An Intensional Type Theory of Coinduction using Copatterns

David Thibodeau
School of Computer Science
Mcgill University, Montreal

December 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Abstract

Reasoning about well-founded programs and finite data is well-understood nowadays. On the other hand, representation and reasoning of interactive or constantly-running systems which exhibit an infinite structure lack fairly behind. This thesis focuses on bridging this gap by proposing a language allowing both representations and reasoning about finite and infinite data alike. To do so, we build on our recent idea of copatterns, a syntactic device to represent infinite computation through observations, and extend it into an indexed type system. Indexed types allow us to encode invariants in types using indices from a decidable domain. In our setting, the index domain is left abstract and we present a general framework for admitting valid domains. We also design a coverage algorithm and use it to prove type preservation and progress. Moreover, we establish termination criteria for programs in our language. We prove the validity of those criteria by showing them sufficient to a normalisation preserving translation from our indexed copattern language to a core calculus with (co)recursors. This calculus is shown to be normalizing by a logical relation argument. We showcase the use of indexed copatterns by implementing Howe's method as a case study in a prototype implementation of copatterns in the proof assistant Beluga.

Résumé

Raisonner au sujet des programmes et données d'ordre fini est bien compris de nos jours. Par opposition, la représentation et le raisonnement de processus interactifs ou à exécution constante dont le comportement s'apparente à des structures infinies ne sont pas aussi bien compris. Cette thèse vise à améliorer notre compréhension en proposant un langage permettant la représentation et le raisonnement de données à la fois finies et infinies. Ce langage bâti sur le concept de comotifs, un élément syntactique permettant la représentation de programmes infinis à travers un processus d'observations. Cette thèse offre une version des comotifs adaptée au systèmes de types annotés. Les types annotés permettent aux programmeurs d'encoder des invariants dans les types en utilisant des annotations provenant de domaines décidables. Dans notre théorie, les domaines sont abstraits et nous présentons une infrastructure pour valider l'utilisation de domaines particuliers. De plus, cette thèse décrit un algorithme de couverture et l'utilise de façon à prouver la préservation des types et le progrès des programmes. Nous établissons aussi des critères de terminaison pour les programmes de notre langage. Nous démontrons la validité de ces critères à travers une traduction préservant la normalisation de notre langage à comotifs vers un calcul ayant accès à des (co)récurseurs. La démonstration de la normalisation de ce calcul est faite grâce à un argument de candidats de réductibilité. Nous avons implémenté un prototype des comotifs annotés dans l'assistant de preuves Beluga et nous faisons usage de ce prototype pour implémenter la méthode de Howe.

Acknowledgements

I first want to thank my advisor Brigitte Pientka who supported me over the last nine years and taught me so much about being a researcher. Prakash Panangaden was always there when I had questions and his curiosity on any new subject was inspiring. Andreas Abel guided me on the project of copatterns at its very beginning and was very patient and positive despite my struggles.

Francisco Ferreira is always there to help whenever I need it and I did need it often. We shared ideas, we programmed together, we had many laughs. Andrew Cave always seemed to know the answers, and more crucially would always ask the right questions to pinpoint where things went wrong. He helped shape how I approach and solve problems. Jacob Errington kept me sane through the last years of this long adventure. I used him sometimes as a rubber duck but he would always listen and engage. There were so many more people who were part of the Complogic lab in the last nine years and who helped me in one way or another: Stefan Knudsen, Rohan Jacob-Rao, Shawn Otis, Steven Thephsourinthone, François Thiré, Agata Murawska, Aïna Linn Georges, Tao Xue, Mathieu Boespflug, Olivier Savary Belanger, Matthias Puech, Aliya Hameer, Jason Hu, Hanneli Andreazzi Tavante, ...

In addition, my friends and my family always were so supportive of my work and always there for me. They might not have understood the details of my research, but it did not matter. Without them I would not have made it this far.

I thank my thesis defence committee for all their time and all their interesting questions: Prakash Panangaden, Clark Verbrugge, Stefan Monnier, and Hamed Hatami. I also want to give special thanks to my external reviewer Herman Geuvers.

Finally, I would not have been able to complete my PhD and write this thesis without the financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Alexander Graham Bell Canada Graduate Scholarship and from McGill University through the Lorne Trottier Fellowship and Lorne Trottier Science Accelerator Fellowship.

Contribution of the Author

- Chapter 2 is based on [Thibodeau et al., 2016] where I am the first author and I developed the ideas under the supervision of my co-authors.
- The calculus and proof of normalisation of Sections 3.1 and 3.2 in Chapter 3 are loosely based on [Jacob-Rao et al., 2018]. My contributions to this publication was to extend the calculus and the proofs to a coinductive setting.
- Chapter 4 is taken from [Momigliano et al., 2019]. My contribution was to extend Beluga to index copatterns and to build the mechanization of the proof (Section 4.2) under the supervision of my co-authors. The development of Section 4.2 was done by Alberto Momigliano.

Contents

List of Figures

Chapter 1

Introduction

Nowadays, computer systems are integral to our modern society. They are in our phones; they handle our money; they control the chain of production. Unlike in the older days where computers would be used to perform a single task to completion such as a calculation, modern infrastructures are ever running complex systems which perform critical tasks often concurrently. The importance of such systems requires us to rely on the correctness of their behaviour: we do not want our private messages to be sent to the wrong person, our banks to miscount our money, or our planes to crash. Correctness cannot be simply assumed. It is critical that those systems be verified.

Verification comes in many forms. Testing is a very common and powerful method, but only offers partial guarantees. It only covers cases that are tested and most of the time it is not possible to test all cases. Alternatively, proof theory allows us to directly prove properties about systems. For complex systems, this is a difficult task. It requires one to describe the large amount of specifications and behaviours of those systems and take on intricate proofs that they indeed satisfy the invariants.

Type theory strikes a balance which allows us to encode specific properties directly as part of the code. Those are proven in the code itself and checked statically when typing the programs. Thus, there are no corner cases that could fall off the cracks as is the case in testing. It is modular enough that you can target specific invariants within the code without

having to specify everything. As such, it scales better to larger systems than full correctness proofs.

Some questions that arise from using such a system for proving invariants are "what are the methods of proof needed to prove such invariants?" and "how can we encode such methods in a programming language so that the type system can automatically check them?". There is not a single answer that fits all invariants. There are nevertheless some general classifications we can make.

Systems such as traditional batch programs have a finite behaviour in nature. They have a starting point and an ending after finitely many steps. This finite behaviour also exists in many data structures such as natural numbers, lists, and trees. Finite structures are very common in computer science due to the finite limitation of time and space. Finite structures and properties can be defined inductively: we define some base cases and, given some existing objects, we can build new ones. For example, lists are built from an empty list and a cons operator that adds an element to the front of an existing list. Reasoning over those is done by the principle of induction: it suffices to show that what we are trying to prove holds on the base cases, and that if it holds on the existing objects, such as the smaller list l, it also holds for the ones we build from those such the list $\mathsf{cons}\ (a, l)$.

Most modern systems however behave very differently: we do not want them to stop but we rather want them to continue indefinitely, always ready to answer the next query. They are our operating systems, our webservers, our streams of messages. Them stopping to respond is a bug rather than a feature. Queries answered by those systems can be modeled as never ending streams of states. A property over them can thus also be seen as infinite in nature as it needs to be proven for every single of those states.

The method to encode and reason about such infinite structures is coinduction, the dual of induction. This duality has a clear semantic intuition that we go over in Section 1.1, but its syntactic representation in the context of type theory is still an open question. This is because we are trying to fit infinite structures into systems that are operate under finite time and space constraints.

Providing a syntactic representation of coinduction in the context of (indexed) type

theory is the focus of this thesis. Specifically, we build and showcase an ML like language with support for indexed (co)inductive types through a syntactic device called copatterns and prove metaproperties about this language and its features.

The rest of this chapter is organized as follows:

- We describe the duality of induction and coinduction and thus provide a semantic definition of coinduction;
- We present an overview of existing languages featuring coinduction;
- We explain what copatterns are and how they work;
- We discuss the details of this thesis' contributions and how the rest of this thesis is organized.

1.1 Coinduction

The first recorded uses of coinduction were by Park [1979, 1981] and Milner [1982] in the context of concurrency. It was used in particular to reason about fair merges of sequences. The name of coinduction was first used by Milner and Tofte [1991] who described a proof method to study semantics of programming languages. It was built on a semantic understanding of the principle of induction from domain theory. We provide an overview of its definition.

Semantic Intuition of Coinduction

Suppose that $\mathcal{L}$ is a complete lattice, that is, a partially ordered set (with ordering $\leq$) in which all subsets have both a supremum and an infimum. It follows that $\mathcal{L}$ itself has both a least and greatest element (respectively denoted $\bot$ and $\top$, respectively). The usual example is the power set of a set A (denoted $\mathcal{P}(A)$) under subset inclusion. The supremum is simply the union of all elements of a subset B of $\mathcal{P}(A)$, while the infimum is simply the intersection of all elements of B. The least element of $\mathcal{P}(A)$ is simply $\emptyset$ while its greatest element is A itself.

A function $\mathcal{F} : \mathcal{L} \to \mathcal{L}$ is said to be monotone if it preserves $\leq$, that is, for all x and y in $\mathcal{L}$, $x \leq y$ implies $\mathcal{F}(x) \leq \mathcal{F}(y)$. We have by (possibly transfinite) induction on n that $\mathcal{F}^{n-1}(\bot) \leq \mathcal{F}^n(\bot)$. In the base case, $\bot \leq \mathcal{F}(\bot)$ since $\bot$ is the least element of $\mathcal{L}$. In the inductive case, if $\mathcal{F}^{n-2}(\bot) \leq \mathcal{F}^{n-1}(\bot)$, then by monotonicity of $\mathcal{F}$, we have $\mathcal{F}^{n-1}(\bot) \leq \mathcal{F}^n(\bot)$. This leads us to conclude that we have the chain.

$$\bot \leq \mathcal{F}(\bot) \leq \mathcal{F}(\mathcal{F}(\bot)) \leq \cdots \leq \mathcal{F}^n(\bot) \leq \ldots$$

By Kleene's fixed-point theorem, the supremum of this chain is the least fixed-point of $\mathcal{F}$, denoted $\mu\mathcal{F}$.

Let us look at an example. Let Ω be the set of all terms in our programming language (as generated by our grammar). Suppose $\mathcal{L} = \mathcal{P}(\Omega)$. Let $\mathcal{F}(X) = 1 + X$, that is, the disjoint union of the one element set 1 and the input set X. Then, our Kleene chain is

$$\begin{aligned}\emptyset \subset \{\mathtt{in}_1\,()\} \subset \{\mathtt{in}_1\,(), \mathtt{in}_2\,(\mathtt{in}_1\,())\} \subset \{\mathtt{in}_1\,(), \mathtt{in}_2\,(\mathtt{in}_1\,()), \mathtt{in}_2\,(\mathtt{in}_2\,(\mathtt{in}_1\,()))\} \\ \subset \cdots \subset \{\mathsf{in}_2^i(\mathtt{in}_1\,()) \mid i = 1, \ldots, n\} \subset \ldots\end{aligned}$$

The supremum of that chain is $\{\mathsf{in}_2^i\ (\mathtt{in}_1\,()) \mid i \in \mathbb{N}\}$, where $\mathsf{in}_2^0\ (\mathtt{in}_1\,())$ is simply $\mathtt{in}_1\,()$. If we define `zero` to be $\mathtt{in}_1\,()$ and `suc` to be $\mathtt{in}_2$, then the least fixed point is the set of all natural numbers.

This is the idea behind inductive datatypes. The operator $\mathcal{F}$ describes the constructors of the type and the set of terms that can be built with those constructors are terms existing in its least fixed point.

Given a complete lattice $\mathcal{L}$ of a set A with ordering $\leq$, we define the lattice $\mathcal{L}^{\mathrm{op}}$ as the set A together with the ordering $\geq$, that is, for all $a, b \in A$ if $a \leq b$, then $b \geq a$. Clearly, the suprema and infima of $\mathcal{L}$ are infima and suprema of $\mathcal{L}^{\mathrm{op}}$, respectively. It follows that $\mathcal{L}^{\mathrm{op}}$ is also a complete lattice. A monotone operator $\mathcal{F} : \mathcal{L} \to \mathcal{L}$ is also a monotone operator $\mathcal{F} : \mathcal{L}^{\mathrm{op}} \to \mathcal{L}^{\mathrm{op}}$. We thus can build the chain

$$\top \geq \mathcal{F}(\top) \geq \mathcal{F}(\mathcal{F}(\top)) \geq \cdots \geq \mathcal{F}^n(\top) \geq \ldots$$

By Kleene's fixed point theorem, the least fixed point of $\mathcal{F}$ is the supremum of this chain. This chain is still valid under the original ordering, with the infimum $\nu\mathcal{F}$ being the greatest fixed point of $\mathcal{F}$.

Going back to our previous example, we can build the chain

$$\Omega \supset (\{\mathtt{in}_1\,()\} \cup \{\mathtt{in}_2\,x \mid x \in \Omega\}) \supset (\{\mathtt{in}_1\,(), \mathtt{in}_2\,(\mathtt{in}_1\,())\} \cup \{\mathtt{in}_2\,(\mathtt{in}_2\,x) \mid x \in \Omega\})$$
$$\supset \cdots \supset \left(\{\mathtt{in}_1\,()\} \cup \{\mathsf{in}_2^i\ (\mathtt{in}_1\,()) \mid i = 1, \ldots, n-1\} \cup \{\mathsf{in}_2^n\ x \mid x \in \Omega\}\right) \supset \ldots$$

The infimum of this chain is the set $\{\mathsf{in}_2^i\ (\mathtt{in}_1\,()) \mid i \in \mathbb{N}\} \cup \{\mathsf{in}_2^\infty\ \}$, that is, the set of all natural numbers, together with the smallest infinite ordinal ω. Those can be described as the coinductive natural numbers.

Coinduction as a Programming Language Feature

Since the world of programming languages and type theories is vast, it would be difficult to give an exhaustive account. As such, we limit our discussion here to a few we deem important. The first occurrences of coinduction in programming languages seem to have been developed independently by Mendler et al. [1986] and Hagino [1987, 1989] by building upon its duality with induction, either from a fixed point semantics, or from an equivalent categorical semantics. We also describe how coinduction is defined in both Coq and Isabelle/HOL

Infinite Objects in NuPRL

Mendler et al. [1986] extends NuPRL with coinduction by defining types $\Box$, which corresponds to Ω in our interpretation above, and $\inf(x.c)$ where x is a variable binding and c is a term which may contain x. The latter is inhabited by a term e if e inhabits approximations $(\lambda x.c)^n(\Box)$ for all n. That is, the infima of our Kleene chain. They also establish that $\inf(x.c)$ is extensionally equal to $[\inf(x.c)/x]c$.

Codatatypes in ML

While Mendler et al. [1986] built coinduction by encoding Kleene chains in their language, Hagino [1987, 1989] designed a syntactic description for coinductive types that was dual to the syntax of inductive types in ML. An inductive type (or datatype) is defined as a series

of constructors:

$$\text{datatype } T = c_1 \text{ of } A_1 \mid \cdots \mid c_n \text{ of } A_n$$

Thus, each constructor c_i acts as a function from $A_i \to T$. If duality in domain theory was to reverse the ordering relation, we understand categorical duality by reversing the direction of the arrows. That would give us a function $T \to A_i$ that can be seen as a destructor of T into type A_i. Coinductive types (called codatatypes by Hagino) are thus defined as:

$$\text{codatatype } T = d_1 \text{ is } A_1 \ \&\ \ldots\ \&\ d_n \text{ is } A_n$$

This definition basically generates recursive records. For example, we can define the type of streams as

$$\text{codatatype } \mathsf{Stream} = \mathsf{head} \text{ is } \mathsf{Nat} \ \&\ \mathsf{tail} \text{ is } \mathsf{Stream}$$

Analyzing datatypes is done through case statements:

```
case t of
  | c1 x1 ⇒ t1
  | c2 x2 ⇒ t2
  ...
```

`t` evaluates to a term `c_i v_i` and the corresponding branch is taken resulting in the term `[v_i/x_i]t_i`. For codatatypes, we build them using merge statements:

```
merge
  & d1 ⇐ t1
  & d2 ⇐ t2
  ...
```

Building a stream simply requires us to provide a head and a tail.

```
fun zeroes = merge
  & head ⇐ zero
  & tail ⇐ zeroes
```

To prevent infinite unfolding in case of recursion, the body of a merge statement is only reduced when applied to a destructor.

Guarded Induction in Coq

Coquand [1994] and Giménez [1995] view inductive types as well-founded trees whose nodes are defined by constructors. Coinductive types are simply viewed as trees for which the well-foundedness requirements are relaxed. Constructors still govern how we define coinductive terms but they are restricted in how they are unfolded. In particular, they are only unfolded lazily. As such, constructors form guards that ensure productivity of terms. This is the approach followed by the Coq proof assistant [INRIA, 2020].

For example, we can define streams with the single constructor `cons : Nat → Stream → Stream`. Coinductive terms can be defined through corecursive definitions using **CoFixpoint**. The stream of zeroes would thus look like the following:

```
CoFixpoint zeroes : Stream := cons 0 zeroes.
```

The laziness is obtained by only allowing reduction of cofixed points under case analysis, as case analysis needs to know which constructor is at the head position. Cofixed point definitions are deemed wellfounded if their corecursive calls are done under a constructor such as `cons`. The reduction rules for a stream then works as follows:

```
match zeroes with (cons x xs) ⇒ t end
      ⇝ match (cons 0 zeroes) with (cons x xs) ⇒ t end

match (cons 0 zeroes) with (cons x xs) ⇒ t end
      ⇝ t[0/x,zeroes/xs]
```

While such reductions ensure programs using lazy definitions do not unfold indefinitely, it fails to satisfy subject reduction [Giménez, 1996, Oury, 2008], the property that evaluation preserves types. We go over the argument leading to its failure below.

(Co)datatypes in Isabelle/HOL

Isabelle/HOL [Nipkow et al., 2002] takes on a definitional approach to development of extensions and libraries. That is, they are defined from existing constructs instead of added as new axioms. There have been several approaches to coinduction in HOL over the years [Paulson,

1994, Hausmann et al., 2005], we only focus on the most recent body of work on bringing codatatypes to Isabelle/HOL [Traytel et al., 2012, Blanchette et al., 2014].

Blanchette et al. [2014] describe a new library for coinduction in Isabelle/HOL in which coinductive terms are built using constructors (such as `nil` and `cons` for lazy lists). However, manipulation of terms is done via existing terms via discriminators and observations. Discriminators are boolean functions to check whether a term has a specific principal constructor (e.g. whether the list is empty or not). Observations are defined for subterms under specific constructors (such as `head` and `tail` for non-empty lists).

Functions producing potentially infinite codatatype terms are built via primitive corecursion which can only be performed under constructors. One cannot pattern match over constructors of codatatypes, but can only use discriminators and observations to obtain informations about them. The library does allow for conditionals on the left-hand side of functions to facilitate writing them. A user can thus specify the conditions for which a list is empty, and what are the values supplied for the head and the tail in the case where it is not empty. This generalizes merge definitions from Hagino [1987, 1989].

They assert equality for coinductive terms via a coinduction rule. Two coinductive terms s and t are deemed equal if there is a relation R such that $s\ R\ t$ and R is a bisimulation. That is, for all t_1 and t_2 such that $t_1\ R\ t_2$, we have either t_1 and t_2 are both empty, or $t_1.\mathsf{head} = t_2.\mathsf{head}$ and $t_1.\mathsf{tail}\ R\ t_2.\mathsf{tail}$. Bisimulation generalizes definitional equality to infinite objects by stipulating they behave the same, when observed. While traditional approaches for coinduction in Isabelle/HOL require the user to provide a specific relation R manually, Blanchette et al. [2014] are able in many cases to automatically generate a canonical relation.

The Failure of Coinduction in Coq

The failure of Coq is one of the main inspirations of the body of work leading to this thesis: offering a strong candidate for coinduction in intensional type theories. Those type theories rely on a weak equality limited to terms sharing a same normal form. This equality was one of the important factor leading to its success: equalities can be computed automatically. However, it also imposes severe limitations that sometimes are hard to bypass. For example,

it does not allow function extensionality and thus would not allow NuPRL's definition of coinduction. It is also the cause of the failure to preserve types.

Let us define a type `U` with a single constructor **in** taking a `U` as its single argument.

```
CoInductive U : Type :=
| in : U → U.
```

The only inhabitant of this type is thus defined by the cofixed point

```
CoFixpoint u : U := in u.
```

We shall build a term `eq_u : u ≡ in u`.

Here, $\equiv$ refers to the propositional equality. Propositional equality is an inductive relation defined via a single constructor `refl` which stands for reflexivity and thus states that $x \equiv x$ for any x. In a closed context, propositional equality will behave exactly the same as intensional equality as any proof of equality will reduce to reflexivity. It is nevertheless a powerful tool as it allows us to reason directly with equalities and thus prove complicated equations that are not obvious from reductions alone.

The key part of the failure of subject reduction lies in this term `eq_u`. We will build it in such a way that it evaluates to `refl`. However, `refl` has type $x \equiv x$, which is not the same equality as `u ≡ in u` since `u` itself not equal to `in u` definitionally. Thus, the type of `eq_u` is not preserved under the reduction.

Let us now build `eq_u`. First, we need to be able to force unfolding of `u`. This is done through a `force` operation which simply uses case analysis.

```
Definition force (x : U) : U :=
match x with in y ⇒ in y
end
```

Since `force` places `x` under a case analysis, it will be unfolded and matched against `in y`. From this, we can build a proof term of type $\forall$`x:U.x ≡ force x` as follows:

```
Definition eq (x : U) : x ≡ force x :=
match x with in y ⇒ refl
end
```

Now, **Definition** `eq_u : u ≡ force u := eq u`, which reduces to `refl` while its type reduces to `u ≡ in u`, as described above.

There is no easy way to fix definitional equality to save subject reduction. It has been argued by McBride [2009] and subsequently formalized by Berger and Setzer [2018] that any equality admitting unfolding of codata is undecidable. Decidable equality is an important property on which a dependent type theory such as Coq's is relying to ensure that type checking itself is decidable: since Coq allows terms to inhabit types, type checking needs to be able to compare terms for equality.

We noted above that NuPRL's treatment of coinduction treated unfoldings as equal. NuPRL's equality is undecidable by very design; they admit an equality reflection rule allowing equalities proven propositionally to be used definitionally. By contrast, in Coq, one must use explicit conversions when facing propositional equalities. However, this allows them to handle conversions of definitional equality automatically via reductions. In a similar vein, one cannot convert a bisimulation into equality in Coq as it is the case in Isabelle/HOL, but would have to prove a substitutivity property for a given bisimulation.

Thus, the problem of coinduction in Coq lies in the paradigm of defining codata using constructors mixed with dependent pattern matching which naturally refines types based on the information learned by the unfolding. A natural solution is to find shelter in the dual paradigm: definition by means of observations.

Copatterns

Pattern matching offers more than just an induction principle for inductive data; it allows simultaneous and deep matching and is quite a powerful abstraction. On the other hand, coinduction defined by destructors (or observations) as described by Hagino [1987, 1989] is in contrast pretty low level. Observation matching takes the form of a simple record indicating the value at each observation. If multiple observations are needed to be applied one after the other, then several nested record definitions need to be provided. To offer a convenient high level abstraction, we invented copatterns [Abel et al., 2013, Abel and Pientka, 2013, 2016, Setzer et al., 2014, Thibodeau et al., 2016]. The interest in this approach has already

started to grow; support for copatterns has been added to the proof assistant Agda [Norell, 2007] in version 2.3.4 [Agda team, 2014] and has since then become the default way to handle coinduction. An extension to copatterns was also added to OCaml by Laforgue and Régis-Gianas [2017].

Despite its name, copatterns are not the dual of patterns but instead subsume both patterns and observations. We define coinductive types by observations; streams are made of a `head` and a `tail`. Definitions of coinductive terms still are done by supplying a term for each observation. However, the machinery we provide to build such term is more advanced.

Let us introduce copatterns by means of interactive program development. Say we want to build a stream of natural numbers starting at a number of our choice and decreasing until it reaches 0. When it does, it goes to a constant, say 5, and decreases again to zero, and so on. Thus, the stream might look like the following:

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

Now, this stream `cycleNats` has type `Nat → Stream` where the input denotes the starting number.

```
fun cycleNats : Nat → Stream = ?
```

Given its type, we introduce a variable of type `Nat` to the left hand side

```
fun cycleNats : Nat → Stream =
  | x ⇒ ?
```

Now, the hole has type `Stream`. We can thus split on the result which will provide us with the observations `head` and `tail`.

```
fun cycleNats : Nat → Stream =
  | x .head ⇒ ?
  | x .tail ⇒ ?
```

We can complete the first branch as the head of the stream is simply the input `x`. For the `tail` branch, we need to know if the current number is 0 or not. We thus want to match on `x`.

```
fun cycleNats : Nat → Stream =
  | x .head ⇒ x
  | zero .tail ⇒ ?
  | (suc y) .tail ⇒ ?
```

The two last branches continue the stream and thus will make a recursive call. In the case when `x` is 0, we simply go to our constant 5. If `x` was `suc y` for some `y`, then we simply decrease the number by 1, which is the variable `y`.

```
fun cycleNats : Nat → Stream =
  | x .head ⇒ x
  | zero .tail ⇒ cycleNats 5
  | (suc y) .tail ⇒ cycleNats y
```

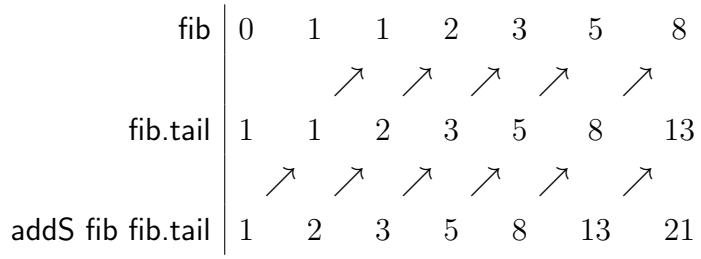
More generally, copatterns allow for any mixing of patterns and observations, and it allows for deep patterns and deep observation matching. A simple example of deep observations is the Fibonacci stream:

```
fun fib : Stream =
  | .head ⇒ 0
  | .tail .head ⇒ 1
  | .tail .tail ⇒ addS fib fib.tail
```

where `addS` denotes the pointwise addition between the two streams.

```
fun addS : Stream → Stream → Stream =
  | s1 s2 .head ⇒ s1.head + s2.head
  | s1 s2 .tail ⇒ addS s1.tail s2.tail
```

We can see how the addition work using the following diagram:



Applying the observation `tail` to `fib` in the last copattern branch is safe because `fib` only reduce if another observation is applied later. It is also important that the definition of `addS` does not unfold its input in an ungarded way, which is not the case here as they are unfolded only when the resulting stream is being unfolded.

The current line of work on copatterns is limited to the simply-typed case. This thesis thus extends the theory to indexed copatterns to allow us not only to encode infinite structures but to also allow coinductive reasoning.

1.2 Contributions

The contributions of this thesis are the following:

- In Chapter 2, we present a first order type theory with support for induction and coinduction through copatterns. We handle dependencies on (co)data from indices using explicit equalities. During type checking, these are accumulated in the context by means of unification, thus yielding linear (co)patterns. This is an alternative to inaccessible patterns [Brady et al., 2004, Goguen et al., 2006b]. We define a non deterministic coverage algorithm following interactive splitting à la Agda [Norell, 2007]. This type theory is proven to be type preserving and to satisfy progress.
- In Section 2.2, We describe a framework for admitting arbitrary index domains to reason over in the type theory. The framework relies on a small of number of requirements on the typing and unification of the provided domain. We use as an example a simple domain of natural numbers, but we believe it can accommodate a number of more involved domains such as contextual LF Nanevski et al. [2008], Pientka [2008], Pientka and Dunfield [2008] or even coinductive domains.
- In Chapter 3, we provide criteria for termination for our copattern language. We prove the subset of programs satisfying these criteria to be terminating via a normalisation preserving translation to a normalising core calculus. This core calculus supports (co)induction through Mendler-style (co)recursors [Mendler, 1991] and is proven normalizing through argument following the technique of Tait [1967] and Girard [1972].

- We implemented a prototype of indexed copatterns over contextual LF by extending the proof assistant Beluga [Pientka and Dunfield, 2010]. Using this prototype, we describe in Chapter 4 as a case study a mechanization of Howe's method to show that bisimilation of programs in a simply typed λ-calculus with recursion over (lazy) lists is a congruence.

Chapter 2

Indexed Codata Types

This chapter lays the foundation of an indexed type theory with copatterns. Indexed types [Zenger, 1997, Xi and Pfenning, 1999] separate the language of indices from the language of types and programs. It is a restriction on dependent types and, by virtue of its simplicity, it thus facilitates answering the metatheoretical questions surrounding the extension of copatterns with dependencies. While indexed data types allow us to for example specify and statically enforce properties about finite lists and trees, indexed codata types allow us to specify and statically enforce properties about streams and traces.

The contributions of this chapter are the following:

- We extend an ML like language with support for indexed (co)data types and deep (co)pattern matching. To keep the design modular, we keep the index domain abstract and specify the key properties it must satisfy. In particular, we require for the index domain a decision procedure to reason about equalities and a unification procedure to compute the most general unifier of two index objects. We illustrate these properties by considering the domain of natural numbers.
- The core language provides a clean foundation for dependent (co)pattern matching where we track and reason with dependencies among indices using equality constraints that are accumulated in a context. Our equality context may contain both satisfiable and contradictory equality constraints. This provides a foundation for (co)pattern

matching that may serve as an alternative to existing approaches [Brady et al., 2004, Goguen et al., 2006a, Pientka and Dunfield, 2008].

- We describe the operational semantics as a small-step semantics and prove type preservation. In addition, we provide a sound non-deterministic algorithm to generate covering sets of copatterns and use it to prove progress.

The flexibility of the index domain allows us not only to build a first step towards dependent copatterns, but also to provide foundations for extensions of indexed languages to a corecursive settting. The language itself can be seen as a foundation for extending languages such as DML [Xi and Pfenning, 1999] and ATS [Xi, 2004] to support indexed codata. Choosing LF [Harper et al., 1993] as an index language, our work serves as a general foundation for writing both inductive and coinductive definitions and proofs about formal systems.

The remainder of this chapter is organized as follows. We illustrate the main ideas of indexed (co)data types through several examples in Section 2.1. In Section 2.2, we discuss the framework allowing us to define index domains. In Section 2.3, we introduce the language supporting both indexed data types and codata types together with (co)pattern matching in a symmetric way. Section 2.4 describes the operational semantics and prove type preservation. Section 2.5 presents coverage, proves its soundness and decidability, and proves progress.

2.1 Main Idea

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

```
data Msg (n:Nat) : type =
| nil : n = zero * 1
```

```
| cons : Σm:Nat. n = suc m * Bit * Msg m
```

The type `Msg n` defines messages inductively. The constructor `nil` describes an empty message and so its length `n` must be zero. The constructor allows us to build a message of length `suc m` by appending a `Bit` to an existing message of length `m`. We denote the existential binding of `m` using a Σ-type and require the user to show that `n = suc m`. As in ML-like languages, we require that constructors that correspond to the base case in our inductive definition take in formally an argument of type unit (denoted by `1`). When we pattern match on a message `w` of type `Msg n`, we need to consider the following two cases: if `m` stands for an empty message, written as `nil (e, ())`, then we also obtain an equality proof `e` that `n = zero`; if `m` stands for a message `cons <m, (e, h, w')>` where `m` is the witness for the existential in the definition of `cons` and `e` stands for the equality proof `n = suc m`. In both cases, we can further pattern match on the equality proof `e`, writing $\wp$ as the witness which forces the type checker to solve the accumulated constraints setting in the base case `n` to zero and in the step case `n` to `suc m`. As our index domain is restricted to a decidable domain, equality proofs can always be derived and reconstructed when elaborating a surface program into our core language.

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

```
codata Str (n:Nat) : type =
| get-bit : Πm:Nat. n = suc m → Bit
| next-bits : Πm:Nat. n = suc m → Str m
| done : n = zero → NextMsg

and data Next-Msg : type =
| next-msg : Σn:Nat. Str n
```

Given a stream with index `n`, we can observe the next bits (`next-bits`) and get the current bit (`get-bit`), provided that we supply some number `m` (which we universally quantify over using a Π-type) and an equality proof that `n = suc m`. We are done reading all bits belonging to our message if `n = zero`, i.e. we can get the next message as long as we provide a proof for

`n = zero`. This definition of a stream allows us to enforce that we read the correct number of bits belonging to a message.

When we pattern match on a data type, we also learn about equality constraints that must hold. When we make observations on a codata type, we must supply an equality proof that satisfies the equality constraint that guards the observation.

Message Processing Using Deep (Co)Pattern Matching

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

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

```
fun read-msg: Π n:Nat. Str n → Msg n * Str zero =
 | zero s ⇒ (nil (℘, ()), s)
 | (suc m) s ⇒
  let c = s.get-bit m ℘ in
  let (w, s') = read-msg m (s.next-bits m ℘) in
  (cons <m, (℘, (c, w))>, s')
```

The program `read-msg` is written by pattern matching on the index object `n`. Using **fun**-abstractions we pattern match on multiple input arguments simultaneously. If `n` is zero, then we are finished reading all bits belonging to the message and we simply return the empty message together with the remaining stream `s`. When we construct a message `nil`, we also must supply a proof that `zero = zero` which is done via $\wp$. $\wp$ is a proof term for equalities and generalizes reflexivity with the rules provided by the index domain. In the case of our index domain of natural numbers, it is equivalent to reflexivity. If `n` is not zero, but of the form `suc m`, we observe the first element `c`, the bit at position `suc m`, in the stream using the observation `.get-bit`. We then read the rest of the message `w` by making the recursive call `read-msg m (s.next-bits m` $\wp$`)` and then build the actual message by consing `c` to the front of `w`. Note that in order to make the observation `get-bit` or `next-bits` we must supply two arguments, namely `m` and a proof that `suc m = suc m`. When constructing the message, we provide as a witness `m` together with a proof that `suc m = suc m`. It seems reasonable to assume that these arguments and equality proofs can be inferred in practice; however we make them explicit in our core language to emphasize their dual role in indexed (co)data types.

So far we have seen how to make observations about streams and use them. Next, we show how to build a stream which is aware of how many bits belong to a message effectively turning it into a stream of messages. This is accomplished via two mutually recursive functions (shown in Figure 2.1) mixing pattern and copattern matching: the first marshals the size of the message with the message stream and the second one continues to create the message stream. We assume that we have parametric polymorphism here (which we do not treat in our foundation).

The function `get-msg` takes in a stream `s` of bits and a stream of natural numbers that tells us the size of a message. It then returns a message of the required size by reading the appropriate number of bits from `s` using the function `msg-str` and creating a stream of type `Str n` where `n` is the size of the message. The function `msg-str` is defined by (co)pattern matching; the first branch says we can only make the observation `done` provided that `n` is zero; in this case we are done reading all bits belonging to the message. The second branch

```
codata 'a Stream : type =
| head : 'a
| hail : 'a Stream

fun get-msg: Bit Stream → Nat Stream → Next-Msg =
| s ns ⇒ let n = ns.head in next-msg <n, msg-str n s ns.tail>

and msg-str: Π n:Nat. Bit Stream → Nat Stream → Str n =
| zero s ns .done ℘ ⇒ get-msg s ns
| (suc n) s ns .get-bit m ℘ ⇒ s.head
| (suc n) s ns .next-bits m ℘ ⇒ msg-str n s.tail ns
```

Figure 2.1: Building indexed streams

tells us if the size of the message is `suc n`, we can make the observation `get-bit` provided we have a proof $\wp$ showing that `suc m` is equal to `suc n`. Note that our (co)pattern remains linear – the fact that `m` is forced to be equal to `n` is guaranteed by the equality proof $\wp$ that solves the resulting constraint `suc n = suc m`. The pattern $\wp$ indicates to the type checker that it needs to unify equality constraints. We exploit here the fact that equality and unification is decidable in our index domain. If we can solve the constraint, as is the case here, we keep track of the solution `n := m : Nat` in our context of assumptions and continue to type check the body of the branch under this constraint; if we can disprove the arising equality constraint, we keep a contradiction in our context of assumption and continue to check the body. This allows for an elegant treatment of linear (co)patterns in the presence of dependent types.

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

```
fun gen-bit-str : Str (suc (suc zero)) =
 | .get-bit   (suc zero) ℘ ⇒ random-bit-gen ()
 | .next-bits (suc zero) ℘ .get-bit zero ℘ ⇒ random-bit-gen ()
```

```
| .next-bits (suc zero) ℘ .next-bits zero ℘ .done ℘ ⇒
    next-msg (suc (suc z)) gen-bit-str
```

Revisiting the Duality of (Co)Inductive Definitions

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

An important question to clarify is whether (co)data type definitions need or should be exhaustive, i.e. provide a constructor or observation for each possible index. What does it mean to have no constructor for a possible index? And dually, what does it mean to have no observation for a possible index? We discuss these questions by looking at how we define even numbers inductively and coinductively.

```
data Even (n:Nat) : type =
| ev-z : n = zero * 1
| ev-ss : Σm:Nat. n = suc (suc m) * Even m
```

Above is the inductive version of the definition. Clearly, inductive definitions do not need to be covering. No case for `suc zero` is shown here. `Even n` states that we can construct a proof that `zero` is even using `ev-z` provided we have a proof that `n = zero`. For clarity, we define the type of the constructor `ev-z` as `n = zero * 1` where `1` stands for unit (or top). Similarly, we can construct a proof that `n` is even, if there exists a number `m` s.t. `n = suc (suc m)` and `m` is even.

The set of terms inhabiting this predicate is the least fixed point defining even numbers. Note that there is no way that we can construct a witness for `Even (suc zero)` and this type is empty. Modelling the empty type `data 0 : type` by declaring no constructors, we could make this more explicit by adding a constructor `ev-sz` of type `n = (suc zero) * 0`. This explicitly states that `Even (suc zero)` cannot be constructed without any assumptions,

since `0` is not inhabitated. We typically omit such a case in the definition of our inductive types, but these impossible cases might arise when we pattern match on elements of the type `Even`.

Now, we wish to define `Even` coinductively. We simply proceed by taking the dual of the definitions. That is, we build the greatest fixed point. We start with the set containing all natural numbers and refine it until we get only even numbers. This refinement process is obtained by simply applying observations which restrict our options. We have an observation to reject `suc zero` and another one that ensures a given number is even only if it is of the form `suc (suc n)` for an even number `n`. This leads us to the following definition:

```
codata Coeven (n:Nat) : type =
| cev-sz : n = suc zero → 0
| cev-ss : Πm:Nat. n = suc (suc m) → Coeven m
```

If `n = suc zero` then we return the empty type. If we make an observation `cev-sz` and have a proof that `n = suc zero` then we have arrived at a contradiction. The observation `cev-ss` extracts a proof of `Coeven m` from a proof of `Coeven (suc (suc m))`.

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

We now prove that both interpretations give us the same set of terms. First we show that `Even n` implies `Coeven n`:

```
fun ev-to-coev : Πn:Nat. Even n → Coeven n =
 | n (ev-z (℘, x))          .cev-sz ℘
 | n (ev-z (℘, x))          .cev-ss m ℘
 | n (ev-ss <m, (℘, e)>) .cev-sz ℘
 | n (ev-ss <m, (℘, e)>) .cev-ss k ℘ ⇒ ev-to-coev e
```

We write this function by pattern matching on `Even n`. In the case where `Even zero`, we want to return `Coeven zero`. As elements of `Coeven zero` are defined by the observations we

can make about it, we consider two sub-cases. If we try to make the observation `cev-sz`, we must provide a proof that `zero = suc zero`. This will be refuted by the decision procedure in our index domain, i.e. the decision procedure will succeed, but add a contradiction to the context of assumptions, from which anything follows. Again, as a pattern, $\wp$ causes the type checker to try to solve the equality constraint. If they are satisfiable, it adds the constraints to the context of assumptions. If they are unsatisfiable, it adds a marker that there is a contradiction. In latter cases, we can simply omit the right hand side, since an unsatisfiable equality constraint makes the branch unreachable.

If we try to make the observation `cev-sz`, then we must provide a term for the type

```
Πm:Nat. zero = suc (suc m) → Coeven m.
```

Abstracting on the left, we have an argument `zero = suc (suc m)` for which there is no proof. From this contradiction, the case holds trivially.

Finally, we consider the case where `Even (suc (suc m))`. In this case, we can again make two possible observations, `cev-sz` and `cev-ss`. In the first case, we again arrive at a contradiction, since `suc (suc n) = suc zero` is false. In the last case, we accumulate and solve two equality constraints while type checking the (co)pattern: `n = suc (suc m)` and `n = suc (suc k)`. Then we proceed to check the body of the branch in the context where `n := suc (suc k):Nat` and `k := m:Nat`.

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

Can we also prove that `Coeven n` implies `Even n`? – For an arbitrary predicate it does not hold that the coinductive interpretation implies the inductive one. The greatest fixed point could be strictly larger than the least. In the case of `Even` and `Coeven` however, we can indeed show this property by induction on `n`.

```
fun coev-to-ev : Πn:Nat. Coeven n → Even n =
 | zero c ⇒ ev-z (℘, ())
 | (suc zero) c ⇒ abort (c.cev-sz ℘)
 | (suc (suc n)) c ⇒ ev-ss <n, (℘ , coev-to-ev n (c.cev-ss n ℘))>
```

When `n = suc zero`, we assume `Coeven (suc zero)` and observe `(c.cev-sz` $\wp$`)` which results in an object of type `0` – however, we know that this type is not inhabited and hence we abort. **abort** eliminates `0` and serves as an abbreviation for a function matching on a term without constructors that would have no branches.

Lambda Terms as Index Language: Reasoning about Divergence

So far, all of our examples used the natural numbers as our index domain. In this example we will use an encoding of the λ-calculus using contextual LF [Nanevski et al., 2008, Pientka, 2008]. We represent λ-terms as the following LF signature:

```
Exp : type.
app : Exp → Exp → Exp.
lam : (Exp → Exp) → Exp.
```

This signature creates a type `Exp` with applications `app m n`, λ-abstractions `lam (`λ`x.m x[x])`, and variables `x`. LF signatures make use of higher-order abstract syntax to encode variable binders using the LF function type.

Now, terms of the index domain are pairs of an LF term together with its binding context, written `g` $\vdash$ `t`. If the term is closed and the context is empty, we will simply omit the turnstile and write t. We distinguish bound variables x defined within the λ-calculus of our index-domain with meta-variables m, n, and p that denote terms from our index domain which have of type `g` $\vdash$ `Exp`. Meta-variables are paired with a substitution, written `m[s]`, that serves to mediate between the expected context for the meta-variable and the ambient context in which it occurs. These substitutions are usually simultaneous substitutions. However, our example will only feature terms depending on a single variable. For readability, we will often omit writing the substitutions when they are identity or weakening substitutions.

For our needs, we will simply consider the fixed signature defined above instead of all of contextual LF. However, the meta-properties that contextual LF [Nanevski et al., 2008, Pientka, 2008] satisfies include the criteria for an index domain that we will present in Section 2.2 and its framework thus justifies our representation of the λ-calculus. The system

Beluga [Pientka and Dunfield, 2010] provides an implementation of such an indexed language over contextual LF.

Now, let us consider a data type denoting the big-step evaluation of lambda terms.

```
data Eval : (m : Exp) (n : Exp) : type =
| ev-lam : Σp:(x:Exp ⊢ Exp).m = (lam λx.p[x]) * n = (lam λx.p[x])
| ev-app : Σm1:Exp.Σm2:Exp.Σp:(x:Exp ⊢ Exp).m = (app m1 m2)
              * (Eval m1 (lam λx.p[x]) * Eval p[m2/x] n)
```

`Eval` is a predicate on two terms that are closed. Thus, there is no case for variables. `ev-lam` states that lambda abstractions evaluate to themselves; if the left-hand side `m` is a λ-abstraction `lam λx.p[x]`, then so is the right-hand side. The variable `p` has type `x : Exp ⊢ Exp` indicating its dependency on a variable `x`, as it is used under the binding for `x`. `ev-app` requires the input to be `app m1 m2` for some closed $\mathtt{m}_1$ and $\mathtt{m}_2$. The evaluation of an application simply evaluates the left-hand side to a λ-abstraction `lam λx.p[x]` and then evaluates the body of the abstraction under the substitution $\mathtt{m}_2$ for `x`, leading to the output.

