (**rec** length.	;	nil
fun (nil ()) ⇒ **produce** (zero ())		
| (cons (x, xs)) ⇒ ((**force** length) xs) **to** y. **produce** (suc y)))		
(cons (suc (zero ()), nil))		
to y. **produce** (suc y)		

The evaluation then continues, consuming the list one by one. We leave the details to the reader.

2.2.5 Subject reduction

We now get one of to the main statements we want to prove.

Theorem 2.4 (Subject reduction)**.** *If* $\vdash m; K : N$ *and* $m; K \longrightarrow m'; K'$ *then* $\vdash m'; K' : N$*.*

Proof. By case analysis on the evaluation rules. The only case that requires our attention is the rule

$$\frac{q_i \doteq K \searrow (\sigma; K')}{\textbf{fun}\ (q_i \mapsto m_i); K \longrightarrow [\sigma]m_i; K'}$$

By inversion on the typing rule for **fun** $(q_i \mapsto m_i); K$, we have $\Gamma \vdash_{N_1} q_i : N'$ and $\vdash_N K : N'$. By Lemma 2.3, we thus have that $\vdash_N K' : N_1$ and $\vdash \sigma : \Gamma$. Then, by Lemma 2.2, $\vdash [\sigma]m_i : N_1$ and so $\vdash [\sigma]m_i; K : N$. □

2.2.6 Coverage

The second property we require for our type soundness is the concept of progress which states that computations will not get stuck along the evaluation sequence. More specifically, for each well-typed term, either we can make another step, or we are in a terminal state such as a produced value.

We note that in the presence of (co)pattern matching, it is always possible to define a partial set of copatterns against which some valid terms cannot be matched, blocking the

evaluation to ever proceed. To prevent those cases, we require a notion of coverage which ensures that a covering copattern will always be able to match on any input.

Before we actually define coverage, we need to introduce a bit of notation. Let @ denote the append function for copatterns. Let k and k' be terms of the form v or $.d$ or $[]$ **to** $x.n$. Then, the append operation obeys the following rules.

$$(k\ K)@k' \quad = \quad k\ (K@k') \qquad\qquad \cdot@k \quad = \quad k\ \cdot$$

For example, if $K = v_1\ v_2$ .tail nil, then $K@v = v_1\ v_2$ .tail v nil.

Using this notation, we define an algorithm for coverage which acts by splitting a particular copattern in our current copattern set. The initial copattern set contains only the empty copattern. The algorithm is presented in Figure 2.9. We define a judgment $N \lhd| Q$ to mean that the type N is covered by the copattern set Q. We show in Lemma 2.12 that our notion of coverage is sound.

The judgment $N \lhd| Q$ is read from top to bottom. We start from the empty copattern $\cdot$ which covers any type N. The algorithm then makes a non deterministic choice of a particular copattern and splits on it via the judgment $(\Gamma \vdash q \rightrightarrows N') \Longrightarrow Q$. The result is a set of copatterns Q which is then added to our current covering set. The splitting can be done either on the resulting type N' which is used to introduce observations for the type $\nu Y.R$ or a variable for $P \to N''$, or splits on a given variable in the context Γ. We recall that we treat Γ as an unordered set and so the notation $\Gamma, x : P$ only serves at pointing out a variable x from the context rather than focusing only at the last variable introduced. The notation $\Gamma \vdash q \rightrightarrows N'$ represents a copattern whose typing is $\Gamma \vdash_{N'} q : N$ for some type N. This type N is determined by the input N in the judgment $N \lhd| Q$.

By design, it follows how the interactive splitting works in Agda [Norell, 2007] or Idris [Brady, 2013] and creates non-overlapping patterns ensuring evaluation is deterministic. This makes the algorithm simple and intuitive to create covering copatterns. The price for this simplicity is that it is not practically suitable for verifying if an existing set of copatterns is covering since it requires us to create possible splittings until we find the right one (or until we have exhausted the set of possible ones). For our theoretical purposes, however, this limitation will not be a problem and so we will leave out further considerations of this issue.

$\boxed{(\Gamma \vdash q \rightrightarrows N) \Longrightarrow Q}$ Copattern q is split into copattern set Q.

$$\begin{array}{rcll}
(\Gamma \vdash q \rightrightarrows P \to N) & \Longrightarrow & \{\Gamma, x : P \vdash q@x \rightrightarrows N\} & \mathrm{c}_{\mathrm{arr}} \\
(\Gamma \vdash q \rightrightarrows \nu Y.R) & \Longrightarrow & \{\Gamma \vdash q@.d \rightrightarrows [\nu Y.R/Y]R_d\}_{\forall d \in R} & \mathrm{c}_{\nu} \\
(\Gamma, x : P_1 \times P_2 \vdash q \rightrightarrows N) & \Longrightarrow & \{\Gamma, x_1 : P_1, x_2 : P_2 \vdash [(x_1, x_2)/x]q \rightrightarrows N\} & \mathrm{c}_{\mathrm{pair}} \\
(\Gamma, x : \mu X.D \vdash q \rightrightarrows N) & \Longrightarrow & \{\Gamma, x' : [\mu X.D/X]D_c \vdash [c\ x'/x]q \rightrightarrows N\}_{\forall c \in D} & \mathrm{c}_{\mu}
\end{array}$$

$\boxed{N \lhd\!| \ Q}$ Type N is covered by copattern set Q.

$$\frac{}{N \lhd\!| \ \{\cdot \vdash \cdot \rightrightarrows N\}}\ \mathrm{c}_{\mathrm{nil}} \qquad \frac{N \lhd\!| \ (Q \uplus \{\Gamma \vdash q \rightrightarrows N'\}) \quad (\Gamma \vdash q \rightrightarrows N') \Longrightarrow Q'}{N \lhd\!| \ Q \cup Q'}\ \mathrm{c}_{\mathrm{extend}}$$

Figure 2.9: Coverage algorithm

Example 8. In order to make more precise how this algorithm works, we will use it to obtain the copattern split for the cycleNats example.

We start with a single empty copattern to eliminate the type $\mathsf{Nat} \to \mathsf{Stream}$.

$$\mathsf{Nat} \to \mathsf{Stream} \lhd\!| \ (\cdot \vdash \cdot : \mathsf{Nat} \to \mathsf{Stream})$$

We eliminate the arrow type, introducing a variable for it in every copattern.

$$\mathsf{Nat} \to \mathsf{Stream} \lhd\!| \ (x : \mathsf{Nat} \vdash x\ \cdot : \mathsf{Stream})$$

We split it into two, one for each observation on streams. They are

$$\mathsf{Nat} \to \mathsf{Stream} \lhd\!| \ (x : \mathsf{Nat} \vdash x\ .\mathsf{head}\ \cdot : \mathsf{Nat})\ (x : \mathsf{Nat} \vdash x\ .\mathsf{tail}\ \cdot : \mathsf{Stream})$$

As the head is simply the input value, we are done. The tail will split on x, substituting a new term for each constructor.

$$\mathsf{Nat} \to \mathsf{Stream} \lhd\!| \ (x : \mathsf{Nat} \vdash x\ .\mathsf{head}\ \cdot)\ (x : 1 \vdash (\mathsf{zero}\ x)\ .\mathsf{tail}\ \cdot)\ (x : \mathsf{Nat} \vdash (\mathsf{suc}\ x)\ .\mathsf{tail}\ \cdot)$$

Now, we do not split further as we have obtained the copattern set we wanted.

We want to define a notion of coverage indicating that for any series of well-typed terms or observations applied to a copattern abstraction, it will not get stuck but rather step. It is however possible that the copattern abstraction only steps when it is applied a certain number of terms or observations. Thus, a particular evaluation context might not trigger unless more is added to it. We thus want to add some judgment that expresses that.

Now, we can say that $\vdash_N K : N'$ is covered by the copattern set Q, denoted $K \triangleleft Q : N$ if there is an extension of K that matches against Q. We thus define recursively the judgment $K \triangleleft Q : N$ in Figure 2.10.

$$\frac{\exists q \in Q \quad q \doteq K \searrow (\sigma; K')}{K \triangleleft Q : N} \qquad \frac{\forall v \in P \quad K@v \triangleleft Q : N}{K \triangleleft Q : P \to N}$$

$$\frac{\forall d \in R \quad K@.d \triangleleft Q : R_d[\nu Y.R/Y]}{K \triangleleft Q : \nu Y.R}$$

Figure 2.10: Covering of evaluation contexts

We first prove some useful lemmas.