A proof of evaluation is then just a series of constructors representing a particular step in the evaluation. Thus, the proof forms a trace of the evaluation sequence. For example, the simple term `app (lam λx.x) (lam λx.x)` evaluates by simply doing a β-reduction since both the left-hand side of the application and the result of the substitution are `lam λx.x` which reduces to itself. This is captured by the term

```
ev-app ⟨lam λx.x, ⟨lam λx.x, ⟨(x:Exp ⊢ x),
          (℘, ev-lam ⟨(x:Exp ⊢ x), (℘, ℘)⟩,
               ev-lam ⟨(x:Exp ⊢ x), (℘, ℘)⟩)⟩⟩⟩
```

which has type `Eval (app (lam λx.x) (lam λx.x)) (lam λx.x)`. Here, `x:Exp ⊢ x` is the instantiation for `p` from the declaration of `Eval`. Since `p` had type `x:Exp ⊢ Exp` which depends on a variable x, the term we provide also exists in the context `x:Exp`. In our case, the provided term is simply the variable `x`. We provide both terms of the application and use reflexivity to show it is equal to the expected type. The left-hand side of the application being the λ-abstraction `lam λx.x`, we use `ev-lam` as it evaluates to iself. Similarly, the resulting substitution `x[lam λx.x/x]` becomes `lam λx.x` which also evaluates to itself.

If a term in our indexed language forms a trace of evaluation, and the evaluation of a given λ-expression diverges, such a term will be infinite. It appears naturally that divergence should be described coinductively. However, it is not obtained by defining a coinductive version of `Eval` as we did with `Coeven` but rather by describing the negation of it. To make sense of it, let us have a look at the definition.

```
codata Div (m:Exp) : type =
| div-lam : Πp:(x:Exp ⊢ Exp).m = lam λx.p[x] → 0
| div-app : Πm₁:Exp.Πm₂:Exp.m = app m₁ m₂
      → Div m₁ + (Σp:(x:Exp ⊢ Exp).Eval m₁ (lam λx.p[x]) * Div p[m₂/x])
```

Studying `div-app` first, once the equality is assured we either obtain a proof of divergence of $\mathtt{m}_1$ or the evaluation of $\mathtt{m}_1$ into `lam` λ`x.p[x]` and the divergence of `p[`$\mathtt{m}_2$`/x]`. This choice is represented through a disjunction. Now, if we want to build a term of `Div (app` $\mathtt{m}_1$ $\mathtt{m}_2$`)` for some given $\mathtt{m}_1$ and $\mathtt{m}_2$, we need to provide either proofs of divergence ($\mathtt{m}_1$, or `p[`$\mathtt{m}_2$`/x]` given that `Eval` $\mathtt{m}_1$ `(lam` λ`x.p)`).

As for `div-lam`, we stated above that λ-abstractions evaluate trivially to themselves. Hence, they can never diverge. Thus, we simply need to provide a term of type `0`.

With such a predicate, it is natural to want to prove divergence of some terms. Let us start by proving that given `Div` $\mathtt{e}_1$ then `Div (app` $\mathtt{e}_1$ $\mathtt{e}_2$`)` for all $\mathtt{e}_2$.

```
fun div1 : Πe₁.Πe₂. Div e₁ → Div (app e₁ e₂) =
| e₁ e₂ d .div-lam p ℘
| e₁ e₂ d .div-app e₁' e₂' p ℘ ⇒ inl d
```

Now, we are trying to prove divergence of applications, hence we trivially fulfill the branch for `div-lam` using the empty function. For applications, the equality holds so we know $\mathtt{m}_1$ `= app` $\mathtt{n}_1$ $\mathtt{n}_2$. Then, we use `d` through the left injection **inl** to obtain the proof we require. This is a very simple proof that does not make use of recursion.

Let us build a concrete λ-expression that makes use of a coinduction hypothesis; we will prove that the term `app (lam` λ`x.app x x) (lam` λ`x.app x x)` diverges. The left-hand side of the term is already a λ-abstraction so it evaluates to itself. Then, the substitution

gives us the term `app (lam λx.app x x) (lam λx.app x x)` which is the original term. Now we found a loop and thus obviously this term diverges. As a program, it is defined as follows:

```
fun om-div : Div (app (lam λx.app x x) (lam λx.app x x)) =
| .div-lam p ℘
| .div-app (lam λx.app x x) (lam λx.app x x) ℘ ⇒
  inr ⟨(x:Exp ⊢ app x x),
          (ev-lam ⟨(x:Exp ⊢ app x x), (℘, ℘)⟩), om-div⟩
```

The `div-lam` case is handled in the same way it was handled in `div1` as we expect an application instead of an abstraction. The application case follows the right injection as the left-hand side of our term evaluates to itself, described by `ev-lam ⟨x:Exp ⊢ app x x, (℘, ℘)>` Then, the result of the substitution is the same as the original term, so it can be proved using the recursive call. This call is guarded as it is done under the observation `div-app` from the left-hand side and thus is a valid use of the coinduction hypothesis.

2.2 Index Domain

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

$$\text{Index-Type } U ::= \mathsf{Nat}$$
$$\text{Index-Term } C ::= u \mid \mathsf{zero} \mid \mathsf{suc}\ C$$

Variables that occur in index-terms must be declared in an index-context Δ and they are

instantiated via an index-substitution θ. Their definition is concrete in the framework:

$$\begin{array}{ll} \text{Index-Context } \Delta & ::= \cdot \mid \Delta, u : U \mid \Delta, u := C : U \mid \Delta, \# \\ \text{Index-Substitution } \theta & ::= \cdot \mid \theta, C/u \end{array}$$

In our setting, the index-context also contains equality constraints. The constraint $u{:=}C{:}U$ says that the index-variable u is equal to the index-term C. Such constraints arise in typing (co)patterns (see the function `msgStr` from Section 2.1). The index-context also keeps track of contradictions, written #, that may arise when we encounter in a (co)pattern a constraint that can never be satisfied.

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

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

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

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

Typing of Index Domain

We define the well-formedness of index-contexts and index-substitutions in Fig. 2.2. The definition of index-contexts is mostly straightforward noting that $\Delta, u{:=}C{:}U$ is a well-formed index-context if Δ is well-formed, the index-type U is well-formed in Δ, and the index-term C has index-type U. We make sure that there are no circularities in Δ. An index-substitution θ provides a mapping for declarations in the index-context Δ' and guarantees that all instantiations have the expected index-type and are compatible with existing constraints. Our

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

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

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

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

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

Figure 2.2: Index-Contexts and index-Substitution

judgment $\Delta \vdash \theta : \Delta'$ states that a given instantiation θ, computed via pattern matching at run-time, matches the assumptions in Δ' that were made statically during type checking. It is defined inductively on the domain Δ'. Although all instantiations computed by pattern matching are ground (i.e. Δ is empty), we state the relationship between θ and Δ more generally. If Δ' contained a contradiction, the contradiction must also be present in Δ. We still require in this case that θ provides consistent and well-typed instantiations for all the remaining declarations in Δ'.

The presentation of an index domain must include kinding and typing rules, which we will denote by the judgments $\Delta \vdash U : \mathsf{Type}$ and $\Delta \vdash C : U$, respectively. We provide the rules for the example of natural numbers in Figure 2.3.

Our index domain must satisfy several properties. The first one is the substitution property which we list here as a requirement. We provide proofs of each of the requirements for our example domain Nat.

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

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

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

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

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

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

Figure 2.3: Index-Types and index-Terms for Nat

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

$$\frac{}{\Delta \vdash \mathsf{zero} = \mathsf{zero}} \quad \frac{\Delta \vdash C_1 = C_2}{\Delta \vdash \mathsf{suc}\ C_1 = \mathsf{suc}\ C_2} \quad \frac{}{\Delta \vdash u = u} \quad \frac{\# \in \Delta}{\Delta \vdash C_1 = C_2}$$

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

Figure 2.4: Equality of index terms for Nat

In addition to the typing rules we also require that equality on the index language is decidable and takes into account the equality constraints in Δ. Equality is defined using the judgment $\Delta \vdash C_1 = C_2$. To illustrate we give the definition of equality for natural numbers in Fig. 2.4. We note that if a contradiction lives in the context, any two index-terms are equal. An index-variable u can be equal to an arbitrary index-term C if the context contains a constraint $u := C'$ and C is equal to C'.

Typing Judgement for Equality in Patterns Type checking of (co)patterns will need to solve equations $C_1 = C_2$ using unification on our index domain, and thus introduce term assignments to variables in Δ, yielding Δ'. Unification for the index domain is represented

by the judgment $\Delta \vdash C_1 = C_2 \searrow \Delta'$. We define unification for the index domain of natural numbers in Figure 2.5. Note that the unification should always succeed in producing an index-context Δ'. If C_1 and C_2 are unifiable, then the arising equality constraints are recorded in Δ'. However, if C_1 and C_2 are not unifiable, we should return a Δ' that contains a contradiction #. There are two possible sources of failure: either the two terms are syntactically different or the occurs check fails. As we keep track of constraints in Δ, checking whether u occurs in u' where $u' := C : U \in \Delta$, we must check whether u occurs in C. Our well-formedness of Δ guarantees that our context is not cyclic and hence the occurs check will terminate. Our definition of unification is then straightforward. As we must guarantee that Δ remains well-formed, we may permute it to an equivalent well-formed context (written as $\Delta \sim \Delta'$) when unifying u with an index-term C.

Properties about Unification

As we alluded during our examples, typing of (co)patterns will rely on solving equality constraints. We therefore rely on the correctness of the unification algorithm. In particular, we require that unification will always succeed, possibly yielding a context containing a contradiction.

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

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

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

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

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

Lemma 2.1. *If $\Delta \vdash C_1 = C_2 \searrow \Delta'$ and $\Delta_1 \vdash \theta : \Delta$ then there is a Δ_1' such that $\Delta_1 \vdash [\theta]C_1 = [\theta]C_2 \searrow \Delta_1'$ and $\Delta_1' \vdash \theta : \Delta'$.*

$\boxed{\Delta \vdash C_1 = C_2 \searrow \Delta'}$ Given the index-terms C_1 and C_2 in context Δ, we synthesize the most general index-context Δ' such that $\Delta \prec \Delta'$ and $\Delta' \vdash C_1 = C_2$.

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

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

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

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

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

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

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

Figure 2.5: Unification of Index Terms

Proof. By Requirements 2 and 3. □

2.3 Syntax

We now present the syntax of types and terms of our copattern language. We shall first discuss the grammar or types and kinds and show kinding, before moving to typing of terms and (co)patterns.

Types (Figure 2.6) admit the standard unit types 1, pairs of types $T_1 \times T_2$, and function types $S \to T$. We make use of the indices using dependent function types $\Pi u{:}U.T$ and dependent product types $\Sigma u{:}U.T$. Index objects from our index domain U can be embedded

$$
\begin{array}{lll}
\text{Kinds} & K & ::= \mathsf{type} \mid \Pi u{:}U.K \\
\text{Types} & S,T & ::= X \mid 1 \mid T_1 \times T_2 \mid S \to T \mid C_1 = C_2 \mid \Pi u{:}U.T \mid \Sigma u{:}U.T \mid T\ \vec{C} \\
 & & \quad \mid \Lambda u.T \mid \mu X.T \mid \nu X.T \mid D \mid R \\
\text{Variants} & D & ::= \langle c_1\ T_1 \mid \cdots \mid c_n\ T_n \rangle \\
\text{Records} & R & ::= \{d_1 : T_1, \ldots, d_n : T_n\}
\end{array}
$$

Figure 2.6: Grammar of types and kinds

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

The main feature of our type system is that it supports indexed recursive and indexed corecursive types. We defined recursive types from our examples using the types $(\mu X.\Lambda\vec{u}.D)\ \vec{C}$. The type variable X denotes recursive occurrences of the type itself inside D. The binding $\Lambda\vec{u}.D$ binds an arbitrary number of index variables u and stands for the iterated application $\Lambda u_1.\Lambda u_2.\cdots.\Lambda u_n.D$. The variant D is a labelled sum (written as $\langle c_1\ T_1 \mid \cdots \mid c_n\ T_n \rangle$) which represents the constructors and the types of the arguments they expect. The arguments $\vec{C}$ at the end are the parameters passed to the type to instantiate the recursive type family. In Section 2.1, we had the type `Eval p[`$\mathtt{m}_2$`/x] n` as part of the constructor `ev-app`. The arguments $\vec{C}$ in this case are `p[`$\mathtt{m}_2$`/x]` and `n`. While the grammar and kinding rules show those elements of what we call a recursive type to be separate, typing will force them to be build like that. Dually, in the corecursive type will be constructed as $(\nu X.\Lambda\vec{u}.R)\ \vec{C}$ which builds a record R (denoted as $\{d_1 : T_1, \ldots, d_n : T_n\}$) where the labels d_i are the observations we can make.

Example 1 (Indexed Recursive Types). Datatypes $C = \mu X.\lambda\vec{u}.D$ for $D = \langle c_1\ T_1 \mid \cdots \mid c_n\ T_n\rangle$ describe least fixed points. Choosing as index domain natural numbers, we can model our previous definition of `Msg` as follows in our core language.

$$\begin{aligned}\mu\text{Msg}.\lambda u.\langle \mathsf{Nil} \quad &: u = \mathsf{zero} \times 1\ ,\\ \mathsf{Cons} &: \Sigma u'{:}\mathsf{Nat}.u = \mathsf{suc}\ u' \times (\mathsf{Bit} \times \text{Msg}\ u')\rangle\end{aligned}$$

Example 2 (Indexed Corecursive Types). Record types $C = \nu X.\lambda\vec{u}.R$ with $R = \{d_1 : T_1, \ldots, d_n : T_n\}$ are recursive labeled products and describe infinite data. As for data, non-recursive record types are encoded by a void ν-abstraction $\nu\_.\lambda\vec{u}.R$. Consider our previous codata type definition for indexed streams, i.e. `Str`, with the three observations, `GetBit`, `NextBits`, and `Done`. Depending on the index `n` we choose the corresponding observation. It directly translates to the following:

$$\begin{aligned}\nu\text{Str}.\lambda m.\ \{\ \mathsf{Done} \quad &: \quad m = \mathsf{zero} \ \to \mathsf{NextMsg}\ ,\\ \mathsf{NextBits} \ &: \ \Pi n{:}\mathsf{Nat}.m = \mathsf{suc}\ n \to \text{Str}\ n\ ,\\ \mathsf{GetBit} \ &: \ \Pi n{:}\mathsf{Nat}.m = \mathsf{suc}\ n \to \text{Bit} \qquad \}\\ \mu\text{NextMsg}.\ \langle\ \mathsf{NextMsg} \ &: \ \Sigma n{:}\mathsf{Nat}.\mathsf{Str}\ n\ \rangle\end{aligned}$$

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

In order to facilitate reading in the rest of the thesis, we will adopt the following convention for recursive and corecursive types. For a type T depending on a single type variable X, we build the functional $\mathcal{F}$ defined as $X \mapsto T$. Then, if T is $\Lambda\vec{u}.D$, we can denote the recursive type $\mu X.\Lambda\vec{u}.D$ as $\mu\mathcal{F}$ and the resulting substitution $[\overrightarrow{C/u}; (\mu X.\Lambda\vec{u}.D)/X]D_c$ as $\mathcal{F}_c(\mu\mathcal{F})\ \vec{C}$ where D_c denotes the type in D for the constructor c. Moreover, we will iterate over all $c \in \mathcal{F}$ to actually mean all $c \in D$. We equivalently will use $\mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$ to denote the type in D at position i. In a similar fashion, for $\mathcal{F} = \Lambda\vec{u}.R$, we use $\nu\mathcal{F}$ for the corecursive type $\nu X.\Lambda\vec{u}.R$.

Kinding is expressed using the judgment $\Delta; \Xi \vdash T : K$ that indicates that type T has kind K. Since types can contain indices with free variables, the judgment carry the index-context Δ. In addition, the context Ξ is used to accumulate type variables introduces by (co)recursive types. The kinding rules appear in Figure 2.7. Product types, arrow types,

$\boxed{\Delta;\Xi \vdash T : K}$ Type T has kind K in contexts Δ and Ξ

$$\frac{}{\Delta;\Xi \vdash 1 : \mathsf{type}} \quad \frac{\Delta;\Xi \vdash T_1 : \mathsf{type} \quad \Delta;\Xi \vdash T_2 : \mathsf{type}}{\Delta;\Xi \vdash T_1 \times T_2 : \mathsf{type}} \quad \frac{\Delta;\Xi \vdash S : \mathsf{type} \quad \Delta;\Xi \vdash T : \mathsf{type}}{\Delta;\Xi \vdash S \to T : \mathsf{type}}$$

$$\frac{\Delta \vdash U : \mathsf{Type} \quad \Delta, u{:}U;\Xi \vdash T : \mathsf{type}}{\Delta;\Xi \vdash \Pi u{:}U.T : \mathsf{type}} \quad \frac{\Delta \vdash U : \mathsf{Type} \quad \Delta, u{:}U;\Xi \vdash T : \mathsf{type}}{\Delta;\Xi \vdash \Sigma u{:}U.\, T : \mathsf{type}}$$

$$\frac{\Delta;\Xi \vdash T_i : \mathsf{type}}{\Delta;\Xi \vdash \langle c_1\ T_1 \mid \cdots \mid c_n\ T_n\rangle : \mathsf{type}} \quad \frac{\Delta;\Xi \vdash T_i : \mathsf{type}}{\Delta;\Xi \vdash \{d_1 : T_1, \ldots, d_n : T_n\} : \mathsf{type}}$$

$$\frac{X{:}K \in \Xi}{\Delta;\Xi \vdash X : K} \quad \frac{\Delta;\Xi \vdash T : \Pi u{:}U.\, K \quad \Delta \vdash C : U}{\Delta;\Xi \vdash T\,C : [C/u]K} \quad \frac{\Delta, u{:}U;\Xi \vdash T : K}{\Delta;\Xi \vdash \Lambda u.T : \Pi u{:}U.K}$$

$$\frac{\Delta;\Xi, X{:}K \vdash T : K}{\Delta;\Xi \vdash \mu X{:}K.\, T : K} \quad \frac{\Delta;\Xi, X{:}K \vdash T : K}{\Delta;\Xi \vdash \nu X{:}K.\, T : K} \quad \frac{\Delta \vdash M : U \quad \Delta \vdash N : U}{\Delta;\Xi \vdash M = N : \mathsf{type}}$$

Figure 2.7: Kinding rules

variants and record types are well kinded against type if all of their components are kinded against type. Π-types and Σ-types are kinded against type if their bodies are kinded against type in the extended index-context. Equality types $M = N$ are kinded against type if M and N are typed against the same index-type U. Types variables X are well-kinded against kind K if $\Xi(X) = K$. Type applications $T\ C$ are kinded against $[C/u]K$ if T is kinded against $\Pi u : U.K$. (Co)recursive types $\mu X{:}K.T$ and $\nu X{:}K.T$ types are kinded against K if T is kinded against K in the extended type variable context $\Xi, X{:}K$.

Terms and Typing

Terms are simply applications of heads together with spines $h \star E$. If the spine is empty (aka $\cdot$), we simply write the head h by itself. Heads include variables x, unit (written as ()), pairs (written as (v_1, v_2)), dependent pairs (written as $\mathtt{pack}\,\langle C, v\rangle$), the canonical term for equalities which denotes reflexivity (written as $\wp$), and function definitions (written

Terms	t	$::= h \star E$
Heads	h	$::= x \mid () \mid (t_1, t_2) \mid \wp \mid c\ t \mid \texttt{pack}\,\langle C, t\rangle \mid \texttt{fun}\ f.\vec{b}$
Spines	E	$::= \cdot \mid t\ E \mid .d\ E \mid C\ E$
Branches	b	$::= q \mapsto t \mid q$
Patterns	p	$::= x \mid (p_1, p_2) \mid \wp \mid c\ p \mid \texttt{pack}\,\langle u, p\rangle$
Copatterns	q	$::= \cdot \mid p\ q \mid u\ q \mid .d\ q$

Figure 2.8: Grammar of terms and copatterns

as $\texttt{fun}\ f.\vec{b}$). All but function definitions and variables are introduction forms which are eliminated via pattern matching; dually, we make observations about functions, universals and coinductive types through applications, which are carried by spines E.

Simultaneous (co)patterns are described using a spine that is built out of patterns (written as p) and observations (written as $.d$). Patterns themselves are derived from terms and can be defined using pattern variables x, pairs (written as (p_1, p_2)), pattern instances (written as $\texttt{pack}\,\langle C, p\rangle$) and patterns formed with a data constructor c.

Branches in case-expressions are modelled by $q \mapsto t$. We also allow branches with no body – they will only succeed if the copattern q is impossible, i.e. we arrived at some equality constraints that lead to a contradiction. Strictly speaking, it is not necessary as we could always write some arbitrary expression for the body which would be inaccessible and thus can never be reached. In the remainder of this thesis we will often describe functions as $\texttt{fun}\ f.\overrightarrow{q \mapsto t}$ in order to expose its copatterns or q_i, or even it's right-hand side t. This not does preclude that some (or even all) branches are missing right-hand sides. It is merely done for the sake of notational convenience.

Notation 1. Given two spines E and E', we will denote their concatenation simply by $E\ E'$. Similarly, we will append a term t, a index-term C or an observation $.d$ at the end of a spine by simply writing $E\ t$, $E\ C$, or $E\ .d$, respectively.

Moreover, if we have a term t which represents the application $h \star E$ and a spine E', we will use the notation $t \star E$ to denote the concatenation of E to the spine E'. Hence $t \star E$ is

equivalent to writing $h \star E\ E'$.

As our last piece of notation, if the spine of a term is empty, that is $h \star \cdot$, we will simply omit it entirely and consider the head h as a term.

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

$$\frac{\Delta; \Gamma \vdash h : T' \quad \Delta; \Gamma; T' \vdash E \searrow T}{\Delta; \Gamma \vdash h \star E : T} \qquad \frac{\Delta; \Gamma \vdash t : T' \quad \Delta \vdash T = T'}{\Delta; \Gamma \vdash t : T}$$

$\boxed{\Delta; \Gamma \vdash h : T}$ In index-context Δ and context Γ, head h has type T.

$$\frac{\Gamma(x) = T}{\Delta; \Gamma \vdash x : T} \qquad \frac{}{\Delta; \Gamma \vdash () : 1} \qquad \frac{\Delta; \Gamma \vdash t_1 : T_1 \quad \Delta; \Gamma \vdash t_2 : T_2}{\Delta; \Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \frac{\Delta \vdash C : U \quad \Delta; \Gamma \vdash t : [C/u]T}{\Delta; \Gamma \vdash \mathtt{pack}\, \langle C, t \rangle : \Sigma u{:}U.T}$$

$$\frac{\Delta; \Gamma \vdash t : \mathcal{F}_c(\mu\mathcal{F})\ \vec{C}}{\Delta; \Gamma \vdash c\ t : \mu\mathcal{F}\ \vec{C}} \qquad \frac{\Delta \vdash C_1 = C_2}{\Delta; \Gamma \vdash \wp : C_1 = C_2} \qquad \frac{\text{for each } i\ \Delta; \Gamma, f{:}T \vdash b_i : T}{\Delta; \Gamma \vdash \mathtt{fun}\ f.\vec{b} : T}$$

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

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

$\boxed{\Delta; \Gamma; T \vdash E \searrow T'}$ In index-context Δ and context Γ, spine E eliminates function of type T into type T'

$$\frac{}{\Delta; \Gamma; T \vdash \cdot \searrow T} \qquad \frac{\Delta \vdash C : U \quad \Delta; \Gamma; [C/u]T \vdash E \searrow T'}{\Delta; \Gamma; \Pi u{:}U.T \vdash C\ E \searrow T'}$$

$$\frac{\Delta; \Gamma \vdash t : S \quad \Delta; \Gamma; T \vdash E \searrow T'}{\Delta; \Gamma; S \to T \vdash t\ E \searrow T'} \qquad \frac{\Delta; \Gamma; \mathcal{F}_d(\nu\mathcal{F})\ \vec{C} \vdash E \searrow T}{\Delta; \Gamma; (\nu\mathcal{F})\ \vec{C} \vdash .d\ E \searrow T}$$

Figure 2.9: Typing rules for terms

The typing rules appear in Figure 2.9. Applications $h\ E$ type against T simply if heads h type against T' and spines E eliminate T' into T. As we have constraints in Δ, we also include

a type conversion rule. The judgment $\Delta \vdash T_1 = T_2$ is defined recursively on the structure of types. When we compare $\Delta \vdash (T_1\ \vec{C}) = (T_2\ \vec{C'})$, we simply compare $\Delta \vdash T_1 = T_2$ and for all i we have $\Delta \vdash C_i = C'_i$ falling back to the comparison on index types.

Most rules for heads are standard. Typing of index object C refers to typing of index-terms as described in Section 2.2. A constructor c takes a term of type $\mathcal{F}_c(\mu\mathcal{F})\ \vec{C}$, yielding a term of type $\mu\mathcal{F}\ \vec{C}$. Dually, an observation $.d$ takes a term of type $\nu\mathcal{F}\ \vec{C}$ yielding a term of type $\mathcal{F}_d(\nu\mathcal{F})\ \vec{C}$. The witness for an equality $C_1 = C_2$ is simply $\wp$ provided C_1 and C_2 are equal in our index domain using our rules from Figure 2.4.

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

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

Spines are essentially typed pointwise. $t\ E$ eliminates a function type $S \to T$ into T' where t has type S and E eliminates T into T'. $C\ E$ eliminates a universal type $\Pi u{:}U.T$ into T' where C has type U and E eliminates $[C/u]T$ into T'. $.d\ E$ eliminates a codata type $(\nu\mathcal{F})\ \vec{C}$ into T where $.d$ is an observation of $\mathcal{F}\ E$ eliminates $\mathcal{F}_d(\nu\mathcal{F})\ \vec{C}$ into T'. Since non empty spines only provide arguments or observations, the only heads that are well typed against a non empty spines are variables and functions. For any term other term, term typing simply reduces into head typing.

The typing rules for (co)patterns (see Fig. 2.10) are defined using the following two

$\boxed{\Delta;\Gamma \vdash p : T \searrow \Delta';\Gamma'}$ Pattern p of type T extends contexts $\Delta;\Gamma$ into $\Delta';\Gamma'$.

$$\frac{}{\Delta;\Gamma \vdash x : T \searrow \Delta;\Gamma, x{:}T} \qquad \frac{\Delta;\Gamma \vdash p : \mathcal{F}_c(\mu\mathcal{F})\ \vec{C} \searrow \Delta';\Gamma'}{\Delta;\Gamma \vdash c\ p : \mu\mathcal{F}\ \vec{C} \searrow \Delta';\Gamma'}$$

$$\frac{\Delta;\Gamma \vdash p_1 : T_1 \searrow \Delta';\Gamma' \quad \Delta';\Gamma' \vdash p_2 : T_2 \searrow \Delta'';\Gamma''}{\Delta;\Gamma \vdash (p_1, p_2) : T_1 \times T_2 \searrow \Delta'';\Gamma''}$$

$$\frac{\Delta, u{:}U;\Gamma \vdash p : P \searrow \Delta';\Gamma'}{\Delta;\Gamma \vdash \mathtt{pack}\,\langle u, p\rangle : \Sigma u{:}U.P \searrow \Delta';\Gamma'} \qquad \frac{\Delta \vdash C_1 = C_2 \searrow \Delta'}{\Delta;\Gamma \vdash \wp : C_1 = C_2 \searrow \Delta';\Gamma}$$

$\boxed{\Delta;\Gamma;T \vdash q \searrow \Delta';\Gamma';T'}$ Copattern q eliminates type T into type T' and extending contexts $\Delta;\Gamma$ into $\Delta';\Gamma'$.

$$\frac{}{\Delta;\Gamma;T \vdash \cdot \searrow \Delta;\Gamma;T} \qquad \frac{\Delta;\Gamma \vdash p : S \searrow \Delta';\Gamma' \quad \Delta';\Gamma';T \vdash q \searrow \Delta'';\Gamma'';T'}{\Delta;\Gamma;S \to T \vdash p\ q \searrow \Delta'';\Gamma'';T'}$$

$$\frac{\Delta;\Gamma;\mathcal{F}_d(\nu\mathcal{F})\ \vec{C} \vdash q \searrow \Delta';\Gamma';T}{\Delta;\Gamma;\nu\mathcal{F}\ \vec{C} \vdash .d\ q \searrow \Delta';\Gamma';T} \qquad \frac{\Delta, u{:}U;\Gamma;T \vdash q \searrow \Delta';\Gamma';T'}{\Delta;\Gamma;\Pi u{:}U.T \vdash u\ q \searrow \Delta';\Gamma';T'}$$

Figure 2.10: Type Checking for Patterns

judgments:

$$\begin{array}{llll} \Delta;\Gamma & \vdash\ p : T & \searrow \Delta';\Gamma' & \text{Typing for pattern } p \\ \Delta;\Gamma;T & \vdash\ q & \searrow \Delta';\Gamma';T' & \text{Typing for copattern } q \end{array}$$

In both typing judgments, the index-context Δ and the context Γ contain variable declarations that were introduced at the outside. We assume that all variables occurring in the (co)pattern are fresh with respect to Δ and Γ and occur linearly, although this is not explicitly enforced in our rules. When we check a pattern p against a type T in the index-context Δ and context Γ, we synthesize an index-context Δ' such that Δ' is an extension of Δ (i.e. $\Delta \prec \Delta'$) and Γ' is an extension of Γ. We note that as we check the pattern p we may update and constrain some of the variables already present in Δ. This happens in the rule for $\mathtt{pack}\,\langle C, p\rangle$ where we fall back to type checking patterns in our domain and in the rule

for $\wp$ where we unify two index objects C_1 and C_2, and return a new index-context Δ' such that $\Delta' \vdash C_1 = C_2$. For simplicity, we thread through both the index-context Δ and the context Γ, although only Δ may actually be refined.

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

Copattern spines allow us to make observations on a type T in the index-context Δ and context Γ. As we process the copattern spine from left to right, we synthesize a type T'. Intuitively, T' is the suffix of T. As copattern spines also contain patterns we also return a new index-context Δ' and context Γ'.

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

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

2.4 Operational Semantics

In order to set the stage for the operational semantics, we first discuss value typing as values will guide the flow of the semantics in order to make it deterministic. Value typing is similar to term typing and is split into judgments for terms, heads, and spines (Figure 2.11). Unlike term typing, value typing is done exclusively on closed terms. Terms with empty spines $v \star \cdot$

$\boxed{t \text{ value}}$ Term t is a value.

$$\frac{v \text{ value}}{v \star \cdot \text{ value}} \qquad \frac{v \text{ value} \quad E \text{ value} \quad v \diamond E}{v \star E \text{ value}}$$

$\boxed{h \text{ value}}$ Head h is a value.

$$\frac{}{() \text{ value}} \qquad \frac{v_1 \text{ value} \quad v_2 \text{ value}}{(v_1, v_2) \text{ value}} \qquad \frac{v \text{ value}}{c\ v \text{ value}} \qquad \frac{}{\wp \text{ value}} \qquad \frac{v \text{ value}}{\texttt{pack}\,\langle C, v\rangle \text{ value}} \qquad \frac{\text{for each } i\ b_i \neq \cdot \mapsto t_i}{\texttt{fun}\ f.\vec{b} \text{ value}}$$

$\boxed{E \text{ value}}$ Spine E is a value

$$\frac{}{\cdot \text{ value}} \qquad \frac{E \text{ value}}{C\ E \text{ value}} \qquad \frac{v \text{ value} \quad E \text{ value}}{v\ E \text{ value}} \qquad \frac{E \text{ value}}{.d\ E \text{ value}}$$

Figure 2.11: Definition of values

are values at type T if their heads v are values at type T. If the spine E is not empty, we require both the heazvd h and the spine to be themselves values, and for the head to not match with the spine, denoted by $h \diamond E$. We shall discuss this judgment after we introduce matching.

Spines are values if terms in them are values. Unit () and reflexivity $\wp$ are values. Pairs of values (v_1, v_2) are themselves values. Packing an index with a value $\texttt{pack}\,\langle C, v\rangle$ is also value. Constructors applied to values c $v$are values. Functions are always values as long as no branch has an empty copattern as its left-hand side. This last condition is important as a function with an empty copattern would be able to reduce against any spine.

Matching appears in Figure 2.12. It is split between a matching of valued terms against patterns, and of valued spines against copatterns. In the former case, we return substitutions θ and σ which make both sides equal. We note that while matching is done on terms, unless the pattern is a variable, the matching will effectively be done against the head, with the spine being empty. Variables match against anything. Constructors match if they share the same constructor and their bodies match. Pairs match pointwise. Dependent pairs match if their

$\boxed{v = [\theta;\sigma]p}$ Value v matches against pattern p yielding substitutions $\theta;\sigma$.

$$\frac{}{v = [\cdot; v/x]x,} \quad \frac{v \cdot = [\theta;\sigma]p}{(c\ v) \cdot = [\theta;\sigma]c\ p} \quad \frac{}{\wp = [\cdot;\cdot]\wp}$$

$$\frac{v_1 \cdot = [\theta_1;\sigma_1]p_1 \quad v_2 \cdot = [\theta_2;\sigma_2]p_2}{(v_1, v_2) \cdot = [\theta_1,\theta_2;\sigma_1,\sigma_2](p_1, p_2)} \quad \frac{v \cdot = [\theta;\sigma]p}{(\mathtt{pack}\,\langle C, v\rangle) \cdot = [\theta, C/u;\sigma]\mathtt{pack}\,\langle u, p\rangle}$$

$\boxed{E = [\theta;\sigma]q@E'}$ Spine E matches against copattern q yielding substitutions $\theta;\sigma$ and leftover spine E'.

$$\frac{}{E = [\cdot;\cdot] \cdot @E} \quad \frac{v = [\theta_1;\sigma_1]p \quad E = [\theta_2;\sigma_2]q@E'}{v\ E = [\theta_1,\theta_2;\sigma_1,\sigma_2](p\ q)@E'}$$

$$\frac{E = [\theta;\sigma]q@E'}{.d\ E = [\theta;\sigma](.d\ q)@E'} \quad \frac{E = [\theta;\sigma]q@E'}{C\ E = [\theta, C/u;\sigma](u\ q)@E'}$$

$\boxed{E \stackrel{?}{=} q}$ Spine E partially matches against copattern q.

$$\frac{q \neq \cdot}{\cdot \stackrel{?}{=} q} \quad \frac{v = [\theta;\sigma]p \text{ for some } \theta;\sigma \quad E \stackrel{?}{=} q}{v\ E \stackrel{?}{=} p\ q} \quad \frac{E \stackrel{?}{=} q}{.d\ E \stackrel{?}{=} .d\ q} \quad \frac{E \stackrel{?}{=} q}{C\ E \stackrel{?}{=} u\ q}$$

Figure 2.12: Matching

right-hand sides match; we simply substitute the index-term C in ther pattern variable u. Matching of spines against copatterns returns substitutions θ and σ, and a leftover spine E'. The copattern may be shorter than the spine and thus only consume part of it. Observations must simply be the same on both sides. Patterns are simply matched against values using the previous judgment. We simply associate index-terms to index-variables in copatterns and add them to the substitution.

It will happen that a spine E does not match with a copattern q, written $E \neq q$ if there are no substitutions $\theta;\sigma$ or spines E' such that $E = [\theta;\sigma]q@E'$. We note that we can decide this negation of matching simply because we can decide matching, which is shown in the

following lemma:

Lemma 2.2. *The judgments $v = [\theta;\sigma]p$ and $E = [\theta;\sigma]q@E'$ are decidable.*

Proof. By induction on p and q and case analysis on v and E, respectively. □

As we mentioned above, the judgment $h \diamond E$ indicates that the head h will not form a reduction with the spine E and thus the term $h \star E$ is a value. In our copattern language, it has the single rule

$$\frac{\text{for all } i,\ E \neq q_i \quad \text{for some } i,\ E \stackrel{?}{=} q_i}{(\texttt{fun}\ f.\overrightarrow{q \mapsto t}) \diamond E}$$

We encode this rule in its own judgment even though there is a single rule in order to maintain a uniform representation between this language and our core calculus from Chapter 3. The operational semantics of the latter requires more than rule to indicate when matching of a head and a spine fails to produce a reduction.

Let us focus for now on this one rule. The first assumption simply indicates the spine does not match against any of the copatterns, since a matching branch would trigger a reduction. The second assumption requires that at least one branch represents a partial match. Partial matches are spines matching a prefix of q but not the whole of it. We simply require the spine E to be of shorter length than the copattern, and we need to make sure the observations match, and the values match their corresponding patterns. The reason we require partial matches is to reject stuck terms from a non-covering function to be considered values. This way, we distinguish between terms that are stuck from terms that are waiting for additional inputs. The progress theorem we define below thus will reject stuck terms but accept terms that are awaiting additional inputs.

Let us now discuss the definition and application of substitutions (Figure 2.13). Its definition follows the definition of index-substitutions from Section 2.2. Substitutions σ are lists of pairs of variables x and values v. When applied to x, it replaces it with its corresponding value v. If the variable it is applied to is not in σ, then it just leaves it alone. As such, the empty substitution $\cdot$ acts simply as a identity substitution. Applying substitutions to terms is done pointwise. We assume that copatterns q in $\texttt{fun}\ f.\overrightarrow{q \mapsto t}$ only

$\boxed{[\theta;\sigma]t = t'}$ Applying substitutions θ and σ to term t yields term t'

$$\begin{array}{lll} [\theta;\sigma]x & = \sigma(x) & [\theta;\sigma](h \star E) = [\theta;\sigma]h \star [\theta;\sigma]E \\ [\theta;\sigma]() & = () & [\theta;\sigma](t_1, t_2) = ([\theta;\sigma]t_1, [\theta;\sigma]t_2) \\ [\theta;\sigma](c\ t) & = c\ [\theta;\sigma]t & [\theta;\sigma](\mathtt{pack}\,\langle C, t\rangle) = \mathtt{pack}\,\langle [\theta]C, [\theta;\sigma]t\rangle \\ [\theta;\sigma]\wp & = \wp & [\theta;\sigma](\mathtt{fun}\ f.\overrightarrow{q \mapsto t}) = \mathtt{fun}\ f.\overrightarrow{q \mapsto [\theta;\sigma]t} \end{array}$$

$\boxed{\sigma(x) = v}$ Variable x is bound to value v in substitution σ.

$$\begin{array}{ll} (\sigma, v/x)(x) & = t \\ (\sigma, v/y)(x) & = \sigma(x) \text{ if } x \neq y \\ (\cdot)(x) & = x \end{array}$$

$\boxed{\Delta;\Gamma \vdash \sigma : \Gamma'}$ Substitution σ maps context Γ' to context Γ in index-context Δ.

$$\frac{}{\Delta;\Gamma \vdash \sigma : \cdot} \qquad \frac{\Delta;\Gamma \vdash \sigma : \Gamma' \quad \Delta;\Gamma \vdash \sigma(x) : T}{\Delta;\Gamma \vdash \sigma : \Gamma', x{:}T}$$

Figure 2.13: Typing and application of substitutions

introduce fresh variables. Alternatively, one would apply α-renaming to copatterns and their right-hand sides to keep variable fresh. Thus, pushing the substitution in will keep those bound variables intact.

The typing of substitutions also follows the one for index-substitutions. It is denoted by the judgment $\Delta;\Gamma \vdash \sigma : \Gamma'$. It is defined inductively on the context Γ'. Any well-formed substitution σ can be used as weakening substitutions from the empty context $\cdot$ as there will be no free variable to which apply σ. A substitution σ from the context $\Gamma', x{:}T$ to context Γ in Δ is well typed if $\Delta;\Gamma \vdash \sigma(x) : T$ and σ is a valid substitution from Γ' to Γ.

Even though typing is done inductively on Γ', we admit the following typing derivations for $\cdot$ and $\sigma, v/x$:

Lemma 2.3. *The empty substitution $\cdot$ has typing $\Delta;\Gamma,\Gamma' \vdash \cdot : \Gamma'$.*

$\boxed{t \longrightarrow t'}$ Term t steps to term t'

$$\frac{h \longrightarrow h'}{h \star E \longrightarrow h' \star E} \quad \frac{h \text{ value} \quad E \longrightarrow E'}{h \star E \longrightarrow v \star E'} \quad \frac{E \text{ value} \quad E = [\theta;\sigma]q_i@E'}{(\texttt{fun } f.\overrightarrow{q \mapsto t}) \star E \longrightarrow [\theta;\sigma,(\texttt{fun } f.\overrightarrow{q \mapsto t})/f]t \star E'}$$

$\boxed{h \longrightarrow h'}$ Head h steps to head h'

$$\frac{t_1 \longrightarrow t_1'}{(t_1,t_2) \longrightarrow (t_1',t_2)} \quad \frac{v_1 \text{ value} \quad t_2 \longrightarrow t_2'}{(v_1,t_2) \longrightarrow (v_1,t_2')} \quad \frac{t \longrightarrow t'}{\texttt{pack}\,\langle C,t\rangle \longrightarrow \texttt{pack}\,\langle C,t'\rangle} \quad \frac{t \longrightarrow t'}{c\ t \longrightarrow c\ t'}$$

$\boxed{E \longrightarrow E'}$ Spine E steps to spine E'

$$\frac{t \longrightarrow t'}{t\ E \longrightarrow t'\ E} \quad \frac{v \text{ value} \quad E \longrightarrow E'}{v\ E \longrightarrow v\ E'} \quad \frac{E \longrightarrow E'}{.d\ E \longrightarrow .d\ E'} \quad \frac{E \longrightarrow E'}{C\ E \longrightarrow C\ E'}$$

Figure 2.14: Operational Semantics

Proof. By induction on Γ'. □

Lemma 2.4. *If* $\Delta;\Gamma \vdash \sigma : \Gamma'$ *and* $\Delta;\Gamma \vdash v : T$ *then* $\Delta;\Gamma \vdash \sigma, v/x : \Gamma', x{:}T$.

Proof. By induction on $\Delta;\Gamma \vdash \sigma : \Gamma'$. The inductive case makes use of the exchange property for Γ'. □

We recall that variables are heads, and substitutions replace them with terms. Thus, if $\sigma(x) = h \star E$, the substitution $[\theta;\sigma](x \star E')$ will be $(h \star E) \star [\theta;\sigma]E'$. As we mentioned on Page 36, this is notation for $h \star E\ [\theta;\sigma]E'$, which is a well defined term.

Now that everything is in place, we can discuss the operational semantics. It appears in Figure 2.14. We step applications pointwise. Heads and spines simply follow congruence rules on subterms. In order to obtain a deterministic operational semantics, we restrict the flow of the semantics to only be applied when a subterm v is a value.

Since we are under closed terms, the only applications with non empty spines have functions as heads. We do β-reduction by matching E against a q_i, yielding substitutions θ

and σ and leftover spine E'. Since copatterns are linear, the substitutions are simply obtained by associating to the variables their corresponding terms in E. The leftover spine E' is then obtained by simply taking the tail of the spine E which exceeds in length the matching q_i. The resulting step yields the application $[\theta; \sigma, (\texttt{fun}\ f.\overrightarrow{q \mapsto t})/f]t_i \star E'$. Once again, we recall the notation introduced at Page 36 to justified this last term being well-formed even though $[\theta; \sigma, (\texttt{fun}\ f.\overrightarrow{q \mapsto t})/f]t_i$ is itself a term.

We note that by forcing some rules to be usable only when some subterms are values, we ensure the operational semantics is deterministic. For example, applications will first evaluate the left-hand side, then evaluate the right-hand side.