Lemma 2.5. *The following hold.*

1. *Let* $\Gamma, x : P_1 \times P_2 \vdash p : P$ *and* $\vdash v : P$. *If* $p \doteq v \searrow \sigma, v'/x$, *then* $v' = (v_1, v_2)$, *and* $[(x_1, x_2)/x]p \doteq v \searrow \sigma, v_1/x_1, v_2/x_2$.

2. *Let* $\Gamma, x : \mu X.D \vdash p : P$ *and* $\vdash v : P$. *If* $p \doteq v \searrow \sigma, v'/x$, *then* $v' = c\ v''$, *and* $[c\ x'/x]p \doteq v \searrow \sigma, v''/x'$ *for some* $c \in D$.

Proof. We will prove Statement 1. Statement 2 is analogous and left to the reader. The proof is done by induction on the derivation for $p \doteq v \searrow \sigma$.

Case $\overline{y \doteq v \searrow v/y}$

If $\Gamma, x : P_1 \times P_2 \vdash y : P$, then $\Gamma = \cdot$, $x = y$ and $P = P_1 \times P_2$ by inversion on pattern typing. Thus, $\vdash v : P_1 \times P_2$. By inversion on the typing derivation of v, we have $v = (v_1, v_2)$ where $\vdash v_1 : P_1$ and $\vdash v_2 : P_2$. Hence, we can use the following derivation.

$$\frac{\overline{x_1 \doteq v_1 \searrow v_1/x_1} \quad \overline{x_2 \doteq v_2 \searrow v_2/x_2}}{(x_1, x_2) \doteq (v_1, v_2) \searrow v_1/x_1, v_2/x_2}$$

Case $\dfrac{p \doteq v \searrow \sigma, v'/x}{c\ p \doteq c\ v \searrow \sigma, v'/x}$

Our induction hypothesis gives us that $[(x_1, x_2)/x]p \doteq v \searrow \sigma, v''/x'$ where $v' = c\ v''$, and so $[(x_1, x_2)/x](c\ p) \doteq c\ v \searrow \sigma, v''/x'$ by definition of substitution.

The last case is similar and thus left to the reader. □

Lemma 2.6. *If $K \lhd \{q\} : N$, then there is a K' such that $q \doteq K' \searrow (\sigma; K'')$.*

Proof. The proof is done by induction on the derivation of $K \lhd \{q\} : N$.

Case $\dfrac{q \doteq K \searrow (\sigma; K')}{K \lhd \{q\} : N}$

Trivially, $K' = K$ and $q \doteq K \searrow (\sigma; K')$.

Case $\dfrac{\forall v \in P \quad K@v \lhd \{q\} : N}{K \lhd \{q\} : P \to N}$

By induction, there is a K' such that $K' \doteq q \searrow (\sigma; K'')$.

Case $\dfrac{\forall d \in R \quad K@.d \lhd \{q\} : R_d[\nu Y.R/Y]}{K \lhd \{q\} : \nu Y.R}$

By induction, there is a K' such that $K' \doteq q \searrow (\sigma; K'')$.

This concludes the proof. □

Lemma 2.7. *The following hold.*

1. *Let $\Gamma, x : P_1 \times P_2 \vdash_{N'} q : N$ and $\vdash_{N''} K : N$. If $q \doteq K \searrow (\sigma, v/x; K')$, then $v = (v_1, v_2)$ and $[(x_1, x_2)/x]q \doteq K \searrow (\sigma, v_1/x_1, v_2/x_2; K')$.*

2. *Let* $\Gamma, x : \mu X.D \vdash_{N'} q : N$ *and* $\vdash_{N''} K : N$. *If* $q \doteq K \searrow (\sigma, v/x; K')$, *then* $v = c\ v'$ *and* $[c\ x'/x]q \doteq K \searrow (\sigma, v'/x'; K')$ *for some* $c \in D$.

3. *Let* $\Gamma \vdash_{P \to N'} q : N$ *and* $\vdash_{N''} K : N$. *If* $q \doteq K \searrow (\sigma; K')$, *then* $K \triangleleft \{q@x\} : N''$.

4. *Let* $\Gamma \vdash_{\nu Y.R} q : N$ *and* $\vdash_{N''} K : N$. *If* $q \doteq K \searrow (\sigma; K')$, *then* $K \triangleleft \{q@.d\}_{\forall d \in R} : N''$.

Proof. Statements 1 and 2 are proved by induction on the derivation $q \doteq K \searrow (\sigma, v/x; K')$ and use Lemma 2.5. Statements 3 and 4 are proved by induction on $q@k \doteq K \searrow (\sigma; K')$ where $k = x$ and $k = .d$, respectively.

Case $\dfrac{}{\cdot \doteq K \searrow (\cdot; K)}$

We have $\vdash_{N''} K : P \to N'$. Then, by inversion on its typing derivation $K = \cdot$ or $K = v\ K'$ for some $\vdash v : P$ and some $\vdash_{N''} K' : N'$. In the former case, for any $v \in P$, we have $x\cdot \doteq v \cdot \searrow (v/x; \cdot)$ which implies that $\cdot@v \triangleleft \{\cdot@x\} : N''$. In the latter case, we have the following derivation.

$$\frac{x \doteq v \searrow v/x \quad \cdot \doteq K' \searrow (\cdot; K')}{x \cdot \doteq v\ K' \searrow (v/x; K')}$$

It follows that $K \triangleleft \{\cdot@x\} : N''$.

Case $\dfrac{p \doteq v \searrow \sigma \quad q \doteq K \searrow (\sigma'; K')}{p\ q \doteq v\ K \searrow (\sigma, \sigma'; K')}$

Our induction hypothesis is $K \triangleleft \{q@x\} : N''$. Hence, by Lemma 2.6 there is a K' such that $q@x \doteq K' \searrow (\sigma''; K'')$ and so $p\ (q@x) \doteq v\ K' \searrow (\sigma, \sigma''; K'')$. We conclude that $v\ K \triangleleft \{(p\ q)@x\} : N''$ since $(p\ q)@x = p\ (q@x)$.

Case $\dfrac{q \doteq K \searrow (\sigma; K')}{.d\ q \doteq .d\ K \searrow (\sigma; K')}$

The proof is identical to the one of the last case and is thus left to the reader. □

The next two lemmas introduce results about inverse substitutions on (co)patterns and (co)pattern matching on extended copatterns by append operations. The notion of inverse

substitutions is well-defined due to the linear nature of (co)patterns, These results will be useful in Section 3.3.

Lemma 2.8. *The following hold.*

1. *Let* $\Gamma, x_1 : P_1, x_2 : P_2 \vdash [(x_1, x_2)/x]p : P$. *If* $[(x_1, x_2)/x]p \doteq v \searrow \sigma, v_1/x_1, v_2/x_2$, *then* $p \doteq v \searrow \sigma, (v_1, v_2)/x$

2. *Let* $\Gamma, x' : D_c[\mu X.D/X] \vdash [c\ x'/x]p : P$. *If* $[c\ x'/x]p \doteq v \searrow \sigma, v/x'$, *then* $p \doteq v \searrow \sigma, (c\ v)/x$.

Proof. Both statements are proved by case analysis on p, then by induction the derivation $p \doteq v \searrow \sigma$. We only do the following case of Statement 1. The rest is left to the reader.

Case $[(x_1, x_2)/x]p = (p_1, p_2)$.

If $[(x_1, x_2)/x]p = (x_1, x_2)$, then $(x_1, x_2) \doteq (v_1, v_2) \searrow v_1/x_1, v_2/x_2$. Trivially, $x \doteq (v_1, v_2) \searrow (v_1, v_2)/x$. If $[(x_1, x_2)/x]p \neq (x_1, x_2)$, then by inversion on the pattern matching rule, we have either

$$\frac{[(x_1, x_2)/x]p_1 \doteq v_1 \searrow \sigma_1, v_1/x_2, v_2/x_2 \quad [(x_1, x_2)/x]p_2 \doteq v_2 \searrow \sigma_2}{[(x_1, x_2)/x](p_1, p_2) \doteq (v_1, v_2) \searrow \sigma_1, \sigma_2, v_1/x_1, v_2/x_2}$$

or

$$\frac{[(x_1, x_2)/x]p_1 \doteq v_1 \searrow \sigma_1 \quad [(x_1, x_2)/x]p_2 \doteq v_2 \searrow \sigma_2, v_1/x_2, v_2/x_2}{[(x_1, x_2)/x](p_1, p_2) \doteq (v_1, v_2) \searrow \sigma_1, \sigma_2, v_1/x_1, v_2/x_2}$$

since, by linearity of patterns, the variable x can occur only in one of p_1 and p_2. Suppose the former holds. Then, $p_2 \doteq v_2 \searrow \sigma_2$ as $[(x_1, x_2)/x]p_2 = p_2$. By induction, $p_1 \doteq v_1 \searrow \sigma_1, (v_1, v_2)/x$. Hence, $(p_1, p_2) \doteq (v_1, v_2) \searrow \sigma_1, (v_1, v_2)/x, \sigma_2$. We recall from Section 2.2.2 that we do not consider contexts and substitutions to be ordered sets and so $\sigma_1, (v_1, v_2)/x, \sigma_2 = \sigma_1, \sigma_2, (v_1, v_2)/x$. Thus, we are done. □

Lemma 2.9. *The following hold.*

1. *Let* $\Gamma, x_1 : P_1, x_2 : P_2 \vdash_{N'} [(x_1, x_2)/x]q : N$. *If* $[(x_1, x_2)/x]q \doteq K \searrow (\sigma, v_1/x_1, v_2/x_2; K')$, *then* $q \doteq K \searrow (\sigma, (v_1, v_2)/x; K')$.