We also define the multi steps judgments $t \longrightarrow^* t'$ and $h \longrightarrow^* h'$ and $E \longrightarrow^* E'$. They are simply defined as the reflexive transitive closure of their one step equivalent $t \longrightarrow t'$. We only provide the rules for one of them. The others follow the same pattern.

$$\frac{}{t \longrightarrow^* t} \qquad \frac{t \longrightarrow t_1 \quad t_1 \longrightarrow^* t'}{t \longrightarrow^* t'}$$

Additionally, we will use a variant of this judgment in which the number of steps is explicit: $t \longrightarrow^n t'$. Once again, it has equivalent formulations for heads and spines. The rules for them are as expected:

$$\frac{}{t \longrightarrow^0 t} \qquad \frac{t \longrightarrow t_1 \quad t_1 \longrightarrow^n t'}{t \longrightarrow^{n+1} t'}$$

Type Preservation

Let us prove that the operational semantics preserve types. As a notational convenience, we define combined typing judgments for substitutions and evaluation contexts as follow:

$$\frac{\Delta \vdash \theta : \Delta' \quad \Delta; \Gamma \vdash \sigma : [\theta]\Gamma'}{\Delta; \Gamma \vdash (\theta; \sigma) : (\Delta'; \Gamma')} \qquad \frac{\vdash (\theta; \sigma) : (\Delta; \Gamma) \quad [\theta]T \vdash_v E \searrow T'}{\vdash (\theta; \sigma; E) : (\Delta; \Gamma; T) \searrow T'}$$

Lemma 2.5 ((Co)patterns are stable under substitutions)**.** *The following hold:*

1. *If* $\Delta; \Gamma \vdash p : T \searrow \Delta'; \Gamma'$ *and* $\Delta_1; \Gamma_1 \vdash (\theta; \sigma) : (\Delta; \Gamma)$ *then there are* Δ'_1 *and* Γ'_1 *such that* $\Delta_1; \Gamma_1 \vdash p : [\theta]T \searrow \Delta'_1; \Gamma'_1$ *and* $\Delta'_1; \Gamma'_1 \vdash (\theta; \sigma) : (\Delta'; \Gamma')$.

2. *If* $\Delta;\Gamma;T \vdash q \searrow \Delta';\Gamma';T'$ *and* $\Delta_1;\Gamma_1 \vdash (\theta;\sigma) : (\Delta;\Gamma)$ *then there are* Δ'_1 *and* Γ'_1 *such that* $\Delta_1;\Gamma_1;[\theta]T \vdash q \searrow \Delta'_1;\Gamma'_1;[\theta]T'$ *and* $\Delta'_1;\Gamma'_1 \vdash (\theta;\sigma) : (\Delta';\Gamma')$.

Proof. Implicitly we assume that the variables bound in p and q are fresh for $\mathrm{dom}(\theta;\sigma)$. The proofs are done by induction on $\Delta;\Gamma \vdash p : T \searrow \Delta';\Gamma'$ and $\Delta;\Gamma;T \vdash q \searrow \Delta';\Gamma';T'$. The case for $\wp$ in Statement 1 makes use of Lemma 2.1. □

Lemma 2.6 (Substitution lemma)**.** *The following hold:*

1. *If* $\Delta;\Gamma \vdash t : T$ *and* $\Delta';\Gamma' \vdash (\theta;\sigma) : (\Delta;\Gamma)$ *then* $\Delta';\Gamma' \vdash [\theta;\sigma]t : [\theta]T$.
2. *If* $\Delta;\Gamma \vdash h : T$ *and* $\Delta';\Gamma' \vdash (\theta;\sigma) : (\Delta;\Gamma)$ *then* $\Delta';\Gamma' \vdash [\theta;\sigma]h : [\theta]T$.
3. *If* $\Delta;\Gamma;T \vdash E \searrow T'$ *and* $\Delta';\Gamma' \vdash (\theta;\sigma) : (\Delta;\Gamma)$ *then* $\Delta';\Gamma';[\theta]T \vdash [\theta;\sigma]E \searrow [\theta]T'$.

Proof. By mutual induction on the derivations $\Delta;\Gamma \vdash t : T$, and $\Delta;\Gamma \vdash h : T$, and $\Delta;\Gamma;T \vdash E \searrow T'$, respectively. Cases involving indices use Req. 1. The function case uses Lemma 2.5. □

Lemma 2.7. *If* $\Delta;\Gamma;T \vdash E \searrow T'$ *and* $\Delta;\Gamma;T' \vdash E' \searrow T''$ *then* $\Delta;\Gamma;T \vdash E\ E' \searrow T''$.

Proof. By induction on $\Delta;\Gamma;T \vdash E \searrow T'$. □

We recall that if $t = h \star E'$, then we defined the operation $t \star E$ as $h \star E'\ E$. The following lemma shows this operation preserves typing.

Lemma 2.8. *If* $\Delta;\Gamma \vdash t : T$ *and* $\Delta;\Gamma;T \vdash E \searrow T'$ *then* $\Delta;\Gamma \vdash t \star E : T'$.

Proof. By induction on $\Delta;\Gamma \vdash t : T$. At the base case is $t = h \star E'$ and $\Delta;\Gamma \vdash h : S$ and $\Delta;\Gamma;S \vdash E' \searrow T$. By Lemma 2.7, we have $\Delta;\Gamma;S \vdash E'\ E \searrow T'$ and so $\Delta;\Gamma \vdash h \star E'\ E : T'$. □

Lemma 2.9 (Adequacy of (co)pattern matching)**.** *the following hold:*

1. *Suppose* $\vdash v : T$ *where* v *is a value. If* $\vdash p : T \searrow \Delta;\Gamma$ *and* $v = [\theta;\sigma]p$, *then* $\vdash (\theta;\sigma) : (\Delta;\Gamma)$.

2. *Suppose* $T \vdash E \searrow T''$ *where* E *is a value. If* $T \vdash q \searrow \Delta; \Gamma; T'$ *and* $E = [\theta; \sigma]q@E'$ *then* $\vdash (\theta; \sigma; E') : (\Delta; \Gamma; T') \searrow T''$.

Proof. By induction on the judgments $v = [\theta; \sigma]p$ and $E = [\theta; \sigma]q@E'$. The case for $\wp$ uses Req. 3. □

Theorem 2.10 (Type preservation)**.** *The following hold:*

1. *If* $\vdash t : T$ *and* $t \longrightarrow t'$ *then* $\vdash t' : T$.
2. *If* $\vdash h : T$ *and* $h \longrightarrow h'$ *then* $\vdash h' : T$.
3. *If* $T \vdash E \searrow T'$ *and* $E \longrightarrow E'$ *then* $T \vdash E' \searrow T'$.

Proof. By induction on the stepping derivation. The interesting case is

$$\frac{E \text{ value} \quad E = [\theta; \sigma]q_i@E' \quad \sigma' = \sigma, (\texttt{fun}\ f.\overrightarrow{q \mapsto t})/f}{(\texttt{fun}\ f.\overrightarrow{q \mapsto t})\ E \longrightarrow [\theta; \sigma']t_i \star E'}$$

$\vdash \texttt{fun}\ f.\overrightarrow{q_i \mapsto t_i} : T'$ and $T' \vdash E \searrow T$	by inversion on typing.
$f{:}T'; T' \vdash q_i \searrow \Delta_i; \Gamma_i; T_i$ and $\Delta_i; \Gamma_i \vdash t_i : T_i$	by inversion on typing.
$T' \vdash q_i \searrow \Delta_i; \Gamma'_i; T_i$ where $\Gamma_i = \Gamma'_i, f{:}T'$	by Lemma 2.5.
$\vdash (\theta; \sigma; E') : (\Delta_i; \Gamma'_i; T_i) \searrow T$	by Lemma 2.9.
$\vdash (\theta; \sigma, (\texttt{fun}\ f.\overrightarrow{q \mapsto t})/f; E') : (\Delta_i; \Gamma_i; T_i) \searrow T$	by definition of substitution.
$\vdash [\theta; \sigma, (\texttt{fun}\ f.\overrightarrow{q \mapsto t})/f]t_i : [\theta]T_i$	by Lemma 2.6.
$\vdash [\theta; \sigma]t_i \star E' : T$	by Lemma 2.8.

□

2.5 Coverage and Progress

In this section, we define a notion of coverage for copatterns. That is, applying any spine long enough to a function with a covering copattern set will yield a match. The goal of coverage is twofold. On the one hand, it is required to prove progress, that is, that evaluation does not get stuck because of missing copattern cases. The proof of progress apppears at the end

of this section. This also ensures that the encoding of a proof covers all possible cases. In addition, coverage will be used in Chapter 3 to establish a normalisation criterion for our language.

Together with the definition of coverage, we provide proofs of that it is sound and decidable. The former states that copatterns sets accepted by the coverage algorithm are indeed covering all inputs of a given type. The latter ensures we can decide for any copattern set whether it is covering based on the algorithm.

$\boxed{E \triangleleft Q : T}$ Copattern set Q at type T matches against a spine E or its extension

$$\frac{\exists q \in Q \quad E = [\theta;\sigma]q@E'}{E \triangleleft Q : T} \qquad \frac{\forall d \in \mathcal{F} \quad E@.d \triangleleft Q : \mathcal{F}(\nu\mathcal{F})\ \vec{C}}{E \triangleleft Q : \nu\mathcal{F}\ \vec{C}}$$

$$\frac{\forall v : S \quad E@v \triangleleft Q : T}{E \triangleleft Q : S \to T} \qquad \frac{\forall C : U \quad E@C \triangleleft Q : [C/u]T}{E \triangleleft Q : \Pi u{:}U.T}$$

Figure 2.15: Covering of a copattern set

Let us talk first about what it means to be covering all inputs. Given Type T, let Q be a set of tuples $q \searrow \Delta';\Gamma';T'$ such that for each tuple, the judgment $T \vdash q \searrow \Delta';\Gamma';T'$ holds. We shall write $T \vdash Q$ when the type the copatterns in Q eliminate is relevant. Let us choose a valued spine E and a copattern set Q such that $T \vdash_v E \searrow T'$ and $T \vdash Q$ hold. Then the judgment $E \triangleleft Q : T'$ indicates that the copattern set Q covers E or its extensions. That is, either there is a $q \searrow \Delta_i;\Gamma_i;T_i \in Q$ where $E = [\theta;\sigma]q@E'$ for some $\theta;\sigma;E'$, or we can add any well typed element to the tail of E until one of them does. This brings us to the following definitions:

Definition 2.1 (Covering sets and functions)**.** A copattern set $T \vdash Q$ is said to be *covering* if for all values $T \vdash E \searrow T'$ we have $E \triangleleft Q : T'$. Moreover, a function $\vdash$ `fun` $f.\overrightarrow{q \mapsto t} : T$ is *covering* if its copattern set $T \vdash \{\vec{q}\}$ is covering.

We note that we can discuss coverage of a function in an open context by simply choosing some arbitrary instantiations θ and σ and looking at the copattern set $[\theta]T \vdash \{\vec{q}\}$. This follows from Lemma 2.5.

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

For copattern $T \vdash q \searrow \Delta; \Gamma; T'$ and copattern set $T \vdash Q$, we define judgments $T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* Q$ and $q \searrow \Delta; \Gamma; T' \Longrightarrow Q$ both defined in Figure 2.16. The main judgment $T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* Q$ means that the (finite) set Q of copatterns is the coverage-preserving refinement of the copattern q. The copattern q is being refined by applying successively any of the single refinement rules defined by the second judgment $q \searrow \Delta; \Gamma; T' \Longrightarrow Q$.

There are two different types of refinements. The first one introduces a variable or observation by analysis on the resulting type. If we have an arrow type $S \to T$, we introduce a variable of that type, yielding the copattern $q \mathbin{@} x$. If we have a corecursive type $\nu\mathcal{F}\ \vec{C}$, for each observation $d \in \mathcal{F}$, we create a new copattern $q \mathbin{@} .d$.

The second type of refinement splits on a variable. We expose a variable occuring in q, and its type in Δ or Γ. We write $q[x]$ for a copattern q with a single distinguished position in which the variable x occurs. We consider in this judgment the contexts to be unordered, so the notation $\Gamma, x : T$ (or $\Delta, u : U$) is simply to expose any variable $x \in \Gamma$ ($X \in \Delta$, respectively), no matter its actual position in the context. The splitting is done by examining the type of the exposed variable. If $x : T_1 \times T_2$, we introduce two new variables $x_1 : T_1$ and $x_2 : T_2$ and perform the instantiation $q[(x_1, x_2)]$. If the variable is of recursive type $\mu\mathcal{F}\ \vec{C}$, we introduce a new copattern for each constructor $c \in \mathcal{F}$ with the variable replaced by $c\ x'$ where $x' : \mathcal{F}_c(\mu\mathcal{F})\ \vec{C}$. If we have an equality constraint $C_1 = C_2$, we replace it by

$\boxed{q \searrow \Delta; \Gamma; T \Longrightarrow Q}$ Copattern q refines into copatterns Q.

(Co)Pattern Introduction

$$
\begin{array}{lcl}
(q \searrow \Delta; \Gamma; \Pi u{:}U.T) & \Longrightarrow & \{q@u \searrow \Delta, u{:}U; \Gamma; T\} \\
(q \searrow \Delta; \Gamma; S \to T) & \Longrightarrow & \{q@x \searrow \Delta; \Gamma, x{:}S; T\} \\
(q \searrow \Delta; \Gamma; \nu\mathcal{F}\ \vec{C}) & \Longrightarrow & \{q@.d_i \searrow \Delta; \Gamma; \mathcal{F}_i(\nu\mathcal{F})\ \vec{C}\}_i
\end{array}
$$

Pattern Refinement

$$
\begin{array}{lcl}
(q[x] \searrow \Delta; \Gamma, x : C_1 = C_2; T) & \Longrightarrow & \{q[\wp] \searrow \Delta'; \Gamma; T\} \quad \text{provided } \Delta \vdash C_1 = C_2 \searrow \Delta' \\
(q[x] \searrow \Delta; \Gamma, x : T_1 \times T_2; T) & \Longrightarrow & \{q[(x_1, x_2)] \searrow \Delta; \Gamma, x_1{:}T_1, x_2{:}T_2; T\} \\
(q[x] \searrow \Delta; \Gamma, x : \Sigma u{:}U.T'; T) & \Longrightarrow & \{q[\mathtt{pack}\,\langle u, x'\rangle] \searrow \Delta, u{:}U; \Gamma, x'{:}T'; T\} \\
(q[x] \searrow \Delta; \Gamma, x : \mu\mathcal{F}\ \vec{C}; T) & \Longrightarrow & \{q[c_i\ x_o] \searrow \Delta; \Gamma, x'{:}\mathcal{F}_i(\mu\mathcal{F})\ \vec{C}; T\}_i
\end{array}
$$

$\boxed{T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* Q}$ Copattern $q \searrow \Delta; \Gamma; T'$ can be refined in multiple steps into copatterns Q.

$$\frac{}{T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* \{q \searrow \Delta; \Gamma; T'\}}$$

$$\frac{q \searrow \Delta; \Gamma; T' \Longrightarrow Q' \quad \text{for all } (q_i \searrow \Delta_i; \Gamma_i; T_i) \in Q', \quad T \vdash q_i \searrow \Delta_i; \Gamma_i; T_i \Longrightarrow^* Q_i}{T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* \bigcup_i Q_i}$$

Figure 2.16: Coverage

$\wp$. As such, we keep all the branches no matter whether they turned out to be unreachable. This is needed to be able to decide coverage, which is in turn necessary for the normalisation criteria in Chapter 3.

Theorem 2.11 (Decidability of coverage)**.** *The judgment $T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* Q$ is decidable.*

Proof. Each copattern refinement either introduces a new element in the copattern spine (a variable, a meta-variable or an observation), or introduces a language construct ($\wp$, (de-

pendent) pairs, constructors). Thus, a derivation for the copattern set will be bounded by the sum of the lengths of all copatterns in the set plus all language constructs in all the copatterns. Since this number is finite, $T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* Q$ is decidable. □

Let us now prove soundness of the copattern refinement rules. Soundness ensures that a covering set will indeed match any input of adequate length. Moreover, if an input is too short to trigger a match, then any long enough extension will trigger one. We encode that idea in the judgmentFirst, we prove that refinements preserve coverage. This, combined with the fact that the empty copattern covers all input, will ensure that any copattern set resulting from copattern refinements will be covering.

Lemma 2.12. *If $E = [\theta; \sigma]q$ and $E' = [\theta'; \sigma']q'$ then $E\ E' = [\theta, \theta'; \sigma, \sigma'](q\ q')$.*

Proof. By induction on $E = [\theta; \sigma]q$. □

Lemma 2.13. *The following hold:*

1. *If $E = [\theta; \sigma]q@(v\ E')$, then $E = [\theta; \sigma, v/x](q\ x)@E'$.*
2. *If $E = [\theta; \sigma]q@(C\ E')$, then $E = [\theta, C/u; \sigma](q\ u)@E'$.*
3. *If $E = [\theta; \sigma]q@(.d\ E')$, then $E = [\theta; \sigma](q\ .d)@E'$.*
4. *If $E = [\theta; \sigma](q[x])@E'$, then $E = [\theta; \sigma'](q[\wp])@E'$ where $\sigma = \sigma', \wp/x$.*
5. *If $E = [\theta; \sigma](q[x])@E'$, then $E = [\theta; \sigma'](q[(x_1, x_2)])@E'$ where $\sigma' = \sigma'', v_1/x_1, v_2/x_2$ and $\sigma = \sigma'', (v_1, v_2)/x$ for some σ''.*
6. *If $E = [\theta'; \sigma](q[x])@E'$, then $E = [\theta; \sigma'](q[\mathtt{pack}\,\langle u, x'\rangle])@E'$ where $\sigma' = \sigma'', v/x'$ and $\sigma = \sigma'', (\mathtt{pack}\,\langle C, v\rangle)/x$ for some σ'' and $\theta = \theta', C/u$.*
7. *If $E = [\theta; \sigma](q[x])@E'$, then $E = [\theta; \sigma'](q[c\ x'])@E'$ where $\sigma' = \sigma'', v/x'$ and $\sigma = \sigma'', (c\ v)/x$ for some σ''.*

Proof. By induction on the matching derivations. □

Lemma 2.14. *If $q \searrow \Delta; \Gamma; T' \Longrightarrow Q$ and $T \vdash E \searrow T''$ where E is a value and $E = [\theta; \sigma]q@E'$, for some $\theta; \sigma; E'$, then $E \triangleleft Q : T''$.*

Proof. By case analysis on the judgment $q \searrow \Delta; \Gamma; T' \Longrightarrow Q$.

Case: $q \searrow \Delta; \Gamma; \Pi u{:}U.T_1 \Longrightarrow \{q\ u \searrow \Delta, u{:}U; \Gamma; T_1\}$

By Lemma 2.9, we have $\Pi u{:}[\theta]U.[\theta]T_1 \vdash E' \searrow T''$. By inversion, either $E' = \cdot$ or $E' = C\ E''$ for some $\vdash C : [\theta]U$. In the former case, let us choose some $C'{:}[\theta]U$. By definition of matching, we have $C' = [C'/u]u$. By Lemma 2.12, $E\ C = [\theta, C'/u; \sigma](q\ u)$. Hence, $E@C' \triangleleft \{q@u \searrow \Delta, u{:}U; \Gamma; T_1\} : [\theta, C'/u]T_1$. Since the choice of C' was arbitrary, we can conclude $E \triangleleft \{q\ u \searrow \Delta, u{:}U; \Gamma; T_1\} : \Pi u{:}[\theta]U.[\theta]T_1$. In the latter case, we have $E = [\theta; \sigma]q@(C\ E')$. By Lemma 2.13, we obtain $E = [\theta, C/u; \sigma](q\ u)@E'$ and thus $E \triangleleft \{q\ u \searrow \Delta, u{:}U; \Gamma; T_1\} : [\theta, C/u]T_1$.

The other cases for (co)pattern introduction are similar. The cases for pattern refinement follow directly from Lemma 2.13. □

Theorem 2.15 (Soundness of coverage)**.** *If $T \vdash q \searrow \Delta; \Gamma; T' \Longrightarrow^* Q$ and $T \vdash E \searrow T''$ where E is a value and $E = [\theta; \sigma]q@E'$, for some $\theta; \sigma; E'$, then $E \triangleleft Q : T''$.*

Proof. By induction on the derivation $T \vdash q \Longrightarrow^* Q$. The inductive case is proved using Lemma 2.14. □

Corollary 2.16. *If $T \vdash \cdot \Longrightarrow^* \{\vec{q}\}$, then $\vdash \texttt{fun}\ f.\overrightarrow{q \mapsto t} : T$ is covering.*

Progress

Now we can move on to progress. We want to prove that well typed terms do not get stuck. In order for the theorem to hold, we require every function to be covering.

Lemma 2.17. *The following hold:*

1. *If $E\ v = [\theta; \sigma]q$, then $E \stackrel{?}{=} q$.*
2. *If $E\ C = [\theta; \sigma]q$, then $E \stackrel{?}{=} q$.*
3. *If $E\ .d = [\theta; \sigma]q$, then $E \stackrel{?}{=} q$.*

Proof. All statements are proved the same way. We only prove the first one. By induction on E $v = [\theta;\sigma]q$. The base case is $v \cdot = [\theta;\sigma](p \cdot)$. Then $E = \cdot$ and we can build $\cdot \stackrel{?}{=} p \cdot$. The inductive cases simply appeal to the induction hypothesis. □

Lemma 2.18. *If $E \triangleleft Q : T$ then there is a $q \in Q$ such that either $E = [\theta;\sigma]q@E'$ or $E \stackrel{?}{=} q$.*

Proof. By induction on $E \triangleleft Q : T$. In the base, case we trivially have that $E = [\theta;\sigma]q@E'$. The inductive cases are all similar, so we only show the case for $E \triangleleft Q : S \to T$. By induction hypothesis we have that for some q, either $E\ v = [\theta;\sigma]q@E'$ or $E\ v \stackrel{?}{=} q$. In the former case, if $E' = \cdot$ then by Lemma 2.17, we have $E \stackrel{?}{=} q$. Otherwise, we simply have $E' = E''\ v$ and so $E = [\theta;\sigma]q@E''$. □

Theorem 2.19 (Progress)**.** *Suppose for every function* $\texttt{fun}\ f.\overrightarrow{q \mapsto t}$*, we have* $\cdot \searrow \cdot;\cdot;T' \Longrightarrow^* \{q_i\}$*. Then the following hold:*

1. *If $\vdash t : T$ then either there is a t' such that $t \longrightarrow t'$ or t is a value.*
2. *If $\vdash h : T$ then either there is a h' such that $h \longrightarrow h'$ or h is a value.*
3. *If $T' \vdash E \searrow T$ then either there is a E' such that $E \longrightarrow E'$ or E is a value.*

Proof. By mutual induction on the derivations of $\vdash t : T$, and $\vdash h : T$, and $T' \vdash E \searrow T$. Most cases simply make appeal to the induction hypotheses. We present the interesting case:

$$\textbf{Case:}\quad \frac{\vdash \texttt{fun}\ f.\overrightarrow{q \mapsto h\ E} : T' \quad T' \vdash E \searrow T}{\vdash \texttt{fun}\ f.\overrightarrow{q \mapsto t} \star E : T}$$

By induction hypothesis on $T' \vdash E \searrow T$ we get either E is a value or some E' such that $E \longrightarrow E'$. In the latter case, we are done.

In the former case, by Lemma 2.2, we can decide if $E = [\theta;\sigma]q_i@E'$ for some q_i. If one of them matches, then $\texttt{fun}\ f.\overrightarrow{q \mapsto h\ E} \star E \longrightarrow [\theta;\sigma]t_i \star E'$. If none of them matches, then by Lemma 2.18, we must have $E \stackrel{?}{=} q$. It follows that $\texttt{fun}\ f.\overrightarrow{q \mapsto t} \diamond E$ and so $\texttt{fun}\ f.\overrightarrow{q \mapsto t} \star E$ is a value. □

2.6 Related Work

Our work models finite data using dependent sums and infinite data using dependent records where fields share a given index. This is in contrast to dependent records that allow a particular field to depend on previous ones [Betarte, 1998].

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

Dependent type theories provide in principle support to track data dependencies on infinite data, although this has not received much attention in practice. Agda [Norell, 2007], a dependently typed proof and programming environment based on Martin-Löf's type theory, has support for copatterns since version 2.3.4 [Agda team, 2014]. We can directly define equality guards and using large eliminations we can match on index arguments. Cockx and Abel [2018, 2020] recently refined Agda's termination checking in the presence of copatterns and developed the underlying theory. They assert coverage via splitting trees which bear close resemblance to our coverage derivations.

Agda uses inaccessible patterns (also called dot-patterns) [Brady et al., 2004, Goguen et al., 2006b] to maintain linear pattern matching in a dependently typed setting. Inaccessible patterns mark patterns that are fully determined by their type. They do not bind additional variables not already occuring in the rest of the pattern, which is then linear. Our approach offers an alternative view which is mostly notational where relationships between arguments in a pattern are kept in the index context while the pattern is fully linear. The equality checks are done during type checking and the constraints are irrelevant at run-time since a matching branch will always satisfy all of its constraints.

Our work draws on the distinction between finite data defined by constructors and infinite

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

Our development of indexed patterns and copatterns builds on the growing body of work [Zeilberger, 2007, Licata et al., 2008] which relates focusing and linear logic to programming language theory via the Curry-Howard isomorphism. Zeilberger [2008] and Krishnaswami [2009] have argued that focusing calculi for propositional logic provide a proof-theoretic foundation for pattern matching in the simply-typed setting. Our work extends and continues this line of work to first-order logic (= indexed types) with (co)recursive types and equality. Our work also takes inspiration from the proof theory described by Baelde [2012], Baelde et al. [2010] and the realization of this work in the Abella system [Baelde et al., 2014b]. While Baelde's proof theory supports coinductive definitions and equality, coinduction is defined by a non-wellfounded unfolding of a coinductive definition. Proofs in this work would correspond to programs written by (co)iteration. This is in contrast to our work, which is centered around the duality of (co)data types and supports simultaneous deep (co)pattern matching.

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

2.7 Conclusion

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

We note that we defined typing rules as a type assignment system without regard for decidability of type checking as a property of our language. While we do not treat it in the context of this thesis, we believe it would be straightforward to convert our typing rules to a bidirectional type checker and prove it to be decidable. This hinges on the requirement for our index domains to be decidable.

The rest of this thesis builds on the language we presented above. Chapter 3 describes a normalisation criterion to ensure that programs are terminating and thus can be considered as proofs. Chapter 4 expands on the idea of using contextual LF as an index domain by going over a case study of Howe's method.

Chapter 3

Normalisation

The propositions-as-types principle is the underlying concept allowing us to treat programs as proofs. To ensure validity of proofs, the system used to define proofs must be consistent. In addition, a proof must cover all cases and all calls to (co)induction hypotheses must be justified. A programming language used to encode proofs must equivalently be able to guarantee those properties. We touched on coverage of our copattern language in the previous chapter. However, we haven't discussed whether our language yielded a consistent proof system or how to distinguish valid uses of (co)inductive hypotheses.

In a program, (co)inductive hypotheses correspond to (co)recursive calls. Valid calls to induction hypotheses must be recursive calls made on smaller outputs while valid calls to coinduction hypotheses are corecursive calls made in guarded positions. This is not guaranteed by our type system: our copattern language implements a general recursion scheme which does not disallow uses of (co)recursive calls based on size of the input, or on any form of guardedness. In particular, we can easliy define the (well-typed) term:

```
fun loop : A → A
  | a ↦ loop a
```

for any type `A`. This function, when applied to any term $a : A$, produces an infinite loop. A similar problem arises with the function:

```
fun eager : Stream
```

```
| .head ↦ 0
| .tail ↦ eager.tail
```

Taking `eager.tail` just leads to the infinite unfolding $\mathsf{eager.tail} \longrightarrow \mathsf{eager.tail}$ because the corecursive call is not guarded. Thus, we need to make an external check to ensure functions are safe. This check will look at every recursive call and verify that they are made on smaller arguments than the input, or that the corecursive call is guarded by an observation on the left-hand side and is not unfolded on the right-hand side. Safe functions will be deemed structurally (co)recursive. We shall go over the specifics of the definition in Section 3.3.

This structural (co)recursiveness check, paired together with coverage and well-typedness of programs will serve as conditions to show the language is normalizing. Normalisation will in turn ensure consistency, allowing us to treat our language as a proof assistant. This proof of normalisation is the core of this chapter.

Rather than proving normalisation on our subset language directly, we will show it holds for a core calculus. This core calculus defines (co)recursion using Mendler-style operators [Mendler, 1991] whose typing ensures (co)recursive calls are safe. Thus, in this language every well typed program is terminating. We prove normalisation of this calculus using a logical relation argument following the technique of Tait [1967] and Girard [1972].

Once we have proven that this calculus is normalizing, we need to relate it to our copattern language. To do so, we define a translation from the latter to the former and prove this translation to be normalisation preserving, thus ensuring the translatable subset of our copattern language is itself normalizing. The key part is that for the translation to be normalisation preserving it can only translate a subset of our typable terms as we need to weed out non-terminating programs. To that end, we show that well-typedness, coverage, and structural (co)recursiveness are sufficient conditions for the translation to succeed. Hence, programs satisfying those conditions are normalizing.

This chapter presents the pieces of the proof in the following order. We first introduce the calculus in Section 3.1 and prove its normalisation in Section 3.2. We discuss the criteria for the translation in Section 3.3 and the translation itself in Section 3.4. We show the criteria to be sufficient in Section 3.5, and that the translation preserves normalisation in

Section 3.6.

3.1 Core Calculus

We now describe the core calculus, its syntax and semantics. We prove a substitution lemma, subject reduction, and progress. We reuse the type system from Section 2.3. We will largerly ignore labels attached to variants and records, using instead positional information. For convenience, we copy here the grammar:

$$
\begin{array}{lll}
\text{Kinds} & K & ::= \mathsf{type} \mid \Pi u{:}U.K \\
\text{Types} & S,T & ::= X \mid 1 \mid T_1 \times T_2 \mid S \to T \mid C_1 = C_2 \mid \Pi u{:}U.T \mid \Sigma u{:}U.T \mid T\ \vec{C} \\
 & & \quad \mid \Lambda \vec{u}.T \mid \mu X.T \mid \nu X.T \mid D \mid R \\
\text{Variants} & D & ::= \langle c_1\ T_1 \mid \cdots \mid c_n\ T_n \rangle \\
\text{Records} & R & ::= \{d_1 : T_1, \ldots, d_n : T_n\}
\end{array}
$$

Terms, on the other hand, differ from the ones in our copattern language. Namely, we add elimination forms for terms represented by patterns and we have introduction forms for variables and observations matching.

$$
\begin{array}{lll}
\text{Terms} & t,s & ::= h \star E \\
\text{Heads} & h & ::= x \mid () \mid \lambda x.t \mid \Lambda u.t \mid (t_1; \ldots; t_n) \mid (t_1, t_2) \mid \mathtt{split}\, s\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t \\
 & & \quad \mid \mathtt{in}_i\, t \mid (\mathtt{case}\, t\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i}) \mid \mathtt{pack}\, \langle C, t \rangle \mid \mathtt{unpack}\, t\, \mathtt{as}\, \langle u, x \rangle\, \mathtt{in}\, s \\
 & & \quad \mid \wp \mid \mathtt{eq}\, s\, \mathtt{with}\, t \mid \mathtt{eq\_abort}\, s \mid \mathtt{in}_\mu\, t \mid \mathtt{out}_\mu \mid \mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t \\
 & & \quad \mid \mathtt{in}_\nu \mid \mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t \\
\text{Spines} & E & ::= \cdot \mid .\mathtt{out}_i\, E \mid .\mathtt{out}_\nu\, E \mid t\ E \mid C\ E
\end{array}
$$

We have function abstractions $\lambda x.t$ and $\Lambda u.t$ to introduce term and index variables, respectively. $\mathtt{split}\, s\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t$, and $\mathtt{unpack}\, t\, \mathtt{as}\, \langle u, x \rangle\, \mathtt{in}\, s$ eliminate (dependent) pairs by binding each component to a variable. Equalities are eliminated by $\mathtt{eq}\, s\, \mathtt{with}\, t$ and $\mathtt{eq\_abort}\, s$. The former takes an equality proof s and a body t which is valid under this equality. The latter aborts when the equality s is false.

Instead of using constructors, terms of recursive types are introduced by the "fold" syntax $\mathtt{in}_\mu$ while the i-th branch of a variant is chosen using injections $\mathtt{in}_i$. Elimination of recursive

$\boxed{\Delta; \Xi; \Gamma \vdash t : T}$ Term t is of type T

$$\frac{\Delta; \Xi; \Gamma \vdash t : T' \quad \Delta; \Xi; \Gamma; T' \vdash E \searrow T}{\Delta; \Xi; \Gamma \vdash t \star E : T} \qquad \frac{\Delta; \Xi; \Gamma \vdash t : T' \quad \Delta; \Xi; \Gamma \vdash T = T'}{\Delta; \Xi; \Gamma \vdash t : T}$$

$\boxed{\Delta; \Xi; \Gamma; T \vdash E \searrow T'}$ Spine eliminates function of type T into type T'

$$\frac{\Delta; \Xi; \Gamma \vdash t : S \quad \Delta; \Xi; \Gamma; T \vdash E \searrow T'}{\Delta; \Xi; \Gamma; S \to T \vdash t\ E \searrow T'} \qquad \frac{\Delta \vdash C : U \quad \Delta; \Xi; \Gamma; [C/u]T \vdash E \searrow T'}{\Delta; \Xi; \Gamma; \Pi u{:}U.T \vdash C\ E \searrow T'}$$

$$\frac{}{\Delta; \Xi; \Gamma; T \vdash \cdot \searrow T} \qquad \frac{\Delta; \Xi; \Gamma; \mathcal{F}(\nu\mathcal{F})\ \vec{C} \vdash E \searrow T}{\Delta; \Xi; \Gamma; (\nu\mathcal{F})\ \vec{C} \vdash .\mathtt{out}_\nu\, E \searrow T} \qquad \frac{\Delta; \Xi; \Gamma; R_i \vdash E \searrow T}{\Delta; \Xi; \Gamma; R \vdash .\mathtt{out}_i\, E \searrow T}$$

Figure 3.1: Typing rules for the target language

types is done in two possible ways. The first one is a simple non recursive projection $\mathtt{out}_\mu\, t$ while the second employs a Mendler-style recursor $\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t$. Similarly, we replace observations with a single "unfold" $.\mathtt{out}_\nu$. We handle the different observations by simply projecting the corecursive term afterwards from its product structure using $.\mathtt{out}_i$. Introduction of corecursive types is then done through the injection $\mathtt{in}_\nu\, t$ or via the Mendler-style corecursor $\mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t$.

Typing rules appear in Figures 3.1 and 3.2. Typing judgments for the target language use three contexts: the index-context Δ, the type variable context Ξ, and the term context Γ. For spines, $.\mathtt{out}_\nu$ simply unfolds the corecursive type $\nu\mathcal{F}\ \vec{C}$ into $\mathcal{F}(\nu\mathcal{F})\ \vec{C}$ while $\mathtt{out}_i$ unfolds the record type R into R_i, effectively splitting observations into two parts. Similarly, $\mathtt{in}_\mu$ folds $\mathcal{F}(\mu\mathcal{F})\ \vec{C}$ into $\mu\mathcal{F}\ \vec{C}$, while $\mathtt{in}_i$ injects D_i into D, again effectively splitting the work of constructors into two parts.

We introduce function types and Π-types using λ-abstractions $\lambda x.t$ and $\Lambda u.t$, respectively. Records are simply lazy products $(t_1; \ldots; t_n)$. We eliminate variants using case analysis, and we split (dependent) pairs into their components. Elimination of equalities is done via $\mathtt{eq}\, s\, \mathtt{with}\, t$ if the context Δ makes C_1 and C_2 equal without the use of a contradiction, and

$\boxed{\Delta;\Xi;\Gamma \vdash h : T}$ Head h is of type T

$$\frac{}{\Delta;\Xi;\Gamma \vdash () : 1} \quad \frac{x{:}T \in \Gamma}{\Delta;\Xi;\Gamma \vdash x : T} \quad \frac{\Delta;\Xi;\Gamma, x{:}S \vdash t : T}{\Delta;\Xi;\Gamma \vdash \lambda x.t : S \to T} \quad \frac{\Delta, u : U;\Xi;\Gamma \vdash t : T}{\Delta;\Xi;\Gamma \vdash \Lambda u.t : \Pi u{:}U.T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t_1 : T_1 \quad \Delta;\Xi;\Gamma \vdash t_2 : T_2}{\Delta;\Xi;\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \quad \frac{\Delta;\Xi;\Gamma \vdash p : T_1 \times T_2 \quad \Delta;\Xi;\Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t : T}{\Delta;\Xi;\Gamma \vdash \mathtt{split}\, p\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t : T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t : D_i}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_i\, t : D} \quad \frac{\Delta;\Xi;\Gamma \vdash t : D \quad \Delta;\Xi;\Gamma, x_i{:}D_i \vdash t_i : S}{\Delta;\Xi;\Gamma \vdash (\mathtt{case}\, t\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i}) : S} \quad \frac{\Delta;\Xi;\Gamma \vdash t_i : R_i}{\Delta;\Xi;\Gamma \vdash (t_1; \ldots; t_n) : R}$$

$$\frac{\Delta \vdash C : U \quad \Delta;\Xi;\Gamma \vdash t : T[C/u]}{\Delta;\Xi;\Gamma \vdash \mathtt{pack}\, \langle C, t\rangle : \Sigma u{:}U.\, T} \quad \frac{\Delta;\Xi;\Gamma \vdash t : \Sigma u{:}U.\, T \quad \Delta, u{:}U;\Xi;\Gamma, x{:}T \vdash s : S}{\Delta;\Xi;\Gamma \vdash \mathtt{unpack}\, t\, \mathtt{as}\, \langle u, x\rangle\, \mathtt{in}\, s : S}$$

$$\frac{\Delta \vdash C_1 = C_2}{\Delta;\Xi;\Gamma \vdash \wp : C_1 = C_2} \quad \frac{\Delta;\Xi;\Gamma \vdash s : C_1 = C_2 \quad \Delta \vdash C_1 = C_2 \searrow \Delta' \quad \# \in \Delta'}{\Delta;\Xi;\Gamma \vdash \mathtt{eq\_abort}\, s\ : T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash s : C_1 = C_2 \quad \Delta \vdash C_1 = C_2 \searrow \Delta' \quad \Delta';\Xi;\Gamma \vdash t : T}{\Delta;\Xi;\Gamma \vdash \mathtt{eq}\, s\, \mathtt{with}\, t : T}$$

$$\frac{\Delta;\Xi;\Gamma \vdash t : \mathcal{F}(\mu\mathcal{F})\, \vec{C}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_\mu\, t : (\mu\mathcal{F})\, \vec{C}} \quad \frac{\Delta;\Xi;\Gamma \vdash t : (\mu\mathcal{F})\, \vec{C}}{\Delta;\Xi;\Gamma \vdash \mathtt{out}_\mu\, t : \mathcal{F}(\mu\mathcal{F})\, \vec{C}} \quad \frac{\Delta;\Xi;\Gamma \vdash t : \mathcal{F}(\nu\mathcal{F})\, \vec{C}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_\nu\, t : (\nu\mathcal{F})\, \vec{C}}$$

$$\frac{\Gamma' = \begin{array}{l}\Gamma, f{:}\Pi\overrightarrow{u{:}U}.X\, \vec{C} \to T, x{:}\mathcal{F}\, X\, \vec{C},\\ \iota{:}\Pi\overrightarrow{u{:}U}.X\, \vec{u} \to \mathcal{F}\, X\, \vec{u}, \rho{:}\Pi\vec{u}.X\, \vec{u} \to (\mu\mathcal{F})\, \vec{u}\end{array} \quad \Delta, \overrightarrow{u{:}U};\Xi, X{:}K;\Gamma' \vdash t : T}{\Delta;\Xi;\Gamma \vdash \mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t : \Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\, \vec{C} \to T}$$

$$\frac{\Gamma' = \begin{array}{l}\Gamma, f : \Pi\overrightarrow{u{:}U}.T \to X\, \vec{C}, x : T,\\ \iota : \Pi\overrightarrow{u{:}U}.\mathcal{F}\, X\, \vec{u} \to X\, \vec{u}, \rho : \Pi\vec{u}.(\nu\mathcal{F})\, \vec{u} \to X\, \vec{u}\end{array} \quad \Delta, \overrightarrow{u{:}U};\Xi, X{:}K;\Gamma' \vdash t : \mathcal{F}\, X\, \vec{C}}{\Delta;\Xi;\Gamma \vdash \mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t : \Pi\overrightarrow{u{:}U}.T \to (\nu\mathcal{F})\, \vec{C}}$$

Figure 3.2: Typing rules for the target language (continued)

$\texttt{eq\_abort}\, s$ otherwise.

We can eliminate constructors using Mendler-style recursion $\texttt{rec}\, f, \iota, \rho, \vec{u}, x.\, t$ or unfolds $\texttt{out}_\mu\, t$. Similarly, we can build constant coinductive terms via injections $\texttt{in}_\nu\, t$ or corecursive terms via Mendler-style corecursion $\texttt{corec}\, f, \iota, \rho, \vec{u}, x.\, t$. Recursors and corecursors take in index arguments that are bound to $\vec{u}$, and an input which is bound to x. In the recursive case, x is the argument over which the recursion is done, and thus has type $\mathcal{F} X\, \vec{C}$. Here, the X is the recursive type variable and is being abstracted over. The symbol f stands for the recursive call and has type $\Pi \overrightarrow{u{:}U}.X\, \vec{C} \to T$. This ensures only valid recursive calls can be made as the only terms with the type X are subterms of the input x. The recursor binds extra terms: ι and ρ. Each of them allows us to manipulate the input as it maps the abstract type X into a specific type. $\iota : \Pi \overrightarrow{u{:}U}.X\, \vec{u} \to \mathcal{F}\, X\, \vec{u}$ allows us to unfold further the input and thus perform deep matching. The term $\rho : \Pi \overrightarrow{u{:}U}.X\, \vec{u} \to \mu\mathcal{F}\, \vec{u}$ acts as an identity function, allowing us to output directly the input without having to reconstruct it from the bottom up. At runtime, it gets replaced by the identity function $\Lambda\vec{u}.\lambda x.x$.

The terms ι and ρ extend the basic Mendler-style scheme [Mendler, 1991] for convenience, which will in term simplify the translation by allowing a more direct correspondence. They are not new ideas either; Mendler [1987] himself described first a version of (co)recursors featuring an operator similar to our ρ constant, while we defined ι based on the work of Ahn and Sheard [2011].

Let us have a closer look at what they allow us to do. Using ρ, we can return constants built directly from the input without having to fully reconstruct them. For example, we can build the predecessor function on natural numbers $\mathsf{Nat} = \mu X.1 + X$ directly.

```
pred = rec f, ι, ρ, x.case x of
         | inl y₁ ⇒ inl ()
         | inr y₂ ⇒ ρ ⋆ y₂
```

The variable $\texttt{y}_2$ has type X and thus $\rho \star \texttt{y}_2$ has type Nat.

The term ι on the other hand encodes course-of-value (co)recursion which has also been more commonly described via histomorphisms and futomorphisms [Uustalu and Vene, 1999]. Course-of-value recursion allows us to work with information from the previous cases when

building the current one and is emulated by allowing a form of deep matching. For example, we can define a division by two as follows:

```
div2 = rec f, ι, ρ, x.case x of
         | inl y₁ ⇒ inl ()
         | inr y₂ ⇒ case ι ⋆ y₂ of
                    | inl z₁ ⇒ inl ()
                    | inr z₂ ⇒ inr (f ⋆ z₂)
```

Terms of coinductive types are built in a very dual way. We can build them corecursively using the Mendler-style corecursor $\texttt{corec}\, f, \iota, \rho, \vec{u}, x.\, t$, or we can define constant terms using $\texttt{in}_\nu\, t$. In the former case, the corecursor binds x, the input of arbitrary type that is used as a seed to build the corecursive term. The term t is a term of type $\mathcal{F}\, X\, \vec{C}$ which requires us to provide a term of abstract type in corecursive positions. $f : \Pi\vec{u}.T \to X\, \vec{C}$ gives us access to the corecursive call, taking a new seed and producing such term of type $X\, \vec{C}$. The operator $\iota : \Pi\vec{u}.\mathcal{F}\, X\, \vec{u} \to X\, \vec{u}$ allows us to perform deep observation matching. The operator $\rho : \Pi\vec{u}.\nu\mathcal{F}\, \vec{u} \to X\, \vec{u}$ acts as an identity function, allowing us to put a constant in places expecting a term built via corecursion.