2. *Let* $\Gamma, x' : [\mu X.D/X]D_c \vdash_{N'} [c\ x'/x]q : N$. *If* $[c\ x'/x]q \doteq K \searrow (\sigma, v/x'; K')$, *then* $q \doteq K \searrow (\sigma, (c\ v)/x; K')$.

3. *If* $q@x \doteq K \searrow (\sigma, v/x; K')$, *then* $q \doteq K \searrow (\sigma; v\ K')$.

4. *If* $q@.d \doteq K \searrow (\sigma; K')$, *then* $q \doteq K \searrow (\sigma; .d\ K')$.

Proof. Statements 1 and 2 are proved by induction on the derivation $[p/x]q \doteq K \searrow (\sigma; K')$ for the appropriate p and use Lemma 2.8. Statements 3 and 4 are proved by case analysis on q and induction on $q@x \doteq K \searrow (\sigma; K')$ (or $q@.d \doteq K \searrow (\sigma; K')$, respectively). We only prove Statement 3 as the proof for Statement 4 is identical.

Case $q = \cdot$.

Thus,

$$\frac{x \doteq v \searrow v/x \quad \overline{\cdot \doteq K \searrow (\cdot; K)}}{x \cdot \doteq v\ K \searrow (v/x; K)}$$

Trivially, $\cdot \doteq v\ K \searrow (\cdot; v\ K)$.

Case $q = y\ q'$.

Thus, the only possible case is

$$\frac{y \doteq v' \searrow \sigma \quad q'@x \doteq K \searrow (\sigma', v/x; K')}{y\ (q'@x) \doteq v'\ K \searrow (\sigma, \sigma', v/x; K')}$$

Our induction hypothesis is thus $q' \doteq K \searrow (\sigma; v\ K')$. Hence $y\ q' \doteq v'\ K \searrow (\sigma, \sigma'; K')$.

Case $q = .d\ q'$.

This case is identical to the previous one. This concludes the proof. □

Lemma 2.10. *If* $\Gamma \vdash_{N'} q : N$ *and* $\vdash_{N''} K : N$ *and* $(\Gamma \vdash q \rightrightarrows N') \Longrightarrow Q'$ *and* $q \doteq K \searrow (\sigma; K')$, *then* $K \triangleleft Q : N''$.

Proof. The proof is done by case analysis on $(\Gamma \vdash q \rightrightarrows N') \Longrightarrow Q'$. Each case is solved by calling the corresponding statement in Lemma 2.7. □

Corollary 2.11. *If* $\Gamma \vdash_{N'} q : N$ *and* $\vdash_{N''} K : N$ *and* $(\Gamma \vdash q \rightrightarrows N') \Longrightarrow Q'$ *and* $K \lhd (Q \uplus \{\Gamma \vdash q \rightrightarrows N'\}) : N''$*, then* $K \lhd (Q \cup Q') : N''$*.*

Proof. By induction on the derivation of $K \lhd (Q \uplus \{\Gamma \vdash q \rightrightarrows N'\}) : N''$. The base case is proved using Lemma 2.10. □

Lemma 2.12 (Soundness of Coverage)**.** *If* $\vdash_{N'} K : N$ *and* $N \lhd| Q$*, then* $K \lhd Q : N'$*.*

Proof. By induction on the derivation $N \lhd| Q$.

Case $\dfrac{}{N \lhd| \{\cdot \vdash \cdot : N\}}$

Since $\cdot \doteq K \searrow (\cdot; K)$, we have trivially $K \lhd \{\cdot\} : N'$.

Case $\dfrac{N \lhd| (Q \uplus \{\Gamma \vdash q \rightrightarrows N'\}) \quad (\Gamma \vdash q \rightrightarrows N') \Longrightarrow Q'}{N \lhd| Q \cup Q'}$

By Corollary 2.11. □

2.2.7 Progress

From now on, we assume any copattern set Q has a derivation $N \lhd| Q$. We can now formally define progress. We define it through the judgment $m; K$ **safe at** N in Figure 2.11.

$\boxed{m; K \textbf{ safe at } N}$ Term $\vdash m : N'$ for some N' together with evaluation context $\vdash_N K : N'$ progress.

$$\frac{}{\textbf{produce}\ v; \mathsf{nil} \textbf{ safe at } \uparrow P} \qquad \frac{\forall v : P \quad m; K@v \textbf{ safe at } N}{m; K \textbf{ safe at } P \to N}$$

$$\frac{m; K \longrightarrow m'; K'}{m; K \textbf{ safe at } N} \qquad \frac{\forall d \in R \quad m; K@.d \textbf{ safe at } R_d[\nu Y.R/Y]}{m; K \textbf{ safe at } \nu Y.R}$$

Figure 2.11: Progress

It appears obvious that a produced value with an empty stack does not have to step at all. It is obvious that a configuration that steps satisfies the idea of progress. The other

two rules follow the idea that a term might be underapplied at the current time and thus extensions of the evaluation context will trigger a reduction rule. It is done through this recursive relation.

We handle the case for copattern abstractions in the following lemma.

Lemma 2.13. *If $Q = \{q_i\}_{\forall i}$ where $(q_i \mapsto m_i) = u_i$ and $K \lhd Q : N$, then*

$$\textsf{fun}\ \vec{u}; K\ \textbf{\textit{safe at}}\ N.$$

Proof. The proof is done by induction on the derivations for $K \lhd Q : N$.

Case

$$\frac{\exists q \in Q \quad q \doteq K \searrow (\sigma; K')}{K \lhd Q : N}$$

Since we have a pattern match, we can take $u = \Gamma.q \mapsto m$ and we have that $\textsf{fun}\ \vec{u}; K \longrightarrow [\sigma]m; K'$. Thus, $\textsf{fun}\ \vec{u}; K$ **safe at** N.

Case

$$\frac{\forall v \in P \quad K@v \lhd Q : N}{K \lhd Q : P \to N}$$

By induction hypothesis, we have $\textsf{fun}\ \vec{u}; K@v$ **safe at** N for all $v : P$. By our definition of progress, $\textsf{fun}\ \vec{u}; K$ **safe at** $P \to N$ follows.

The other case is analoguous. □

This leads us to proving the progress theorem.

Theorem 2.14 (Progress)**.** *If $\vdash m; K : N$ and for all copattern abstraction $\textsf{fun}\ \vec{u}$ in m, we have a covering derivation $N' \lhd| Q$ for $\vec{u}$, then $m; K$ **safe at** N.*

Proof. The proof is done by case analysis on m.

If m is of the form $m'.d$, or $m'\ v$, or $\textsf{force}\ (\textsf{thunk}\ m')$, or $m\ \textsf{to}\ '.xn$, or $\textsf{rec}\ f.m'$, then there is a stepping rule and so it progresses. If m is $\textsf{produce}\ v$, then there are several possibilites for K. If K is nil, we are done by assumption. If K is $([]\ \textsf{to}\ x.n)\ K'$ for some

K', it will step to $[v/x]n; K'$. The cases where K is of the form $.d\ K$ or $v\ K$ are impossible since K must have type $\uparrow P$ for some P.

This leaves us with m being of the form **fun** $\vec{u}$. By assumption, we have $N' \lhd| Q$. Since $\vdash_N K : N'$, we have that $K \lhd Q : N$ by Lemma 2.12. Hence, we can use Lemma 2.13 to obtain that **fun** $\vec{u}; K$ **safe at** N. This concludes the proof. □

Chapter 3

Compilation

This chapter focuses on the question of compilation of CBPV$^{\text{copat}}$ into CBPV. The main part of the difference between the two languages is the deep (co)pattern matching which gets replaced into simple case analysis on the different value term constructs. Under this scope, CBPV can be used as an intermediate language in the compilation of the programs into machine code. This language is presented in detail in Section 3.1.