Let us build an example on streams. The type $\nu X{:}K.\, \mathsf{Nat} \times X$ denotes streams of natural numbers. We will build a stream `cycleNats` starting at input value `x` and going down by one until it reaches zero. After that, it will go back to five, before continuing down and repeating. The stream will thus look like $\mathsf{cycleNats}\ 2 = 2, 1, 0, 5, 4, 3, 2, 1, 0, 5, 4, \ldots$ The copattern definition looks like the following:

```
fun cycleNats.
  | x       .head ⇒ x
  | zero    .tail ⇒ cycleNats ⋆ 5
  | (suc x) .tail ⇒ cycleNats ⋆ x
```

The corresponding corecursive definition would thus be:

```
cycleNats = corec f, ι, ρ, x.
  (x; case outμ x of
```

```
                | in1 y1 ↦ f ⋆ 5
                | in2 y2 ↦ f ⋆ y2)
```

In the `head` case, we simply return x. In the `tail` case, we perform matching on $\texttt{out}_\mu$ `x`. The unfolding $\texttt{out}_\mu$ allows us to expose the sum type. If the input is the left injection $\textbf{in}_1$ $\texttt{y}_1$ (that is, the number 0) we make a corecursive call using 5; this is done with the corecursive function f. In the other case, we need to build our stream with seed y_2, which we pass to f. In both cases, we needed to provide a term of type X, which is obtained via f.

Suppose instead of looping back, we simply want to output zero forever once we reach it, that is, the sequence $2, 1, 0, 0, 0, \ldots$. The function would become:

```
countdown = corec f, ι, ρ, x.
  (x; case outμ x of
                | in1 y1 ↦ ρ ⋆ zeroes
                | in2 y2 ↦ f ⋆ y2)
```

Here, `zeroes` is the stream of zeroes that is being inserted as a constant in the case `x` is already zero. Now, we can use ι when building a stream enumerating the integers as the sequence $0, 1, -1, 2, -2, 3, -3, \ldots$ The code building that stream would be as follows:

```
integersFrom = corec f, ι, ρ, x. (x, ι ⋆ (-x, f ⋆ (suc x)))
integers = inν (0, integersFrom ⋆ 1)
```

The function `integersFrom` builds a stream two numbers at a time. First it puts the positive number `x`, then its negative equivalent `-x` and calls itself on `suc x`. We can build the stream two steps at a time by wrapping the tail of the main stream using ι. The final function `integers` builds the constant stream starting at zero, then invokes `integersFrom` at 1. This avoids having 0 and -0 in the stream.

Operational Semantics

Similarly to our copattern language, our core calculus has a small-step operational semantics. Once again we start with values (Figure 3.3). The main differences between the copattern language and the core language is the addition of our extra terms. Records, λ-abstractions,

$\boxed{t \text{ value}}$ Term t is a value.

$$\frac{v \text{ value}}{v \star \cdot \text{ value}} \qquad \frac{v \text{ value} \quad E \text{ value} \quad v \diamond E}{v \star E \text{ value}}$$

$\boxed{h \text{ value}}$ Head h is a value.

$$\frac{}{() \text{ value}} \quad \frac{}{\wp \text{ value}} \quad \frac{v_1 \text{ value} \quad v_2 \text{ value}}{(v_1, v_2) \text{ value}} \quad \frac{v \text{ value}}{\mathtt{pack}\,\langle C, v\rangle \text{ value}}$$

$$\frac{}{\Lambda u.t \text{ value}} \quad \frac{v \text{ value}}{\mathtt{in}_\mu\, v \text{ value}} \quad \frac{v \text{ value}}{\mathtt{in}_i\, v \text{ value}} \quad \frac{}{\mathtt{in}_\nu\, t \text{ value}} \quad \frac{}{(t_1; \ldots; t_n) \text{ value}}$$

$$\frac{}{\lambda x.t \text{ value}} \quad \frac{}{\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t \text{ value}} \quad \frac{}{\mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t \text{ value}}$$

$\boxed{h \diamond E}$ Head h does not match against spine E.

$$\frac{}{\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t \diamond \vec{C}} \quad \frac{}{\mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t \diamond \vec{C}} \quad \frac{}{\mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t \diamond \vec{C}\ v}$$

$\boxed{E \text{ value}}$ Spine E is a value.

$$\frac{}{\cdot \text{ value}} \quad \frac{E \text{ value}}{.\mathtt{out}_\nu\, E \text{ value}} \quad \frac{E \text{ value}}{.\mathtt{out}_i\, E \text{ value}} \quad \frac{v \text{ value} \quad E \text{ value}}{v\ E \text{ value}} \quad \frac{E \text{ value}}{C\ E \text{ value}}$$

Figure 3.3: Definition of values

and (co)recursors are also values. As they form closures, they don't require their bodies to be values themselves.

The judgment $h \diamond E$ that ensures the head does not match the spine is reused here. Instead of referring to matching, we use it to denote values of (co)recursors when they are not applied to enough arguments. A recursor needs the adequate amount of index-terms, together with a value that will be passed to x to trigger a reduction, while a corecursor needs an observation at the end of a spine formed of index-terms, a seed, and an unfolding.

The stepping rules appear in Figure 3.4. There are only congruence rules for spines, while terms and heads have reduction rules. We recall the notation from Page 36. We denote $t \star E$

$\boxed{t \longrightarrow t'}$ Term t steps to term t'

$$\frac{h \longrightarrow h'}{h \star E \longrightarrow h' \star E} \quad \frac{t \text{ value} \quad E \text{ value}}{(\mathtt{in}_\nu\, t) \star .\mathtt{out}_\nu\, E \longrightarrow t \star E} \quad \frac{E \text{ value}}{(t_1; \ldots; t_n) \star .\mathtt{out}_i\, E \longrightarrow t_i \star E}$$

$$\frac{h \text{ value} \quad E \longrightarrow E'}{h \star E \longrightarrow h \star E'} \quad \frac{v \text{ value} \quad E \text{ value}}{(\lambda x.t) \star v\ E \longrightarrow [v/x]t \star E} \quad \frac{E \text{ value}}{(\Lambda u.t) \star C\ E \longrightarrow [C/u]t \star E}$$

$$\frac{\begin{array}{c}\sigma_X = (\Lambda \vec{u}'.\lambda x.\mathtt{out}_\mu\, x)/\iota_X, (\Lambda \vec{u}'.\lambda x.x)/\rho_X \\ v \text{ value} \quad E \text{ value} \quad t' = [\overrightarrow{C/u}; (\mathtt{rec}\, f, \iota_X, \rho_X, \vec{u}, x.\, t)/f, \sigma_X, v/x]t\end{array}}{(\mathtt{rec}\, f, \iota_X, \rho_X, \vec{u}, x.\, t) \star \vec{C}\ (\mathtt{in}_\mu\, v)\ E \longrightarrow t' \star E}$$

$$\frac{\begin{array}{c}\sigma_X = (\Lambda \vec{u}'.\lambda x.\mathtt{in}_\nu\, x)/\iota_X, (\Lambda \vec{u}'.\lambda x.x)/\rho_X \\ v \text{ value} \quad E \text{ value} \quad t' = [\overrightarrow{C/u}; (\mathtt{corec}\, f, \iota_X, \rho_X, \vec{u}, x.\, t)/f, \sigma_X, v/x]t\end{array}}{(\mathtt{corec}\, f, \iota_X, \rho_X, \vec{u}, x.\, t) \star \vec{C}\ v\ .\mathtt{out}_\nu\, E \longrightarrow t' \star E}$$

$\boxed{h \longrightarrow h'}$ Head h steps to head h'

$$\frac{t_1 \longrightarrow t_1'}{(t_1, t_2) \longrightarrow (t_1', t_2)} \quad \frac{t_1 \longrightarrow t_1'}{\mathtt{split}\, t_1\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t_2 \longrightarrow \mathtt{split}\, t_1'\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t_2}$$

$$\frac{t_2 \longrightarrow t_2'}{(t_1, t_2) \longrightarrow (t_1, t_2')} \quad \frac{v_1 \text{ value} \quad v_2 \text{ value}}{\mathtt{split}\, (v_1, v_2)\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t \longrightarrow [v_1/x_1, v_2/x_2]t}$$

$$\frac{t \longrightarrow t'}{\mathtt{pack}\, \langle C, t\rangle \longrightarrow \mathtt{pack}\, \langle C, t'\rangle} \quad \frac{t_1 \longrightarrow t_1'}{\mathtt{unpack}\, t_1\, \mathtt{as}\, \langle u, x\rangle\, \mathtt{in}\, t_2 \longrightarrow \mathtt{unpack}\, t_1'\, \mathtt{as}\, \langle u, x\rangle\, \mathtt{in}\, t_2}$$

$$\frac{t \longrightarrow t'}{\mathtt{in}_i\, t \longrightarrow \mathtt{in}_i\, t'} \quad \frac{v \text{ value}}{\mathtt{unpack}\, (\mathtt{pack}\, \langle C, v\rangle)\, \mathtt{as}\, \langle u, x\rangle\, \mathtt{in}\, t \longrightarrow [C/u; v/x]t}$$

$$\frac{t \longrightarrow t'}{\mathtt{case}\, t\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i} \longrightarrow \mathtt{case}\, t'\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i}} \quad \frac{v \text{ value}}{\mathtt{case}\, (\mathtt{in}_j\, v)\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i} \longrightarrow [v/x_j]t_j}$$

$$\frac{s \longrightarrow s'}{\mathtt{eq}\, s\, \mathtt{with}\, t \longrightarrow \mathtt{eq}\, s'\, \mathtt{with}\, t} \quad \frac{}{\mathtt{eq}\, \wp\, \mathtt{with}\, t \longrightarrow t}$$

$$\frac{t \longrightarrow t'}{\mathtt{in}_\mu\, t \longrightarrow \mathtt{in}_\mu\, t'} \quad \frac{t \longrightarrow t'}{\mathtt{out}_\mu\, t \longrightarrow \mathtt{out}_\mu\, t'} \quad \frac{v \text{ value}}{\mathtt{out}_\mu\, (\mathtt{in}_\mu\, v) \longrightarrow v}$$

$\boxed{E \longrightarrow E'}$ Spine E steps to spine E

$$\frac{t \longrightarrow t'}{t\ E \longrightarrow t'\ E} \quad \frac{E \longrightarrow E'}{t\ E \longrightarrow t\ E'} \quad \frac{E \longrightarrow E'}{.\mathtt{out}_i\, E \longrightarrow .\mathtt{out}_i\, E'} \quad \frac{E \longrightarrow E'}{.\mathtt{out}_\nu\, E \longrightarrow .\mathtt{out}_\nu\, E'} \quad \frac{E \longrightarrow E'}{C\ E \longrightarrow C\ E'}$$

Figure 3.4: Stepping rules

to mean $h_0 \star E_0\ E$ if t is of the form $h_0 \star E_0$. This allows us to directly write the substitutions $[v/x]t \star E$ without having to extract $[v/x]t$ into a head and a tail. Similarly as we did for our copattern language, we restrict the usage of some rules when specific subterms are values so that the semantics is deterministic.

We unfold constant corecursive terms $\mathtt{in}_\nu\, t$ when applied to unfolding $.\mathtt{out}_\nu$. We project the lazy pair $(t_1; \ldots; t_n)$ to the i-th term t_i when applied to the projection $.\mathtt{out}_i$. We perform the usual β-reduction when applying *lambda*-abstractions. Recursor reduce when applied to indices $\vec{C}$, and a value of the form $\mathtt{in}_\mu\, v$. This leads to the substitutions $\overrightarrow{C/u}$ and $(\mathtt{rec}\, f, \iota_X, \rho_X, \vec{u}, x.\, t)/f, \sigma_X, v/x$. The variable f takes in the recursor itself. x is taking the input stripped of its fold. We also have the substitution σ_X which gives the standard instantiations of $\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x$ for ι_X and $\Lambda\vec{u}'.\lambda x.x$ for ρ_X. Similarly, corecursors reduce when applied to indices $\vec{C}$, a value v and an unfolding $\mathtt{out}_\nu$. The substitution is similar to the one for recursors, although the term used for ι_X is $\Lambda\vec{u}.\lambda x.\mathtt{in}_\nu\, x$. We shall reuse the notation σ_X to denote those standard instantiations of ι_X and ρ_X throughout this chapter. Moreover, for a type variable context Ξ, we shall denote σ_Ξ as the concanetation of all σ_X for $X \in \Xi$.

The reductions for heads work as follow: splitting a pair (v_1, v_2) into (x_1, x_2) in t yields the substitution $[v_1/x_1, v_2/x_2]t$. Unpacking a dependent pair works the same way. Performing a case analysis $\mathtt{case}\, \mathtt{in}_j\, v\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i}$ with a scrutinee $\mathtt{in}_j\, v$ results in the branch $[v/x_j]t_j$. The equality elimination $\mathtt{eq}\, \wp\, \mathtt{with}\, t$ on reflexivity simply reduces into t. Notably absent is an evalution rule for $\mathtt{eq\_abort}\, t$. This term is used in a branch of a case split or a record that we know statically to be impossible. Such branches are never reached at run time, so there is no need for an evaluation rule. For example, consider a type-safe "head" function, which receives a nonempty vector as input. As we write each branch of a case split explicitly, the empty list case must use $\mathtt{eq\_abort}\, t$, but is never executed.

Type Preservation and Progress

Let us now prove type preservation and progress. We first need to show a substitution lemma.

Lemma 3.1 (Substitution lemma)**.** *The following hold:*

1. *If* $\Delta; \Xi; \Gamma \vdash t : T$ *and* $\Delta'; [\theta]\Xi; \Gamma' \vdash (\theta; \sigma) : (\Delta; \Gamma)$ *then* $\Delta'; [\theta]\Xi; \Gamma' \vdash [\theta; \sigma]t : [\theta]T$.
2. *If* $\Delta; \Xi; \Gamma \vdash h : T$ *and* $\Delta'; [\theta]\Xi; \Gamma' \vdash (\theta; \sigma) : (\Delta; \Gamma)$ *then* $\Delta'; [\theta]\Xi; \Gamma' \vdash [\theta; \sigma]h : [\theta]T$.
3. *If* $\Delta; \Xi; \Gamma; T \vdash E \searrow T'$ *and* $\Delta'; [\theta]\Xi; \Gamma' \vdash (\theta; \sigma) : (\Delta; \Gamma)$ *then* $\Delta'; [\theta]\Xi; \Gamma'; [\theta]T \vdash [\theta; \sigma]E \searrow [\theta]T'$.

Proof. By mutual induction on the derivations $\Delta; \Xi; \Gamma \vdash t : T$, and $\Delta; \Xi; \Gamma \vdash h : T$, and $\Delta; \Xi; \Gamma; T \vdash E \searrow T'$, respectively. Cases involving indices use Req. 1. Furthermore, eliminators for equalities make use of Lemma 2.1. □

Theorem 3.2 (Type preservation)**.** *The following hold:*

1. *If* $\vdash t : T$ *and* $t \longrightarrow t'$ *then* $\vdash t' : T$.
2. *If* $\vdash h : T$ *and* $h \longrightarrow h'$ *then* $\vdash h' : T$.
3. *If* $T \vdash E \searrow T'$ *and* $E \longrightarrow E'$ *then* $T \vdash E' \searrow T'$.

Proof. By induction on the stepping derivation. We show the cases for equality elimination and reduction of recursors.

Case: $\mathtt{eq}\,\wp\,\mathtt{with}\,t' \longrightarrow t'$

$\vdash \mathtt{eq}\,\wp\,\mathtt{with}\,t' : T$ by assumption.
$\vdash \wp : C_1 = C_2$ and $\vdash C_1 = C_2 \searrow \Delta$ and $\Delta; \cdot; \cdot \vdash t' : T$ by inversion on the typing derivation.
$\vdash C_1 = C_2$ by inversion on $\vdash \wp : C_1 = C_2$.
By Req. 3, choosing the empty context $\cdot$ for Δ_0 and the identity substitution $\cdot$ for θ, we have $\vdash \cdot : \Delta$. By inversion on this judgment, we must have $\Delta = \cdot$. Hence, $\vdash t' : T$.

Case: $$\frac{v \text{ value} \quad E \text{ value} \quad t'' = [\overrightarrow{C/u}; (\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t')/f, \sigma_X, v/x]t'}{(\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t') \star \vec{C}\ (\mathtt{in}_\mu\, v)\ E \longrightarrow t'' \star E}$$

$\vdash (\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t') \star \vec{C}\ (\mathtt{in}_\mu\, v)\ E : T$ by assumption.
$\vdash \mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t' : \Pi \overrightarrow{u{:}U}.\mu\mathcal{F}\vec{C}' \to T'$ and $\Pi \overrightarrow{u{:}U}.\mu\mathcal{F}\vec{C}' \to T' \vdash \vec{C}\ (\mathtt{in}_\mu\, v)\ E \searrow T$
by inversion on typing derivation.

Let $\Gamma' = f{:}\Pi\overrightarrow{u{:}U}.X\,\vec{C} \to T', \iota{:}\Pi\overrightarrow{u{:}U}.X\,\vec{u} \to \mathcal{F}\,X\,\vec{u}, \rho{:}\Pi\overrightarrow{u{:}U}.X\,\vec{u} \to (\mu\mathcal{F})\,\vec{u}, x{:}\mathcal{F}\,X\,\vec{C}'$
$\overrightarrow{u{:}U}; X{:}K; \Gamma' \vdash t' : T'$ by inversion on typing derivation.
$\overrightarrow{u{:}U}; \cdot; [\mu\mathcal{F}/X]\Gamma' \vdash t : [\mu\mathcal{F}/X]T'$ by the substitution property of type variables.
$\vdash C_i : U_i$ and $\vdash \mathtt{in}_\mu\, v : \mu\mathcal{F}\vec{C}'$ and $[\overrightarrow{C/u}][\mu\mathcal{F}/X]T' \vdash E \searrow T$ by inversion on typing on spine.
$\vdash [\overrightarrow{C/u}; (\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t)/f, \sigma_X, v/x]t : [\overrightarrow{C/u}][\mu\mathcal{F}/X]T'$ by substitution lemma.
Thus, $\vdash [\overrightarrow{C/u}; (\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t)/f, \sigma_X, v/x]t \star E : T$. □

Theorem 3.3 (Progress)**.** *The following hold:*

1. *If $\vdash t : T$ then either there is a t' such that $t \longrightarrow t'$ or t is a value.*
2. *If $\vdash h : T$ then either there is a h' such that $h \longrightarrow h'$ or h is a value.*
3. *If $T' \vdash E \searrow T$ then either there is a E' such that $E \longrightarrow E'$ or E is a value.*

Proof. By mutual induction on the derivations of $\vdash t : T$, and $\vdash h : T$, and $T' \vdash E \searrow T$.

We discuss the case of $\vdash h \star E : T$. By inversion we have $\vdash h : T'$ and $T' \vdash E \searrow T$. By Statement 2, either $h \longrightarrow h'$ or $\vdash_v h : T'$. In the former case, we are done. In the latter case, we have by Statement 3, either $E \longrightarrow E'$ or E is a value. In the former case we are also done. We are also done if the spine is empty. In the latter case, we need to decide whether both together reduce. We proceed by case analysis on the head, then by inversion on the spine. Let us show an example. The others are similar. Suppose the head is:

$$\frac{\Gamma' = \begin{array}{l} f{:}\Pi\overrightarrow{u{:}U}.X\,\vec{C} \to T, x{:}\mathcal{F}\,X\,\vec{C}, \\ \iota{:}\Pi\overrightarrow{u{:}U}.X\,\vec{u} \to \mathcal{F}\,X\,\vec{u}, \rho{:}\Pi\vec{u}.X\,\vec{u} \to (\mu\mathcal{F})\,\vec{u} \end{array} \qquad \overrightarrow{u{:}U}; X{:}K; \Gamma' \vdash t : T}{\vdash \mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, t : \Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\,\vec{C} \to T}$$

By inversion on the spine, there are 3 possibilities for E. We could have $E = \cdot$ or $E = \vec{C}'$. If so, we have $h \diamond E$ and $h \star E$ is a value. Otherwise we have have $E = \vec{C}'\ (\mathtt{in}_\mu\, v)\ E'$ and so $\mathtt{rec}\, f, \iota, \vec{\rho}, \vec{u}, x.\, t' \star \vec{C}'\ (\mathtt{in}_\mu\, v)\ E' \longrightarrow [\overrightarrow{C'/u}; (\mathtt{rec}\, f, \iota, \vec{\rho}, \vec{u}, x.\, t')/f, v/x]t' \star E'$. □

3.2 Normalisation of Core Language

We now describe termination of evaluation of the core calculus. Our proof uses the logical predicate technique of Tait [1967] and Girard [1972]. We follow loosely the presentations of Jacob-Rao et al. [2018] and Abel [2004] to build our semantics and proof. We interpret each language construct (index types, kinds, types, etc.) into a semantic model of sets and functions.

Interpretation of Index Language

We start with the interpretations for index-types and index-spines. In general, our index language may be dependently typed, as it is if we choose contextual LF. Hence our interpretation for index types U must take into account an environment θ containing instantiations for index variables u. Such an index environment θ is simply a grounding substitution $\vdash \theta : \Delta$.

Definition 3.1 (Interpretation of index-types $[\![U]\!]$ and index-spines $[\![\overrightarrow{u{:}U}]\!]$)**.**

$$\begin{array}{lcl} [\![U]\!](\theta) & = & \{C \mid \cdot \vdash C : [\theta]U\} \\ [\![(\cdot)]\!](\theta) & = & \{\cdot\} \\ [\![(u_0{:}U_0, \overrightarrow{u{:}U})]\!](\theta) & = & \{C_0, \vec{C} \mid C_0 \in [\![U_0]\!](\theta), \vec{C} \in [\![(\overrightarrow{u{:}U})]\!](\theta, C_0/u_0)\} \end{array}$$

The interpretation of an index type U under environment θ is the set of closed terms of type $[\theta]U$. The interpretation lifts to index spines $(\overrightarrow{u{:}U})$. With these definitions, the following lemma follows from the substitution principles of index terms (Req. 1).

Lemma 3.4 (Interpretation of index substitution)**.** *The following hold:*

1. *If* $\Delta \vdash C : U$ *and* $\vdash \theta : \Delta$ *then* $[\theta]C \in [\![U]\!](\theta)$.
2. *If* $\Delta \vdash \vec{C} : (\overrightarrow{u{:}U})$ *and* $\vdash \theta : \Delta$ *then* $\overrightarrow{[\theta]C} \in [\![(\overrightarrow{u{:}U})]\!](\theta)$.

Lattice Interpretation of Kinds

We now describe the lattice structure that underlies the interpretation of kinds in our language. The idea is that types are interpreted as sets of strongly normalizing terms and

type constructors as functions taking indices to sets of those terms. The set of all strongly normalizing terms will be denoted as Ω and is defined inductively using the following rules:

$$\frac{t \longrightarrow t' \quad t' \in \Omega}{t \in \Omega} \qquad \frac{t \not\longrightarrow}{t \in \Omega}$$

We note by Theorem 3.3, if $t \not\longrightarrow$, then t is a value. We also recall that our operational semantics is deterministic. Thus, we only need to consider the single step $t \longrightarrow t'$ in our inductive case. We denote the power set of Ω as $\mathcal{P}(\Omega)$. The interpretation is defined inductively on the structure of kinds.

Definition 3.2 (Interpretation of kinds $[\![K]\!]$)**.**

$$\begin{array}{lcl} [\![\mathsf{type}]\!](\theta) & = & \mathcal{P}(\Omega \times \Omega) \\ [\![\Pi u{:}U.\, K]\!](\theta) & = & \{\mathcal{E} \mid \forall C \in [\![U]\!](\theta).\, \mathcal{E}(C) \in [\![K]\!](\theta, C/u)\} \end{array}$$

In the base case, $[\![\mathsf{type}]\!](\theta) = \mathcal{P}(\Omega \times \Omega)$ is a *complete lattice* under the subset ordering, with meet and join given by intersection and union respectively. For a kind $K = \Pi u{:}U.\, K'$, we induce a complete lattice structure on $[\![K]\!](\theta)$ by lifting the lattice operations pointwise. Precisely, we define

$$\mathcal{A} \leq_{[\![K]\!](\theta)} \mathcal{B} \quad \text{iff} \quad \forall C \in [\![U]\!](\theta).\, \mathcal{A}(C) \leq_{[\![K']\!](\theta, C/u)} \mathcal{B}(C).$$

The meet and join operations can similarly be lifted pointwise.

We note that sets in $[\![\mathsf{type}]\!](\theta)$ contain pairs of normalizing terms. We choose to define the interpretation as sets of relations as such relations will be needed to prove that our translation preserves normalisation in Section 3.6. For the purpose of proving normalisation of our calculus, we simply prove that well-typed terms are related to themselves in the interpretation. Thus, they are normalizing.

Interpretation of Types

Fixed Points in Semantics

Before we move on to the definition of interpretation, we do need some apparatus for (co)inductive types. The semantics of type operators $\mathcal{F}$ are monotone operators on sets.

We make use of Kleene's fixed point theorem in the style of Mendler [1991] and Abel [2004] in order to build the fixed points by means of transfinite induction.

Let $\Phi : \mathcal{L} \to \mathcal{L}$ and $\Psi : \mathcal{L} \to \mathcal{L}$ be operators on our lattice. For an ordinal α, we define the α-iterate Φ^α and Ψ^α at kind K inductively on K and then by transfinite recursion on α, as follows:

For $K = \mathsf{type}$, then $[\![\mathsf{type}]\!](\theta) = \mathcal{P}(\Omega \times \Omega)$.

$$\begin{array}{ll} \Phi^0 = \emptyset & \Psi^0 = \Omega \times \Omega \\ \Phi^{\alpha+1} = \Phi(\Phi^\alpha) & \Psi^{\alpha+1} = \Psi(\Psi^\alpha) \\ \Phi^\lambda = \bigcup_{\alpha<\lambda} \Phi^\alpha \text{ for } \lambda \text{ limit ordinal} & \Psi^\lambda = \bigcap_{\alpha<\lambda} \Psi^\alpha \text{ for } \lambda \text{ limit ordinal} \end{array}$$

For $K = \Pi u{:}U.K'$, if $C \in [\![U]\!](\theta)$, then we have:

$$\begin{array}{ll} \Phi^0(C) = \emptyset & \Psi^0(C) = \Omega \times \Omega \\ \Phi^{\alpha+1}(C) = \Phi(\Phi^\alpha)(C) & \Psi^{\alpha+1}(C) = \Psi(\Psi^\alpha)(C) \\ \Phi^\lambda(C) = \bigcup_{\alpha\leq\lambda} \Phi^\alpha(C) \text{ for } \lambda \text{ limit ordinal} & \Psi^\lambda(C) = \bigcap_{\alpha\leq\lambda} \Psi^\alpha(C) \text{ for } \lambda \text{ limit ordinal} \end{array}$$

If Φ and Ψ are monotone, then Kleene's fixed point theorem ensures they will reach a fixed point at some ordinal. Given our types, we will only ever need to reach the least uncountable ordinal ω_1. Thus, $\Phi(\Phi^{\omega_1}) = \Phi^{\omega_1}$ and $\Psi(\Psi^{\omega_1}) = \Psi^{\omega_1}$. Moreover, since $\emptyset \subset \Phi(\emptyset)$ and $\Phi(\Omega \times \Omega) \subset \Omega \times \Omega$ by monotonicity, then $\Phi^\alpha \subset \Phi^\beta$ and $\Psi^\beta \subset \Psi^\alpha$ for all $\alpha < \beta$.

Defining the Interpretation

Following the interpretation of kinds, the interpretation of types is a relation, rather than a set. For a type T, the interpretation $[\![T]\!](\theta;\eta)$ denotes pairs of values of type T behaving the same with respect to evaluation. We will use the notation $v_1 \geq v_2 \in [\![T]\!](\theta;\eta)$ to denote pairs (v_1, v_2) in the interpretation $[\![T]\!](\theta;\eta)$. We extend it to arbitrary terms through the concept of saturation. Given an interpretation $\mathcal{E}$, we define the *saturation* of $\mathcal{E}$ as:

$$\mathcal{E}^* = \{t_1 \geq t_2 \mid \exists v_1, v_2 \text{ such that } t_1 \longrightarrow^{n+k} v_1 \text{ and } t_2 \longrightarrow^n v_2 \text{ and } v_1 \geq v_2 \in \mathcal{E}\}.$$

The relation imposes that the left-hand term takes at least as many steps as the right-hand term takes to reach a value, and that both of those values are themselves related. This

condition allows us to impose an upper bound on the number of steps. This will be needed later on when we relate the Mendler style system to the copattern one. For the purpose of this section, it only matters that it contains normalizing terms.

Before we move on to the definition of $[\![T]\!](\theta;\eta)$, let us introduce some more notation. We lift term-level $\mathtt{in}$ and $\mathtt{out}$ tags to the level of sets and functions in the lattice $[\![K]\!](\theta)$. We define the lifted tags $\mathtt{in}^* : [\![K]\!](\theta) \to [\![K]\!](\theta)$ inductively on kind K. If $\mathcal{E} \in [\![\mathsf{type}]\!](\theta) = \mathcal{P}(\Omega \times \Omega)$ then $\mathtt{in}^*\,\mathcal{E} = \mathtt{in}\,\mathcal{E} = \{\mathtt{in}\,v_1 \geq \mathtt{in}\,v_2 \mid v_1 \geq v_2 \in \mathcal{E}\}$. If $\mathcal{E} \in [\![\Pi u{:}U.\,K']\!](\theta)$ then $(\mathtt{in}^*\,\mathcal{E})(C) = \mathtt{in}^*\,(\mathcal{E}(C))$ for all $C \in [\![U]\!](\theta)$. Essentially, the $\mathtt{in}^*$ function attaches a tag to every element in the set produced after the index arguments are received. Dually we define $\mathtt{out}^* : [\![K]\!](\theta) \to [\![K]\!](\theta)$. If $\mathcal{E} \in [\![\mathsf{type}]\!](\theta) = \mathcal{P}(\Omega \times \Omega)$ then $\mathtt{out}^*\,\mathcal{E} = \mathtt{out}\,\mathcal{E} = \{t_1 \geq t_2 \mid t_1 \star .\mathtt{out} \geq t_2 \star .\mathtt{out} \in \mathcal{E}^*\}$. We note that $t.\mathtt{out} \geq t.\mathtt{out}$ is defined in the saturation of $\mathcal{E}$ rather than $\mathcal{E}$ itself, as the application of an observation will likely trigger a reduction. If $\mathcal{E} \in [\![\Pi u{:}U.\,K']\!](\theta)$ then $(\mathtt{out}^*\,\mathcal{E})(C) = \mathtt{out}^*\,(\mathcal{E}(C))$ for all $C \in [\![U]\!](\theta)$.

This leads us to the interpretation of types T (Figure 3.5), under environments θ and η. This is done inductively on the structure of T. Interpretations of unit type 1, products $T_1 \times T_2$, variants D, and existentials $\Sigma u{:}U.T$ simply contain their canonical values which are related pointwise. Interpretations of records R, Π-types $\Pi u{:}U.T$, and function types $T_1 \to T_2$ are represented by the application of their elimination forms. Applying an elimination form might not result in a value and thus we assume the result is in the saturation of the interpretation. In the particular case of $T_1 \to T_2$, we pick a value v such that $v \geq v \in [\![T_1]\!](\theta;\eta)$, following usual definitions about applicative simulations. This definition is justified by the following lemma:

Lemma 3.5. *The following hold:*

1. *If $v_1 \geq v_2 \in [\![T]\!](\theta;\eta)$, then $v_1 \geq v_1 \in [\![T]\!](\theta;\eta)$;*
2. *If $v_1 \geq v_2 \in [\![T]\!](\theta;\eta)$, then $v_2 \geq v_2 \in [\![T]\!](\theta;\eta)$.*

Proof. By induction on T. □

Type level abstractions $\Lambda u.T$ lead to functions from the interpretation of the index domain to a set, denoted by $C \mapsto \_$, while type level applications $T\ C$ simply correspond to

$$
\begin{array}{lcl}
[\![1]\!](\theta;\eta) & = & \{() \geq ()\} \\
[\![C_1 = C_2]\!](\theta;\eta) & = & \{\wp \geq \wp\} \\
[\![T_1 \times T_2]\!](\theta;\eta) & = & \{(t_1,t_2) \geq (t_1',t_2') \mid t_1 \geq t_1' \in [\![T_1]\!](\theta;\eta) \text{ and } t_2 \geq t_2' \in [\![T_2]\!](\theta;\eta)\} \\
[\![D]\!](\theta;\eta) & = & \bigcup_i \mathtt{in}_i^* \, [\![D_i]\!](\theta;\eta) \\
[\![R]\!](\theta;\eta) & = & \bigcap_i \mathtt{out}_i^* \, [\![R_i]\!](\theta;\eta) \\
[\![\Pi u{:}U.T]\!](\theta;\eta) & = & \{t_1 \geq t_2 \mid \forall C \in [\![U]\!](\theta), t_1 \star C \geq t_2 \star C \in [\![T]\!]^*(\theta, \overrightarrow{C/u};\eta)\} \\
[\![S \to T]\!](\theta;\eta) & = & \{t_1 \geq t_2 \mid \forall s \geq s \in [\![S]\!](\theta;\eta), t_1 \star s \geq t_2 \star s \in [\![T]\!]^*(\theta;\eta)\} \\
[\![\Sigma u{:}U.T]\!](\theta;\eta) & = & \{\mathtt{pack}\,\langle C, t_1\rangle \geq \mathtt{pack}\,\langle C, t_2\rangle \mid C \in [\![U]\!](\theta) \text{ and } t_1 \geq t_2 \in [\![T]\!](\theta, C/u;\eta)\} \\
[\![T\ C]\!](\theta;\eta) & = & [\![T]\!](\theta;\eta)([\theta]C) \\
[\![\Lambda u.T]\!](\theta;\eta) & = & C \mapsto [\![T]\!](\theta, C/u;\eta) \\
[\![X]\!](\theta;\eta) & = & \eta(X) \\
[\![\mu\mathcal{F}]\!](\theta;\eta) & = & \Phi^{\omega_1}_{\mathcal{F},\theta,\eta} \\
[\![\nu\mathcal{F}]\!](\theta;\eta) & = & \Psi^{\omega_1}_{\mathcal{F},\theta,\eta}
\end{array}
$$

where

$$
\begin{array}{lcl}
\Phi_{\mathcal{F},\theta,\eta}(Q) & = & \mathtt{in}_\mu^* \, [\![\mathcal{F}X]\!](\theta;\eta,Q/X) \\
\Psi_{\mathcal{F},\theta,\eta}(Q) & = & \mathtt{out}_\nu^* \, [\![\mathcal{F}X]\!](\theta;\eta,Q/X)
\end{array}
$$

Figure 3.5: Interpretation of types

the application of those functions.

Inductive and coinductive types are interpreted as Φ^{ω_1} and Ψ^{ω_1}. Those operators compute the interpretation of the one step unfolding $\mathcal{F}X$ lifted respectively with $\mathtt{in}_\mu^*$ and $\mathtt{out}_\nu^*$. In order for the fixed points Φ^{ω_1} and Ψ^{ω_1} to exist, we need Φ and Ψ to be monotone. Thus we require $\mathcal{F}X$ to only have positive occurrences of X. Moreover, since the operators are monotone, the iterations are cumulative. In particular, we can prove the following:

Lemma 3.6. *If* $\mathtt{in}_\mu\, v \in \Phi^\alpha(\vec{C})$ *then* $\mathtt{in}_\mu\, v \in \Phi^{\beta+1}(\vec{C})$ *for some ordinal* $\beta < \alpha$.

Proof. By transfinite induction on α. □

We define the interpretation of contexts $[\![\Gamma]\!](\theta;\eta)$ not as a relation but simply as a set. We only use the interpretation of contexts in order to obtain safe grounding substitutions.

In Section 3.6, we will define a generalization of open simulation and a subsequently a corresponding simulation for arbitrary substitutions. The interpretation of empty contexts contains any well-formed substitution σ as any such substitution maps from an empty context to any other one. This is because there are no variables in the empty context for σ to replace. The interpretation of a context $\Gamma, x : T$ contains substitutions σ that are in the interpretation of Γ and are such that $\sigma(x) \geq \sigma(x)$ is in the interpretation of T. As such, applying it to x will result in a safe term.

$$\begin{aligned}
[\![\cdot]\!](\theta;\eta) &= \{\sigma \mid \sigma \text{ is a well-formed substitution}\} \\
[\![\Gamma, x : T]\!](\theta;\eta) &= \{\sigma \mid \sigma \in [\![\Gamma]\!](\theta;\eta) \text{ and } \sigma(x) \geq \sigma(x) \in [\![T]\!](\theta;\eta)
\end{aligned}$$

Moreover, it will become useful, in particular in Section 3.6, to lift the interpretation of types to support spines. We thus define the interpretation $[\![S \searrow T]\!](\theta;\eta)$ as

$$\{E_1 \geq E_2 \mid \text{for all } v \geq v \in [\![S]\!](\theta;\eta), v \star E_1 \geq v \star E_2 \in [\![T]\!]^*(\theta;\eta)\}.$$

Proving Normalisation

There are two main statements we need to prove to show well-typed terms are normalizing. The first is a lemma stating that if a type T is well-kinded against K, then the interpretation of T is in the interpretation of K. This ensures that terms in the interpretation of T are in Ω and thus normalizing. Then it suffices to show well-typed terms are in the interpretation of their corresponding types to complete the picture.

Before we move on to results, we first need some helpful lemmas. The first two lemmas will help us directly reason about evaluation. We show that the small-step semantics mimicks a big-step behaviour. The second one shows specific sub-terms of normalizable terms are normalizing.

Lemma 3.7 (Concatenation of multi-step rules)**.** *Suppose* $t_1 \longrightarrow^* t_2$ *and* $t_2 \longrightarrow^* t_3$ *then* $t_1 \longrightarrow^* t_3$.

Proof. By induction on $t_1 \longrightarrow^* t_2$. □

Lemma 3.8 (Big-step behaviour of stepping rules)**.** *Suppose $t \longrightarrow^* v$. Then, the following hold:*

1. *If $t' \longrightarrow^* v'$, then $(t, t') \longrightarrow^* (v, v')$*
2. $\mathtt{pack}\,\langle C, t\rangle \longrightarrow^* \mathtt{pack}\,\langle C, v\rangle$
3. $\mathtt{in}_\mu\, t \longrightarrow^* \mathtt{in}_\mu\, v$
4. $\mathtt{in}_i\, t \longrightarrow^* \mathtt{in}_i\, v$
5. $\mathtt{out}_\mu\, t \longrightarrow^* \mathtt{out}_\mu\, v$
6. $t \star E \longrightarrow^* v \star E$
7. $\mathtt{split}\, t\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t' \longrightarrow^* \mathtt{split}\, v\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, t'$
8. $\mathtt{unpack}\, t\, \mathtt{as}\, \langle u, x\rangle\, \mathtt{in}\, t' \longrightarrow^* \mathtt{unpack}\, v\, \mathtt{as}\, \langle u, x\rangle\, \mathtt{in}\, t'$
9. $\mathtt{case}\, t\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i} \longrightarrow^* \mathtt{case}\, v\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto t_i}$
10. $\mathtt{eq}\, t\, \mathtt{with}\, t' \longrightarrow^* \mathtt{eq}\, v\, \mathtt{with}\, t'$

Proof. By induction on the stepping derivations. □

Lemma 3.9 (Normalisation of shortened spines)**.** *The following hold:*

1. *If $h \star E\ t \in \Omega$, then $h \star E \in \Omega$;*
2. *If $h \star E\ C \in \Omega$, then $h \star E \in \Omega$;*
3. *If $h \star E\ .\mathtt{out}_\nu\ \in \Omega$, then $h \star E \in \Omega$;*
4. *If $h \star E\ .\mathtt{out}_i\ \in \Omega$, then $h \star E \in \Omega$.*

Proof. By induction on the derivation $h \star E\ \alpha \in \Omega$ where α stands for t, C, $\mathtt{out}_\nu$, or $\mathtt{out}_i$. □

The following two lemmas let us manipulate interpretations of type.

Lemma 3.10 (Type-level substitution associates with interpretation)**.** *The following hold:*

1. *If $\Delta; \Xi \vdash T : K$ and $\Delta' \vdash \Theta : \Delta$ and $\vdash \theta : \Delta'$ and $\eta \in [\![[\Theta]\Xi]\!](\theta)$, then $[\![[\Theta]\Xi]\!](\theta) = [\![\Xi]\!]([\theta]\Theta)$ and $[\![[\Theta]T]\!](\theta; \eta) = [\![T]\!]([\theta]\Theta; \eta)$.*

2. *If* $\Delta; \Xi, X{:}K' \vdash T : K$ *and* $\Delta; \Xi \vdash S : K'$ *and* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$*, then* $[\![[S/X]T]\!](\theta;\eta) = [\![T]\!](\theta;\eta, [\![S]\!](\theta;\eta)/X)$.

Proof. Both statements hold by induction on the structure of T. □

Lemma 3.11 (Interpretation preserves (co)recursive unfoldings)**.** *Let* $\Delta; \Xi, X{:}K \vdash \mathcal{F}X : K$ *and* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$*. The following hold:*

1. $[\![\mathcal{F}(\mu\mathcal{F})\ \vec{C}]\!](\theta;\eta) = [\![\mathcal{F}X\vec{C}]\!](\theta;\eta, \Phi^{\omega_1}/X)$
2. $[\![\mathcal{F}(\nu\mathcal{F})\ \vec{C}]\!](\theta;\eta) = [\![\mathcal{F}X\vec{C}]\!](\theta;\eta, \Psi^{\omega_1}/X)$

where $\Phi = \Phi_{\mathcal{F};\theta;\eta}$ *and* $\Psi = \Psi_{\mathcal{F};\theta;\eta}$.

Proof. As a reminder, $\mathcal{F}X = \Lambda\vec{u}.T$. Using Lemma 3.10, we can deduce:

$$\begin{aligned}
[\![\mathcal{F}(\mu\mathcal{F})\ \vec{C}]\!](\theta;\eta) &= [\![[\mu\mathcal{F}/X](\Lambda\vec{u}.T)\ \vec{C}]\!](\theta;\eta) = [\![[\mu\mathcal{F}/X](\Lambda\vec{u}.T)]\!](\theta;\eta)\ (\overrightarrow{[\theta]C}) \\
&= [\![\Lambda\vec{u}.T]\!](\theta;\eta, [\![\mu\mathcal{F}]\!](\theta;\eta)/X)\ (\overrightarrow{[\theta]C}) = [\![\mathcal{F}X]\!](\theta;\eta, [\![\mu\mathcal{F}]\!](\theta;\eta)/X)\ (\overrightarrow{[\theta]C}) \\
&= [\![\mathcal{F}X]\!](\theta;\eta, \Phi^{\omega_1}/X)\ (\overrightarrow{[\theta]C}) = [\![\mathcal{F}X\vec{C}]\!](\theta;\eta, \Phi^{\omega_1}/X)
\end{aligned}$$

The proof is analogous for $\nu\mathcal{F}$. □

We now prove that the interpretation of types contains normalising terms.

Lemma 3.12 (Interpretation of kinds contains interpretation of types)**.** *If* $\Delta; \Xi \vdash T : K$ *and* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$*, then* $[\![T]\!](\theta;\eta) \in [\![K]\!](\theta)$.

Proof. By induction on $\Delta; \Xi \vdash T : K$. □

This leads us to the main result: well-typed terms are contained the interpretation of their type.

Theorem 3.13 (Normalisation)**.** *Let* $\vdash \theta : \Delta$ *and* $\eta \in [\![\Xi]\!](\theta)$ *and* $\sigma \in [\![\Gamma]\!](\theta;\eta)$*. The following hold:*

1. *If* $\Delta; \Xi; \Gamma \vdash t : T$ *then* $[\theta;\sigma]t \geq [\theta;\sigma]t \in [\![T]\!]^*(\theta;\eta)$.
2. *If* $\Delta; \Xi; \Gamma \vdash h : T$ *then* $[\theta;\sigma]h \geq [\theta;\sigma]h \in [\![T]\!]^*(\theta;\eta)$.

3. *If* $\Delta; \Xi; \Gamma; S \vdash E \searrow T$, *then* $[\theta;\sigma]E \geq [\theta;\sigma]E \in [\![S \searrow T]\!]^*(\theta;\eta)$.