One of the main advantage of the CBPV representation over CBPV$^{\text{copat}}$ is that the former does not require a check for coverage. In itself, we can see the translation to be a form of coverage checking as the translation would succeed only if the patterns are indeed covering. We discuss this translation in Section 3.2.

We note that we need to be careful when doing such translation if we want to preserve the programs users write and the properties they hold. In Section 3.3, we prove that our translation preserves evaluation.

3.1 Target language: Levy's CBPV

We now present in detail Levy's original call-by-push-value language. We define the types in Section 3.1.1, the terms and their typing rules in Section 3.1.2, and the evaluation rules in Section 3.1.3.

P	$::=$	X	Positive type variable
	$\mid$	$P_1 \times P_2$	Product type
	$\mid$	$\mu X.P$	Data type
	$\mid$	1	Unit type
	$\mid$	D	Variant
	$\mid$	$\downarrow N$	Embedding negative type
N	$::=$	Y	Negative type variable
	$\mid$	$P \to N$	Function type
	$\mid$	$\nu Y.N$	Codata type
	$\mid$	R	Record type
	$\mid$	$\uparrow P$	Embedding positive type
D	$::=$	$\langle c_1\ P_1 \mid \cdots \mid c_n\ P_n \rangle$	Labeled sum
R	$::=$	$\{d_1 : N_1\ \&\ \ldots\ \&\ d_n : N_n\}$	Labeled product

Figure 3.1: Types of CBPV

3.1.1 Types

The types of CBPV are defined in Figure 3.1. The main difference with our presentation in Chapter 2 is that variants and records can appear by themselves and are not restricted to only be used when constructing (co)data types.

While it might seem to generalize the type system by allowing it to express types that could not be encoded in our version, we can always convert them by simply wrapping variants and records in data and codata types, respectively. We can also create a single constructor or observation to join to data and codata types, respectively, to get back our original types. Thus, the two languages are equivalent in terms of the types they can express.

$$
\begin{array}{lcl}
\text{Nat} & = & \mu X.\, \langle \text{zero } 1 \mid \text{succ } X \rangle \\
\text{List} & = & \mu X.\, \langle \text{nil } 1 \mid \text{cons Nat} \times X \rangle \\
\text{Bool} & = & \langle \text{true } 1 \mid \text{false } 1 \rangle \\
\text{Option} & = & \langle \text{none } 1 \mid \text{some Nat} \rangle \\
\text{Stream} & = & \nu Y.\, \{\text{head } : \uparrow\text{Nat} \;\&\; \text{tail } : \text{Stream}\} \\
\text{Vector} & = & \{\text{list } : \uparrow\text{List} \;\&\; \text{length } : \uparrow\text{Nat}\} \\
\text{LazyList} & = & \nu Y.\uparrow \langle \text{nil } 1 \mid \text{cons Nat} \times \downarrow Y \rangle
\end{array}
$$

Figure 3.2: Recursives types in CBPV

3.1.2 Typing rules

While the types are similar, the terms, however, are quite different (see Figure 3.3). On the value side, we have a split between the constructors (denoting sums) and the **foldv**-statement (denoting the recursive types). On the computational side, we split (co)pattern matching, creating a term per (co)pattern matching rule. Variable introduction is done through the usual function abstraction. There are case analysis constructs to eliminate sums, values of recursive types and pairs. Records are defined by providing a term for each observation and eliminated using projections. Computations of recursive types can be folded and unfolded using appropriate constructs.

Example 9. Let us recall the length function in CBPV$^{\text{copat}}$:

```
rec length.
    fun (nil x) ⇒ produce (zero ())
      | (cons (x, xs)) ⇒ ((force length) xs) to y. produce (suc y)))
```

In CBPV, it is expanded to this:

```
rec length.λx.pm x as foldv y.pm y as
  < nil z.produce (zero ())
  , cons z.pm z as (x, xs).((force length) xs) to y. produce (suc y))) >
```

$\Gamma \vdash v : P$ Value v has type P in context Γ.

$$\frac{\Gamma(x) = P}{\Gamma \vdash x : P} \quad \frac{}{\Gamma \vdash () : 1} \quad \frac{\Gamma \vdash m : N}{\Gamma \vdash \textsf{thunk}\ m : \downarrow N}$$

$$\frac{\Gamma \vdash v_1 : P_1 \quad \Gamma \vdash v_2 : P_2}{\Gamma \vdash (v_1, v_2) : P_1 \times P_2} \quad \frac{\Gamma \vdash v : D_c}{\Gamma \vdash c\ v : D} \quad \frac{\Gamma \vdash v : [\mu X.P/X]P}{\Gamma \vdash \textsf{foldv}\ v : \mu X.P}$$

$\Gamma \vdash m : N$ Computation m has type N in context Γ.

$$\frac{\Gamma \vdash v : P}{\Gamma \vdash \textsf{produce}\ v : \uparrow P} \quad \frac{\Gamma \vdash v : \downarrow N}{\Gamma \vdash \textsf{force}\ v : N} \quad \frac{\Gamma \vdash m : P \to N \quad \Gamma \vdash v : P}{\Gamma \vdash m\ v : N} \quad \frac{\Gamma, f : \downarrow N \vdash m : N}{\Gamma \vdash \textsf{rec}\ f.m : N}$$

$$\frac{\Gamma \vdash m : R}{\Gamma \vdash m.d : R_d} \quad \frac{\Gamma \vdash m_1 : \uparrow P \quad \Gamma, x : P \vdash m_2 : N}{\Gamma \vdash m_1\ \textsf{to}\ x.m_2 : N} \quad \frac{\Gamma, x : P \vdash m : N}{\Gamma \vdash \lambda x.m : P \to N}$$

$$\frac{\Gamma \vdash v : D \quad \text{for all c. } \Gamma, x : P_c \vdash m_c : N}{\Gamma \vdash \textsf{pm}\ v\ \textsf{as}\ \langle \ldots, c\ x.m_c, \ldots \rangle : N} \quad \frac{\text{for all d} \quad \Gamma \vdash m_d : R_d}{\Gamma \vdash \lambda\{\ldots, d.m_d, \ldots\} : R}$$

$$\frac{\Gamma \vdash v : P_1 \times P_2 \quad \Gamma, x : P_1, y : P_2 \vdash m : N}{\Gamma \vdash \textsf{pm}\ v\ \textsf{as}\ (x, y).m : N} \quad \frac{\Gamma \vdash m : R}{\Gamma \vdash m.d : R_d} \quad \frac{m : [\nu Y.N/Y]N}{\Gamma \vdash \textsf{foldc}\ m : \nu Y.N}$$

$$\frac{\Gamma \vdash v : \mu X.P \quad \Gamma, x : [\mu X.P/X]P \vdash m : N}{\Gamma \vdash \textsf{pm}\ v\ \textsf{as foldv}\ x.m : N} \quad \frac{\Gamma \vdash m : \nu Y.N}{\Gamma \vdash \textsf{unfold}\ m : [\nu Y.N/Y]N}$$

Figure 3.3: Typing rules of CBPV

3.1.3 Evaluation

The operational semantics of CBPV is still a small-step semantics with evaluation contexts just as CBPV$^{\text{copat}}$. Those evaluation contexts, however, require an additional construct as **unfold** has been separated from record definitions. We thus add to the evaluation context the following rule.

$$\frac{\vdash_N K : [\nu Y.N'/Y]N'}{\vdash_N \textsf{unfold}\ K : \nu Y.N'}$$

The rules now differ as we replaced the copattern abstraction by value eliminations done

$$
\begin{array}{rcl}
m \ \textbf{to}\ x.n; K & \longrightarrow & m; ([]\ \textbf{to}\ x.n)\ K \\
\textbf{produce}\ v; ([]\ \textbf{to}\ x.n)\ K & \longrightarrow & [v/x]n; K \\
m.d; K & \longrightarrow & m; .d\ K \\
\lambda\{\ldots, d.m_d, \ldots\}; .d\ K & \longrightarrow & m_d; K \\
m\ v; K & \longrightarrow & m; v\ K \\
\lambda x.m; v\ K & \longrightarrow & [v/x]m; K \\
\textbf{unfold}\ m; K & \longrightarrow & m; \textbf{unfold}\ K \\
\textbf{foldc}\ m; \textbf{unfold}\ K & \longrightarrow & m; K \\
\textbf{force}\ (\textbf{thunk}\ m); K & \longrightarrow & m; K \\
\textbf{rec}\ f.m; K & \longrightarrow & [\textbf{thunk}\ (\textbf{rec}\ f.m)/f]m; K \\
\textbf{pm}\ (v_1, v_2)\ \textbf{as}\ (x, y).m; K & \longrightarrow & [v_1/x, v_2/y]m; K \\
\textbf{pm}\ (\textbf{foldv}\ v)\ \textbf{as foldv}\ x.m; K & \longrightarrow & [v/x]m; K \\
\textbf{pm}\ (c\ v)\ \textbf{as}\ \langle \ldots, c\ x.m_c, \ldots \rangle; K & \longrightarrow & [v/x]m_c; K
\end{array}
$$

Figure 3.4: Operational Semantics

by case analysis, function abstraction and projections. The small-step rules appear in Figure 3.4.

Pattern matching will simply substitute the value under the constructor into the term to continue evaluating. Projections simply choose the part of the record to use. If we encounter an **unfold** , we stash it until we get a **foldc** to cancel them out.[1]

[1]Levy used the rule **unfold** (**foldc** m); $K \longrightarrow m; K$ which appears to be too restrictive as evaluation can get stuck if an unfold is applied to any other computation (as it could as well be). We thus generalized the rules to allow progress to be preserved.

3.2 Translation

The translation for types is quite simple as the only differences between CBPV and CBPV$^{\text{copat}}$ are the fact that the latter restricts the use of variants and records to be directly under a data or codata type bindings, respectively. The translation is then just the embedding of CBPV$^{\text{copat}}$ into CBPV.

$\boxed{\Gamma \vdash v : P \rightsquigarrow w}$ Value v is translated to w.

$$\frac{}{\Gamma \vdash x : P \rightsquigarrow x} \quad \frac{\Gamma \vdash v_1 : P_1 \rightsquigarrow w_1 \quad \Gamma \vdash v_2 : P_2 \rightsquigarrow w_2}{\Gamma \vdash (v_1, v_2) : P_1 \times P_2 \rightsquigarrow (w_1, w_2)} \quad \frac{}{\Gamma \vdash () : 1 \rightsquigarrow ()}$$

$$\frac{\Gamma \vdash m : N \rightsquigarrow n}{\Gamma \vdash \textbf{thunk}\ m : {\downarrow}N \rightsquigarrow \textbf{thunk}\ n} \quad \frac{\Gamma \vdash v : [\mu X.D/X]D_c \rightsquigarrow w}{\Gamma \vdash c\ v : \mu X.D \rightsquigarrow \textbf{foldv}\ (c\ w)}$$

$\boxed{\Gamma \vdash m : N \rightsquigarrow n}$ Term m is translated to n.

$$\frac{\Gamma \vdash v : {\downarrow}N \rightsquigarrow w}{\Gamma \vdash \textbf{force}\ v : N \rightsquigarrow \textbf{force}\ w} \quad \frac{\Gamma \vdash v : P \rightsquigarrow w}{\Gamma \vdash \textbf{produce}\ v : {\uparrow}P \rightsquigarrow \textbf{produce}\ w}$$

$$\frac{\Gamma \vdash m : \nu Y.R \rightsquigarrow n}{\Gamma \vdash m.d : [\nu Y.R/Y]R_d \rightsquigarrow (\textbf{unfold}\ n).d} \quad \frac{\Gamma, x : {\downarrow}N \vdash m : N \rightsquigarrow n}{\Gamma \vdash \textbf{rec}\ x.m : N \rightsquigarrow \textbf{rec}\ x.n}$$

$$\frac{\Gamma \vdash m : P \to N \rightsquigarrow n \quad \Gamma \vdash v : P \rightsquigarrow w}{\Gamma \vdash m\ v : N \rightsquigarrow n\ w} \quad \frac{\Gamma \vdash m_1 : {\uparrow}P \rightsquigarrow n_1 \quad \Gamma \vdash m_2 : N \rightsquigarrow n_2}{\Gamma \vdash m_1\ \textbf{to}\ x.m_2 : N \rightsquigarrow n_1\ \textbf{to}\ x.n_2}$$

$$\frac{\Gamma_i \vdash_{N_i} q_i : N \quad \Gamma, \Gamma_i \vdash m_i \rightsquigarrow m_i' \quad N \lhd| Q \gg \{m_i'\}_{\forall i \in I} \rightsquigarrow m \qquad \text{where } Q = \{q_i\}_{\forall i \in I}}{\Gamma \vdash \textbf{fun}\ (q_i \mapsto m_i)_{\forall i \in I} : N \rightsquigarrow m}$$

Figure 3.5: Translation from CBPV$^{\text{copat}}$ to CBPV

The translation for terms appears in Figure 3.5. It is done through the judgments

$$\Gamma \vdash v : P \rightsquigarrow v' \qquad \text{and} \qquad \Gamma \vdash m : N \rightsquigarrow m'$$

for values and computations, respectively. Most of the cases are returned as is. The two

$\boxed{\mathcal{D} : (N \lhd\mid Q) \gtrdot \mathcal{M} \rightsquigarrow m}$ Copattern coverage derivation $\mathcal{D}$ produces term m from the set of terms $\mathcal{M}$.

$$\frac{}{\mathcal{D} : (N \lhd\mid \{\cdot \vdash \cdot \rightrightarrows N\}) \gtrdot \{m_1\} \rightsquigarrow m_1}$$

$$\frac{\mathcal{C} : ((\Gamma \vdash q_j \rightrightarrows N') \Longrightarrow Q') \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}' \quad \mathcal{D}' : (N \lhd\mid (Q \uplus \{q_j\})) \gtrdot \mathcal{M}' \rightsquigarrow m}{\mathcal{D} : (N \lhd\mid Q \cup Q') \gtrdot \mathcal{M} \rightsquigarrow m}$$

$\boxed{\mathcal{C} : ((\Gamma \vdash q \rightrightarrows N) \Longrightarrow Q) \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}'}$ Copattern splitting rule $\mathcal{C}$ is used to transform set of terms $\mathcal{M}$ into set $\mathcal{M}'$.

$$\frac{\mathcal{M}' = (\mathcal{M} \backslash \{m_q\}) \cup \{\lambda x.m_q\}}{c_{\text{arr}} : ((\Gamma \vdash q \rightrightarrows P \to N) \Longrightarrow \{\Gamma, x : P \vdash q@x \rightrightarrows N\}) \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}'}$$

$$\frac{\mathcal{M}' = (\mathcal{M} \backslash \{m_d\}_{\forall d \in R}) \cup \{\textsf{foldc}\ \lambda\{\ldots, d.m_d, \ldots\}\}}{c_\nu : ((\Gamma \vdash q \rightrightarrows \nu Y.R) \Longrightarrow \{\Gamma \vdash q@.d \rightrightarrows [\nu Y.R/Y]R_d\}_{\forall d \in R}) \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}'}$$

$$\frac{\mathcal{M}' = (\mathcal{M} \backslash \{m_q\}) \cup \{\textsf{pm}\ x\ \textsf{as}\ (x_1, x_2).m_q\}\}}{c_{\text{pair}} : ((\Gamma, x : P_1 \times P_2 \vdash q \rightrightarrows N) \Longrightarrow \{\Gamma, x_1 : P_1, x_2 : P_2 \vdash [(x_1, x_2)/x]q \rightrightarrows N\}) \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}'}$$

$$\frac{\mathcal{M}' = (\mathcal{M} \backslash \{m_c\}_{\forall c \in D}) \cup \{\textsf{pm}\ x\ \textsf{as foldv}\ x.\textsf{pm}\ x\ \textsf{as}\ \langle \ldots, c\ x'.m_c, \ldots \rangle\}\}}{c_\mu : ((\Gamma, x : \mu X.D \vdash q \rightrightarrows N) \Longrightarrow \{\Gamma, x' : [\mu X.D/X]D_c \vdash [c\ x'/x]q \rightrightarrows N\}_{\forall c \in D}) \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}'}$$

Figure 3.6: Copattern translation from CBPV$^{\text{copat}}$ to CBPV

main differences are that $c\ v$ and $m.d$ requires us to introduce folds and unfolds.