Proof. All statements are proved by mutual induction on typing derivations.

Case: $\dfrac{\Delta; \Xi; \Gamma \vdash h : T' \quad \Delta; \Xi; \Gamma; T' \vdash E \searrow T}{\Delta; \Xi; \Gamma \vdash h \star E : T}$

$[\theta;\sigma]h \longrightarrow^* v$ and $v \geq v \in [\![T']\!](\theta;\eta)$ by induction hypothesis
$[\theta;\sigma]E \longrightarrow^* E_v$ and $v \star E_v \geq v \star E_v \in [\![T]\!]^*(\theta;\eta)$ by induction hypothesis
$[\theta;\sigma](h \star E) \geq [\theta;\sigma](h \star E) \in [\![T]\!]^*(\theta;\eta)$ by Lemma 3.8

Case: $\dfrac{\Delta; \Xi; \Gamma \vdash t_1 : T_1 \quad \Delta; \Xi; \Gamma \vdash t_2 : T_2}{\Delta; \Xi; \Gamma \vdash (t_1, t_2) : T_1 \times T_2}$

$[\theta;\sigma]t_1 \geq [\theta;\sigma]t_1 \in [\![T_1]\!]^*(\theta;\eta)$ by induction hypothesis on $\Delta; \Xi; \Gamma \vdash t_1 : T_1$
$[\theta;\sigma]t_2 \geq [\theta;\sigma]t_2 \in [\![T_2]\!]^*(\theta;\eta)$ by induction hypothesis on $\Delta; \Xi; \Gamma \vdash t_2 : T_2$
$[\theta;\sigma](t_1, t_2) \geq [\theta;\sigma](t_1, t_2) \in [\![T_1 \times T_2]\!]^*(\theta;\eta)$ by Lemma 3.8

Case: $\dfrac{\Delta; \Xi; \Gamma, x{:}S \vdash t : T}{\Delta; \Xi; \Gamma \vdash \lambda x.t : S \to T}$

We want to show $[\theta;\sigma](\lambda x.t) \geq [\theta;\sigma](\lambda x.t) \in [\![S \to T]\!]^*(\theta;\eta)$. By $[\![S \to T]\!]^*(\theta;\eta)$, it is equivalent to $[\theta;\sigma](\lambda x.t) \star v \geq [\theta;\sigma](\lambda x.t) \star v \in [\![T]\!]^*(\theta;\eta)$ for all $v \geq v \in [\![S]\!](\theta;\eta)$.
$(\lambda x.[\theta;\sigma]t) \star v \longrightarrow [\theta;\sigma, v/x]t$ by stepping rules
$\sigma, v/x \geq \sigma, v/x \in [\![\Gamma, x{:}S]\!](\theta;\eta)$ since $\sigma \geq \sigma \in [\![\Gamma]\!](\theta;\eta)$ and $v \geq v \in [\![S]\!](\theta;\eta)$
$[\theta;\sigma, v/x]t \geq [\theta;\sigma, v/x]t \in [\![T]\!]^*(\theta;\eta)$ by induction hypothesis

Case: $\dfrac{\Delta; \Xi; \Gamma \vdash s : T_1 \times T_2 \quad \Delta; \Xi; \Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t : T}{\Delta; \Xi; \Gamma \vdash \mathtt{split}\ s\ \mathtt{as}\ (x_1, x_2)\ \mathtt{in}\ t : T}$

$[\theta;\sigma]s \geq [\theta;\sigma]s \in [\![T_1 \times T_2]\!]^*(\theta;\eta)$ by induction hypothesis
$[\theta;\sigma]s \longrightarrow^* v$ and $v \geq v \in [\![T_1 \times T_2]\!](\theta;\eta)$ by definition of $[\![T_1 \times T_2]\!]^*(\theta;\eta)$
$v = (v_1, v_2)$ by definition of $[\![T_1 \times T_2]\!](\theta;\eta)$
$[\theta;\sigma](\mathtt{split}\ s\ \mathtt{as}\ (x_1, x_2)\ \mathtt{in}\ t) \longrightarrow^* \mathtt{split}\ (v_1, v_2)\ \mathtt{as}\ (x_1, x_2)\ \mathtt{in}\ [\theta;\sigma]t$ by Lemma 3.8
$(\mathtt{split}\ (v_1, v_2)\ \mathtt{as}\ (x_1, x_2)\ \mathtt{in}\ [\theta;\sigma]t) \longrightarrow [\theta;\sigma, v_1/x_1, v_2/x_2]t$ by stepping rule
$\sigma, v_1/x_1, v_2/x_2 \geq \sigma, v_1/x_1, v_2/x_2 \in [\![\Gamma, x_1{:}T_1, x_2{:}T_2]\!]$ by assumptions

$[\theta;\sigma,v_1/x_1,v_2/x_2]t \geq [\theta;\sigma,v_1/x_1,v_2/x_2]t \in [\![T]\!]^*(\theta;\eta)$ by induction hypothesis
$[\theta;\sigma,v_1/x_1,v_2/x_2]t \longrightarrow^* v_1$ and $v_1 \geq v_1 \in [\![T]\!](\theta;\eta)$ by definition of $[\![T]\!]^*(\theta;\eta)$
$[\theta;\sigma](\mathtt{split}\, s\, \mathtt{as}\, (x_1,x_2)\, \mathtt{in}\, t) \longrightarrow^* v_1$ by Lemma 3.7
$[\theta;\sigma](\mathtt{split}\, s\, \mathtt{as}\, (x_1,x_2)\, \mathtt{in}\, t) \geq [\theta;\sigma](\mathtt{split}\, s\, \mathtt{as}\, (x_1,x_2)\, \mathtt{in}\, t) \in [\![T]\!]^*(\theta;\eta)$ by $[\![T]\!]^*(\theta;\eta)$

Case: $\dfrac{\Delta;\Xi;\Gamma \vdash t : \mathcal{F}(\mu\mathcal{F})\,\vec{C}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_\mu\, t : (\mu\mathcal{F})\,\vec{C}}$

$[\theta;\sigma]t \geq [\theta;\sigma]t \in [\![\mathcal{F}(\mu\mathcal{F})\,\vec{C}]\!]^*(\theta;\eta)$ by induction hypothesis
$[\theta;\sigma]t \longrightarrow^* v$ and $v \geq v \in [\![\mathcal{F}(\mu\mathcal{F})\,\vec{C}]\!](\theta;\eta)$ by definition of saturation
$[\![\mathcal{F}(\mu\mathcal{F})\,\vec{C}]\!](\theta;\eta) = [\![\mathcal{F}X\vec{C}]\!](\theta;\eta,\Phi^{\omega_1}/X)$ by Lemma 3.11
$[\theta;\sigma](\mathtt{in}_\nu\, t) \longrightarrow^* \mathtt{in}_\nu\, v$ by Lemma 3.8
$\mathtt{in}_\nu\, v \geq \mathtt{in}_\nu\, v \in \mathtt{in}^*_\mu\, [\![\mathcal{F}X\vec{C}]\!](\theta;\eta,\Phi^{\omega_1}/X)$ by definition of $\mathtt{in}^*_\mu$
$[\![(\mu\mathcal{F})\ \vec{C}]\!](\theta;\eta) = \Phi^{\omega_1}(\vec{C}) = \Phi(\Phi^{\omega_1})(\vec{C})$ as Φ^{ω_1} is a fixed point of Φ
$\quad = \mathtt{in}^*_\mu\, [\![\mathcal{F}X\vec{C}]\!](\theta;\eta,\Phi^{\omega_1}/X)$

Case: $\dfrac{\Delta;\Xi;\Gamma \vdash t : \mathcal{F}(\nu\mathcal{F})\,\vec{C}}{\Delta;\Xi;\Gamma \vdash \mathtt{in}_\nu\, t : (\nu\mathcal{F})\,\vec{C}}$

$[\![(\nu\mathcal{F})\,\vec{C}]\!](\theta;\eta) = \Psi^{\omega_1}\,(\overrightarrow{[\theta]C}) = \Psi(\Psi^{\omega_1})\,(\overrightarrow{[\theta]C})$ since Ψ^{ω_1} is a fixed point of Ψ
$\quad = \mathtt{out}^*_\nu\, [\![\mathcal{F}X\vec{C}]\!](\theta;\eta,\Psi^{\omega_1}/X) = \mathtt{out}^*_\nu\, [\![\mathcal{F}(\nu\mathcal{F})\ \vec{C}]\!](\theta;\eta)$ by Lemma 3.11
$[\theta;\sigma](\mathtt{in}_\nu\, t) \star .\mathtt{out}_\nu \longrightarrow [\theta;\eta]t$ by stepping rule
$[\theta;\sigma]t \geq [\theta;\sigma]t \in [\![\mathcal{F}(\nu\mathcal{F})\ \vec{C}]\!]^*(\theta;\eta)$ by induction hypothesis

Case: $\dfrac{\Delta, \overrightarrow{u{:}U};\Xi,X;\Gamma' \vdash t : T}{\Delta;\Xi;\Gamma \vdash \mathtt{rec}\, f,\iota,\rho,\vec{u},x.\, t : \Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\ \vec{u} \to T}$

where $\Gamma' = \Gamma, f : \Pi\overrightarrow{u{:}U}.X\ \vec{u} \to T, \iota : \Pi\overrightarrow{u{:}U}.X\ \vec{u} \to \mathcal{F}X\ \vec{u}, \rho : \Pi\overrightarrow{u{:}U}.X\ \vec{u} \to (\mu\mathcal{F})\ \vec{u}, x : \mathcal{F}X\ \vec{u}$
Let $g = [\theta;\sigma](\mathtt{rec}\, f,\iota,\rho,\vec{u},x.\, t)$. We want to show $g \geq g \in [\![\Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\ \vec{u} \to T]\!]^*$. Since g is a value, it suffices to show $g \geq g \in [\![\Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\ \vec{u} \to T]\!]$. We have that

$$[\![\Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\ \vec{u} \to T]\!](\theta;\eta) = \{v \geq v' \mid \forall \vec{C} \in [\![\vec{U}]\!](\theta), v\ \vec{C} \geq v'\ \vec{C} \in [\![(\mu\mathcal{F})\ \vec{u} \to T]\!]^*(\theta, \overrightarrow{C/u};\eta)\}$$

Let $\vec{C} \in [\![\vec{U}]\!](\theta)$. Since $g \star \vec{C}$ is a value, it suffices to show $g \star \vec{C} \geq g \star \vec{C} \in [\![(\mu\mathcal{F})\ \vec{u} \to$

$T]\!](\theta, \overrightarrow{C/u}; \eta)$. By definition of interpretation of our types, we have

$$[\![(\mu\mathcal{F})\ \vec{u} \to T]\!](\theta, \overrightarrow{C/u}; \eta) = \{v \geq v' \mid \forall w \geq w \in [\![(\mu\mathcal{F})\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta), v \star w \geq v' \star w \in [\![T]\!]^*(\theta, \overrightarrow{C/u}; \eta)\}$$

Moreover, since $\mu\mathcal{F}$ does not depend on $\vec{u}$,

$$\begin{aligned}[\![(\mu\mathcal{F})\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta) &= [\![\mu\mathcal{F}]\!](\theta, \overrightarrow{C/u}; \eta)\ (\vec{C}) = [\![\mu\mathcal{F}]\!](\theta; \eta)\ (\vec{C}) \\ &= \Phi^{\omega_1}\ (\vec{C}) = \textstyle\bigcup_{\alpha<\omega_1} \Phi^\alpha\ (\vec{C})\end{aligned}$$

We proceed by side induction on α. In the case where $\alpha = 0$, then there is nothing to prove as there are no $v \in \emptyset$. Now assume $\alpha > 0$ and that for all $\beta < \alpha$ we have

$$g \star \vec{C} \geq g \star \vec{C} \in [\![X \to T]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X).$$

Let $v \in \Phi^\alpha\ (\vec{C})$. We now want to show $g \star \vec{C}\ v \geq g \star \vec{C}\ v \in [\![T]\!]^*(\theta, \overrightarrow{C/u}; \eta)$. By definition of $\Phi^\alpha\ (\vec{C})$, we have $v = \mathtt{in}_\mu\, w$ with $\mathtt{in}_\mu\, w \geq \mathtt{in}_\mu\, w \in \Phi^\alpha\ (\vec{C})$. Moreover, we have that $\mathtt{in}_\mu\, w \geq \mathtt{in}_\mu\, w \in \Phi^{\beta+1}\ (\vec{C})$ by Lemma 3.6. Thus, $w \geq w \in [\![\mathcal{F}\ X]\!](\theta; \eta, \Phi^\beta/X)(\vec{C})$.

The stepping rules give use that

$$g \star \vec{C}\ (\mathtt{in}_\mu\, w) \longrightarrow [\theta, \overrightarrow{C/u}; \sigma, g/f, (\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x)/\iota, (\Lambda\vec{u}.\lambda x.x)/\rho, w/x]t$$

At this point we can appeal to the induction hypothesis to obtain the desired result. We only need to show

$$\begin{aligned}\sigma, g/f, (\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x)/\iota, (\Lambda\vec{u}.\lambda x.x)/\rho, w/x \geq \sigma, g/f, (\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x)/\iota, (\Lambda\vec{u}.\lambda x.x)/\rho, w/x \\ \in [\![\Gamma']\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X)\end{aligned}$$

By assumption, and since Γ does not depend on $\vec{u}$ or X, we trivially have

$$\sigma \geq \sigma \in [\![\Gamma]\!](\theta; \eta) = [\![\Gamma]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X)$$

Using the side induction as $\beta < \alpha$, and since there are no free occurrences of $\vec{u}$ in $\Pi\overrightarrow{u{:}U}.X\ \vec{u} \to T$, we have

$$g \geq g \in [\![\Pi\overrightarrow{u{:}U}.X\ \vec{u} \to T]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X) = [\![\Pi\overrightarrow{u{:}U}.X\ \vec{u} \to T]\!](\theta; \eta, \Phi^\beta/X)$$

Now we want to show

$$\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x \geq \Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x \in [\![\Pi\overrightarrow{u{:}U}.X\vec{u} \to \mathcal{F}X\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X).$$

Since there are no free occurrences of $\vec{u}$ in $\Pi\overrightarrow{u{:}U}.X\vec{u} \to \mathcal{F}X\ \vec{u}$, we have

$$[\![\Pi\overrightarrow{u{:}U}.X\vec{u} \to \mathcal{F}X\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X) = [\![\Pi\overrightarrow{u{:}U}.X\vec{u} \to \mathcal{F}X\ \vec{u}]\!](\theta; \eta, \Phi^\beta/X)$$

Let $\vec{C} \in [\![\vec{U}]\!](\theta)$. Then $\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x \star \vec{C} \longrightarrow \lambda x.\mathtt{out}_\mu\, x$. Let $v_1 \geq v_1 \in \Phi^\beta\ (\vec{C})$. Then, by definition of Φ^β we have $v_1 = \mathtt{in}_\mu\, w_1$. Since $\mathcal{F}Y$ does not depend on X, and since Φ is cumulative, we have

$$\begin{aligned} w_1 \geq w_1 \in [\![\mathcal{F}Y]\!](\theta; \eta, (\Phi^\beta\ (\vec{C}))/X, (\Phi^{\beta-1}\ (\vec{C}))/Y)\ (\vec{C}) &= [\![\mathcal{F}Y]\!](\theta; \eta, (\Phi^{\beta-1}\ (\vec{C}))/Y)\ (\vec{C}) \\ &= [\![\mathcal{F}Y]\!](\theta; \eta, (\Phi^\beta\ (\vec{C}))/Y)\ (\vec{C}) = [\![\mathcal{F}X]\!](\theta; \eta, (\Phi^\beta\ (\vec{C}))/X)\ (\vec{C}) \end{aligned}$$

Now, $(\lambda x.\mathtt{out}_\mu\, x) \star \mathtt{in}_\mu\, w_1 \longrightarrow \mathtt{out}_\mu\, (\mathtt{in}_\mu\, w_1) \longrightarrow w_1$ which is what we needed.

Now we want to show $\Lambda\vec{u}.\lambda x.x \geq \Lambda\vec{u}.\lambda x.x \in [\![\Pi\overrightarrow{u{:}U}.X\vec{u} \to (\mu\mathcal{F})\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X)$. Since there are no free occurrences of $\vec{u}$ in $\Pi\overrightarrow{u{:}U}.X\vec{u} \to (\mu\mathcal{F})\ \vec{u}$, we have

$$[\![\Pi\overrightarrow{u{:}U}.X\vec{u} \to (\mu\mathcal{F})\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X) = [\![\Pi\overrightarrow{u{:}U}.X\vec{u} \to (\mu\mathcal{F})\ \vec{u}]\!](\theta; \eta, \Phi^\beta/X)$$

Let $\vec{C} \in [\![\vec{U}]\!](\theta)$. Then $\Lambda\vec{u}.\lambda x.x \star \vec{C} \longrightarrow \lambda x.x$. Now we choose v_1 such that

$$v_1 \geq v_1 \in \Phi^\beta\ (\vec{C}) \subset \bigcup_{\alpha<\omega_1} \Phi^\alpha\ (\vec{C}) = [\![\mu\mathcal{F}\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta) = [\![\mu\mathcal{F}\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X)$$

Then $\lambda x.x \star v_1 \longrightarrow v_1$ as we wanted.

This concludes the case for recursors.

Case: $$\frac{\Delta, \overrightarrow{u{:}U}; \Xi, X; \Gamma' \vdash t : \mathcal{F}X\vec{u}}{\Delta; \Xi; \Gamma \vdash \mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t : \Pi\overrightarrow{u{:}U}.T \to (\nu\mathcal{F})\ \vec{u}}$$

where $\Gamma' = \Gamma, f : \Pi\overrightarrow{u{:}U}.T \to X\ \vec{u}, \iota : \Pi\overrightarrow{u{:}U}.\mathcal{F}X\ \vec{u} \to X\ \vec{u}, \rho : \Pi\overrightarrow{u{:}U}.(\nu\mathcal{F})\ \vec{u} \to X\ \vec{u}, x : T$
Let $g = [\theta; \sigma](\mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, t)$. We want to show $g \geq g \in [\![\Pi\overrightarrow{u{:}U}.T \to (\nu\mathcal{F})\ \vec{u}]\!]^*$. Since g is a value, it suffices to show $g \geq g \in [\![\Pi\overrightarrow{u{:}U}.T \to (\mu\mathcal{F})\ \vec{u}]\!]$. We have that

$$[\![\Pi\overrightarrow{u{:}U}.T \to (\nu\mathcal{F})\ \vec{u}]\!](\theta; \eta) = \{v \geq v' \mid \forall \vec{C} \in [\![\vec{U}]\!](\theta), v \star \vec{C} \geq v' \star \vec{C} \in [\![T \to (\nu\mathcal{F})\ \vec{u}]\!]^*(\theta, \overrightarrow{C/u}; \eta)\}$$

Let $\vec{C} \in [\![\vec{U}]\!](\theta)$. Since $g \star \vec{C}$ is a value, it suffices to show $v \star \vec{C} \geq v \star \vec{C} \in [\![T \to (\nu\mathcal{F})\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta)$ By definition of interpretation of our types, we have

$$[\![T \to (\nu\mathcal{F})\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta) = \{v \geq v' \mid \forall w \geq w \in [\![T]\!](\theta, \overrightarrow{C/u}; \eta), v \star w \geq v' \star w \in [\![(\nu\mathcal{F})\ \vec{u}]\!]^*(\theta, \overrightarrow{C/u}; \eta)\}$$

Moreover, since $\nu\mathcal{F}$ does not depend on $\vec{u}$,

$$\begin{aligned}[\![(\nu\mathcal{F})\ \vec{u}]\!]^*(\theta, \overrightarrow{C/u}; \eta) &= [\![\nu\mathcal{F}]\!](\theta, \overrightarrow{C/u}; \eta)\ (\vec{C}) = [\![\nu\mathcal{F}]\!](\theta; \eta)\ (\vec{C}) \\ &= \Psi^{\omega_1}\ (\vec{C}) = \textstyle\bigcap_{\alpha<\omega_1} \Psi^{\alpha}\ (\vec{C})\end{aligned}$$

We proceed by side induction on α. In the case where $\alpha = 0$, then there is nothing to prove as $v' \in \Omega$ for any v'. Now assume $\alpha > 0$ and that for all $\beta < \alpha$ we have

$$g \star \vec{C} \geq g \star \vec{C} \in [\![T \to X]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\beta}/X).$$

Let $v \geq v \in [\![T]\!](\theta, \overrightarrow{C/u}; \eta)$. We now want to show $g \star \vec{C}\ v \geq g \star \vec{C}\ v \in \Psi^{\alpha}\ (\vec{C})$. By definition of $\Psi^{\alpha}\ (\vec{C}) = \Psi(\Psi^{\alpha-1})\ (\vec{C})$, we want to show

$$g \star \vec{C}\ v\ .\mathtt{out}_{\nu} \geq g \star \vec{C}\ v\ .\mathtt{out}_{\nu} \in [\![\mathcal{F}X\vec{C}]\!]^*(\theta; \eta, \Psi^{\alpha-1}/X)$$

The stepping rules give use that

$$g \star \vec{C}\ v\ .\mathtt{out}_{\nu} \longrightarrow [\theta, \overrightarrow{C/u}; \sigma, g/f, (\Lambda\vec{u}.\lambda x.\mathtt{in}_{\nu}\ x)/\iota, (\Lambda\vec{u}.\lambda x.x)/\rho, v/x]t$$

At this point we can appeal to the induction hypothesis to obtain the desired result. We only need to show

$$\begin{aligned}\sigma, g/f, (\Lambda\vec{u}.\lambda x.\mathtt{in}_{\nu}\ x)/\iota, (\Lambda\vec{u}.\lambda x.x)/\rho, v/x \geq \sigma, g/f, (\Lambda\vec{u}.\lambda x.\mathtt{in}_{\nu}\ x)/\iota, (\Lambda\vec{u}.\lambda x.x)/\rho, v/x \\ \in [\![\Gamma']\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X)\end{aligned}$$

By assumption, and since Γ does not depend on $\vec{u}$ or X, we trivially have

$$\sigma \geq \sigma \in [\![\Gamma]\!](\theta; \eta) = [\![\Gamma]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X)$$

Using the side induction as $\alpha - 1 < \alpha$, and since there are no free occurrences of $\vec{u}$ in $\Pi\overrightarrow{u{:}U}.T \to X\ \vec{u}$, we have

$$g \geq g \in [\![\Pi\overrightarrow{u{:}U}.T \to X\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X) = [\![\Pi\overrightarrow{u{:}U}.T \to X\ \vec{u}]\!](\theta; \eta, \Psi^{\alpha-1}/X)$$

Now we want to show $\Lambda\vec{u}.\lambda x.\mathtt{in}_\nu\, x \geq \Lambda\vec{u}.\lambda x.\mathtt{in}_\nu\, x \in [\![\Pi\overrightarrow{u{:}U}.\mathcal{F}X\vec{u} \to X\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X)$. Since there are no free occurrences of $\vec{u}$ in $\Pi\overrightarrow{u{:}U}.\mathcal{F}X\vec{u} \to X\ \vec{u}$, we have

$$[\![\Pi\overrightarrow{u{:}U}.\mathcal{F}X\vec{u} \to X\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X) = [\![\Pi\overrightarrow{u{:}U}.\mathcal{F}X\vec{u} \to X\ \vec{u}]\!](\theta; \eta, \Psi^{\alpha-1}/X)$$

Let $\vec{C} \in [\![\vec{U}]\!](\theta)$. Then $(\Lambda\vec{u}.\lambda x.\mathtt{in}_\nu\, x)\star\vec{C} \longrightarrow \lambda x.\mathtt{in}_\nu\, x$. Let $v_1 \geq v_1 \in [\![\mathcal{F}X\vec{C}]\!](\theta; \eta, \Psi^{\alpha-1}/X)$. We need to show $(\lambda x.\mathtt{in}_\nu\, x) \star v_1 \longrightarrow \mathtt{in}_\nu\, v_1 \in \Psi^{\alpha-1}(\vec{C})$. It suffices to show $\mathtt{in}_\nu\, v_1 \geq \mathtt{in}_\nu\, v_1 \in \Psi^{\alpha}(\vec{C})$ as $\Psi^\alpha\ (\vec{C}) \subset \Psi^{\alpha-1}(\vec{C})$ by cumulativity. Hence, we want to show

$$\mathtt{in}_\nu\, v_1 \star .\mathtt{out}_\nu\ \geq \mathtt{in}_\nu\, v_1 \star .\mathtt{out}_\nu\ \in [\![\mathcal{F}X\vec{C}]\!]^*(\theta; \eta, \Psi^{\alpha-1}/X)$$

But $\mathtt{in}_\nu\, v_1 \star .\mathtt{out}_\nu\ \longrightarrow v_1$ and so we are done.

Now we want to show $\Lambda\vec{u}.\lambda x.x \geq \Lambda\vec{u}.\lambda x.x \in [\![\Pi\overrightarrow{u{:}U}.(\nu\mathcal{F})\ \vec{u} \to X\vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X)$. Since there are no free occurrences of $\vec{u}$ in $\Pi\overrightarrow{u{:}U}.(\nu\mathcal{F})\ \vec{u} \to X\vec{u}$, we have

$$[\![\Pi\overrightarrow{u{:}U}.(\nu\mathcal{F})\ \vec{u} \to X\vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Psi^{\alpha-1}/X) = [\![\Pi\overrightarrow{u{:}U}.(\nu\mathcal{F})\ \vec{u} \to X\vec{u}]\!](\theta; \eta, \Psi^{\alpha-1}/X)$$

Let $\vec{C} \in [\![\vec{U}]\!](\theta)$. Then $\Lambda\vec{u}.\lambda x.x \star \vec{C} \longrightarrow \lambda x.x$. Now we choose v_1 such that

$$v_1 \geq v_1 \in [\![\nu\mathcal{F}\ \vec{u}]\!](\theta, \overrightarrow{C/u}; \eta, \Phi^\beta/X) = [\![\nu\mathcal{F}\ \vec{C}]\!](\theta; \eta,) = \Psi^{\omega_1}\ (\vec{C}) = \bigcap_{\alpha<\omega_1} \Psi^\alpha\ (\vec{C}) \subset \Psi^{\alpha-1}\ (\vec{C})$$

Then $\lambda x.x \star v_1 \longrightarrow v_1$ as we wanted.

Case: $\dfrac{\Delta; \Xi; \Gamma \vdash t : (\mu\mathcal{F})\, \vec{C}}{\Delta; \Xi; \Gamma \vdash \mathtt{out}_\mu\, t : \mathcal{F}(\mu\mathcal{F})\, \vec{C}}$

$[\theta; \sigma]t \longrightarrow^* v$ and $v \geq v \in [\![(\mu\mathcal{F})\ \vec{C}]\!](\theta; \eta)$ by induction hypothesis

$[\![(\mu\mathcal{F})\ \vec{C}]\!](\theta; \eta) = \Phi^{\omega_1}(\overrightarrow{[\theta]C}) = \Phi(\Phi^{\omega_1})(\overrightarrow{[\theta]C})$ as Φ^{ω_1} is a fixed point of Φ

$\quad = \mathtt{in}^*_\mu\, [\![\mathcal{F}X\vec{C}]\!](\theta; \eta, \Phi^{\omega_1}/X)$

$v = \mathtt{in}_\mu\, w$ where $w \geq w \in [\![\mathcal{F}X\vec{C}]\!](\theta; \eta, \Phi^{\omega_1}/X) = [\![\mathcal{F}(\mu\mathcal{F})\ \vec{C}]\!](\theta; \eta)$ by Lemma 3.11

$[\theta; \sigma](\mathtt{out}_\mu\, t) \longrightarrow^* \mathtt{out}_\mu\, (\mathtt{in}_\mu\, w) \longrightarrow w$ by Lemma 3.8

Case: $\dfrac{\Delta; \Xi; \Gamma; \mathcal{F}(\nu\mathcal{F})\ \vec{C} \vdash E \searrow T}{\Delta; \Xi; \Gamma; (\nu\mathcal{F})\ \vec{C} \vdash .\mathtt{out}_\nu\, E \searrow T}$

Let $v \geq v \in [\![(\nu\mathcal{F})\ \vec{C}]\!](\theta;\eta) = \Psi^{\omega_1}\,(\overrightarrow{[\theta]C}) = \Psi(\Psi^{\omega_1})\,(\overrightarrow{[\theta]C})$ since Ψ^{ω_1} is a fixed point of Ψ

$= \mathtt{out}^*_\nu\,[\![\mathcal{F}X\vec{C}]\!](\theta;\eta,\Psi^{\omega_1}/X) = \mathtt{out}^*_\nu\,[\![\mathcal{F}(\nu\mathcal{F})\ \vec{C}]\!](\theta;\eta)$ by Lemma 3.11

$v \star .\mathtt{out}_\nu \longrightarrow^* w$ and $w \geq w \in [\![\mathcal{F}(\nu\mathcal{F})\ \vec{C}]\!]^*(\theta;\eta)$ by definition of $\mathtt{out}^*_\nu$

$w\ [\theta;\sigma]E \geq w\ [\theta;\sigma]E \in [\![T]\!]^*(\theta;\eta)$ by induction hypothesis

□

3.3 Function Criteria

The overall goal of this chapter is to define criteria on which we can rely to assess that a program defined in copatterns is normalising. The first two parts of the criteria are pretty straightforward. We only care about the behaviour of well-typed terms so our first criterion is for the terms to be well-typed. The second criterion is for functions to be covering as per Definition 2.1 as it was already required for progress and we desire to assert our programs as valid proofs.

The last criterion is that functions must either not be recursive or to be structurally (co)recursive. This criterion is based on the structural recursion criterion defined by Coquand [1992] and Goguen et al. [2006a]. We first start with the conditions for a function to be considered not recursive. The main requirement is for the body of the function to not depend on its function symbol. We also add the extra requirement that it cannot have an empty copattern in any of its branches in order to simplify the proofs later. This gives us the following definition:

Definition 3.3 (Not recursive)**.** A function $\mathtt{fun}\ f.\overrightarrow{q \mapsto t}$ is *not recursive* if there are no q_i such that $q_i = \cdot$ and there are no free occurrences of f in any of the terms t_i.

Structural Recursiveness

In order for a function to be structurally recursive, we need to ensure recursive calls are safe. This is described through the concept of inductive guardedness which is denoted by $t \prec_{\text{ind}} h$

and means that the term t is inductively guarded by the head h. The rules are the following:

$$\frac{}{t \prec_{\text{ind}} c\ t} \quad \frac{}{t \prec_{\text{ind}} \texttt{pack}\,\langle C, t\rangle} \quad \frac{}{t_1 \prec_{\text{ind}} (t_1, t_2)} \quad \frac{}{t_2 \prec_{\text{ind}} (t_1, t_2)}$$

$$\frac{E \neq \cdot \quad x \prec_{\text{ind}} h}{x \star E \prec_{\text{ind}} h} \quad \frac{t \prec_{\text{ind}} h' \quad h' \prec_{\text{ind}} h}{t \prec_{\text{ind}} h}$$

Intuitively, inductive guardedness holds if one of the following holds:

- h is a constructor or a (dependent) pair and t is a direct subterm of h,
- the head of t is guarded by h,
- there is a head h' such that t is guarded by h' and h' is guarded by h.

The transitivity rule ensures we can go arbitrarily deep in a subterm. The rule $x \star E \prec_{\text{ind}} h$ if $x \prec_{\text{ind}} h$ follows the tradition of Coquand [1992] and Goguen et al. [2006a] and signifies that applications of subterms to arbitrary terms are still subterms. Such applications are safe as we have the positivity restriction for (co)recursive types. We note that terms are guarded by heads instead of terms. This is because we want to guard terms using patterns and patterns are a subset of heads. In the remainder of this thesis, we will often say $t \prec_{\text{ind}} p$ to mean $t \prec_{\text{ind}} h$ where h is the lifting of p as a head.

Using guardedness we can define what it means to be structurally recursive:

Definition 3.4 (Structurally recursive)**.** A function $\Delta; \Gamma \vdash \texttt{fun}\ f.\overrightarrow{q \mapsto t} : \Pi\overrightarrow{u{:}U}.(\mu\mathcal{F})\ \vec{C} \to T$ is *structurally recursive* if for all branches $q_i \mapsto t_i$, the copattern q_i is of the form $\vec{u}\ (c_i\ p_i)\ q_i'$ and for all recursive calls $f \star \vec{C}\ t\ E$ in the right-hand side t_i, we have that $t \prec_{\text{ind}} p_i$, where the pattern p_i is lifted as a term.

We note that we do not only require the arguments to be inductively guarded but we also ask for both a specific type signature and a specific shape for the copattern of all branches, even if a branch does not have any recursive call. The type forces the recursive argument to be the first non-index argument passed to the function. The shape of copatterns allows us to expose the patterns p_i against which the recursive calls are guarded. Those patterns p_i are called pattern guards. We require every branch to expose a constructor as an artifact of

the translation. We shall go over the details of this restriction in Section 3.4 when we will be discussing the translation itself.

Those requirement do impose a somewhat stringent restriction on the functions that are admitted. For example, take the following function that computes the maximum of two inputs:

```
fun max : Nat → Nat → Nat
| n zero = n
| zero (suc m) = suc m
| (suc n) (suc m) = suc (max ⋆ n m)
```

Both of its arguments are inductively guarded in the single recursive call `max ⋆ n m`. However, the function is not deemed structurally recursive since it does not expose a constructor on its first argument in its branch. Moreover, we only ever consider the first argument and so we cannot use that the second argument does expose a constructor in every branch. Swapping the order of patterns in each branch would solve the problem. Similarly, we could duplicate the first branch and split on `n` resulting in a function:

```
fun max : Nat → Nat → Nat
| zero zero = zero
| (suc n) zero = suc n
| zero (suc m) = suc m
| (suc n) (suc m) = suc (max ⋆ n m)
```

One could even imagine a preprocessing step during which reordering and/or branch duplication is done to relax our definition of structural recursiveness and still follow the translation presented in Section 3.4. While we believe such preprocessing should be possible to automate and integrate as part of termination checking of copatterns, we will not develop this idea further and focus instead on the translation based on the definition based above.

Structural Corecursiveness

Once again, we need to make sure corecursive calls are guarded in some way. The guardedness condition, however, is different; it does not matter what arguments the function is applied to. What matters is where the recursive call is made. In particular, we do not want to unfold corecursive calls in an unsafe way. We recall the function `eager` from the beginning of the chapter.

```
fun eager : Stream
| .head = 0
| .tail = eager.tail
```

Note that there could be instances where the function is guarded by multiple observations and we safely strip away some of them. One such example is the definition of `fib` at Page 12. Our setting will reject such definition for simplicity. In the case of `fib`, there exist rewritings of such function that would pass our criteria. However, the transformations leading to such valid functions often require creativity and thus cannot easily be converted automatically. This is outside the scope of this thesis. In addition, passing the corecursive call to a function is not safe as we do not have guarantees this function will not try to expand the corecursive call we are passing to it.

$\boxed{s \prec_{\text{coind}} t}$ Term s is coinductively guarded by term t.

$$\frac{}{t \prec_{\text{coind}} c\ t} \quad \frac{}{t \prec_{\text{coind}} \texttt{pack}\,\langle C, t\rangle} \quad \frac{}{t_1 \prec_{\text{coind}} (t_1, t_2)} \quad \frac{}{t_2 \prec_{\text{coind}} (t_1, t_2)} \quad \frac{r \prec_{\text{coind}} s \quad s \prec_{\text{coind}} t}{r \prec_{\text{coind}} t}$$

Figure 3.6: Coinductive guardedness

The guardedness condition $s \prec_{\text{coind}} t$ indicates s is guarded by t. It allows corecursive calls to be under constructors, (dependent) pairs, or at the head of applications. Just like for the inductive guardedness, we have a transitivity rule. We note that the definition also restricts nesting a corecursive call inside another function. This is because corecursive calls

have abstract types X in the target language and nesting corecursive calls create situations where we would have functions that expect terms of abstract types X.

Definition 3.5 (Structurally corecursive)**.** A function $\Delta; \Gamma \vdash \texttt{fun}\ f.\overrightarrow{q \mapsto t} : \Pi u{:}U.T \to (\nu\mathcal{F})\ \vec{C}$ is *structurally corecursive* if, all for all branches $q_i \mapsto t_i$, the copattern q_i is of the form $p_i\ .d_i\ q_i'$ and all corecursive calls $f \star \vec{C}\ t'$ in the term t_i satisfy $f \star \vec{C}\ t' \prec_{\text{coind}} t_i$.

In addition to guardedness, we force corecursive functions to have a single argument and for copattern splits to have an observation in each branch. Once again, those conditions are imposed by our translations and could be alleviated by source level translations uncurrying arguments or duplicating non recursive branches.

Navigating Structural (Co)Recursiveness

We note that it is possible for a function to satisfy more than one of these definitions. For example, the following function produces always a constant stream of the predecessor of the input.

```
fun f : Nat → Stream =
| zero .head ⇒ zero
| (suc n) .head ⇒ n
| zero .tail ⇒ zeroes
| (suc n) .tail ⇒ const ⋆ n
```

There are no occurrences of `f` and no empty copatterns so it is not recursive. It is also structurally recursive because all copatterns split on the first non-index argument exposing a constructor, and all recursive calls satisfy trivially inductive guardedness as there are no recursive calls. It is also structurally corecursive as there is a single argument and all copatterns have an observation and there are no corecursive calls so the coinductive guardedness is trivially satisfied.

There are functions that satisfy both inductively and coinductively guarded in a non trivial way. Recall the function `ev-to-coev` from Section 2.1.

```
fun ev-to-coev : Πn:Nat. Even n → Coeven n =
 | zero (ev-z ℘ ())          .cev-sz ℘
 | zero (ev-z ℘ ())          .cev-ss m ℘
 | (suc n) (ev-ss <m, (℘, e)>) .cev-sz ℘
 | (suc n) (ev-ss <m, (℘, e)>) .cev-ss k ℘ ⇒ ev-to-coev⋆ e
```

This function is both structurally recursive and corecursive as the only recursive call `ev-to-coev e` is coinductively guarded while its argument `e` is inductively guarded.

For the purpose of our translation, as structural (co)recursiveness guides how the translation proceeds, we will simply assume there is a consistent ordering in case of overlap. It does not matter what is this ordering as long as it is followed. Prioritizing non recursive functions will likely result in simpler translations as it will not generate (co)recursors when it is not needed. However, we will not worry about it.

Summarizing the Function Criteria

This leads us to the final representation of our criteria:

Definition 3.6 (Function criteria)**.** We say that a term t *satisfies the function criteria* if $\Delta; \Gamma \vdash t : T$ and for all functions g that are subterms of t, the following hold:

- If Q is the copattern set for g then $\cdot \searrow \Delta; \Gamma; T \Longrightarrow Q$;
- one of the following holds:

  1. g is not recursive,
  2. g is structurally recursive,
  3. g is structurally (co)recursive.

Before, we move on to the translation itself, we wish to prove that evaluation does not invalidate the function criteria. This will be needed to show the function criteria are sufficient conditions for termination checking.

Lemma 3.14 (Evaluation preserves the function criteria)**.** *If t satisfies the function criteria and $t \longrightarrow t'$, then t' satisfies the function criteria.*

Proof. By Theorem 2.10, typing is preserved by evaluation. The operational semantics does not modify copattern sets of functions and we do not evaluate under functions. Thus, we can prove by a simple induction that any function g that is a subterm of t' is also a subterm of t and so both coverage and structural (co)recursiveness are preserved. □

3.4 Translation

We have defined a core calculus and proved normalisation for it, and we defined criteria for termination. What remains to be done is to define a translation and prove it preserves normalisation. This section describes the translation.

Challenges

Before we dive into the details of the translation, we first go over some examples in order to discuss specific challenges we face when designing it. Those challenges dictate choices not only in the translation itself, but also in its underlying proofs of metatheory.

Example 4 (Translation not uniquely determined)*.* Suppose we wish to translate the copattern set in the non recursive function which produces a constant stream of the predecessor of the input.

```
fun f : Nat → Stream =
| zero .head ⇒ zero
| (suc n) .head ⇒ n
| zero .tail ⇒ zeroes
| (suc n) .tail ⇒ const ⋆ n
```

Now, our target language splits on sums and record terms one at the time using case splits and observations. In the source language, we can use our coverage derivations to decompose

the source program one at the time. For our example program, we could derive the following coverage derivation:

$$\cdot \Longrightarrow \{n\ \cdot\} \Longrightarrow \left\{ \begin{array}{cc} \mathsf{zero} & \cdot \\ (\mathsf{suc}\, n) & \cdot \end{array} \right\} \Longrightarrow \left\{ \begin{array}{cc} \mathsf{zero} & .\mathsf{head}\ \cdot \\ \mathsf{zero} & .\mathsf{tail}\ \cdot \\ (\mathsf{suc}\, n) & .\mathsf{head}\ \cdot \\ (\mathsf{suc}\, n) & .\mathsf{tail}\ \cdot \end{array} \right\}$$

Alternatively, we could derive the following coverage derivation:

$$\cdot \Longrightarrow \{n\ \cdot\} \Longrightarrow \left\{ \begin{array}{cc} n & .\mathsf{head}\ \cdot \\ n & .\mathsf{tail}\ \cdot \end{array} \right\} \Longrightarrow \left\{ \begin{array}{cc} \mathsf{zero} & .\mathsf{head}\ \cdot \\ (\mathsf{suc}\, n) & .\mathsf{head}\ \cdot \\ \mathsf{zero} & .\mathsf{tail}\ \cdot \\ (\mathsf{suc}\, n) & .\mathsf{tail}\ \cdot \end{array} \right\}$$

Thus, we could build a translation of our copattern set as the following target-level term:

```
λn.case out_μ n of
  | in_1 n' ⇒ in_ν (zero; zeroes)
  | in_2 n' ⇒ in_ν (n'; const ⋆ n')
```

We can also translate it into the following program:

```
λn.in_ν (case out_μ n of
    | in_1 n' ⇒ zero
    | in_2 n' ⇒ n' ;
       case out_μ n of
    | in_1 n' ⇒ zeroes
    | in_2 n' ⇒ const ⋆ n')
```

As we can see the shape of the translated programs follow closely possible coverage derivations for our original program. In fact, we use coverage as a guide to translate copatterns. When multiple possible derivations exist for a given problem, we simply choose one.

Example 5 (Deep patterns translation). As another example, suppose we have the following function we wish to translate:

```
fun f : Nat → Nat
| zero ⇒ zero
| suc zero ⇒ zero
| suc (suc n) ⇒ f ⋆ (suc n)
```

Now the translation would look like the following:

```
rec f, ι, ρ, ρ, x. case x of
| in₁ x' ⇒ zero
| in₂ x' ⇒ case ι ⋆ x' of
             | in₁ y ⇒ zero
             | in₂ y ⇒ ?
```

The main question is what term would we put in place of the hole. Looking at the source code, we would be tempted to write something like `f ⋆ (in`$_\mu$ `(in`$_2$ `y))`. However, `y` has type X and so `in`$_\mu$ `(in`$_2$ `y)` is ill-typed. Instead, the right answer is `f ⋆ x'` where `x'` has type X and so the recursive call is well-typed. We thus need a way to keep track of what variable denotes specific subterms of the pattern. This is done through the mapping φ which, when supplied a term that is a subterm of the pattern guard, yields the variable corresponding to it in the target language. This mapping thus accumulates the needed information to translate adequately recursive calls.