The case for $\textsf{fun}\ \vec{u}$ requires us to navigate the copattern set to know how to properly split it. Since our coverage algorithm is done by a succession of splitting choices, we will use the coverage derivation for $\vec{u}$ as input to guide the translation. Looking back at the coverage algorithm, our coverage derivation is in fact a list of rules $(\Gamma \vdash q \rightrightarrows N) \Longrightarrow Q$ glued together using the rule c_{extend}. Denote this rule with the infix operator $\langle\langle$, and c_{nil} by the notation $\cdot$, we have that a derivation $\mathcal{D} : (N \lhd\mid Q)$ is in fact a sequence $c_1 \langle\langle\ c_2 \langle\langle\ \ldots \langle\langle\ \cdot$, where c_i is

one of the following copattern splitting rules that we used during coverage.

$$\begin{array}{rcll}
(\Gamma \vdash q \rightrightarrows P \to N) & \Longrightarrow & \{\Gamma, x : P \vdash q@x \rightrightarrows N\} & \mathrm{c_{arr}} \\
(\Gamma \vdash q \rightrightarrows \nu Y.R) & \Longrightarrow & \{\Gamma \vdash q@.d \rightrightarrows [\nu Y.R/Y]R_d\}_{\forall d \in R} & \mathrm{c_{\nu}} \\
(\Gamma, x : P_1 \times P_2 \vdash q \rightrightarrows N) & \Longrightarrow & \{\Gamma, x_1 : P_1, x_2 : P_2 \vdash [(x_1, x_2)/x]q \rightrightarrows N\} & \mathrm{c_{pair}} \\
(\Gamma, x : \mu X.D \vdash q \rightrightarrows N) & \Longrightarrow & \{\Gamma, x' : [\mu X.D/X]D_c \vdash [c\ x'/x]q \rightrightarrows N\}_{\forall c \in D} & \mathrm{c_{\mu}}
\end{array}$$

The translation of $\mathbf{fun}\ (q_i \mapsto m_i)_{\forall i \in I}$ uses the contexts Γ_i obtained from copattern typing to translate each m_i into m_i'. Then, it makes use of the judgment $\mathcal{D} : (N \lhd \mid Q) \gg \mathcal{M} \rightsquigarrow m$ where the inputs are $\mathcal{D}$, the coverage derivation for the copattern set $\{q_i\}_{\forall i \in I}$, and $\mathcal{M}$, the set of terms m_i'. The output is simply a term m in CBPV. We note that we have a one-to-one correspondance between the elements of $\mathcal{M}$ and the elements of Q. This is an invariant of the translation. As we walk up the derivation $\mathcal{D}$, the copattern set Q will shrink and so will $\mathcal{M}$. We assume that we can always identify the term $m_i' \in \mathcal{M}$ that matches the copattern $q_i \in Q$. This could be done in practice using a hashtable. When we get to the empty copattern, there is one term left which is the output we are looking for. It is thus simply returned.

The actual transformation of $\mathcal{M}$ is done by looking at the effect of the rules

$$(\Gamma \vdash q \rightrightarrows N) \Longrightarrow Q$$

on the copattern set. It is introduced by the judgment

$$\mathcal{C} : ((\Gamma \vdash q \rightrightarrows N) \Longrightarrow Q) \gg \mathcal{M} \rightsquigarrow \mathcal{M}'.$$

If we have a copattern q and the rule was $\mathrm{c_{arr}}$, then we introduced a new variable to q, giving us $q@x$. We thus choose the term $m \in \mathcal{M}$ corresponding to q and we replace it in $\mathcal{M}$ by a new term $\lambda x.m$. If the rule $\mathrm{c_{\nu}}$ was used, then we take all terms m_d corresponding to the copatterns $q@.d$ and replace them by a single term $\mathbf{foldc}\ \lambda\{\ldots, d.m_d, \ldots\}$. If we have $\mathrm{c_{pair}}$ used on the copattern q giving us $[(x_1, x_2)/x]q$, then we replace the term m_q with $\mathbf{pm}\ x\ \mathbf{as}\ (x_1, x_2).m_q$. If we have $\mathrm{c_{\mu}}$, then we replace all terms m_c for some $c \in D$ by the single term $\mathbf{pm}\ x\ \mathbf{as}\ \mathbf{foldv}\ x.\mathbf{pm}\ x\ \mathbf{as}\ \langle \ldots, c\ x'.m_c, \ldots \rangle$.

Example 10. Now we compile the cycleNats example to exhibit the process. Let us recall what the function looks like.

```
rec cycleNats. fun x .Head ⇒ x
             | 0     .Tail ⇒ (force cycleNats) 5
             | (s x) .Tail ⇒ (force cycleNats) x
```

The rec-statement is translated as is with the translated body. Each branch is translated as is and so we get the set

$$\mathcal{M} = \{x; (\textsf{force cycleNats})\ 5; (\textsf{force cycleNats})\ x\}$$

Our coverage derivation was $c_\mu \langle\langle\ c_\nu \langle\langle\ c_{\text{arr}} \langle\langle\ \cdot$. The rule c_μ split x into 0 and $s\ x$. We thus replace the terms (**force** cycleNats) 5 and (**force** cycleNats) x in $\mathcal{M}$ by the single term

$$\textbf{pm}\ x\ \textbf{as foldv}\ x.\textbf{pm}\ x\ \textbf{as}\ \langle \textsf{z}.(\textbf{force cycleNats})\ 5, \textsf{s}\ x.(\textbf{force cycleNats})\ x\rangle\ .$$

This gives us the set

$$\mathcal{M} = \{x; \textbf{pm}\ x\ \textbf{as foldv}\ x.\textbf{pm}\ x\ \textbf{as}\ \langle \textsf{z}.(\textbf{force cycleNats})\ 5, \textsf{s}\ x.(\textbf{force cycleNats})\ x\rangle\}$$

The rule c_ν split from a single copattern into two. Thus, $\mathcal{M}$ now contains single term

$$\textbf{foldc}\ \lambda\{\textsf{head}.x, \textsf{tail}.\textbf{pm}\ x\ \textbf{as foldv}\ x.\textbf{pm}\ x\ \textbf{as}\ \langle\ \begin{array}{l}\textsf{z}.(\textbf{force cycleNats})\ 5,\\ \textsf{s}\ x.(\textbf{force cycleNats})\ x\ \rangle\}\end{array}$$

Then, the last rule was c_{arr}. We thus prefix the above term by a lambda abstraction. Since we reach the empty copattern after that, we return this term as is. The translation of the copatttern abstraction thus is the following.

$$\lambda x.\textbf{foldc}\ \lambda\{\textsf{head}.x, \textsf{tail}.\textbf{pm}\ x\ \textbf{as foldv}\ x.\textbf{pm}\ x\ \textbf{as}\ \langle\ \begin{array}{l}\textsf{z}.(\textbf{force cycleNats})\ 5,\\ \textsf{s}\ x.(\textbf{force cycleNats})\ x\ \rangle\}\end{array}$$

The final result, when readding the rec-prefix, is thus the function below.

```
rec cycleNats. λx.foldc
λ{ head.x,
   tail.pm x as foldv x.pm x as
        < z.(force cycleNats) 5,
          s x.(force cycleNats) x>}
```

In order to be able to reason about the relationship between the evaluation in each language, we need to have a translation acting on configurations and substitutions rather than only translate terms. These translations are defined in Figure 3.7. Evaluation contexts are translated pointwise. Values in evaluation contexts are translated using the value translation presented above. Observations are translated by adding to the evaluation context an **unfold** on top of the observation. **to**-statements are translated by simply translating the computation n in them. nil are translated as nil. Configurations $m; K$ are translated by translating both m and K separately and then creating a new configuration out of the translations of each of them. Substitutions are translated pointwise, using the value translation above for each value in the substitution.