Example 6 (Translations for functions that are both recursive and corecursive). We mentioned in the last section that the function `ev-to-coev` is both structurally recursive and corecursive.

```
fun ev-to-coev : Πn:Nat. Even n → Coeven n =
 | n (ev-z (℘, x))          .cev-sz ℘
 | n (ev-z (℘, x))          .cev-ss k ℘
 | n (ev-ss <m, (℘, e)>) .cev-sz ℘
 | n (ev-ss <m, (℘, e)>) .cev-ss k ℘ ⇒ ev-to-coev ⋆ m e
```

This leads us to two different possible translations. If we consider the function recursively, we would generate the term

```
rec ev-to-coev, ι,ρ,n,x₀.case x of
| in₁ x' ⇒ split x' as (p, x) in
            eq p with
            in_ν (λq.eq_abort q; Λk.λq.eq_abort q)
| in₂ x' ⇒ unpack x' as <m, x''> in
            split x'' as (p,e) in
            eq p with
            in_ν (λq.eq_abort q; Λk.λq.eq q with ev-to-coev ⋆ m e)
```

This is obtained by using the coverage derivation

$$n\ x_0\ \cdot \Longrightarrow \left\{ \begin{array}{l} n\ (\textsf{ev-z}\ x')\ \cdot \\ n\ (\textsf{ev-ss}\ x')\ \cdot \end{array} \right\} \Longrightarrow^* \left\{ \begin{array}{l} n\ (\textsf{ev-z}\ (\wp, x))\ \cdot \\ n\ (\textsf{ev-ss}\ \langle m, (\wp, e)\rangle)\ \cdot \end{array} \right\}$$

$$\Longrightarrow \left\{ \begin{array}{ll} n\ (\textsf{ev-z}\ (\wp, x)) & .\textsf{cev-sz}\ \cdot \\ n\ (\textsf{ev-z}\ (\wp, x)) & .\textsf{cev-ss}\ \cdot \\ n\ (\textsf{ev-ss}\ \langle m, (\wp, e)\rangle) & .\textsf{cev-sz}\ \cdot \\ n\ (\textsf{ev-ss}\ \langle m, (\wp, e)\rangle) & .\textsf{cev-ss}\ \cdot \end{array} \right\} \Longrightarrow^* \left\{ \begin{array}{ll} n\ (\textsf{ev-z}\ (\wp, x)) & .\textsf{cev-sz}\ \wp\ \cdot \\ n\ (\textsf{ev-z}\ (\wp, x)) & .\textsf{cev-ss}\ k\ \wp\ \cdot \\ n\ (\textsf{ev-ss}\ \langle m, (\wp, e)\rangle) & .\textsf{cev-sz}\ \wp\ \cdot \\ n\ (\textsf{ev-ss}\ \langle m, (\wp, e)\rangle) & .\textsf{cev-ss}\ k\ \wp\ \cdot \end{array} \right\}$$

For readability, we skip over some of the steps of the coverage derivation that would simply refine variables. Alternatively, we can build a corecursive translation for `ev-to-coev` as follows

```
corec ev-to-coev, ι,ρ,n,x₀.
(λp.eq p with
    case out_μ x₀ with
    | in₁ x' ⇒ split x' as (q, x) in
                eq_abort q
    | in₂ x' ⇒ unpack x' as <m, x''> in
                split x'' as (q,e) in
                eq_abort q;
Λk.λp.eq p with
    case out_μ x₀ with
```

```
| in1 x' ⇒ split x' as (q, x) in
            eq_abort q
| in2 x' ⇒ unpack x' as <m, x''> in
            split x'' as (q,e) in
            eq q with
            ev-to-coev ⋆ m e)
```

This function is generated using the following coverage derivation:

$$n\ x_0\ \cdot \Longrightarrow \left\{ \begin{array}{l} n\ x_0\ .\mathsf{cev\text{-}sz}\ \cdot \\ n\ x_0\ .\mathsf{cev\text{-}ss}\ \cdot \end{array} \right\} \Longrightarrow^* \left\{ \begin{array}{l} n\ x_0\ .\mathsf{cev\text{-}sz}\ \wp\ \cdot \\ n\ x_0\ .\mathsf{cev\text{-}ss}\ k\ \wp\ \cdot \end{array} \right\}$$

$$\Longrightarrow \left\{ \begin{array}{l} n\ (\mathsf{ev\text{-}sz}\ x')\ .\mathsf{cev\text{-}sz}\ \wp\ \cdot \\ n\ (\mathsf{ev\text{-}ss}\ x')\ .\mathsf{cev\text{-}sz}\ \wp\ \cdot \\ n\ (\mathsf{ev\text{-}sz}\ x')\ .\mathsf{cev\text{-}ss}\ k\ \wp\ \cdot \\ n\ (\mathsf{ev\text{-}ss}\ x')\ .\mathsf{cev\text{-}ss}\ k\ \wp\ \cdot \end{array} \right\} \Longrightarrow^* \left\{ \begin{array}{ll} n\ (\mathsf{ev\text{-}sz}\ (\wp, x)) & .\mathsf{cev\text{-}sz}\ \wp\ \cdot \\ n\ (\mathsf{ev\text{-}ss}\ \langle m, (\wp, e)\rangle) & .\mathsf{cev\text{-}sz}\ \wp\ \cdot \\ n\ (\mathsf{ev\text{-}sz}\ (\wp, x)) & .\mathsf{cev\text{-}ss}\ k\ \wp\ \cdot \\ n\ (\mathsf{ev\text{-}ss}\ \langle m, (\wp, e)\rangle) & .\mathsf{cev\text{-}ss}\ k\ \wp\ \cdot \end{array} \right\}$$

Both coverage derivations are pretty similar, with a particular choice made at a different moment. Obviously, there could be other variations of those derivations alternating when to refine variables and introduce new variables after the observations. The specific details do not matter much for our purposes. It is however important for the translation that, in the recursive case, we match on x_0 first, while in the corecursive case, we split on the observations first. The function criteria guarantee such derivations exist. We prove such properties in Lemmas 3.16 to 3.19.

Example 7 (Evaluation does not preserve translation). Our last example considers how evaluation affects the translation. For this, we take a look at the following nested functions:

```
fun f.
| zero ⇒ fun g. y ⇒ y
| suc x ⇒ fun g. y ⇒ add * x (f * x y)
```

This function when applied to x and y computes the sum $y + \sum_{n=0}^{x-1} n$. If we are to apply it the number 2, it will reduce to the function **fun** `g. y` $\Rightarrow$ `add * 2 (f * 2 y)` where `f` stands for the original function. If we translate both programs we get respectively:

rec `f, ι, ρ, x`$_0$`.` **case** `x`$_0$ **of**
`|` **in**$_1$ `x` $\Rightarrow$ λ`y.y`
`|` **in**$_2$ `x` $\Rightarrow$ λ`y.add * (ρ * x) (f * x y)`

and λ`y.add * 2 (f * 2 y)`. Applying the former to 2 and evaluating it gives the function λ`y.add * ((λz.z) * 2) (f * 2 y)`. Here, we have the extra ρ that was instantiated to λ`z.z`. The programs are thus not equal anymore.

Thus, we cannot commute evaluation and translation directly, but they nevertheless behave the same. This complicates our argument to bind the number of steps in the source language. We thus need to come up with a more sophisticated metric. This will be the relation in our interpretation of types (Figure 3.5), which we will use as a simulation.

Translation of Copatterns

The translation is split into two main components: on the one hand we have the translation of terms, which covers also translation of heads and spines. This is based on both the structure of terms, typing informations, and structural (co)recursiveness. On the other hand we translate copatterns into terms of our core calculus following derivation of coverage.

We first discuss translation of copatterns. The two main judgments for it are

$$(q \searrow \Delta; \Gamma; T; \varphi) \Longrightarrow_b^* Q \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n] \quad \text{and} \quad (q \searrow \Delta; \Gamma; T; \varphi) \Longrightarrow_{\vec{b}} Q \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n].$$

Both judgments mimic the coverage derivation judgments. One can look at them as taking as input a coverage judgment $(q \searrow \Delta; \Gamma; T) \Longrightarrow^* Q$ and outputting a term $t[\cdot_1 \mid \cdots \mid \cdot_n]$ that has n holes where n is the size of the copattern set Q. For example, given a coverage step

$$\{n\ \cdot\} \Longrightarrow \left\{ \begin{array}{c} \mathsf{zero}\ \ \cdot \\ (\mathsf{suc}\, m)\ \cdot \end{array} \right\}$$

we generate the term $\mathtt{case}\, n\, \mathtt{of}\, \mathtt{in}_1\, m \mapsto \cdot_1 \mid \mathtt{in}_2\, m \mapsto \cdot_2$. Then, when we do the steps

$$\left\{ \begin{array}{ll} \mathsf{zero} & \cdot \\ (\mathsf{suc}\, n) & \cdot \end{array} \right\} \Longrightarrow \left\{ \begin{array}{lll} \mathsf{zero} & .\mathsf{head} & \cdot \\ \mathsf{zero} & .\mathsf{tail} & \cdot \\ (\mathsf{suc}\, n) & .\mathsf{head} & \cdot \\ (\mathsf{suc}\, n) & .\mathsf{tail} & \cdot \end{array} \right\}$$

we obtain terms $\mathtt{in}_\nu\, (\cdot_{11}; \cdot_{12})$ and $\mathtt{in}_\nu\, (\cdot_{21}; \cdot_{22})$ that we plug in for $\cdot_1$ and $\cdot_2$, respectively. This gives us the term $\mathtt{case}\, n\, \mathtt{of}\, \mathtt{in}_1\, m \mapsto \mathtt{in}_\nu\, (\cdot_{11}; \cdot_{12}) \mid \mathtt{in}_2\, m \mapsto \mathtt{in}_\nu\, (\cdot_{21}; \cdot_{22})$. Once we have a complete term for the copattern, we fill the remaining holes with the translation of the corresponding right-hand sides.

In addition to a coverage derivation, we carry around the branches $\vec{b}$, and a mapping φ that we use to accumulate valid recursive calls. The contexts Δ and Γ, and type T serve to type the generated term t as opposed to the copattern itself. As such, types in Γ or the type T itself might contain type variables X inherited from the translation of the term containing the copattern set being translated.

Both the initial copattern declaration and the ones in Q carry around the mapping φ we mentioned above. This mapping contains pairs $x := t$ connecting the source level expression t with the intermediate variable x. As such, we require to be able to lookup a term and obtain its corresponding variable. The operation $\varphi(t) = x$ yields the variable x when looking up to term t. While φ accumulates the recursive calls of multiple functions at once, we assume we can always disambiguate them by (silently) associating the type variable corresponding to the recursive call. Thus, the mapping φ can be seen as a list of mappings φ_X iterating over X. We note that it is only needed for structurally recursive functions as corecursive calls accept any arguments and only require to be at the right position in the terms.

The multi-refinements judgment is defined in Figure 3.7. The base case occurs if the copattern we are trying to split on is one of the copatterns in our final copattern set $\vec{q}$. In this case we return a single hole $\cdot$. In the inductive case we appeal to the single refinement judgment to obtain a term t with n holes and for each copattern generated, we produce a new term to fill one of the holes. We note that each new term will itself have holes. For readability, we chose to omit writing them explicitly in the rule.

$\boxed{q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow^*_B Q \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n]}$ Refining copattern q into copattern set Q generates term t using mapping φ.

$$\frac{}{q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow^*_{\{q \mapsto t\}} \{q \searrow \Delta; \Gamma; T; \varphi\} \rightsquigarrow \cdot}$$

$$\frac{q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow_B Q' \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n] \quad B_i = \{(q \mapsto t) \in B \mid q \in Q_i\} \quad \text{for all } (q_i \searrow \Delta_i; \Gamma_i; T_i; \varphi_i) \in Q', (q_i \searrow \Delta_i; \Gamma_i; T_i; \varphi_i) \Longrightarrow^*_{B_i} Q_i \rightsquigarrow t_i}{q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow^*_B \bigcup_i Q_i \rightsquigarrow t[t_1 \mid \cdots \mid t_n]}$$

Figure 3.7: Translation of Copatterns

Just as with coverage, the crux of the work is done through the single refinement rules. They are shown in Figure 3.8. They are once again split between copattern introductions and pattern refinements. Introducing a variable in the copattern leads us to introduce a λ-abstraction in the generated term. Introducing an observation means we need to introduce a coinductive record. How we build the coinductive record depends on the expected type. Two choices are possible depending on whether the function whose copattern set we are refining is structurally corecursive or not. This is reflected in the expected type. Either we expect $\nu\mathcal{F}\ \vec{C}$ which tells us we expect a constant of coinductive type at this particular position, or we expect $X\ \vec{C}$ which indicates the function is coinductively building this term. In the former case, we generate the term $\mathtt{in}_\nu\ (\cdot_1; \ldots; \cdot_n)$ while in the latter we have $\iota_X \star \vec{C}\ (\cdot_1; \ldots; \cdot_n)$. We will always have the variables ι_X and ρ_X in the context Γ if we ever encounter a type variable X as they all are introduced simultaneously by (co)recursors.

Pattern refinement rules introduce a term that takes care of the variable x being refined. Those rules are then subdivided into three main categories. First, we have the rules dealing with variables of equality types. If we do not have a contradiction, we simply provide an equality eliminator. Otherwise, we check if the corresponding branch has a right-hand side. If it does, we provide again an equality eliminator. This eliminator will have a contradiction in its context allowing it to discharge equalities trivially. We introduce an equality eliminator instead of an abort here because some branches can become impossible with evaluation.

$\boxed{q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow_B Q \rightsquigarrow t}$ Copattern refinement $q \searrow \Delta; \Gamma; T \Longrightarrow Q$ generate term t using mapping φ.

(Co)Pattern Introduction

$$\frac{}{(q \searrow \Delta; \Gamma; \Pi u{:}U.T; \varphi) \Longrightarrow_B \{q\ u \searrow \Delta, u{:}U; \Gamma; T; \varphi\} \rightsquigarrow \Lambda u.\cdot}$$

$$\frac{}{(q \searrow \Delta; \Gamma; S \to T; \varphi) \Longrightarrow_B \{q\ x \searrow \Delta; \Gamma, x{:}S; T; \varphi\} \rightsquigarrow \lambda x.\cdot}$$

$$\frac{}{(q \searrow \Delta; \Gamma; \nu\mathcal{F}\ \vec{C}; \varphi) \Longrightarrow_B \{q\ .d_i \searrow \Delta; \Gamma; \mathcal{F}_i(\nu\mathcal{F})\ \vec{C}; \varphi\}_i \rightsquigarrow \mathtt{in}_\nu\,(\cdot_1; \ldots; \cdot_n)}$$

$$\frac{\iota_X \in \Gamma}{(q \searrow \Delta; \Gamma; X\ \vec{C}; \varphi) \Longrightarrow_B \{q\ .d_i \searrow \Delta; \Gamma; \mathcal{F}_i X\ \vec{C}; \varphi\}_i \rightsquigarrow \iota_X \star \vec{C}\ (\cdot_1; \ldots; \cdot_n)}$$

Pattern Refinement

$$\frac{\Delta \vdash C_1 = C_2 \searrow \Delta' \quad \# \notin \Delta'}{(q[x] \searrow \Delta; \Gamma, x : C_1{=}C_2; T; \varphi) \Longrightarrow_B \{q[\wp] \searrow \Delta'; \Gamma; [\wp/x]\varphi\} \ \rightsquigarrow\ \mathtt{eq}\,x\,\mathtt{with}\,\cdot}$$

$$\frac{\Delta \vdash C_1 = C_2 \searrow \Delta' \quad \# \in \Delta' \quad \text{some } b \in B \text{ has a right-hand side}}{(q[x] \searrow \Delta; \Gamma, x : C_1{=}C_2; T; \varphi) \Longrightarrow_B \{q[\wp] \searrow \Delta'; \Gamma; [\wp/x]\varphi\} \ \rightsquigarrow\ \mathtt{eq}\,x\,\mathtt{with}\,\cdot}$$

$$\frac{\Delta \vdash C_1 = C_2 \searrow \Delta' \quad \# \in \Delta' \quad b \text{ has no right-hand side}}{(q[x] \searrow \Delta; \Gamma, x : C_1{=}C_2; T; \varphi) \Longrightarrow_{\{b\}} \{q[\wp] \searrow \Delta'; \Gamma; T; [\wp/x]\varphi\} \rightsquigarrow \mathtt{eq\_abort}\,x}$$

$$\frac{x \notin \varphi}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : T_1{\times}T_2; T; \varphi) \Longrightarrow_B & \{q[(x_1, x_2)] \searrow \Delta; \Gamma, x_1{:}T_1, x_2{:}T_2; T; \varphi\} \\ \rightsquigarrow & \mathtt{split}\,x\,\mathtt{as}\,(x_1, x_2)\,\mathtt{in}\,\cdot\end{array}}$$

$$\frac{x \notin \varphi}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : \Sigma u{:}U.T'; T; \varphi) \Longrightarrow_B & \{q[\mathtt{pack}\,\langle u, x'\rangle] \searrow \Delta, u{:}U; \Gamma, x'{:}T'; T; \varphi\} \\ \rightsquigarrow & \mathtt{unpack}\,x\,\mathtt{as}\,\langle u, x'\rangle\,\mathtt{in}\,\cdot\end{array}}$$

$$\frac{x \notin \varphi}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : \mu\mathcal{F}\ \vec{C}; T; \varphi) \Longrightarrow_B & \{q[c_i\ x_i] \searrow \Delta; \Gamma, \overrightarrow{x_i{:}\mathcal{F}_i(\mu\mathcal{F})\ \vec{C}}; T; \varphi\}_i \\ \rightsquigarrow & \mathtt{case}\,\mathtt{out}_\mu\,x\,\mathtt{of}\,\overrightarrow{\mathtt{in}_i\,x_i \mapsto \cdot_i}\end{array}}$$

$$\frac{x \notin \varphi \quad \iota_X \in \Gamma}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : X\ \vec{C}; T; \varphi) \Longrightarrow_B & \{q[c_i\ x_i] \searrow \Delta; \Gamma, \overrightarrow{x_i{:}\mathcal{F}_i X\ \vec{C}}; T; \varphi\}_i \\ \rightsquigarrow & \mathtt{case}\,\iota_X \star \vec{C}\ x\,\mathtt{of}\,\overrightarrow{\mathtt{in}_i\,x_i \mapsto \cdot_i}\end{array}}$$

The rules for this judgment are continued on the next page.

Figure 3.8: Translation of copatterns using refinements

Pattern Refinement (continued)

$$\frac{\varphi(x) = x \quad \varphi' = [(x_1, x_2)/x]\varphi, x_1{:=}x_1, x_2{:=}x_2}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : T_1{\times}T_2; T; \varphi) \Longrightarrow_B & \{q[(x_1, x_2)] \searrow \Delta; \Gamma, x : T_1{\times}T_2, x_1{:}T_1, x_2{:}T_2; T; \varphi'\} \\ \rightsquigarrow & \mathtt{split}\, x\, \mathtt{as}\, (x_1, x_2)\, \mathtt{in}\, \cdot\end{array}}$$

$$\frac{\varphi(x) = x \quad \varphi' = [(\mathtt{pack}\, \langle u, x'\rangle)/x]\varphi, x'{:=}x'}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : \Sigma u{:}U.T'; T; \varphi) \Longrightarrow_B & \{q[\mathtt{pack}\, \langle u, x'\rangle] \searrow \Delta, u{:}U; \Gamma, x : \Sigma u{:}U.T', x'{:}T'; T; \varphi'\} \\ \rightsquigarrow & \mathtt{unpack}\, x\, \mathtt{as}\, \langle u, x'\rangle\, \mathtt{in}\, \cdot\end{array}}$$

$$\frac{\varphi(x) = x \quad \varphi_i = [(c_i\ x_i)/x]\varphi, x_i{:=}x_i}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : \mu\mathcal{F}\ \vec{C}; T; \varphi) \Longrightarrow_B & \{q[c_i\ x_i] \searrow \Delta; \Gamma, x : \mu\mathcal{F}\ \vec{C}, x_i{:}\mathcal{F}_i(\mu\mathcal{F})\ \vec{C}; T; \varphi_i\}_i \\ \rightsquigarrow & \mathtt{case}\, \mathtt{out}_\mu\, x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto \cdot_i}\end{array}}$$

$$\frac{\varphi(x) = x \quad \varphi_i = [(c_i\ x_i)/x]\varphi, x_i{:=}x_i \quad \iota_X \in \Gamma}{\begin{array}{rl}(q[x] \searrow \Delta; \Gamma, x : X\ \vec{C}; T; \varphi) \Longrightarrow_B & \{q[c_i\ x_i] \searrow \Delta; \Gamma, x : X\ \vec{C}, x_i{:}\mathcal{F}_i X\ \vec{C}; T; \varphi_i\}_i \\ \rightsquigarrow & \mathtt{case}\, \iota_X \star \vec{C}\ x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto \cdot_i}\end{array}}$$

Figure 3.9: Translation of copatterns using refinements (continued)

Thus, if we aborted on all contradictory branches, we would slowly replace branches with aborts during evaluation. This would make relating terms with evaluation more difficult.

In all equality cases, we purge the mapping φ of constant terms after substituting $\wp$ for x in it. This is needed to prevent constant terms from being mapped to variables as we cannot then uniquely determine which variable to use.

The other two categories separate the rules between the ones refining a recursion variable that could be used as argument for recursive calls and refinement of other variables. Prior to being refined, recursion variables are added to φ and are mapped to themselves. Mappings are changed during refinement so if a variable being refined was mapped in φ, it can only be mapped to itself prior to it. If we refine x into p, then we simply substitute p for x in φ and add to φ new associations $x'{:=}x'$ for each variable in p. This methodology is common to every pattern refinement on a recursion variable. We note that recursion is only done on terms of type $\mu\mathcal{F}\ \vec{C}$, but we also add to φ associations for variables of incompatible types.

This is an overhead we accept to keep track of possible new recursion variables without using an additional mechanism.

Let us now have a look at the term we build for specific refinements. The term remains the same whether it is a recursion variable or not. If the variable is of product type, or of existential type, we simply split, or unpack, it accordingly. When dealing with variables of recursive types $\mu\mathcal{F}\ \vec{C}$, we proceed by case analysis on the sum structure $\mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$. We do need to unfold first the variable using $\mathtt{out}_\mu\, x$ in order to expose the sum structure. We could alternatively have a variable x of type $X\ \vec{C}$ if the variable is the result of refinements of the pattern over which we do the recursion. In this case, we also perform a case analysis, but we unfold $X\ \vec{C}$ into $\mathcal{F}X\ \vec{C}$ using $\iota_X \star \vec{C}\ x$ instead of $\mathtt{out}_\mu\, x$. Operationally, they are equivalent as the former will be replaced by the latter during evaluation.

Type-Directed Translation of Terms

Let us move on to the translation of terms. We split it between translation of terms and spines. Heads are handled by the judgment on terms. Let us start with the translation of spines. It is represented with the judgment $\Delta; \Xi; \Gamma; T \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T'$. This judgment states that the spine E translates to $\hat{E}$ and also that $\hat{E}$ is a well-typed spine satisfying $\Delta; \Xi; \Gamma; T \vdash \hat{E} \searrow T'$. Its rules appear in Figure 3.10. For the most part, the translation of spines simply propagates pointwise the translation of terms. It also splits observations $.d_i$ as $.\mathtt{out}_\nu\, .\mathtt{out}_i$. The mapping φ is not directly used by the translation of spines and simply carried around.

Let us move on to the translation of terms (Figure 3.11). The judgment is $\Delta; \Xi; \Gamma \vdash t \rightsquigarrow_\varphi \hat{t} : T$ and means that the term t translates to a term $\hat{t}$ and the latter is well-typed with the judgment $\Delta; \Xi; \Gamma \vdash \hat{t} : T$. Constructors, unit, $\wp$, and (dependent) pairs all have an empty spine. Unit and $\wp$ are translated to themselves, while (dependent) pairs are simply translated pointwise into pairs. Constructors are expanded from $c_i\ t$ to $\mathtt{in}_\mu\, (\mathtt{in}_i\, \hat{t})$ where $\hat{t}$ is the translation of t.

Function bodies are translated through the judgment $\Delta; \Xi; \Gamma \vdash_f \vec{b} \rightsquigarrow_\varphi \hat{t} : S$. We note that functions applied to their spines might appear at positions where a corecursive type

$\boxed{\Delta; \Xi; \Gamma; T \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T'}$ Spine E translates to $\hat{E}$ using contexts $\Delta; \Xi; \Gamma$ and types T and T', yielding $\Delta; \Xi; \Gamma; T \vdash \hat{E} \searrow T'$.

$$\frac{}{\Delta; \Xi; \Gamma; T \vdash \cdot \rightsquigarrow_\varphi \cdot \searrow T} \qquad \frac{\Delta; \Xi; \Gamma \vdash t \rightsquigarrow_\varphi \hat{t} : S \quad \Delta; \Xi; \Gamma; T \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T'}{\Delta; \Xi; \Gamma; S \to T \vdash t\ E \rightsquigarrow_\varphi \hat{t}\ \hat{E} \searrow T'}$$

$$\frac{\Delta; \Xi; \Gamma; [C/u]T \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T'}{\Delta; \Xi; \Gamma; \Pi u{:}U.T \vdash C\ E \rightsquigarrow_\varphi C\ \hat{E} \searrow T'} \qquad \frac{\Delta; \Xi; \Gamma; \mathcal{F}_i(\nu\mathcal{F})\ \vec{C} \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T}{\Delta; \Xi; \Gamma; \nu\mathcal{F}\ \vec{C} \vdash .d_i\ E \rightsquigarrow_\varphi \mathtt{out}_\nu\ \mathtt{out}_i\ \hat{E} \searrow T}$$

Figure 3.10: Translation of spines

is expected. That is, the expected type is $X\ \vec{C}$, or the type depends on some corecursive variable X. Since corecursive calls cannot be nested, the inner function cannot be anything else but a constant function that does not depend on corecursive types. As such, we safely eliminate it with $\mathsf{lift}_\Xi(T) : [\eta]T \to T$ which is the lifting of the corecursive operator ρ to arbitrary types. Here η is the canonical instantiation for variables in Ξ.

The definition of $\mathsf{lift}_\Xi(T)$ appears in Figure 3.12. This lifting builds a deep copy of the input by recursively using ρ's. If we deal with a base type such as 1 or $C_1 = C_2$, we simply have the identity function $\lambda x.x$. In most other cases, we simply deconstruct the term, then rebuild it up. For example, in case of pairs, we split on the input and remake a pair where each side is applied $\mathsf{lift}_\Xi(T_i)$. In the record case, we build a lazy tuple where each component is the i-th projection of the input under $\mathsf{lift}_\Xi(R_i)$. We note that to maintain well-foundedness of recursive and corecursive cases, we introduce a new type variable X for the recursive occurrence $\mathsf{lift}_\Xi(\mathcal{F}\ X\ \vec{C})$. This is where keeping around the context of variables being replaced makes sense. When we come to a type variable case, that is $\mathsf{lift}_\Xi(X\ \vec{C})$, we only replace it by $\rho_X \star \vec{C}$ when $X \in \Xi$. Otherwise, we simply use an identity function. Unless we need to clarify what the context Ξ for $\mathsf{lift}_\Xi(T)$ is, we will often simply write $\mathsf{lift}(T)$. We recall that the recursive ρ_X have type $\Pi\overrightarrow{u{:}U}.X\ \vec{u} \to \mu\mathcal{F}\ \vec{u}$ while the corecursive ρ_X have type $\Pi\overrightarrow{u{:}U}.\nu\mathcal{F}\ \vec{u} \to X\ \vec{u}$. We can thus derive the following general type for $\mathsf{lift}_\Xi(T)$:

$\boxed{\Delta;\Xi;\Gamma \vdash t \rightsquigarrow \hat{t} : T}$ Source term t translates to target term $\hat{t}$ using contexts $\Delta;\Xi;\Gamma$ and type T, yielding typing $\Delta;\Xi;\Gamma \vdash \hat{t} : T$

$$\frac{\Delta;\Xi;\Gamma \vdash t \rightsquigarrow_\varphi \hat{t} : \mathcal{F}_i(\mu\mathcal{F})}{\Delta;\Xi;\Gamma \vdash c_i\ t \rightsquigarrow_\varphi \mathtt{in}_\mu\,(\mathtt{in}_i\,\hat{t}) : \mu\mathcal{F}} \qquad \frac{\Delta;\Xi;\Gamma \vdash t_1 \rightsquigarrow_\varphi \hat{t}_1 : T_1 \quad \Delta;\Xi;\Gamma \vdash t_2 \rightsquigarrow_\varphi \hat{t}_2 : T_2}{\Delta;\Xi;\Gamma \vdash (t_1,t_2) \rightsquigarrow_\varphi (\hat{t}_1,\hat{t}_2) : T_1 \times T_2}$$

$$\frac{}{\Delta;\Xi;\Gamma \vdash \wp \rightsquigarrow_\varphi \wp : C_1 = C_2} \qquad \frac{\Delta \vdash C : U \quad \Delta;\Xi;\Gamma \vdash t \rightsquigarrow_\varphi \hat{t} : [C/u]T}{\Delta;\Xi;\Gamma \vdash \mathtt{pack}\,\langle C,t\rangle \rightsquigarrow_\varphi \mathtt{pack}\,\langle C,\hat{t}\rangle : \Sigma u{:}U.T}$$

$$\frac{}{\Delta;\Xi;\Gamma \vdash () \rightsquigarrow_\varphi () : 1} \qquad \frac{\Delta;\Xi;\Gamma \vdash_f \vec{b} \rightsquigarrow_\varphi \hat{t} : S \quad \Delta;\Xi;\Gamma;S \vdash E \rightsquigarrow_\varphi \hat{E} : [\eta]T}{\Delta;\Xi;\Gamma \vdash \mathtt{fun}\ f.\vec{b} \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (\hat{t} \star \hat{E}) : T}$$

$$\frac{\Gamma(f_X) = \Pi\overrightarrow{u{:}U}.S \to X\ \vec{C}' \quad \Delta \vdash C_i : U_i \quad \Delta;\Xi;\Gamma \vdash t \rightsquigarrow_\varphi \hat{t} : [\overrightarrow{C/u}]S}{\Delta;\Xi;\Gamma \vdash f_X \star \vec{C}\ t \rightsquigarrow_\varphi f_X \star \vec{C}\ \hat{t} : X\ [\overrightarrow{C/u}]\vec{C}'}$$

$$\frac{\begin{array}{c}\Gamma(f_X) = \Pi\overrightarrow{u{:}U}.X\ \vec{C}' \to S \quad \varphi(h) = x \quad \Gamma(x) = S' \quad \Delta \vdash C_i : U_i \\ \Delta;\Xi;\Gamma;S' \vdash E' \rightsquigarrow_\varphi \hat{E}' \searrow X\ \vec{C} \quad \Delta;\Xi;\Gamma;[\overrightarrow{C/u}]S \vdash E \rightsquigarrow_\varphi \hat{E} \searrow [\eta]T\end{array}}{\Delta;\Xi;\Gamma \vdash f_X \star \vec{C}\ (h\ E')\ E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (f_X \star \vec{C}\ (x \star \hat{E}')\ \hat{E}) : T}$$

$$\frac{\Gamma(x) = S \quad \Delta;\Xi;\Gamma;[\eta]S \vdash E \rightsquigarrow_\varphi \hat{E} : [\eta]T}{\Delta;\Xi;\Gamma \vdash x \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (\mathsf{lift}_\Xi(S) \star x\ \hat{E}) : T}$$

Figure 3.11: Translation of terms

Lemma 3.15. *We recall that η_Ξ is the set of canonical instantiations of type variables in Ξ. Suppose T does not contain free variables with respect to Ξ. Then, $\mathsf{lift}_\Xi(T)$ has type $[\eta_{\Xi_c}]T \to [\eta_{\Xi_i}]T$ where Ξ_c and Ξ_i partition Ξ into contexts containing the corecursive and the recursive variables, respectively.*

Proof. By induction on the type T. □

Our translation is very aggressive in applying those lift operators to functions and variables. In most cases, those will simply be identity functions that could very well be omit-

We lift ρ_X by induction on the type S.

$$
\begin{array}{lll}
\mathsf{lift}_\Xi(X\,\vec{C}) & = \begin{cases} \rho_X \star \vec{C} & \text{if } X \in \Xi \\ \lambda x.x & \text{otherwise} \end{cases} & \mathsf{lift}_\Xi(\Pi u{:}U.T) = \lambda f.\Lambda u.\mathsf{lift}_\Xi(T) \star (f \star u) \\
\mathsf{lift}_\Xi(1) & = \lambda x.x & \mathsf{lift}_\Xi(T_1 \to T_2) = \lambda f.\lambda x.\mathsf{lift}_\Xi(T_2) \star (f \star x) \\
\mathsf{lift}_\Xi(C_1 = C_2) & = \lambda x.x & \mathsf{lift}_\Xi(\mu\mathcal{F}\,\vec{C}) = \lambda x.\mathtt{in}_\mu\,(\mathsf{lift}_\Xi(\mathcal{F}\,X\,\vec{C}) \star \mathtt{out}_\mu\,x) \\
 & & \mathsf{lift}_\Xi(\nu\mathcal{F}\,\vec{C}) = \lambda x.\mathtt{in}_\nu\,(\mathsf{lift}_\Xi(\mathcal{F}\,X\,\vec{C}) \star (x \star .\mathtt{out}_\nu\,))
\end{array}
$$

$$\mathsf{lift}_\Xi(T_1 \times T_2) = \lambda x.\mathtt{split}\,x\,\mathtt{as}\,(x_1,x_2)\,\mathtt{in}\,(\mathsf{lift}_\Xi(T_1) \star x_1, \mathsf{lift}_\Xi(T_2) \star x_2)$$

$$\mathsf{lift}_\Xi(D) = \lambda x.\mathtt{case}\,x\,\mathtt{of}\,\overrightarrow{\mathtt{in}_i\,x_i \mapsto \mathtt{in}_i\,(\mathsf{lift}_\Xi(D_i) \star x_i)}$$

$$\mathsf{lift}_\Xi(R) = \lambda x.(\mathsf{lift}_\Xi(R_1)\ (x.\mathtt{out}_1\,); \ldots; \mathsf{lift}_\Xi(R_n) \star (x \star .\mathtt{out}_n\,))$$

$$\mathsf{lift}_\Xi(\Sigma u{:}U.T) = \lambda x.\mathtt{unpack}\,x\,\mathtt{as}\,\langle u, x'\rangle\,\mathtt{in}\,\mathtt{pack}\,\langle u, \mathsf{lift}_\Xi(T) \star x'\rangle$$

Figure 3.12: Lifting ρ to arbritary types

ted. We chose this approach to simplify the presentation of the translation and thus of the metatheory behind it at the expense of the actual translated code. We justify this design choice as our goal is to justify our function criteria rather than to actually compile code into a core. In practice, one could easily optimize this process by limiting lift operators $\mathsf{lift}_\Xi(T)$ to instances of T for which it has a free variable from Ξ. For the purpose of readability, we will omit such lift operators when showing translated code.

The last set of cases operate on terms that have a variable as their head. There are 3 cases for it.

1. The variable could be the corecursive call. In this case, the variable f_X will have a specific signature $\Pi\overrightarrow{u{:}U}.S \to X\ \vec{C}$ and it suffices to translate the seed t that f_X is applied to.

2. It could also be a recursive call. In this case, it is applied to the term we recurse over and is expected to have type $X\ \vec{C}$ in the target language. We recall that there is a challenge to this part of the translation due to deep pattern matching in the source language, exhibited through the following example:

```
fun f : Nat → Nat
| zero ⇒ zero
| suc zero ⇒ zero
| suc (suc n) ⇒ f ⋆ suc n
```

As we mentioned before, we cannot simply translate the right-hand side of the last branch into `f ⋆ in_μ (in_2 n)` because in_μ takes a term of time $1 + \mu\mathcal{F}$ and turns it into $\mu\mathcal{F}$ while in_2 `n` has type $1 + X$, and `f` expects a term of type X. Instead, we use φ to look up the term `suc n` to identify the proper intermediate variable. The end result would look like this:

```
rec f, ι, ρ, x.case x of
    | in_1 y ⇒ in_μ (in_1 ())
    | in_2 y ⇒ case ι ⋆ y of
          | in_1 z ⇒ in_μ (in_1 ())
          | in_2 z ⇒ f ⋆ y
```

Going back to the rule for translation, we only actually care about the head of the term to which the function f is applied to; so only the head is looked up in φ. This is because the term we recurse over can be applied to an arbitrary spine without breaking inductive guardedness. In addition, the result of the recursive call could give rise to a term at a position in the program that expected a corecursive call. Since we make a choice for a function to be handled either recursively or corecursively but not both, this recursive call will not be depending on corecursive variables that could be expected at that position. Hence, we use $\mathsf{lift}_\Xi(T) : T \to [\eta]T$ to align the types.

3. If the variable x is neither, then it could be a recursion variable whose type depends on type variables X that is not used in a recursive call. In addition, it could be a constant that is used in a place where a corecursive call is expect. For this reason, we use two different liftings of ρ's. The first one, $\mathsf{lift}_\Xi(S) : S \to [\eta]S$, allows us to mediate between the type it has in the context and the type $\hat{E}$ is expected to be applied to. The second one, $\mathsf{lift}_\Xi(T) : [\eta]T \to T$, allows us to mediate between the type the expression has and the expected type at the place the variable is used.

$\boxed{\Delta; \Xi; \Gamma \vdash_f \vec{b} \rightsquigarrow_\varphi \hat{t} : T}$ Function f with body $\vec{b}$ translates to term $\hat{t}$ at type T under environments $\Delta; \Xi; \Gamma$ and recursion context φ.

$$\frac{f \text{ is non rec.} \quad \Xi \vdash (\cdot \searrow \Delta; (\Gamma, f{:}T); T; \varphi) \Longrightarrow^*_{\overrightarrow{\{q \mapsto t\}}} \{q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j\}_j \rightsquigarrow \hat{t}}{\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \hat{t} : T}$$

$$\frac{\begin{array}{c} f \text{ is struct. rec.} \quad \bigcup_i Q_i = \{q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j\}_j \quad B_i = \{q \mapsto t \mid q \in Q_i\} \\ \Gamma_X = f_X : \Pi \overrightarrow{u{:}U}.X\,\vec{C} \to T, \iota_X : \Pi \overrightarrow{u{:}U}.X\,\vec{u} \to \mathcal{F}\,X\,\vec{u}, \rho_X : \Pi \vec{u}.X\,\vec{u} \to (\mu\mathcal{F})\,\vec{u} \\ \text{for all } i, \quad \Xi, X \vdash ((\vec{u}\ (c_i\ x_i)) \searrow \Delta, \overrightarrow{u{:}U}; (\Gamma, \Gamma_X, x_i{:}\mathcal{F}_i X\,\vec{C}); T; (\varphi, x_i{:=}x_i)) \Longrightarrow^*_{B_i} Q_i \rightsquigarrow \hat{t}_i \end{array}}{\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \mathtt{rec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, \mathtt{case}\, x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto \hat{t}_i} : \Pi \overrightarrow{u{:}U}.\mu\mathcal{F}\,\vec{C} \to T}$$

$$\frac{\begin{array}{c} f \text{ is struct. corec.} \quad \bigcup_i Q_i = \{q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j\}_j \quad B_i = \{q \mapsto t \mid q \in Q_i\} \\ \Gamma_X = f_X : \Pi \overrightarrow{u{:}U}.T \to X\,\vec{C}, \iota_X : \Pi \overrightarrow{u{:}U}.\mathcal{F}\,X\,\vec{u} \to X\,\vec{u}, \rho_X : \Pi \vec{u}.(\nu\mathcal{F})\,\vec{u} \to X\,\vec{u} \\ \text{for all } i, \quad \Xi, X \vdash ((\vec{u}\ x\ .d_i) \searrow \Delta, \overrightarrow{u{:}U}; (\Gamma, \Gamma_X, x{:}T); \mathcal{F}_i X\,\vec{C}; \varphi) \Longrightarrow^*_{B_i} Q_i \rightsquigarrow \hat{t}_i \end{array}}{\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \mathtt{corec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, (\hat{t}_1; \ldots; \hat{t}_n) : \Pi \overrightarrow{u{:}U}.T \to \nu\mathcal{F}\,\vec{C}}$$

$\boxed{\Xi \vdash q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow^*_B Q \rightsquigarrow t}$ Refinement of copattern $q \searrow \Delta; \Gamma; T$ under mapping φ into the copattern set Q generates the term t such that $\Delta; \Xi; \Gamma \vdash t : T$.

$$\frac{\begin{array}{c} (q \searrow \Delta; \Gamma; T; \varphi) \Longrightarrow^*_B Q \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n] \\ \text{for all } (q_i \searrow \Delta_i; \Gamma_i; T_i; \varphi_i) \in Q, \quad \Delta_i; \Xi; \Gamma_i \vdash t_i \rightsquigarrow_{\varphi_i} \hat{t}_i : T_i \end{array}}{\Xi \vdash (q \searrow \Delta; \Gamma; T; \varphi) \Longrightarrow^*_B Q \rightsquigarrow t[\hat{t}_1 \mid \cdots \mid \hat{t}_n]}$$

Figure 3.13: Function translation

The last part of the translation we need to address is the translation of function bodies. It is represented by the judgment $\Delta; \Xi; \Gamma \vdash_f \vec{b} \rightsquigarrow_\varphi \hat{t} : T$ and the rules are shown in Figure 3.13. We have three different cases for functions depending on whether the function is non-recursive, structurally recursive, or structurally corecursive. If the function is non-recursive, we simply appeal to the translation of copatterns. The judgment we appeal to for

translation of copatterns is not the one we discussed above, but $\Xi \vdash q \searrow \Delta; \Gamma; \varphi \Longrightarrow^*_{\vec{b}} Q \rightsquigarrow \hat{t}$. This judgment has a single rule shown in Figure 3.13. This rule simply generates a term with n holes from the copattern q, and fills out all the holes with the translations of the right-hand sides.

If the function is structurally recursive, we generate a recursor and perform a case analysis on the input x, as the function criteria guarantee that all copatterns match on this argument. Then, we partition the final copattern set into a series of sets Q_i based on the constructor for x. In addition, we partition the branches $\vec{b}$ into sets B_i so that the branches in B_i are the ones whose copatterns are in Q_i. We appeal to the translation of copatterns starting with the copattern $\vec{u}\ (c_i\ x_i)$ to the copattern set Q_i, since we already exposed the constructor using our case analysis. Each resulting term is then placed in the corresponding branch of the case statement. The translation of copattern is called with the extended index-context $\Delta, \overrightarrow{u : U}$ of variable bound by the recursor, and the context Γ to which we add Γ_X which contains typing for f_X, ι_X, and ρ_X, and x_i of type $\mathcal{F}_i X\ \vec{u}$. Moreover, we add to φ, the association $x_i := x_i$ as each x_i is a valid recursive call.

For structurally corecursive functions, we generate a corecursor, together with a lazy record corresponding to each branch of the observations of the coinductive term we are building. We perform the same partition of Q_i's and for each of them appeal to the translation of copatterns starting at the copattern $\vec{u}\ x\ .d_i$. We add the corecursor operators to the context. This time around, the expected type is $\mathcal{F}_i\ X\ \vec{u}$, and we do not add an association to φ as it is only used for structurally recursive functions.

3.5 Function Criteria Are Sufficient for the Translation

We first prove that we can derive a translation for a term as long as this term satisfies the function criteria. This establishes that our criteria are suitable to assert normalisation, as long as the translation is normalisation preserving. This proof has a few pieces. The first one is a set of invariants the translation preserves. Those invariants will allow us to restrict

the cases of our proof.

Definition 3.7 (Translation invariants)**.** We say the pair target language pair $\Gamma \vdash T$ of a context Γ and a type T satisfies the *translation invariants* for Ξ and φ if the following hold:

1. For all $X \in \Xi$, if X is a recursive type variable, then X does not occur free in T.
2. For all $X \in \Xi$, if X is a corecursive type variable, then the only free occurrence of X in Γ is $f_X : \Pi u{:}U.T \to X\ \vec{C}$.
3. For all $X \in \Xi$, if X is a recursive type variable and p is the pattern guard for X and $h \star E \prec_{\text{ind}} p$ and $\Delta; \Gamma \vdash h : T'$, then $h \in \varphi$ and $\Delta; \Xi; \Gamma \vdash \varphi(h) : T'$.

Moreover, we say that the judgments $\Delta; [\eta]\Gamma \vdash t : [\eta]T$ and $\Delta; [\eta]\Gamma; [\eta]S \vdash E \searrow [\eta]T$, where η is the standard instantiation for type variables in Ξ, satisfy the translation invariants if the pairs $\Gamma \vdash S$ and $\Gamma \vdash T$ satisfy it.

The translation invariants is thus defined on contexts Ξ and Γ, the type T, and the mapping φ. The judgment $\Delta; [\eta]\Gamma \vdash t : [\eta]T$ thus is valid for the copattern language. The two first invariants restrict when we can see type-level variables in T and Γ. In addition, the third invariant guarantees that all valid inductively guarded subterms of the pattern guard are part of the mapping φ. All of those invariants are justified by the way our translation works and we shall prove they are preserved when invoking induction hypotheses.