$\boxed{\vdash_N K : N' \rightsquigarrow K'}$ Evaluation context K is translated to K'.

$$\frac{\vdash v : P \rightsquigarrow w \quad \vdash_N K : N' \rightsquigarrow K'}{\vdash_N v\ K : P \to N' \rightsquigarrow w\ K'} \qquad \frac{\vdash_N K : [\nu Y.R/Y]R_d \rightsquigarrow K'}{\vdash_N .d\ K : \nu Y.R \rightsquigarrow \textbf{unfold}\ .d\ K'}$$

$$\frac{x : P \vdash n : N' \rightsquigarrow n' \quad \vdash_N K : N' \rightsquigarrow K'}{\vdash_N ([]\ \textbf{to}\ x.n)\ K : \uparrow P \rightsquigarrow ([]\ \textbf{to}\ x.n')\ K'} \qquad \frac{}{\vdash_N \mathsf{nil} : N \rightsquigarrow \mathsf{nil}}$$

$\boxed{\vdash m; K : N \rightsquigarrow m'; K'}$ Configuration $m; K$ is translated to $m'K'$.

$$\frac{\vdash m : N' \rightsquigarrow m' \quad \vdash_N K : N' \rightsquigarrow K'}{\vdash m; K : N \rightsquigarrow m'; K'}$$

$\boxed{\Delta \vdash \sigma : \Gamma \rightsquigarrow \sigma'}$ Substitution σ is translated to σ'.

$$\frac{}{\Delta \vdash \cdot : \cdot \rightsquigarrow \cdot} \qquad \frac{\Delta \vdash v : P \rightsquigarrow v' \quad \Delta \vdash \sigma : \Gamma \rightsquigarrow \sigma'}{\Delta \vdash \sigma, v/x : (\Gamma, x : P) \rightsquigarrow \sigma', v'/x}$$

Figure 3.7: Translation of evaluation contexts, configurations and substitutions

3.3 Translation preserves evaluation

We need to establish some guarantees on the operational behaviour of programs under the translation. We obtain this guarantee by proving that if a term steps to another one, then the translation of the former will step to the translation of the latter. Thus, the evaluation is preserved by the translation.

Since the translation to CBPV introduces new intermediate terms, we need a definition of a multiple steps relation. We define the multiple step relation, denoted $\longrightarrow^*$, as the transitive closure of the stepping relation $\longrightarrow$. It is defined by the rules:

$$\frac{m; K \longrightarrow m'; K'}{m; K \longrightarrow^* m'; K'} \qquad \frac{m; K \longrightarrow^* m_1; K_1 \quad m_1; K_1 \longrightarrow^* m'; K'}{m; K \longrightarrow^* m'; K'}$$

It applies to both stepping relations for CBPV and CBPV$^{\text{copat}}$, respectively.

By design, the translation from CBPV$^{\text{copat}}$ to CBPV makes so a copattern corresponds a path through several branching terms in CBPV leading to a particular branch. We make explicit in the following lemma that if we match against a copattern, then the translation will reach the term at the end of its corresponding branch.

The following lemma states that the evaluation of terms obtained from the copattern set translation will agree with the said translation.

Lemma 3.1. *Suppose $K_1 \rightsquigarrow K_1'$, and $K_2 \rightsquigarrow K_2'$, and $\sigma \rightsquigarrow \sigma'$. If $N \lhd\mid Q \gtrdot \{m_i\} \rightsquigarrow m$ and $q \doteq K_1 \searrow (\sigma; K_2)$ for some $q \in Q$, then $m; K_1' \longrightarrow^* [\sigma']m_i; K_2'$ for some m_i.*

Proof. By induction on the derivation of $\mathcal{D}(N \lhd\mid Q) \gtrdot \{m_i\} \rightsquigarrow m$.

Case $\dfrac{}{\mathcal{D} : (N \lhd\mid \{\cdot \vdash \cdot \rightrightarrows N\}) \gtrdot \{m_1\} \rightsquigarrow m_1}$

We have $\cdot \doteq K_1 \searrow (\cdot; K_1)$ Trivially, $m_1; K_1' \longrightarrow^* [\cdot]m_1; K_1'$.

Case $\dfrac{\mathcal{C} : (\Gamma \vdash q_j \rightrightarrows N') \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}' \quad \mathcal{D}' : (N \lhd\mid (Q \uplus \{q_j\})) \gtrdot \mathcal{M}' \rightsquigarrow m}{\mathcal{D} : (N \lhd\mid Q \cup Q') \gtrdot \mathcal{M} \rightsquigarrow m}$

If $q \in Q$, then $m_i \in \mathcal{M}'$. Our induction hypothesis gives us $m; K_1' \longrightarrow^* [\sigma']m_i; K_2'$ which is what we wanted.

If $q \in Q' \backslash Q$, we do a nested case analysis on $\mathcal{C} : ((\Gamma \vdash q_j \rightrightarrows N')) \gtrdot \mathcal{M} \rightsquigarrow \mathcal{M}'$.

Subcase $\dfrac{\mathcal{M}' = (\mathcal{M}\backslash\{m_q\}) \cup \{\lambda x.m_q\}}{c_{\text{arr}} : ((\Gamma \vdash q_j \rightrightarrows P \to N) \Longrightarrow \{\Gamma, x : P \vdash q_j@x \rightrightarrows N\}) \gg \mathcal{M} \rightsquigarrow \mathcal{M}'}$

Since $Q' = \{q_j@x\}$, we have by assumption $q_j@x \doteq K_1 \searrow (\sigma, v/x; K_2)$ for some v. By Lemma 2.9, $q_j \doteq K_1 \searrow (\sigma; v\ K_2)$. Our induction hypothesis thus is $m; K_1' \longrightarrow^* [\sigma']\lambda x.m_i; v'\ K_2'$ where $v \rightsquigarrow v'$. Since $\lambda x.m_i; v'\ K_2' \longrightarrow [v'/x]m_i; K_2'$, we have $m; K_1 \longrightarrow^* [\sigma', v'/x]m_i; K_2'$.

Subcase $\dfrac{\mathcal{M}' = (\mathcal{M}\backslash\{m_d\}_{\forall d \in R}) \cup \{\textsf{foldc}\ \lambda\{\ldots, d.m_d, \ldots\}\}}{c_\nu : ((\Gamma \vdash q \rightrightarrows \nu Y.R) \Longrightarrow \{\Gamma \vdash q@.d \rightrightarrows [\nu Y.R/Y]R_d\}_{\forall d \in R}) \gg \mathcal{M} \rightsquigarrow \mathcal{M}'}$

Since $Q' = \{q_j@.d\}_{\forall d \in R}$, we must have $q_j@.d \doteq K_1 \searrow (\sigma; K_2)$ for some $d \in R$. By Lemma 2.9, $q_j \doteq K_1 \searrow (\sigma; .d\ K_2)$. Hence, our induction hypothesis is

$$m; K_1' \longrightarrow^* [\sigma'](\textsf{foldc}\ \lambda\{\ldots, d.m_d, \ldots\}); \textsf{unfold}\ .d\ K_2'.$$

Since

$$(\textsf{foldc}\ \lambda\{\ldots, d.m_d, \ldots\}); \textsf{unfold}\ .d\ K_2' \longrightarrow^* m.d; K_2',$$

we have $m; K_1' \longrightarrow^* [\sigma']m.d; K_2'$.

Subcase

$$\frac{\mathcal{M}' = (\mathcal{M}\backslash\{m_q\}) \cup \{\textsf{pm}\ x\ \textsf{as}\ (x_1, x_2).m_q\}\}}{c_{\text{pair}} : ((\Gamma, x : P_1 \times P_2 \vdash q \rightrightarrows N) \Longrightarrow \{\Gamma, x_1 : P_1, x_2 : P_2 \vdash [(x_1, x_2)/x]q \rightrightarrows N\}) \gg \mathcal{M} \rightsquigarrow \mathcal{M}'}$$

We have $[(x_1, x_2)/x]q_j \doteq K_1 \searrow (\sigma, v_1/x_1, v_2/x_2; K_2)$. By Lemma 2.9, $q_j \doteq K_1 \searrow (\sigma, (v_1, v_2)/x; K_2)$. Hence, we can use the induction hypothesis

$$m; K_1' \longrightarrow^* [\sigma', (v_1', v_2')/x](\textsf{pm}\ x\ \textsf{as}\ (x_1, x_2).m_q); K_2'.$$

We have

$$[\sigma'][(v_1', v_2')/x](\textsf{pm}\ x\ \textsf{as}\ (x_1, x_2).m_q); K_2' = [\sigma'](\textsf{pm}\ (v_1', v_2')\ \textsf{as}\ (x_1, x_2).m_q); K_2'$$

since x doesn't occur in m_q and v_1' and v_2' are closed. Thus,

$$(\textsf{pm}\ (v_1', v_2')\ \textsf{as}\ (x_1, x_2).m_q); K_2' \longrightarrow [v_1'/x_1, v_2'/x_2]m_q; K_2'.$$

It follows that $m; K_1' \longrightarrow^* [\sigma'][v_1'/x_1, v_2'/x_2]m_q; K_2' = [\sigma', v_1'/x, v_2'/x_2]m_q; K_2'$.