Commuting Refinements

As we alluded to in Section 3.4, our function criteria ensure we can derive specifically shaped coverage derivations for structurally (co)recursive functions. Those specific shapes allow us to generate the final copattern sets from the initially split copatterns. The following lemmas show we can commute certain refinements in order to derive the specific derivations we need. The first two lemmas indicate that if we have a split on a copattern q, and a (possibly parallel) split on x, then we can split on x first, then split on q, and yield the same end result. We show it holds both for single step refinements and multi step refinements.

Lemma 3.16 (Single step commuting of constructor split)**.** *If* $\vec{u}\ x\ q \Longrightarrow \{\vec{u}\ x\ q_i\}$ *and if for all* i *we have* $\vec{u}\ x\ q_i \Longrightarrow \{\vec{u}\ (c_j\ x_j)\ q_i\}_j$*, then* $\vec{u}\ x\ q \Longrightarrow \{\vec{u}\ (c_j\ x_j)\ q\}_j$ *and for all* j*, we have* $\vec{u}\ (c_j\ x_j)\ q \Longrightarrow \{\vec{u}\ (c_j\ x_j)\ q_i\}_i$*.*

Proof. By case analysis on the judgment $\vec{u}\ x\ q \Longrightarrow \{\vec{u}\ x\ q'_i\}$. □

Lemma 3.17 (Multi step commuting of constructor split)**.** *If* $\vec{u}\ x\ q \Longrightarrow^* \{\vec{u}\ (c_j\ p_i)\ q_i\}_i$*, then* $\vec{u}\ x\ q \Longrightarrow \{\vec{u}\ (c_j\ x_j)\ q\}_j$ *and for all* j *we have* $\vec{u}\ (c_j\ x_j)\ q \Longrightarrow^* Q_j$ *where* $\bigcup_j Q_j = \{\vec{u}\ (c_j\ p_i)\ q_i\}_i$*.*

Proof. By induction on the derivation of $\vec{u}\ x\ q \Longrightarrow^* \{\vec{u}\ (c_j\ p_i)\ q_i\}_i$.

Case: $$\frac{\vec{u}\ x\ q \Longrightarrow \{\vec{u}\ (c_j\ x_j)\ q\}_j \quad \text{for all } j \quad \overline{\vec{u}\ (c_j\ x_j)\ q \Longrightarrow^* \{\vec{u}\ (c_j\ x_j)\ q\}}}{\vec{u}\ x\ q \Longrightarrow^* \bigcup_j \{\vec{u}\ (c_j\ x_j)\ q\}}$$

The conclusion holds directly.

Case: $$\frac{\vec{u}\ x\ q \Longrightarrow Q' \quad \text{for all } q_i \in Q'\ q_i \Longrightarrow^* Q'_i}{\vec{u}\ x\ q \Longrightarrow^* \bigcup_i Q'_i}$$

If $Q' = \{\vec{u}\ (c_j\ x_j)\ q\}_j$, then we are done. Otherwise, by induction hypothesis:
$\vec{u}\ x\ q_i \Longrightarrow \{\vec{u}\ (c_j\ x_j)\ q_i\}_j$ for all $\vec{u}\ x\ q_i \in Q'$.
$\vec{u}\ (c_j\ x_j)\ q_i \Longrightarrow^* Q_{ij}$ where $Q = \bigcup_{ij} Q_{ij}$ for all j.
By Lemma 3.16, we have the desired result. □

The second set of commuting refinements lemmas show that if we split a pattern p into a set of patterns p_i, and then introduce (possibly in parallel) observations $.d_j$, then we can also introduce those observations first and split on p afterwards.

Lemma 3.18 (Single step commuting of observation introduction)**.** *If* $\vec{u}\ p\ \cdot \Longrightarrow \{\vec{u}\ p_i\ \cdot\}_i$ *and for all* i *we have* $\vec{u}\ p_i\ \cdot \Longrightarrow \{\vec{u}\ p_i\ .d_j\ \cdot\}_j$*, then* $\vec{u}\ p\ \cdot \Longrightarrow \{\vec{u}\ p\ .d_j\ \cdot\}_j$ *and for all* j*, we have* $\vec{u}\ p\ .d_j\ \cdot \Longrightarrow \{\vec{u}\ p_i\ .d_j\ \cdot\}_i$*.*

Proof. By case analysis on the judgment $\vec{u}\ p\ \cdot \Longrightarrow \{\vec{u}\ p_i\ \cdot\}_i$. □

Lemma 3.19 (Multi step commuting of observation introduction)**.** *If* $\vec{u}\ p \cdot \Longrightarrow^* \{\vec{u}\ p_i\ .d_i\ q_i'\}_i$, *then* $\vec{u}\ p \cdot \Longrightarrow \{\vec{u}\ p\ .d_j \cdot\}_j$ *and for all* j *we have* $\vec{u}\ p\ .d_j \cdot \Longrightarrow^* Q_j$ *where* $\bigcup_j Q_j = \{\vec{u}\ p_i\ .d_i\ q_i'\}_i$.

Proof. By induction on the derivation of $\vec{u}\ x\ q \Longrightarrow^* \{\vec{u}\ p_i\ .d_i\ q_i'\}_i$.

$$\textbf{Case:}\quad \frac{\vec{u}\ p \cdot \Longrightarrow \{\vec{u}\ p\ .d_j\ \cdot\}_j \quad \text{for all } j \quad \overline{\vec{u}\ p\ .d_j \cdot \Longrightarrow^* \{\vec{u}\ p\ .d_j\ \cdot\}}}{\vec{u}\ p \cdot \Longrightarrow^* \bigcup_j \{\vec{u}\ p\ .d_j\ \cdot\}}$$

The conclusion holds directly.

$$\textbf{Case:}\quad \frac{\vec{u}\ p \cdot \Longrightarrow Q' \quad \text{for all } q_i \in Q'\ q_i \Longrightarrow^* Q_i'}{\vec{u}\ p \cdot \Longrightarrow^* \bigcup_i Q_i'}$$

If $Q' = \{\vec{u}\ p\ .d_j\ \cdot\}_j$, then we are done. Otherwise, the induction hypothesis gives us:
$\vec{u}\ p_i \cdot \Longrightarrow \{\vec{u}\ p_i\ .d_j\ \cdot\}_j$ for all $\vec{u}\ p_i \cdot \in Q'$.
$\vec{u}\ p_{ij}\ .d_j\ q_j \Longrightarrow Q_{ij}$ where $Q = \bigcup_{ij} Q_{ij}$ for all j.
By Lemma 3.18, we have the desired result. □

Coverage Implies Copattern Translation

Now, we show that if we have a coverage derivation and the initial type and context satisfy the translation invariants, then we can build a translation for this copattern set and the resulting context and type also satisfy the translation invariants. We first start by lemmas that allows us to maintain our third invariant preserving completeness of the mapping φ under coverage refinements.

Lemma 3.20. *If* x *is a variable that appears free in* h *and* $h \star E \prec_{ind} h'$, *then* x *appears free in* h'.

Proof. By induction on $h \star E \prec_{\text{ind}} h'$. □

Lemma 3.21. *Suppose for all* h *and* E, *we have that* $h \star E \prec_{ind} h_0[x]$ *implies* $h \in \varphi$. *Then, for all* h' *and* E', *the following hold:*

1. *if* $h' \star E' \prec_{ind} h_0[c\ x']$, *then* $h' \in \varphi'$ *where* $\varphi' = [(c\ x')/x]\varphi, x' := x'$;
2. *if* $h' \star E' \prec_{ind} h_0[(x_1, x_2)]$, *then* $h' \in \varphi'$ *where* $\varphi' = [(x_1, x_2)/x]\varphi, x_1 := x_1, x_2 := x_2$;

3. *if* $h' \star E' \prec_{ind} h_0[\mathtt{pack}\,\langle C, x'\rangle]$, *then* $h' \in \varphi'$ *where* $\varphi' = [(\mathtt{pack}\,\langle C, x'\rangle)/x]\varphi, x' := x'$.

Proof. All statement are proved by induction on $h' \star E' \prec_{\text{ind}} h_0[-]$. The transitivity case makes use of Lemma 3.20. □

We note that coverage derivations allow us to ensure that copattern translation exists. We first prove it for the single step refinement judgment, then the multi step refinement judgment. We recall that Γ_X was defined as the context containing f_X, ι_X, ρ_X. We define Γ_Ξ recursively on Ξ. If $\Xi = \cdot$, then $\Gamma_\cdot = \cdot$. If $\Xi = \Xi', X$, then $\Gamma_{\Xi',X} = \Gamma_{\Xi'}, \Gamma_X$.

Lemma 3.22 (Single step copattern translation is complete with respect to coverage)**.** *If* $q \searrow \Delta; [\eta]\Gamma; [\eta]T \Longrightarrow \{q_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}_i$ *and* $\Gamma \vdash T$ *and* φ *satisfy the translation invariants for* Ξ*, then*

$$q \searrow \Delta; \Gamma, \Gamma_\Xi; T; \varphi \Longrightarrow \{q_i \searrow \Delta_i; \Gamma_i', \Gamma_\Xi; T_i; \varphi_i\}_i \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n]$$

for some Γ_i' *that extends* Γ_i *and* $\Gamma_i'; T_i; \varphi_i$ *satisfy the translation invariants for* Ξ*.*

Proof. By case analysis on $q \searrow \Delta; [\eta]\Gamma; [\eta]T \Longrightarrow \{q_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}$. We only show the case for introduction of observation and refinements of recursive types. The other cases are similar.

Case: $q \searrow \Delta; [\eta]\Gamma; [\eta]T \Longrightarrow \{q@.d_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}_i$

where $[\eta]T = \nu\mathcal{F}\ \vec{C}$ and $[\eta]T_i = \mathcal{F}_i(\nu\mathcal{F})\ \vec{C}$.

Subcase: $T = \nu\mathcal{F}\ \vec{C}$

Then, $q \searrow \Delta; \Gamma, \Gamma_\Xi; \nu\mathcal{F}\ \vec{C}; \varphi \Longrightarrow \{q@.d_i \searrow \Delta; \Gamma, \Gamma_\Xi; \mathcal{F}_i(\nu\mathcal{F})\ \vec{C}; \varphi\}_i \rightsquigarrow \mathtt{in}_\nu\,(\cdot_1; \ldots; \cdot_n)$
$\Gamma \vdash \mathcal{F}_i(\nu\mathcal{F})\ \vec{C}$ and φ satisfy the translation invariants since $\Gamma \vdash \nu\mathcal{F}\ \vec{C}$ and φ also satisfy them and $\mathcal{F}_i(\nu\mathcal{F})\ \vec{C}$ does not introduce any recursive type variable.

Subcase: $T = X\ \vec{C}$

Then, $q \searrow \Delta; \Gamma, \Gamma_\Xi; X\ \vec{C}; \varphi \Longrightarrow \{q@.d_i \searrow \Delta; \Gamma, \Gamma_\Xi; \mathcal{F}_i X\ \vec{C}; \varphi\}_i \rightsquigarrow \iota_X \star \vec{C}\ (\cdot_1; \ldots; \cdot_n)$

$\Gamma \vdash \mathcal{F}_i X\ \vec{C}$ and φ satisfy the translation invariants since $\Gamma \vdash X\ \vec{C}$ and φ also satisfy them and and $\mathcal{F}_i X\ \vec{C}$ does not introduce any recursive type variable.

Case: $q[x] \searrow \Delta; [\eta]\Gamma; [\eta]T \Longrightarrow \{q[c_i\ x_i] \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}_i$

where $[\eta]\Gamma = \Gamma', x : \mu\mathcal{F}\ \vec{C}$ and $[\eta]\Gamma_i = \Gamma', x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$

Subcase: $\varphi(x) = x$

$\varphi_i = [(c_i\ x_i)/x]\varphi, x_i := x_i$ and we are refining a pattern guard for $X \in \Xi$. Thus, q is of the form $\vec{u}\ (c\ p[x])\ q'$ and $p[x]$ is the pattern guard being refined. By Lemma 3.21, φ_i satisfies the translation invariants for Ξ against the pattern guard $p[c_i\ x_i]$. Depending on whether $x \in \varphi$ and what the type of x is in Γ, we build refinements directly.

Subsubcase: $\Gamma = \Gamma', x : \mu\mathcal{F}\ \vec{C}$

$$q[x] \searrow \Delta; \Gamma', \Gamma_\Xi, x : \mu\mathcal{F}\ \vec{C}; T; \varphi \Longrightarrow \{q[c_i\ x_i] \searrow \Delta; \Gamma', x : \mu\mathcal{F}\ \vec{C}, x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}, \Gamma_\Xi; T; \varphi_i\}_i \rightsquigarrow \mathtt{case}\ \mathtt{out}_\mu\ x\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\ x_i \mapsto \cdot_i}$$

where $\Gamma_i = \Gamma', x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$ and $\Gamma_i' = \Gamma', x : \mu\mathcal{F}\ \vec{C}, x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$.

Subsubcase: $\Gamma = \Gamma', x : X\ \vec{C}$

$$q[x] \searrow \Delta; \Gamma', \Gamma_\Xi, x : X\ \vec{C}; T; \varphi \Longrightarrow \{q[c_i\ x_i] \searrow \Delta; \Gamma', x : \mu\mathcal{F}\ \vec{C}, x_i : \mathcal{F}_i X\ \vec{C}, \Gamma_\Xi; T; \varphi_i\}_i \rightsquigarrow \mathtt{case}\ \iota_X \star \vec{C}\ x\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\ x_i \mapsto \cdot_i}$$

where $\Gamma_i = \Gamma', x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$ and $\Gamma_i' = \Gamma', x : \mu\mathcal{F}\ \vec{C}, x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}$.
Γ_i' satisfies the translation invariants as they don't introduce corecursive type variables to Γ.

Subcase: $x \notin \varphi$

Subsubcase: $\Gamma = \Gamma', x : \mu\mathcal{F}\ \vec{C}$

$$q[x] \searrow \Delta; \Gamma', \Gamma_\Xi, x : \mu\mathcal{F}\ \vec{C}; T; \varphi \Longrightarrow \{q[c_i\ x_i] \searrow \Delta; \Gamma', x_i : \mathcal{F}_i(\mu\mathcal{F})\ \vec{C}, \Gamma_\Xi; T; \varphi_i\}_i \rightsquigarrow \mathtt{case}\ \mathtt{out}_\mu\ x\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\ x_i \mapsto \cdot_i}$$

$\Gamma_i = \Gamma_i'$ and they both satisfy the translation invariants.

Subsubcase: $\Gamma = \Gamma', x : X\ \vec{C}$

$$q[x] \searrow \Delta; \Gamma', \Gamma_\Xi, x : X\ \vec{C}; T; \varphi \Longrightarrow \{q[c_i\ x_i] \searrow \Delta; \Gamma', x_i : \mathcal{F}_i X\ \vec{C}, \Gamma_\Xi; T; \varphi_i\}_i \rightsquigarrow \mathtt{case}\ \iota_X \star \vec{C}\ x\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\ x_i \mapsto \cdot_i}$$

and $\Gamma_i = \Gamma_i'$ and they both satisfy the translation invariants. □

Lemma 3.23 (Multi step copattern translation is complete with respect to coverage)**.** *If* $(q \searrow \Delta; [\eta]\Gamma; [\eta]T) \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}_i$ *and* $\Gamma \vdash_\varphi T$ *satisfy the translation invariants for* Ξ*, then*

$$(q \searrow \Delta; \Gamma, \Gamma_\Xi; T; \varphi) \Longrightarrow^* \{q_i \searrow \Delta_i; \Gamma_i', \Gamma_\Xi; T_i; \varphi_i\}_i \rightsquigarrow t[\cdot_1 \mid \cdots \mid \cdot_n]$$

for some Γ_i' *that extends* Γ_i *and* $\Gamma_i'; T_i; \varphi_i$ *satisfy the translation invariants for* Ξ*.*

Proof. By induction on $q \searrow \Delta; [\eta]\Gamma; [\eta]T \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}$. The inductive case follows from Lemma 3.22. □

Translating Terms using Function Criteria

This leads us to our main theorem which shows we can generate a translation.

Theorem 3.24. *The following hold:*

1. *If* $\Delta; [\eta]\Gamma \vdash t : [\eta]T$ *satisfies the function criteria and the translation invariants, then there exists a* $\hat{t}$ *such that* $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash t \rightsquigarrow_\varphi \hat{t} : T$*.*
2. *If* $\Delta; [\eta]\Gamma; T' \vdash E \searrow [\eta]T$ *satisfies the function criteria and the transition invariants, then there exists an* $\hat{E}$ *such that* $\Delta; \Xi; \Gamma, \Gamma_\Xi; T' \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T$*.*

Proof. By mutual induction on $\Delta; [\eta]\Gamma \vdash t : [\eta]T$ and $\Delta; [\eta]\Gamma; [\eta]T' \vdash E \searrow [\eta]T$. We provide the important cases of Statement 1.

Statement 1

By inversion on $\Delta; [\eta]\Gamma \vdash t : [\eta]T$, we have

$$\frac{\Delta; [\eta]\Gamma \vdash h : T' \quad \Delta; [\eta]\Gamma; T' \vdash E \searrow T''}{\Delta; [\eta]\Gamma \vdash h \star E : [\eta]T}$$

where $[\eta]T = T''$. We proceed by case analysis on the head h.

Case: $\dfrac{\Delta; [\eta]\Gamma \vdash t : \mathcal{F}_c(\mu\mathcal{F})\ \vec{C}}{\Delta; [\eta]\Gamma \vdash c\ t : \mu\mathcal{F}\ \vec{C}}$

$E = \cdot$ and $T' = T''$ by inversion on $\Delta; [\eta]\Gamma; T' \vdash E \searrow T''$.
$T \neq X\ \vec{C}$ for an inductive X by the translation invariants.
Thus $T = T'' = \mu\mathcal{F}\ \vec{C}$
$\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash t \rightsquigarrow_\varphi \hat{t} : \mathcal{F}_c(\mu\mathcal{F})\ \vec{C}$ by induction hypothesis.
$\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash c\ t \rightsquigarrow_\varphi \mathtt{in}_\mu\ (\mathtt{in}_c\ \hat{t}) : \mu\mathcal{F}\ \vec{C}$ by rules of translation.

Case: $\dfrac{[\eta]\Gamma(x) = T'}{\Delta; [\eta]\Gamma \vdash x : T'}$

Let $\Gamma(x) = T_1$ where $[\eta]T_1 = T'$. We proceed by subcase analysis on the variable x.

Subcase: $T_1 = \Pi\overrightarrow{u{:}U}.S \to X\ \vec{C}$ and x is the corecursive call f_X

$E = \vec{C'}\ t'$ where $\Delta \vdash C_i : U_i$ and $\Delta; [\eta]\Gamma \vdash t' : [\eta]S$ by function criteria.
$\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash t' \rightsquigarrow_\varphi \hat{t'} : S$ by induction hypothesis
Thus, $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash x\ \vec{C'}\ t' \rightsquigarrow_\varphi x\ \vec{C'}\ \hat{t'} : X\ \vec{C'}$.

Subcase: $T_1 = \Pi\overrightarrow{u{:}U}.X\ \vec{C} \to S$ and x is the recursive call f_X.

$E = \vec{C'}\ t'\ E'$ where $\Delta; [\eta]\Gamma \vdash t' : [\overrightarrow{C'/u}]([\eta]X\ \vec{C})$
and $\Delta; [\eta]\Gamma; [\overrightarrow{C'/u}]S' \vdash E' \searrow T''$ where $S' = [\eta]S$ by the function criteria.
$t' \prec_{\text{ind}} p$ for the pattern guard p by the function criteria.
$t' = h' \star E''$ and $\Delta; [\eta]\Gamma \vdash h' \star E'' : [\eta]X\ [\overrightarrow{C'/u}]\vec{C}$ and $\Delta; [\eta]\Gamma \vdash h' : S_0$
and $\Delta; [\eta]\Gamma; S_0 \vdash E'' \searrow [\eta]X\ [\overrightarrow{C'/u}]\vec{C}$ by inversion on $\Delta; [\eta]\Gamma \vdash t' : [\overrightarrow{C'/u}]([\eta]X\ \vec{C})$.
$h' \in \varphi$ and $\varphi(h') = x$ where $[\eta]\Gamma(x) = S_0$ by the translation invariants.
Note: S', S_0, and T'' do not depend on variables in Ξ.
Thus, $\Delta; [\eta]\Gamma; [\eta_\Xi][\overrightarrow{C'/u}]S' \vdash E' \searrow [\eta]T''$ and $\Delta; [\eta]\Gamma; [\eta]S_0 \vdash E'' \searrow [\eta](X\ [\overrightarrow{C'/u}]\vec{C})$.
$\Delta; \Xi; \Gamma, \Gamma_\Xi; [\overrightarrow{C'/u}]S \vdash E' \rightsquigarrow_\varphi \hat{E'} \searrow T''$

and $\Delta; \Xi; \Gamma, \Gamma_\Xi; S_0 \vdash E'' \rightsquigarrow_\varphi \hat{E}'' \searrow X\ [\overrightarrow{C'/u}]\vec{C}'$ by induction hypothesis (Statement 2)
$\mathsf{lift}_\Xi(T) : T'' \to T$ by Lemma 3.15, as T only depends on corecursive variables.
Thus, $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash f \star \vec{C}'\ (h'\ E'')\ E' \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (f \star \vec{C}\ (x \star \hat{E}'')\ \hat{E}') : T$

Subcase: x is neither a recursive nor corecursive variable

$\Delta; \Xi; \Gamma, \Gamma_\Xi; T' \vdash E \rightsquigarrow_\varphi \hat{E} \searrow T''$ by induction hypothesis (Statement 2).
$\mathsf{lift}_\Xi(T_1) : T_1 \to [\eta]T_1$ by Lemma 3.15 as T_1 only depends on recursive variables.
Thus, $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash x \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (\mathsf{lift}_\Xi(T_1) \star x\ \hat{E}) : T$.

Case: $$\frac{\text{for each } i\ \Delta; [\eta]\Gamma, f{:}T' \vdash (q_i \mapsto t_i) : T'}{\Delta; [\eta]\Gamma \vdash \mathtt{fun}\ f.\overrightarrow{q \mapsto t} : T'}$$

$\Delta; [\eta]\Gamma, f{:}[\eta]T'; [\eta]T' \vdash q_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i$ and $\Delta_i; [\eta]\Gamma_i \vdash t_i : [\eta]T_i$
by inversion on $\Delta; [\eta]\Gamma, f{:}T' \vdash (q_i \mapsto t_i) : T'$.
We shall obtain translation of copattern judgment by subcase analysis on the structural (co)recursiveness status of f.

Subcase: f is non-recursive

$(\cdot \searrow \Delta; [\eta]\Gamma, f{:}[\eta]T'; [\eta]T') \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta]\Gamma_i; [\eta]T_i\}_i$ by the function criteria.
$(\cdot \searrow \Delta; \Gamma, f{:}T', \Gamma_\Xi; T'; \varphi) \Longrightarrow^*_{\overrightarrow{q \mapsto t}} \{q_i \searrow \Delta_i; \Gamma_i', \Gamma_\Xi; T_i; \varphi_i\}_i \rightsquigarrow \hat{s}[\cdot_1 \mid \cdots \mid \cdot_n]$ by Lemma 3.23.
where Γ_i' extends Γ_i and both Γ_i' and T_i satisfy the translation invariants for Ξ and φ_i.
$\Delta_i; \Xi; \Gamma_i', \Gamma_\Xi \vdash t_i \rightsquigarrow_{\varphi_i} \hat{t}_i : T_i$ by induction hypothesis.
Thus, $\Xi \vdash (\cdot \searrow \Delta; \Gamma, f{:}T', \Gamma_\Xi; T'; \varphi) \Longrightarrow^*_{\overrightarrow{q \mapsto t}} \{q_i \searrow \Delta_i; \Gamma_i', \Gamma_\Xi; T_i; \varphi_i\}_i \rightsquigarrow \hat{s}[\hat{t}_1 \mid \cdots \mid \hat{t}_n]$
and so, $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \hat{s}[\hat{t}_1 \mid \cdots \mid \hat{t}_n] : T'$.
$\Delta; \Xi; \Gamma, \Gamma_\Xi; T' \vdash \hat{E} \searrow T''$ by induction hypothesis (Statement 2).
Thus, $\Delta; \Xi; \Gamma \vdash (\mathtt{fun}\ f.\overrightarrow{q \mapsto t}) \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (\hat{s}[\hat{t}_1 \mid \cdots \mid \hat{t}_n] \star \hat{E}) : T$.

Subcase: f is structurally recursive

$(\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_\Xi]\Gamma, f{:}[\eta_X]T_0', x{:}[\eta_X]X\ \vec{C}; S) \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta_{\Xi,X}]\Gamma_i; T_i\}_i$
and $(\cdot \searrow \Delta; [\eta_\Xi]\Gamma, f{:}T'; T') \Longrightarrow^* \{\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_\Xi]\Gamma, f{:}[\eta_X]T_0', x{:}[\eta_X]X\ \vec{C}; S\}$

and $T' = \Pi\overrightarrow{u{:}U}.\mu\mathcal{F}\,\vec{C} \to S$, where $q_i = \vec{u}\ (c_i\ p_i)\ q_i'$ and $T_0' = \Pi\overrightarrow{u{:}U}.X\,\vec{C} \to S$

by the function criteria.

$(\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}]\Gamma, f{:}[\eta_{\Xi,X}]T_0', x{:}[\eta_{\Xi,X}]X\,\vec{C}; [\eta_{\Xi,X}]S) \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta_{\Xi,X}]\Gamma_i; [\eta_{\Xi,X}]T_i\}_i$

by weakening.

$(\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}](\Gamma, f{:}T_0', x{:}X\,\vec{C}); [\eta_{\Xi,X}]S) \Longrightarrow$
$\quad \{\vec{u}\ (c_j\ x_j) \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}](\Gamma, f{:}T_0', x_j{:}\mathcal{F}_j\,X\,\vec{C}); [\eta_{\Xi,X}]S\}_j$

and for all j, $(\vec{u}\ (c_j\ x_j) \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}](\Gamma, f{:}T_0', x_j{:}\mathcal{F}_j\,X\,\vec{C}); [\eta_{\Xi,X}]S) \Longrightarrow^* Q_j$

where $\bigcup_j Q_j = \{q_i \searrow \Delta_i; [\eta_{\Xi,X}]\Gamma_i; [\eta_{\Xi,X}]T_i\}_i$ by Lemma 3.17.

for each j,

$(\vec{u}\ (c_j\ x_j) \cdot \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, f{:}T_0', x_j{:}\mathcal{F}_j\,X\,\vec{C}, \Gamma_{\Xi,X}; S; \varphi)$
$\quad \Longrightarrow^*_{\overrightarrow{q \mapsto t}} \{q_i \searrow \Delta_i; \Gamma_i, \Gamma_{\Xi,X}; T_i; \varphi_i\}_i \rightsquigarrow \hat{s}_j[\cdot_1 \mid \cdots \mid \cdot_n]$ where Γ_i' extends Γ_i

and $\Gamma_i' \vdash T_i$ satisfy the translation invariants for Ξ, X and φ_i by Lemma 3.23.

$\Delta_i; \Xi; \Gamma_i, \Gamma_{\Xi,X} \vdash t_i \rightsquigarrow_{\varphi_i} \hat{t}_i : T_i$ by induction hypothesis.

Thus, $\Xi, X \vdash (\vec{u}\ (c_j\ x_j) \cdot \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, f{:}T_0', x_j{:}\mathcal{F}_j\,X\,\vec{C}, \Gamma_{\Xi,X}; S; \varphi)$
$\quad \Longrightarrow^*_{\overrightarrow{q \mapsto t}} \{q_i \searrow \Delta_i; \Gamma_i, \Gamma_{\Xi,X}; T_i; \varphi_i\}_i \rightsquigarrow \hat{t}[\cdot_1 \mid \cdots \mid \cdot_n]$

and so, $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, \mathtt{case}\, x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_j\, x_j \mapsto \hat{s}_j[\hat{t}_1 \mid \cdots \mid \hat{t}_n]} : T'$.

$\Delta; \Xi; \Gamma, \Gamma_\Xi; T' \vdash \hat{E} \searrow T''$ by induction hypothesis (Statement 2).

Thus, $\Delta; \Xi; \Gamma \vdash (\mathtt{fun}\ f.\overrightarrow{q \mapsto t}) \star E \rightsquigarrow_\varphi$
$\quad \mathsf{lift}_\Xi(T) \star ((\mathtt{rec}\, f, \iota, \rho, \vec{u}, x.\, \mathtt{case}\, x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_j\, x_j \mapsto \hat{s}_j[\hat{t}_1 \mid \cdots \mid \hat{t}_n]}) \star \hat{E}) : T$.

Subcase: f is structurally corecursive

$(\cdot \searrow \Delta; [\eta_\Xi]\Gamma, f{:}T'; T') \Longrightarrow^* \{\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_\Xi]\Gamma, f{:}[\eta_X]T_0', x{:}S; [\eta_X]X\,\vec{C}\}$
and $(\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_\Xi]\Gamma, f{:}[\eta_X]T_0', x{:}S; [\eta_X]X\,\vec{C}) \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta_\Xi]\Gamma_i; [\eta_X]T_i\}_i$
and $T' = \Pi\overrightarrow{u{:}U}.S \to \nu\mathcal{F}\ \vec{C}$, where $q_i = \vec{u}\ p_i\ .d_i\ q_i'$ and $T_0' = \Pi\overrightarrow{u{:}U}.S \to X\ \vec{C}$

by the function criteria.

$(\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}]\Gamma, f{:}[\eta_{\Xi,X}]T_0', x{:}S; [\eta_{\Xi,X}]X\,\vec{C}) \Longrightarrow^* \{q_i \searrow \Delta_i; [\eta_{\Xi,X}]\Gamma_i; [\eta_{\Xi,X}]T_i\}_i$

by weakening.

$(\vec{u}\ x \cdot \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}](\Gamma, f{:}T_0', x{:}S); [\eta_{\Xi,X}](X\,\vec{C})) \Longrightarrow$
$\quad \{\vec{u}\ x\ .d_j \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}](\Gamma, f{:}T_0', x{:}S); [\eta_{\Xi,X}](\mathcal{F}_j\,X\,\vec{C})\}_j$

and for all j, we have $\vec{u}\ x\ .d_j \searrow \Delta, \overrightarrow{u{:}U}; [\eta_{\Xi,X}](\Gamma, f{:}T'_0, x{:}S); [\eta_{\Xi,X}](\mathcal{F}_j\ X\ \vec{C}) \Longrightarrow^* Q_j$
where $\bigcup_j Q_j = \{q_i \searrow \Delta_i; [\eta_{\Xi,X}]\Gamma_i; [\eta_{\Xi,X}]T_i\}_i$ by Lemma 3.19.
for each j,
$(\vec{u}\ x\ .d_j \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, f{:}T'_0, x{:}S, \Gamma_{\Xi,X}; (\mathcal{F}_j\ X\ \vec{C}); \varphi) \Longrightarrow^*_{\overrightarrow{q\mapsto t}}$
$\{q_i \searrow \Delta_i; \Gamma'_i, \Gamma_{\Xi,X}; T_i\}_i \rightsquigarrow \hat{s}_j[\cdot_1 \mid \cdots \mid \cdot_n]$ where Γ'_i extends Γ_i
and $\Gamma'_i \vdash T_i$ satisfy the translation invariants for Ξ, X and φ_i by Lemma 3.23.
$\Delta_i; \Xi; \Gamma'_i, \Gamma_{\Xi,X} \vdash t_i \rightsquigarrow_{\varphi_i} \hat{t}_i : T_i$ by induction hypothesis.
Thus, $\Xi, X \vdash (\vec{u}\ x\ .d_j \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, f{:}T'_0, x{:}S, \Gamma_{\Xi,X}; (\mathcal{F}_j\ X\ \vec{C}); \varphi) \Longrightarrow^*_{\overrightarrow{q\mapsto t}}$
$\{q_i \searrow \Delta_i; \Gamma'_i, \Gamma_{\Xi,X}; T_i\}_i \rightsquigarrow \hat{s}_j[\hat{t}'_1 \mid \cdots \mid \hat{t}'_n]$
and so, $\Delta; \Xi; \Gamma, \Gamma_\Xi \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, (\hat{s}_1; \ldots; \hat{s}_n) : T'$.
$\Delta; \Xi; \Gamma, \Gamma_\Xi; T' \vdash \hat{E} \searrow T''$ by induction hypothesis (Statement 2).
Thus, $\Delta; \Xi; \Gamma \vdash (\mathtt{fun}\ f.\overrightarrow{q \mapsto t}) \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star ((\mathtt{corec}\, f, \iota, \rho, \vec{u}, x.\, (\hat{s}_1; \ldots; \hat{s}_n)) \star \hat{E}) : T$. □

3.6 Commuting Translation and Evaluation

The last part of our setup requires us to show the translation is normalisation preserving. To do so, we bind the number of steps a term can take in the source language with the number of steps its translation can take. Since the latter will be finite, so will be the former. However, as our translation introduces operators ρ and $\mathsf{lift}(T)$ which are instantiated with closures, it is not the case in general that translation will commute with evaluation. That is, if $s \longrightarrow t$ and $\vdash s \rightsquigarrow \hat{s} : T$ and $\vdash t \rightsquigarrow \hat{t} : T$, then it is not necessarily true that $\hat{s} \longrightarrow^+ \hat{t}$.

We discussed that problem in Example 7: we define nested functions:

```
fun f.
| zero ⇒ fun g. y ⇒ y
| suc x ⇒ fun g. y ⇒ add * x (f * x y)
```

They would translate into the program:

```
rec f, ι, ρ, x₀. case x₀ of
| in₁ x ⇒ λy.y
| in₂ x ⇒ λy.add * (ρ * x) (f * x y)
```

But if we were to apply the original program to the number 2, it would reduce to **fun** `g. y ⇒ add * 2 (f * 2 y)`, yielding a translation `λy.add * 2 (f * 2 y)`. However, if we apply the number 2 to the first translated program, we instead get `λy.add * ((λz.z) * 2) (f * 2 y)`. The two translations are thus different.

We can however see that both programs will still behave the same. When applied to the same value, they will still yield the same result. As such, we can establish a simulation between the two. The simulation we use is the relation we defined for our logical relation $[\![T]\!]^*(\theta;\eta)$ from Section 3.2. The saturation was defined as follows:

$$\{t_1 \geq t_2 \mid \exists v_1, v_2 \text{ such that } t_1 \longrightarrow^{n+k} v_1 \text{ and } t_2 \longrightarrow^{n} v_2 \text{ and } v_1 \geq v_2 \in [\![T]\!](\theta;\eta)\}$$

This relation establishes computational equivalence and binds the number of steps the left-hand side takes with the number of steps the right-hand side takes. A key property we need for this relation is precongruence.

Precongruence ensures that the relation behaves well with respect to the structure of the language. For each term construct $C(\cdot_1, \ldots, \cdot_n)$, if we have $t_i \geq t_i' \in [\![T_i]\!](\theta_i;\eta_i)$ for $i = 1, \ldots, n$, then we must have $C(t_1, \ldots, t_n) \geq C(t_1', \ldots, t_n') \in [\![T]\!](\theta;\eta)$. For example, if $t_1 \geq t_2 \in [\![S]\!](\theta;\eta)$ and $E_1 \geq E_2 \in [\![S \searrow T]\!](\theta;\eta)$, then $t_1 \star E_1 \geq t_2 \star E_2 \in [\![T]\!](\theta;\eta)$. Now, if we want to the same for λ-abstractions, we need need terms a relation over open terms as the body of λ-abstractions $\lambda x.$ depend on a variable x.

Let us generalize our simulation to open terms. We obtain the generalization by simply abstracting over all closing substitutions.

Definition 3.8 (Open simulation)**.** We define the open simulation between terms t_1 and t_2 using the judgment $\Delta; \Xi; \Gamma \vdash t_1 \geq t_2 : T$. This judgment states that for all substitutions $\vdash \theta : \Delta$ and $\eta \in [\![\Xi]\!]$ and $\sigma \in [\![\Gamma]\!](\theta;\eta)$ we have $[\theta;\sigma]t_1 \geq [\theta;\sigma]t_2 \in [\![T]\!]^*(\theta;\eta)$.

Similarly define the open simulation of spines E_1 and E_2 as the judgment $\Delta; \Xi; \Gamma; S \vdash E_1 \geq E_2 \searrow T$. This judgment states that for all substitutions $\vdash \theta : \Delta$ and $\eta \in [\![\Xi]\!]$ and $\sigma \geq \sigma \in [\![\Gamma]\!](\theta;\eta)$, we have $[\theta;\sigma]E_1 \geq [\theta;\sigma]E_2 \in [\![S \searrow T]\!]^*(\theta;\eta)$.

We note that our main statement for normalisation (Theorem 3.13) is exactly reflexivity of open simulation.

Now, we can properly define precongruence. A proof of precongruence for simulation is usually pretty involved, especially given the size of our language. One of the main proof techniques to achieve that is Howe's method which builds an auxiliary relation from the simulation in such a way that it is easily proven to be precongruent. Then, it requires to show that the two relations are equivalent. We provide a more in depth description of the method in our case study in Chapter 4. For the purpose of this chapter, we omit such proof and instead conjecture precongruence. We specifically state two cases:

Conjecture 3.25 (Precongruence case for application)**.** *If* $\Delta; \Xi; \Gamma \vdash t_1 \geq t_2 : S$ *and* $\Delta; \Xi; \Gamma; S \vdash E_1 \geq E_2 \searrow T$ *then* $\Delta; \Xi; \Gamma \vdash t_1 \star E_1 \geq t_2 \star E_2 : T$*.*

Conjecture 3.26 (Precongruence case for λ-abstraction)**.** *If* $\Delta; \Xi; \Gamma, x{:}S \vdash t_1 \geq t_2 : T$*, then* $\Delta; \Xi; \Gamma \vdash \lambda x.t_1 \geq \lambda x.t_2 : S \to T$*.*

Conjecture 3.27 (Precongruence)**.** *The simulation* $\geq$ *is a precongruence.*

Translation Preserves Evaluation

Now that we have a simulation to properly bind our evaluation, we shall move on to prove that terms related by evaluation are, when translated, related by simulation. This is done by establishing several lemmas about matching, evaluation, simulation and their relation to the translation.

We first recall that φ associates patterns from the source languages that are valid terms for recursive calls to the variable in the target language they correspond to. Thus, recursive calls in the target language will not be applied to a translation of their original terms but to those variables instead. If we look at the resulting substitution of a matching in the target language, it needs to have instantiations for those variables as well as the original pattern variables as those can occur in the right-hand side.

We thus define a version of (co)pattern matching that takes the mapping φ into account to adequately output a target level substitution. The enhanced pattern matching is denoted by the judgment $v =_\varphi [\theta; \sigma]p$. The inference rules for it appear in Figure 3.14. It works very much like our original matching that was define in Figure 2.12. We simply make

$\boxed{v =_\varphi [\theta;\sigma]p}$

$$\frac{}{v =_\varphi [\cdot; v/x]x} \quad \frac{v =_\varphi [\theta;\sigma]p \quad c\ p \notin \varphi}{c\ v =_\varphi [\theta;\sigma](c\ p)} \quad \frac{v_1 =_\varphi [\theta_1;\sigma_1]p_1 \quad v_1 =_\varphi [\theta_1;\sigma_1]p_1 \quad (p_1,p_2) \notin \varphi}{(v_1,v_2) =_\varphi [\theta_1,\theta_2;\sigma_1,\sigma_2](p_1,p_2)}$$

$$\frac{}{\wp =_\varphi [\cdot;\cdot]\wp} \quad \frac{v =_\varphi [\theta;\sigma]p \quad \varphi(c\ p) = x}{c\ v =_\varphi [\theta;\sigma,(c\ v)/x](c\ p)} \quad \frac{v_1 =_\varphi [\theta_1;\sigma_1]p_1 \quad v_1 =_\varphi [\theta_1;\sigma_1]p_1 \quad \varphi(p_1,p_2) = x}{(v_1,v_2) =_\varphi [\theta_1,\theta_2;\sigma_1,\sigma_2,(v_1,v_2)/x](p_1,p_2)}$$

$$\frac{v =_\varphi [\theta;\sigma]p \quad \mathtt{pack}\,\langle u,p\rangle \notin \varphi}{\mathtt{pack}\,\langle C,v\rangle =_\varphi [\theta,C/u;\sigma](\mathtt{pack}\,\langle u,p\rangle)} \quad \frac{v =_\varphi [\theta;\sigma]p \quad \varphi(\mathtt{pack}\,\langle u,p\rangle) = x}{\mathtt{pack}\,\langle C,v\rangle =_\varphi [\theta,C/u;\sigma,\mathtt{pack}\,\langle C,v\rangle\,/x](\mathtt{pack}\,\langle u,p\rangle)}$$

Figure 3.14: Enhanced (co)pattern matching

an additional check in our inductive cases. If the pattern occurs in φ, then we add an additional substitution for its variable. As such, the substitution will provide instantiations for all variables in φ. The judgment for enhanced copattern matching $E =_\varphi [\theta;\sigma]q@E'$ works the same way our original one does, but simply appeals to the enhanced pattern matching judgment instead of the original one.

Before we move on to proving the translation is normalisation preserving, we need to prove some lemmas about (co)pattern matching. The following statements involve the original source language matching $E = [\theta;\sigma]q@E$ and $v = [\theta;\sigma]p$ without any φ, but they hold equivalently for our enhanced matching judgments.

Lemma 3.28 (Uniqueness of matching)**.** *If $E = [\theta_1;\sigma_1]q@E_1$ and $E = [\theta_2;\sigma_2]q@E_2$, then $\theta_1 = \theta_2$ and $\sigma_1 = \sigma_2$ and $E_1 = E_2$.*

Proof. By induction on $E = [\theta_1;\sigma_1]q@E_1$ and inversion on $E = [\theta_2;\sigma_2]q@E_2$. □

Lemma 3.29 (Inversion of pattern matching)**.** *The following hold:*

1. *If $v = [\theta;\sigma](p[\wp])$, then $v = [\theta;\sigma,\wp/x](p[x])$.*
2. *If $v = [\theta;\sigma](p[(x_1,x_2)])$, then $v = [\theta;\sigma'](p[x])$ where $\sigma = \sigma'', v_1/x_1, v_2/x_2$ and $\sigma' = \sigma'', (v_1,v_2)/x$ for some σ''.*

3. *If* $v = [\theta;\sigma](q[\mathtt{pack}\,\langle u, x'\rangle])$, *then* $v = [\theta';\sigma'](q[x])$ *where* $\sigma = \sigma'', v/x'$ *and* $\sigma' = \sigma'', (\mathtt{pack}\,\langle C, v\rangle)/x$ *for some* σ'' *and* $\theta = \theta', C/u$.
4. *If* $v = [\theta;\sigma](q[c\ x'])$, *then* $v = [\theta;\sigma'](q[x])$ *where* $\sigma = \sigma'', v/x'$ *and* $\sigma' = \sigma'', (c\ v)/x$ *for some* σ''.

Proof. All statements are proven by induction on the matching judgments using linearity of copatterns. □

Lemma 3.30 (Inversion of copattern matching)**.** *The following hold:*

1. *If* $E = [\theta;\sigma](q\ x)@E'$, *then* $E = [\theta;\sigma']q@(v\ E')$ *where* $\sigma = \sigma', v/x$.
2. *If* $E = [\theta;\sigma](q\ u)@E'$, *then* $E = [\theta';\sigma]q@(C\ E')$ *where* $\theta = \theta', C/u$.
3. *If* $E = [\theta;\sigma](q\ .d)@E'$, *then* $E = [\theta;\sigma]q@(.d\ E')$.
4. *If* $E = [\theta;\sigma](q[\wp])@E'$, *then* $E = [\theta;\sigma,\wp/x](q[x])@E'$.
5. *If* $E = [\theta;\sigma](q[(x_1, x_2)])@E'$, *then* $E = [\theta;\sigma'](q[x])@E'$ *where* $\sigma = \sigma'', v_1/x_1, v_2/x_2$ *and* $\sigma' = \sigma'', (v_1, v_2)/x$ *for some* σ''.
6. *If* $E = [\theta;\sigma](q[\mathtt{pack}\,\langle u, x'\rangle])@E'$, *then* $E = [\theta';\sigma'](q[x])@E'$ *where* $\sigma = \sigma'', v/x'$ *and* $\sigma' = \sigma'', (\mathtt{pack}\,\langle C, v\rangle)/x$ *for some* σ'' *and* $\theta = \theta', C/u$.
7. *If* $E = [\theta;\sigma](q[c\ x'])@E'$, *then* $E = [\theta;\sigma'](q[x])@E'$ *where* $\sigma = \sigma'', v/x'$ *and* $\sigma' = \sigma'', (c\ v)/x$ *for some* σ''.