Subcase

$$\frac{\mathcal{M}' = (\mathcal{M} \backslash \{m_c\}_{\forall c \in D}) \cup \{\textsf{pm}\ x\ \textsf{as foldv}\ x.\textsf{pm}\ x\ \textsf{as}\ \langle \dots, c\ x'.m_c, \dots \rangle\}\}}{c_\mu : ((\Gamma, x : \mu X.D \vdash q \rightrightarrows N) \Longrightarrow \{\Gamma, x' : [\mu X.D/X]D_c \vdash [c\ x'/x]q \rightrightarrows N\}_{\forall c \in D}) \gg \mathcal{M} \rightsquigarrow \mathcal{M}'}$$

This case is similar to the last one and is left to the reader. □

Theorem 3.2. *Suppose* $m_1; K_1 \rightsquigarrow m_1'; K_1'$ *and* $m_2; K_2 \rightsquigarrow m_2'; K_2'$. *If* $m_1; K_1 \longrightarrow^* m_2; K_2$, *then* $m_1'; K_1' \longrightarrow^* m_2'; K_2'$.

Proof. We prove it by induction on the stepping derivation.

Case $\dfrac{m_1; K_1 \longrightarrow^* m_3; K_3 \quad m_3; K_3 \longrightarrow^* m_2; K_2}{m_1; K_1 \longrightarrow^* m_2; K_2}$

Let $m_3; K_3 \rightsquigarrow m_3'; K_3'$. By induction, we have $m_1'; K_1' \longrightarrow^* m_3'; K_3'$ and $m_3'; K_3' \longrightarrow^* m_2'; K_2'$ and so $m_1'; K_1' \longrightarrow^* m_2'; K_2'$.

Case $\dfrac{m_1; K_1 \longrightarrow m_2; K_2}{m_1; K_1 \longrightarrow^* m_2; K_2}$

This case is proved by nested case analysis on the single step judgment. Most of the cases are really simple and and translate into a one step derivation. From those, we only show the case for **to**-statements. We also present the case for observations which creates an additional intermediate step and the case for copattern abstractions.

Subcase $m\ \textsf{to}\ x.n; K \longrightarrow m; (\square\ \textsf{to}\ x.n)\ K$

Then, the left-hand side translates to $m'\ \textsf{to}\ x.n'; K'$ where $m \rightsquigarrow m'$, and $n \rightsquigarrow n'$ and $K \rightsquigarrow K'$. The right-hand side translates to $m'; (\square\ \textsf{to}\ x.\rightsquigarrow n')\ \rightsquigarrow K'$. But the stepping rule in CBPV is

$$m'\ \textsf{to}\ x.n'; K' \longrightarrow m'; (\square\ \textsf{to}\ x.n')\ \rightsquigarrow K'$$

Subcase $m.d; K \longrightarrow m; .d\ K$

The left-hand side translates to $(\textsf{unfold}\ m').d; K'$ where $m \rightsquigarrow m'$ and $K \rightsquigarrow K'$. The right-hand side becomes $m'; \textsf{unfold}\ .d\ K'$. Then

$$(\textsf{unfold}\ m').d; K' \longrightarrow \textsf{unfold}\ m'; .d\ K' \longrightarrow m'; \textsf{unfold}\ .d\ K'$$

Subcase $\dfrac{q_i \doteq K_1 \searrow (\sigma; K_2)}{\textsf{fun}\ (q_i \mapsto m_i); K_1 \longrightarrow [\sigma]m_i; K_2}$

By assumption, we have that for $Q = \{q_i\}_{\forall i}$, $N \lhd| Q$ and so we have

$$\mathcal{D} : (N \lhd| Q) \gtrdot \{m_i\}_{\forall i} \rightsquigarrow m.$$

Let $m_i \rightsquigarrow m_i'$ and $K_1 \rightsquigarrow K_1'$ and $K_2 \rightsquigarrow K_2'$. Then, by Lemma 3.1, $m; K_1' \longrightarrow^* [\sigma']m_i'; K_2'$. $\square$

Chapter 4

Conclusion

This thesis presented CBPV$^{\text{copat}}$, an extension of call-by-push-value with copatterns. Call-by-push-value is a language designed by Levy [2001] to subsume the call-by-value and call-by-name semantics by making explicit choices for the evaluation order using the term structure. Copatterns are a notion introduced in previous joint work [Abel et al., 2013] which defines and eliminites recursive and corecursive data types in a symmetric fashion by making use of the dual nature of their categorical counterparts: initial albegras and terminal coalgebras, respectively. Since CBPV supports (co)recursive definitions, our extension to copatterns moves away from a basic folding and case analysis style of programming to a notion of deep (co)pattern matching more suitable for a high level syntax.

We defined a non deterministic coverage algorithm inspired by the splitting mechanisms offered by interactive modes of languages such as Agda [Norell, 2007] or Idris [Brady, 2013]. Our algorithm starts with an empty copattern which covers any type and preserves coverage at each step.

We proved subject reduction which states that types are preserved by the evaluation relation. We also defined a notion of safety through a type-directed recursive predicate which carries the concept that terms don't get stucks. We proved that every well-typed term is safe and so every well-typed term will progress.

We defined a translation from CBPV$^{\text{copat}}$ into CBPV. This translation makes use of

the coverage judgment used to define copattern abstractions to split them into Levy's core constructs such as case analysis, records and lambda abstractions. We prove that this relation preserves evaluation which ensures that the translation is well behaved with respect to the language's semantics.

4.1 Future work

Our long term goal for this line of work is to define an adequate notion of coinduction for dependently typed languages. We believe copatterns to be a suitable candidate for this endeavor. In order to get to this point, we have several steps in mind.

We intend to mechanize our copattern language and of its successive extensions towards dependent types in a proof assistant such as Agda or Beluga [Pientka, 2008, Cave and Pientka, 2012]. The latter is particularly adapted for language mechanizations due to its intrinsic support for contexts and bindings.

We wish to define a categorical semantics for the productive fragment of our copattern language and for dependently typed languages in order to get insights of how the duality between induction and coinduction scales to dependently typed programming.

We already have defined in joint work with Andrew Cave and Brigitte Pientka an extension of CBPV$^{\text{copat}}$ with indexed types [Thibodeau et al., 2015] that can be used to define invariants about our (co)recursive definitions. We wish to extend this language to a proof language using (co)inductive predicates qualifying the index domain. Using a logical system as the index domain (such as the logical framework LF [Harper et al., 1993]) would allow us to write coinductive proofs about this system. This requires us to define a notion of productivity checking for indexed types. We thus plan to extend the work by Abel and Pientka [2013] to our indexed types setting.

In particular, we want to extend the Beluga language with indexed copatterns as it is a powerful indexed language specialized in meta proofs about languages. We desire to write a prototype implementation of coinductive Beluga in order to put our theory into application.

Further, we plan on defining an extension of Martin-Löf type theory with dependent

copatterns as a full blown proof assistant to program with and reason about coinductive definitions.

For each iteration of our copattern language, we intent to perform case studies using examples on coinductive objects and proofs such as bisimulations, type systems with self referential closures, divergence of lambda terms, etc.

Bibliography

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.

T. Altenkirch and N. A. Danielsson. Termination checking in the presence of nested inductive and coinductive types. note supporting presentation given at the workshop on partiality and recursion in interactive theorem provers. In *In Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications, LDTA 2009*, pages 18–33, 2009.

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

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013. ISSN 1469-7653.

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

A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. URL `http://adam.chlipala.net/cpdt/`. `http://adam.chlipala.net/cpdt/`.

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.

W. R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, UK, 1991. Springer-Verlag.

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.

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

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, 1987a.

T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987b. AAID-80470.

T. Hagino. Codatatypes in ML. *Journal of Symbolic Computation*, 8(6):629–650, Dec. 1989. ISSN 0747-7171.

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

B. Jacobs. Objects and classes, coalgebraically. In *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ, 1995.

D. Kimura and M. Tatsuta. Dual calculus with inductive and coinductive types. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), Brasília, Brazil*, pages 224–238, 2009.

N. R. Krishnaswami. Focusing on pattern matching. In *Proc. of the 36th ACM Symp. on Principles of Programming Languages, POPL 2009*, pages 366–378, 2009.

N. R. Krishnaswami. Focusing is not call-by-push-value. Post on his blog http://semantic-domain.blogpost.ca, Oct. 2014.

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.

R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

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

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

S. Peyton-Jones. *Haskell 98 language and libraries : the revised report*. Cambridge University Press, 2003. ISBN 9780521826143.

B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

D. Thibodeau, A. Cave, and B. Pientka. Indexed codata types. Draft, July 2015.

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

N. Zeilberger. Focusing and higher-order abstract syntax. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 359–369, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9.

N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-matching*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009. AAI3358066.