Proof. Statements 1 to 3 are proven by case analysis on q and induction on the matching judgments. The other statements are proven by induction on the matching judgments, linearity of copatterns, and Lemma 3.29. □

Lemma 3.31 (Coverage splits preserve matching)**.** *If* $(q \searrow \Delta;\Gamma;T) \Longrightarrow^* Q$ *and for some* $(q' \searrow \Delta';\Gamma';T') \in Q$ *we have* $E = [\theta;\sigma]q'@E'$, *then* $E = [\theta';\sigma']q@E''$ *for some* θ' *and* σ' *and* E''.

Proof. By induction on $(q \searrow \Delta;\Gamma;T) \Longrightarrow^* Q$. The inductive case is proved by case analysis on the single step refinement $(q \searrow \Delta;\Gamma;T) \Longrightarrow Q'$ using Lemma 3.30. □

Lemma 3.32 (Equivalence of (co)pattern matching judgments)**.** *The following hold:*

1. *$v = [\theta;\sigma]p$ if and only if for all φ there is a σ' such that $v =_\varphi [\theta;\sigma']p$.*
2. *If $E = [\theta;\sigma]q@E'$ if and only if for all φ there is a σ' such that $E =_\varphi [\theta;\sigma']q@E'$.*

Proof. Each statement is proved by induction on the matching judgment. □

We now can move on to prove that translation preserves normalisation. We first start with translation of copatterns. The following lemma states that matching on a copattern refinement is the same has stepping to translation of the right-hand sides of branches in the translation of the copattern. That is, that translation of copattern is well-behaved with respect to operational semantics.

Lemma 3.33 (Copattern translation preserves matching)**.** *Suppose $q \searrow \Delta;\Gamma;T;\varphi \Longrightarrow^*_B Q \rightsquigarrow \hat{t}[\cdot_1 \mid \cdots \mid \cdot_n]$ and $T_0 \vdash_v E_0 \searrow T'$ and $E_0 =_\varphi [\theta;\sigma]q@E'$ and for all $(q_i \mapsto t_i) \in B$ we have $\Delta_i;\Xi;\Gamma_i \vdash t_i \rightsquigarrow \hat{t}_i : T_i$. If there is some $(q_i \searrow \Delta_i;\Gamma_i;T_i;\varphi_i) \in Q$ such that $E_0 =_{\varphi_i} [\theta_i;\sigma_i]q_i@E'_i$, then*

$$[\theta;\hat{\sigma},\sigma_\Xi]\hat{t}[\hat{t}_1 \mid \cdots \mid \hat{t}_n] \star [\sigma_\Xi]\hat{E}' \longrightarrow^* [\theta_i;\hat{\sigma}_i,\sigma_\Xi]\hat{t}_i \star [\sigma_\Xi]\hat{E}'_i$$

where $\vdash \sigma \rightsquigarrow \hat{\sigma} : \Delta$ and $\vdash \sigma_i \rightsquigarrow \hat{\sigma}_i : \Delta$ and $\Xi;\Gamma_\Xi;T \vdash E' \rightsquigarrow \hat{E}' \searrow T'$ and $\Xi;\Gamma_\Xi;T_i \vdash E'_i \rightsquigarrow \hat{E}'_i \searrow T'$.

Proof. By induction on $(q \searrow \Delta;\Gamma;T;\varphi) \Longrightarrow^*_{\Xi;B} Q \rightsquigarrow \hat{t}$.

Case:
$$\frac{(q \mapsto t) \in B \quad \Delta;\Xi;\Gamma \vdash t : T \rightsquigarrow \hat{t}}{q \searrow \Delta;\Gamma;T;\varphi \Longrightarrow^*_{\Xi;B} \{q \searrow \Delta;\Gamma;T;\varphi\} \rightsquigarrow \hat{t}}$$

$\theta = \theta_i$ and $\sigma = \sigma_i$ and $E' = E_i$ by determinacy of matching.
$[\theta;\hat{\sigma},\sigma_\Xi]\hat{t} \star [\sigma_\Xi]\hat{E}' \longrightarrow^* [\theta;\hat{\sigma},\sigma_\Xi]\hat{t} \star [\sigma_\Xi]\hat{E}'$ by reflexive rule of $\longrightarrow^*$

Case:
$$\frac{q \searrow \Delta;\Gamma;T \Longrightarrow Q' \rightsquigarrow s[\cdot_1 \mid \cdots \mid \cdot_n] \quad \text{for all } q_i \in Q', q_i \Longrightarrow^*_{\Xi;\vec{b}} Q_i \rightsquigarrow \bar{t}_i}{q \searrow \Delta;\Gamma;T \Longrightarrow^*_{\Xi;\vec{b}} \bigcup_i Q_i \rightsquigarrow s[\bar{t}_1 \mid \cdots \mid \bar{t}_n]}$$

We proceed by case analysis on $q \searrow \Delta;\Gamma;T \Longrightarrow Q' \rightsquigarrow s[\cdot_1 \mid \cdots \mid \cdot_n]$. We showcase some of

the interesting subcases.

Subcase: $\overline{q \searrow \Delta; \Gamma; \Pi u{:}U.T \Longrightarrow \{q\ u \searrow \Delta, u{:}U; \Gamma; T\} \rightsquigarrow \Lambda u.\cdot}$

$E_0 =_\varphi [\theta'; \sigma'](q\ u)@E''$ as coverage splits preserve matching (Lemma 3.31)

$E' = C\ E''$ and $\theta' = \theta, C/u$ and $\sigma' = \sigma$ by inversion of copat. (Lemmas 3.28 and 3.30)

$[\theta; \hat{\sigma}, \sigma_\Xi](\Lambda u.\bar{t}_1) \star C\ [\sigma_\Xi]\hat{E}''$

$\longrightarrow [\theta, C/u; \hat{\sigma}, \sigma_\Xi]\bar{t}_1 \star [\sigma_\Xi]\hat{E}''$ by stepping rule

$\longrightarrow^* [\theta_i; \hat{\sigma}_i, \sigma_\Xi]\hat{t}_i \star [\sigma_\Xi]\hat{E}'_i)$ by induction hypothesis

Subcase: $\overline{q \searrow \Delta; \Gamma; X\ \vec{C} \Longrightarrow \{q\ .d_j \searrow \Delta; \Gamma; \mathcal{F}_j X\ \vec{C}\}_j \rightsquigarrow \iota_X \star \vec{C}\ (\cdot_1; \ldots; \cdot_n)}$

$E_0 =_\varphi [\theta'; \sigma'](q\ .d_j)@E''$ as coverage splits preserve matching (Lemma 3.31)

$E' = .d_i\ E''$ and $\theta' = \theta$ and $\sigma' = \sigma$ by inversion of copat. (Lemmas 3.28 and 3.30)

$[\theta; \hat{\sigma}, \sigma_\Xi]\iota_X \star [\theta]\vec{C}\ [\theta; \hat{\sigma}, \sigma_\Xi](\bar{t}_1; \ldots; \bar{t}_n)\ .\mathtt{out}_\nu\ .\mathtt{out}_j\ [\sigma_\Xi]\hat{E}''$

$= \Lambda\vec{u}.\lambda x.\mathtt{in}_\nu\ x \star [\theta]\vec{C}\ [\theta; \hat{\sigma}, \sigma_\Xi](\bar{t}_1; \ldots; \bar{t}_n)\ .\mathtt{out}_\nu\ .\mathtt{out}_j\ [\sigma_\Xi]\hat{E}''$ by definition of σ_Ξ

$\longrightarrow \mathtt{in}_\nu\ [\theta; \hat{\sigma}, \sigma_\Xi](\bar{t}_1; \ldots; \bar{t}_n) \star .\mathtt{out}_\nu\ .\mathtt{out}_j\ [\sigma_\Xi]\hat{E}''$

$\longrightarrow [\theta; \hat{\sigma}, \sigma_\Xi](\bar{t}_1; \ldots; \bar{t}_n) \star .\mathtt{out}_j\ [\sigma_\Xi]\hat{E}''$

$\longrightarrow [\theta; \hat{\sigma}, \sigma_\Xi]\bar{t}_j \star [\sigma_\Xi]\hat{E}''$ by stepping rules

$\longrightarrow^* [\theta_i; \hat{\sigma}_i, \sigma_\Xi]\hat{t}_i \star [\sigma_\Xi]\hat{E}'_i$ by induction hypothesis

Subcase: $\dfrac{\Delta \vdash C_1 = C_2 \searrow \Delta' \quad \# \notin \Delta'}{q[x] \searrow \Delta; \Gamma, x : C_1{=}C_2; T \Longrightarrow \{q[\wp] \searrow \Delta'; \Gamma; T\} \rightsquigarrow \mathtt{eq}\, x\, \mathtt{with}\, \cdot}$

$E =_\varphi [\theta'; \sigma'](q[\wp])@E''$ as coverage splits preserve matching (Lemma 3.31)

$E' = E''$ and $\theta' = \theta$ and $\sigma' = \sigma, \wp/x$ by inversion of copat. (Lemmas 3.28 and 3.30)

$[\theta; \hat{\sigma}, \wp/x, \sigma_\Xi](\mathtt{eq}\, x\, \mathtt{with}\, \bar{h}_1\ \bar{E}_1) \star [\sigma_\Xi]\hat{E}''$

$= \mathtt{eq}\, \wp\, \mathtt{with}\, [\theta; \hat{\sigma}, \wp/x, \sigma_\Xi](\bar{h}_1\ \bar{E}_1) \star [\sigma_\Xi]\hat{E}''$

$\longrightarrow [\theta; \hat{\sigma}, \wp/x, \sigma_\Xi]\bar{t}_1 \star [\sigma_\Xi]\hat{E}''$ by stepping rules

$= [\theta; \hat{\sigma}, \sigma_\Xi]\bar{t}_1 \star [\sigma_\Xi]\hat{E}''$ as $\bar{t}_1$ does not depend on x

$\longrightarrow^* [\theta_i; \hat{\sigma}_i, \sigma_\Xi]\hat{t}_i \star [\sigma_\Xi]\hat{E}'_i$ by induction hypothesis

Subcase: $$\frac{\Delta \vdash C_1 = C_2 \searrow \Delta' \quad \# \in \Delta'}{q[x] \searrow \Delta; \Gamma, x : C_1{=}C_2; T \Longrightarrow \{q[\wp] \searrow \Delta'; \Gamma; T\} \rightsquigarrow \mathtt{eq\_abort}\, x}$$

Thus, $E_0 =_\varphi [\theta; \sigma](q[x])@E'$ and $E_0 =_\varphi [\theta'; \sigma'](q[\wp])@E''$.

$\theta' = \theta$ and $\sigma = \sigma', \wp/x$ and $E' = E''$ by inversion of copat. (Lemma 3.30).

$\vdash \wp : [\theta]C_1 = [\theta]C_2$ by typing.

$\vdash [\theta]C_1 = [\theta]C_2$ by inversion on typing.

$\vdash \theta : \Delta$ implies $\vdash \theta : \Delta'$ by Req 3.

$\# \in \cdot$ since $\# \in \Delta'$ by typing of substitutions.

This is a contradiction. Thus, this case is impossible.

Subcase: $$q[x] \searrow \Delta; \Gamma, x : \Sigma u{:}U.T'; T \Longrightarrow \{q[\mathtt{pack}\,\langle u, x'\rangle] \searrow \Delta, u{:}U; \Gamma, x'{:}T'; T\} \rightsquigarrow \mathtt{unpack}\, x\, \mathtt{as}\, \langle u, x'\rangle\, \mathtt{in}\, \cdot$$

$E =_\varphi [\theta'; \sigma'](q[\mathtt{pack}\,\langle u, x'\rangle])@E''$ as coverage splits preserve matching (Lemma 3.31)

$E' = E''$ and $\theta' = \theta$ and $\sigma' = \sigma, (\mathtt{pack}\,\langle C, v\rangle)/x$ by inversion of copat. (Lemmas 3.28 and 3.30)

$\Xi; \Gamma_\Xi \vdash v : S \rightsquigarrow \hat{v}$ by inversion on $\Xi; \Gamma_\Xi \vdash \sigma' \rightsquigarrow \hat{\sigma}' : \Gamma, x'{:}T'$

For convenience, let $\hat{\sigma}' = \hat{\sigma}, \mathtt{pack}\,\langle C, [\sigma_\Xi]\hat{v}\rangle\, /x, [\sigma_\Xi]\hat{v}/x', \sigma_\Xi$ and $\hat{\sigma}'' = \hat{\sigma}, [\sigma_\Xi]\hat{v}/x', \sigma_\Xi$

$$\begin{aligned}
&[\theta; \hat{\sigma}, (\mathtt{pack}\,\langle C, [\sigma_\Xi]\hat{v}\rangle)/x, \sigma_\Xi](\mathtt{unpack}\, x\, \mathtt{as}\, \langle u, x'\rangle\, \mathtt{in}\, \bar{t}_1) \star [\sigma_\Xi]\hat{E}'' \\
&\quad = \mathtt{unpack}\,(\mathtt{pack}\,\langle C, [\sigma_\Xi]\hat{v}\rangle)\, \mathtt{as}\, \langle u, x'\rangle\, \mathtt{in}\, [\theta; \hat{\sigma}, (\mathtt{pack}\,\langle C, [\sigma_\Xi]\hat{v}\rangle)/x, \sigma_\Xi]\bar{t}_1 \star [\sigma_\Xi]\hat{E}'' \\
&\quad \longrightarrow [\theta, C/u; \hat{\sigma}']\bar{t}_1 \star [\sigma_\Xi]\hat{E}'' \\
&\quad = [\theta, C/u; \hat{\sigma}'']\bar{t}_1 \star [\sigma_\Xi]\hat{E}'' && \text{as } \bar{t}_1 \text{ does not depend on } x \\
&\quad \longrightarrow^* [\theta_i; \hat{\sigma}_i, \sigma_\Xi]t_i \star [\sigma_\Xi]\hat{E}'_i && \text{by induction hypothesis}
\end{aligned}$$

Subcase: $$q[x] \searrow \Delta; \Gamma, x : X\ \vec{C}; T \Longrightarrow \{q[c_j\ x'] \searrow \Delta; \Gamma, x_j{:}\mathcal{F}_j X\ \vec{C}; T\}_j \rightsquigarrow \mathtt{case}\, \iota_X \star \vec{C}\ x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto \cdot}_i$$

$E =_\varphi [\theta'; \sigma'](q[c_j\ x_j])@E''$ as coverage splits preserve matching (Lemma 3.31)

$E' = E'$ and $\theta' = \theta$ and $\sigma' = \sigma, (c_j\ v_j)/x$ by inversion of copat. (Lemmas 3.28 and 3.30)

$\Xi; \Gamma_\Xi \vdash v_j \rightsquigarrow \hat{v}_j : S$ by inversion on $\Xi; \Gamma_\Xi \vdash \sigma' \rightsquigarrow \hat{\sigma}' : \Gamma, x_j{:}\mathcal{F}_j X\ \vec{C}$

For convenience, let $\hat{\sigma}' = \hat{\sigma}, (\mathtt{in}_\mu\,(\mathtt{in}_j\,[\sigma_\Xi]v_j))/x, \sigma_\Xi$ and $\hat{\sigma}'' = \hat{\sigma}, [\sigma_\Xi]v_j/x_j, \sigma_\Xi$

$$
\begin{aligned}
&[\theta;\hat{\sigma}'](\mathtt{case}\,\iota_X \star \vec{C}\ x\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\, x_i \mapsto \bar{t}_i})\ [\sigma_\Xi]\hat{E}'' \\
&\quad = \mathtt{case}\,((\Lambda\vec{u}.\lambda x.\mathtt{out}_\mu\, x) \star \vec{C}\ (\mathtt{in}_\mu\,(\mathtt{in}_j\,[\sigma_\Xi]v_j)))\ \mathtt{of}\ \overrightarrow{\mathtt{in}_j\, x_j \mapsto [\theta;\hat{\sigma}']\bar{t}_j} \star [\sigma_\Xi]\hat{E}'' \\
&\quad \longrightarrow^* \mathtt{case}\,\mathtt{out}_\mu\,(\mathtt{in}_\mu\,(\mathtt{in}_j\,[\sigma_\Xi]v_j))\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\, x_i \mapsto [\theta;\hat{\sigma}']\bar{t}_i} \star [\sigma_\Xi]\hat{E}'' \\
&\quad \longrightarrow \mathtt{case}\,\mathtt{in}_j\,[\sigma_\Xi]v_j\ \mathtt{of}\ \overrightarrow{\mathtt{in}_i\, x_i \mapsto [\theta;\hat{\sigma}']\bar{t}_i} \star [\sigma_\Xi]\hat{E}'' \\
&\quad \longrightarrow [\theta;\hat{\sigma}', v_i/x_i]\bar{t}_j \star [\sigma_\Xi]\hat{E}'' \\
&\quad = [\theta;\hat{\sigma}'']\bar{t}_j \star [\sigma_\Xi]\hat{E}'' && \text{as } \bar{t}_j \text{ does not depend on } x \\
&\quad \longrightarrow^* [\theta_i;\hat{\sigma}_i, \sigma_\Xi]\hat{t}_i \star \hat{E}'_i && \text{by induction hypothesis}
\end{aligned}
$$

□

Lemma 3.34. *If* $\vdash t : T$, *then* $\vdash \mathit{lift}_{-}(T)\ t \geq t : T$.

Proof. By induction on T. □

Lemma 3.35 (Precongruence of $\geq$ on terms with holes)**.** *If* $\Delta_i;\Xi_i;\Gamma_i \vdash t_i \geq t'_i : T_i$ *and* $\Delta;\Xi;\Gamma \vdash t[\cdot_1 \mid \cdots \mid \cdot_n] : T$, *then we have* $\Delta;\Xi;\Gamma \vdash t[t_1 \mid \cdots \mid t_n] \geq t[t'_1 \mid \cdots \mid t'_n] : T$.

Proof. By induction on $\Delta;\Xi;\Gamma \vdash t[\cdot_1 \mid \cdots \mid \cdot_n] : T$ and by appeal to precongruence for $\geq$ (Conjecture 3.27). □

The next lemma shows that commuting translation and substitutions results in similar terms. In order for this statement to hold, we need to limit ourselves to substitutions σ that agree with φ. By agreeing, we mean that if $\varphi(v) = x$, then $\sigma(x) = v$. We note that by the very definition of enhanced matching $E =_\varphi [\theta;\sigma]q@E'$, the resulting substitution σ will agree with φ.

Lemma 3.36 (Commuting substitutions with translation)**.** *Suppose* $\Delta' \vdash \theta : \Delta$ *and* $\Delta';[\theta]\Xi;\Gamma' \vdash \sigma \rightsquigarrow \hat{\sigma} : [\theta]\Gamma$, *where* σ *agrees with* φ. *The following hold:*

1. *If* $\Delta;\Xi;\Gamma,\Gamma_\Xi \vdash t \rightsquigarrow_\varphi \hat{t} : T$ *and* $\Delta';[\theta]\Xi;\Gamma',\Gamma_{[\theta]\Xi} \vdash [\theta;\sigma]t \rightsquigarrow \hat{t}' : [\theta]T$, *then we have* $\Delta';[\theta]\Xi;\Gamma',\Gamma_{[\theta]\Xi} \vdash [\theta;\hat{\sigma}]\hat{t} \geq \hat{t}' : [\theta]T$.
2. *If* $\Delta;\Xi;\Gamma,\Gamma_\Xi;S \vdash E \rightsquigarrow_\varphi \hat{E} : T$ *and* $\Delta';[\theta]\Xi;\Gamma',\Gamma_{[\theta]\Xi};[\theta]S \vdash [\theta;\sigma]E \rightsquigarrow \hat{E}' : [\theta]T$, *then we have* $\Delta';[\theta]\Xi;\Gamma',\Gamma_{[\theta]\Xi};[\theta]S \vdash [\theta;\hat{\sigma}]\hat{E} \geq \hat{E}' : [\theta]T$.

Proof. By induction on the derivations $\Delta; \Xi; \Gamma \vdash t \rightsquigarrow \hat{t} : T$ and $\Delta; \Xi; \Gamma, \Gamma_\Xi; S \vdash E \rightsquigarrow \hat{E} \searrow T$.

Case: $$\frac{\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \hat{t} : S \quad \Delta; \Xi; \Gamma; S \vdash E \rightsquigarrow_\varphi \hat{E} : [\eta]T}{\Delta; \Xi; \Gamma \vdash (\texttt{fun}\ f.\overrightarrow{q \mapsto t}) \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (\hat{t} \star \hat{E}) : T}$$

By definition of substitution, and inversion on $\vdash [\theta;\sigma]t \rightsquigarrow \hat{t}' : [\theta;\eta]T$, we have

$$\frac{\Delta'; \Xi; \Gamma' \vdash_f \overrightarrow{q \mapsto [\theta;\sigma]t} \rightsquigarrow \hat{t}' : [\theta]S \quad \Delta'; \Xi; \Gamma'; [\theta]S \vdash [\theta;\sigma]E \rightsquigarrow \hat{E}' : [\theta;\eta]T}{\Delta'; \Xi; \Gamma' \vdash (\texttt{fun}\ f.\overrightarrow{q \mapsto [\theta;\sigma]t}) \star [\theta;\sigma]E \rightsquigarrow \mathsf{lift}.([\theta]T) \star (\hat{t}' \star \hat{E}') : [\theta]T}$$

$\Delta'; \Xi; \Gamma'; S \vdash [\theta]S \vdash [\theta;\hat{\sigma}]\hat{E} \geq \hat{E}' : [\theta;\eta]T$ by induction hypothesis (Statement 2).
We now proceed by subcase analysis on $\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \hat{t} : S$. The methodology of the proof remains the same across subcases, so we only show a single one.

Subcase:

$$\frac{\begin{array}{c} f \text{ is struct. rec.} \quad \bigcup_i Q_i = \{q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j\}_j \quad B_i = \{q \mapsto t \mid q \in Q_i\} \\ \text{for all } i, \quad \Xi, X \vdash (\vec{u}\ (c_i\ x_i)) \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, \Gamma_X, x_i{:}\mathcal{F}_i X\, \vec{C}; T; \varphi, x_i{:=}x_i \Longrightarrow^*_{B_i} Q_i \rightsquigarrow \hat{t}_i \end{array}}{\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \texttt{rec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, \texttt{case}\, x\, \texttt{of}\, \overrightarrow{\texttt{in}_i\, x_i \mapsto \hat{t}_i} : \Pi\overrightarrow{u{:}U}.\mu\mathcal{F}\,\vec{C} \to T}$$

By inversion on copattern translation, we have

$$\frac{\begin{array}{c} q \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, \Gamma_X, x_i{:}\mathcal{F}_i\ X\ \vec{C}; T; \varphi, x_i{:=}x_i \Longrightarrow^*_{B_i} Q_i \rightsquigarrow \hat{t}_i[\cdot_1 \mid \cdots \mid \cdot_n] \\ \text{for all } (q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j) \in Q_i, \quad \Delta_j; \Xi; \Gamma_j \vdash t_j \rightsquigarrow_{\varphi_j} \hat{t}'_j : T_j \end{array}}{\Xi, X \vdash q \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, \Gamma_X, x_i{:}\mathcal{F}_i\ X\ \vec{C}; T; \varphi, x_i{:=}x_i \Longrightarrow^*_{B_i} Q_i \rightsquigarrow \hat{t}_i[\hat{t}'_1 \mid \cdots \mid \hat{t}'_n]}$$

$\Delta'_j \vdash \theta : \Delta_j$ and $\Delta'_j; \Gamma'_j \vdash \sigma : [\theta]\Gamma_j$ for some Δ'_j and Γ'_j as copat. are stable under subst. (Lemma 2.5).

Thus, $\Delta'_j; \Xi, X; \Gamma'_j \vdash [\theta;\sigma]t'_j \rightsquigarrow \hat{t}''_j : [\theta]T_j$ for some $\hat{t}''_j$.
$\Delta'_j; \Xi, X; \Gamma'_j \vdash [\theta;\sigma]\hat{t}'_j \geq \hat{t}''_j : [\theta]T_j$ by induction hypothesis.
Thus, $\hat{t}$ is $\texttt{rec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, \texttt{case}\, x\, \texttt{of}\, \overrightarrow{\texttt{in}_i\, x_i \mapsto \hat{t}_i[\hat{t}'_1 \mid \cdots \mid \hat{t}'_n]}$.
while $\hat{t}'$ is $\texttt{rec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, \texttt{case}\, x\, \texttt{of}\, \overrightarrow{\texttt{in}_i\, x_i \mapsto \hat{t}_i[\hat{t}''_1 \mid \cdots \mid \hat{t}''_n]}$.
$\Delta'; \Xi; \Gamma' \vdash \hat{t} \geq \hat{t}' : [\theta]S$ by precongruence (Lemma 3.35).
$\Delta'; \Xi; \Gamma' \vdash \mathsf{lift}([\theta]T) \star (\hat{t} \star [\theta;\hat{\sigma}]\hat{E}) \geq \mathsf{lift}([\theta]T) \star (\hat{t}' \star \hat{E}') : [\theta]T$ by precongruence (Conj. 3.27).

□

Main Result

This leads us to our main result. That is, if a term satisfies the function criteria, then it is terminating. Our main theorem simply states that if a term s steps to t in the source language and both s and t have translations to our core calculus, then the number of steps the translation of s takes to reach a value is strictly higher than the number of steps the translation of t. Then, each step taken from s has a translation taking an ever smaller number of steps before reaching a value. Since each of those numbers are finite, it will hold that s has a finite number of step before reaching a value.

Theorem 3.37. *If $s \longrightarrow t$ and $\vdash s \rightsquigarrow \hat{s} : T$ and $\vdash t \rightsquigarrow \hat{t} : T$, then there are values v and w such that $\hat{s} \longrightarrow^n v$ and $\hat{t} \longrightarrow^m w$ and $n > m$.*

Proof. We know by Theorem 3.13 that there are values v and w such that $\hat{s} \longrightarrow^* v$ and $\hat{t} \longrightarrow^* w$. It suffices to establish the bounds. We proceed by induction on the derivation $s \longrightarrow t$. The only case that cannot be handled by a simple appeal to the induction hypothesis is the function case that we showcase here.

Case: $$\frac{E \text{ value} \quad E = [\theta;\sigma]q_i@E' \quad \sigma' = \sigma, (\texttt{fun}\ f.\overrightarrow{q \mapsto t})/f}{(\texttt{fun}\ f.\overrightarrow{q \mapsto t}) \star E \longrightarrow [\theta;\sigma']t_i \star E'}$$

By inversion on the translation of the left-hand side, we have

$$\frac{\Delta;\Xi;\Gamma \vdash_f \vec{b} \rightsquigarrow_\varphi \hat{t} : S \quad \Delta;\Xi;\Gamma;S \vdash E \rightsquigarrow_\varphi \hat{E} : [\eta_\Xi]T}{\Delta;\Xi;\Gamma \vdash (\texttt{fun}\ f.\vec{b}) \star E \rightsquigarrow_\varphi \mathsf{lift}_\Xi(T) \star (\hat{t} \star \hat{E}) : T}$$

while the translation judgment for the right-hand side is $\vdash [\theta;\sigma']t_i \rightsquigarrow \hat{t}' : T$ Inversion on the judgment $\Delta;\Xi;\Gamma \vdash_f \vec{b} \rightsquigarrow_\varphi \hat{t} : S$ leads to three possible subcases:

Subcase: $$\frac{f \text{ is non rec.} \quad \Xi \vdash \cdot \searrow \Delta;\Gamma, f{:}T;T;\varphi \Longrightarrow^*_{\overrightarrow{q \mapsto t}} \{q_j \searrow \Delta_j;\Gamma_j;T_j;\varphi_j\}_j \rightsquigarrow \hat{t}}{\Delta;\Xi;\Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \hat{t} : T}$$

By inversion on $\Xi \vdash \cdot \searrow \Delta; \Gamma, f{:}T; T; \varphi \Longrightarrow^*_{\overrightarrow{q \mapsto \hat{t}}} \{q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j\}_j \rightsquigarrow \hat{t}$, we have

$$\frac{q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow^*_{\overrightarrow{q \mapsto \hat{t}}} Q \rightsquigarrow \hat{t}[\cdot_1 \mid \cdots \mid \cdot_n] \qquad \text{for all } (q_i \searrow \Delta_i; \Gamma_i; T_i; \varphi_i) \in Q, \quad \Delta_i; \Xi; \Gamma_i \vdash t_i \rightsquigarrow_{\varphi_i} \hat{t}_i : T_i}{\Xi \vdash q \searrow \Delta; \Gamma; T; \varphi \Longrightarrow^*_{\overrightarrow{q \mapsto \hat{t}}} Q \rightsquigarrow \hat{t}[\hat{t}_1 \mid \cdots \mid \hat{t}_n]}$$

$\hat{t} \star \hat{E} \longrightarrow^* [\theta; \hat{\sigma}]\hat{t}_i \star \hat{E}'$ as copattern translation preserves matching (Lemma 3.33).
$\vdash [\theta; \hat{\sigma}]\hat{t}_i \star \hat{E}' \geq \hat{t}' : T$ by commuting of substitution with translation (Lemma 3.36).
$\vdash \mathsf{lift}(T) \star ([\theta; \hat{\sigma}]\hat{t}_i \star \hat{E}') \geq \hat{t}' : T$ as terms are related to their lifting (Lemma 3.34).
The final result is obtained by unfolding the definition of $\geq$.

Subcase:

$$\frac{f \text{ is struct. rec.} \quad \bigcup_i Q_i = \{q_j \searrow \Delta_j; \Gamma_j; T_j; \varphi_j\}_j \qquad \text{for all } i, \quad \Xi, X \vdash (\vec{u}\ (c_i\ x_i)) \searrow \Delta, \overrightarrow{u{:}U}; \Gamma, \Gamma_X, x_i{:}\mathcal{F}_i X\, \vec{C}; T; \varphi, x_i{:=}x_i \Longrightarrow^*_{\overrightarrow{q_i \mapsto t_i}} Q_i \rightsquigarrow \hat{t}_i}{\Delta; \Xi; \Gamma \vdash_f \overrightarrow{q \mapsto t} \rightsquigarrow_\varphi \mathtt{rec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, \mathtt{case}\, x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto \hat{t}_i} : \Pi \overrightarrow{u{:}U}.\mu\mathcal{F}\, \vec{C} \to T}$$

Let $g = \mathtt{rec}\, f_X, \iota_X, \rho_X, \vec{u}, x.\, \mathtt{case}\, x\, \mathtt{of}\, \overrightarrow{\mathtt{in}_i\, x_i \mapsto \hat{t}_i}$.
$E = \vec{C}\ (c_j\ v')\ E''$ by inversion on $E =_\varphi [\theta; \sigma]q_i@E'$.
$\vec{C}\ (\mathtt{in}_\mu\, (\mathtt{in}_j\, \hat{v}'))\ \hat{E}''$ by inversion on the translation for E.
Thus, $g \star \hat{E} \longrightarrow^* [\overrightarrow{C/u}; g/f_X, \sigma_X, (\mathtt{in}_j\, \hat{v}')/x, \hat{v}'/x_j]\hat{t}_j \star \hat{E}''$.
$[\overrightarrow{C/u}; g/f_X, \sigma_X, (\mathtt{in}_j\, \hat{v}')/x, \hat{v}'/x_j]\hat{t}_j\ \hat{E}'' \longrightarrow^* [\theta; g/f_X, \hat{\sigma}, \sigma_X]\bar{t}_i \star \hat{E}'$ by Lemma 3.33.
$\vdash [\theta; g/f_X, \hat{\sigma}, \sigma_X]\bar{t}_i \star \hat{E}' \geq \hat{t}' : T$ by Lemma 3.36.
$\vdash \mathsf{lift}(T) \star ([\theta; g/f_X, \hat{\sigma}, \sigma_X]\bar{t}_i \star \hat{E}') \geq \hat{t}' : T$ by Lemma 3.34.

The proof for the corecursive case is very similar and thus is omitted. □

Corollary 3.38. *If $\vdash t : T$ satisfies the function criteria, then there is a $\vdash_v v : T$ such that $t \longrightarrow^* v$.*

Proof. $\vdash t \rightsquigarrow \hat{t} : T$ for some $\hat{t}$ as function criteria imply translation (Theorem 3.24).
By the progress theorem (2.19), either there is a value v such that $t \longrightarrow^* v$ or there exists an infinite sequence such that $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_i \longrightarrow \ldots$

Case: $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_i \longrightarrow \ldots$

$\vdash t_i \rightsquigarrow \hat{t}_i : T$ for each t_i as evaluation preserves function criteria (Lemma 3.14 and Theorem 3.24).

$\hat{t} \longrightarrow^{n_0} v_0$ and $\hat{t}_i \longrightarrow^{n_i} v_i$ such that $n_0 > n_1 > n_2 > \cdots > n_i > \ldots$ by Theorem 3.37.

As each n_i is a non-negative integer, this sequence cannot be infinite. This is a contradiction.

Case: $t \longrightarrow^* v$

Trivially, we are done. □

3.7 Related Work

The earliest work on termination criteria for copattern languages has been done by Abel and Pientka [2013, 2016]. They enhanced the type system for copatterns from simple types to allow size annotations and quantification over them. This is a powerful extension that can be leveraged to build sophisticated termination metrics. It comes at the cost of a significant overhead in complexity of the type system. In addition, without the existence of inference mechanisms for sizes, a lot of overhead is put on the user to correctly annotate datatypes and proofs with the appropriate sizes. In contrast, our approach offers more limited termination criteria that will accept a more limited set of programs (which we still deem acceptable) while being mostly transparent for the user.

The design behind our dependent copattern matching expands on the work of Coquand [1992] and Goguen et al. [2006a], via the extension to copatterns. The methodology of Goguen et al. [2006a] in particular inspired our development. They build splitting trees which bear close resemblance with our coverage derivations. They translate those trees into a core calculus extending UTT [Luo, 1994] which has primitive elimination operators for inductive types. This calculus is assumed to be terminating and they prove the translation to be normalisation preserving. The translation relies crucially on dependent types to build course-of-values recursors to translate pattern matching functions into. The power of the type system of their target language allows them a more direct translation that directly

commutes with evaluation.

The approach of proving termination of copattern languages using splitting trees has also been used by Cockx and Abel [2018, 2020] for the Agda language. Motivated by specific implementation concerns, they focus strictly on the elaboration to valid splitting trees, but do not translate their proof trees into a normalizing calculus. This results in a robust elaboration procedure and answers some bugs in the prior Agda implementation. Our approach does not focus on implementation concerns and instead focuses on a theoretical proof of the validity of our criteria.

Our copattern language is designed to be as close as possible to a surface language a user would program in. By contrast, Basold and Geuvers [2016] built a minimal language for dependent (co)inductive types based on dialgebras. Constructs such as products, sums, Π-types, and Σ-types are shown to be derivable from the (co)inductive types. The language is also shown to be normalizing. While the focus of the two approaches seems to be opposite, we believe the expressivity between dependent copatterns and their type theory to be similar. Their language could potentially serve as a suitable core calculus for dependent copatterns to translate into.

Our core calculus implements coinductive types via a corecursion scheme first designed by Mendler [1987, 1991]. Mendler-style recursion schemes for term-indexed languages have been investigated by Ahn [2014]. Ahn describes an extension of System F_ω with erasable term indices, called F_i. He combines this with fixed points of type operators, as in the Fix_ω language by Abel and Matthes [2004], to produce the core language Fix_i. In Fix_i, one can embed Mendler-style recursion over term-indexed data types by Church-style encodings.

Normalisation of a calculus with Mendler-style recursors has also been done by Abel [2004]. Abel uses subtyping ordering to define programs with course-of-value recursion instead of using explicit term-level operators like we do. This simplifies the writing of terms at the cost of a more sophisticated type system. We took inspiration from his normalisation proof, in particular in the representation of a semantics for (co)inductive types.

3.8 Conclusion

This chapter described a terminating subset of our copattern language by the identification of a set of statically checked criteria: type checking, coverage checking, and structural (co)recursiveness checking. We designed a core calculus with Mendler-style corecursors and proved it to be normalizing using a logical relation argument. We defined a (partial) translation from our copattern language into this core calculus. We showed that the criteria we chose were sufficient for the translation to succeed. We also showed that the translation is normalisation preserving. We did that by binding the number of steps a program takes in the source language by the number of steps its translation takes. We established such bound using a simulation relation on terms.

In the following chapter we present a case study of coinductive proofs in our prototype implementation of indexed copatterns: Beluga. The case study goes over Howe's method that is used to show that a simulation is a precongruence. We leverage Beluga's advanced support for open terms and substitutions to carry out such proof.

Chapter 4

Case Study: Howe's Method

Logical frameworks such as LF [Harper et al., 1993] and λProlog [Miller and Nadathur, 2012] provide a meta-language for representing formal systems given via axioms and inference rules, factoring out common and recurring issues such as modelling variable bindings. They exploit an idea, dating back to Church, where we use a lambda calculus as the meta-language to uniformly encode variable binding in our formal system. This technique is now commonly known as higher-order abstract syntax (HOAS) or (in a slightly weaker setting) λ-tree syntax [Miller and Palamidessi, 1999]. In particular, we can encode uniformly variable binding operators by mapping them to the lambda binder of the meta-language. As a consequence variables in the object language are represented by variables in the meta-language and inherit thereby α-renaming and substitution from it. Moreover, this encoding technique scales to representing formal systems that use hypothetical and parametric reasoning by providing generic support for managing hypotheses and the corresponding substitution lemmas. As users do not need to build up all this basic mathematical infrastructure, it is easier to prototype proof environments and mechanize formal systems. It can also have substantial benefits for proof checking and proof search.

While representing formal systems is a first step, the interesting question is how we can *reason* about HOAS representations (co)inductively. Meta-languages such as LF or λ-Prolog that are used for representing object language are weak calculi and do not include

case analysis, recursion or (co)inductive definitions. This is in fact essential to achieve an adequate representation of the object language where each object language term uniquely corresponds to a given representation in the meta-language. So, how can we still reason about such representations?

One solution to this conundrum is the so-called "two-level" approach, as advocated by McDowell and Miller [1997], where we distinguish between a specification language and a reasoning logic above it, which supports at least some form of induction. The cited paper presented $FOLD^{\mathcal{N}}$, which is basically a first order logic with *definitions* (fixed points) and natural number induction. Object logics are encoded in a specification language, which may vary and often is based on (possibly sub-structural) fragments of hereditary Harrop formulas. The method was tested on classical benchmarks such as subject reduction for PCF and its imperative variants.

Notably, one of Dale Miller's motivating examples has been the (meta)theory of process calculi, in particular the π-calculus. This brought to the forefront the issue of representing and reasoning about *infinite* behaviour. In fact, McDowell et al. [1996] were concerned with the representation of transition systems and their bisimulation: in agreement with Milner's original presentation in *A Calculus of Communicating Systems*, bisimulation was captured *inductively* by computing the greatest fixed point starting from the universal relation and closing downwards by intersection. This is doable, but notoriously awkward to work with and in fact Milner swiftly adopted *coinduction* in his subsequent *Communication and Concurrency*.

In the late 1990, several reasonably large case studies involving coinduction were carried out in Isabelle/HOL and Coq, not without some difficulties [Ambler and Crole, 1999, Honsell et al., 2001, Hirschkoff, 1997]. These case studies further demonstrated the challenges in modelling variable bindings and building up such an infrastructure, as lambda-tree syntax is fundamentally incompatible with the foundations of these proof systems. It turns out instead that it is quite natural to step from $FOLD^{\mathcal{N}}$ to support (co)inductive reasoning; Momigliano and Tiu [2003] adopted the view of definitions as least and greatest fixed points adding rules for fixed point induction. This was later shown to be consistent in [Tiu and Momigliano,

2012]. With the orthogonal ingredient of ∇-quantifier to abstract over variable names [Miller and Tiu, 2005], this line of research culminated in the Abella proof assistant [Baelde et al., 2014a], which until recently was, in fact, the only proof assistant supporting natively both HOAS and coinduction, as exemplified in some non-insignificant case studies [Tiu and Miller, 2010, Momigliano, 2012].

The other main player in HOAS logical frameworks is LF [Harper et al., 1993]: Pfenning advocated using it as a *meta*-logical framework by representing inductive proofs as relations. To ensure that a relation describes a valid inductive proof, external checks guarantee that the implemented relation constitutes a total function, i.e. covers all cases and all appeals to the induction hypothesis are well-founded. This lead to the proof environment Twelf [Pfenning and Schürmann, 1999], which has been used widely, see for a major case study [Lee et al., 2007]. However, Twelf did not seem to lend itself to coinductive reasoning.

To address these and other shortcomings, Pientka [2008] designed a reasoning logic on top of LF that allows us to directly analyze and manipulate LF objects. Beluga [Pientka and Dunfield, 2010] implements this idea. To model derivation trees that depend on assumptions, LF objects are paired with their surrounding context [Nanevski et al., 2008, Pientka, 2008, Pientka and Dunfield, 2008]. Inductive proofs are then implemented as *recursive* functions that directly pattern match on contextual LF objects. Beluga provides a proof language that makes explicit context reasoning via built-in contexts and simultaneous substitutions together with their equational theory. Moreover, it supports (co)inductive and stratified definitions in addition to higher-order functions [Cave and Pientka, 2012, Pientka and Cave, 2015, Thibodeau et al., 2016, Jacob-Rao et al., 2018], thereby going substantially beyond the expressive power of Twelf.

One might say that the proof and the type-theoretic approaches are converging towards a core reasoning logic that supports least and greatest fixed points and equality within first-order logic. (Co)pattern matching in a program is interpreted as case analysis in a proof and recursive calls on structurally smaller objects or guarded by observations correspond valid appeals to the induction or coinduction hypothesis, respectively.

As a contribution to a better understanding of the relationship between the logical and

computational interpretation of coinductive proofs, this chapter reappraises the proof that similarity in the call-by-name lambda calculus with lists is a pre-congruence using Howe's method [Howe, 1996]. This is a challenging proof since it requires a combination of inductive and coinductive reasoning on open terms. We mechanize this proof in Beluga which was extended to serve as a prototype for the theory of Chapter 2 where the index domain is instantiated by contextual LF. This mechanization relies on three key ingredients:

1. we give a HOAS encoding of lambda-terms together with their operational semantics as *intrinsically typed* terms, thereby avoiding not only the need to deal with binders, renaming and substitutions, but keeping all typing invariants implicit;
2. we take advantage of Beluga's support for representing open terms using built-in contexts and simultaneous substitutions: this allows us to directly state a notion such as open simulation without resorting to the usual inductive closure operation and to encode neatly notoriously painful proofs such as the substitutivity of the Howe relation;
3. we exploit the possibility of reasoning by coinduction in Beluga's reasoning logic.

The end result is, in our opinion, succinct and elegant, thanks to the high-level abstractions and primitives Beluga provides.

The paper starts in Section 4.1 with a summary description of Howe's method and discusses the challenges that it poses to its mechanization. The latter is detailed in Section 4.2, together with a discussion on coinductive Beluga, a proof of adequacy of our encoding of similarity, and a example derivation of two terms being similar. We review related work in Section 4.3 and conclude in Section 4.4.

The entire formal development can be retrieved from `https://github.com/Beluga-lang/Beluga/tree/master/examples/codatatypes/howes-method`.

4.1 A summary of Howe's method

First let us fix our programming languages as the simply-typed λ-calculus with recursion over (lazy) lists, which we call PCFL following Pitts [1997]. Its types consist of the unit type

(written as $\top$), function types, and lists (written as $\mathsf{list}(\tau)$).

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \top \mid \tau \to \tau \mid \mathsf{list}(\tau) \\
\text{Terms} & m, n, p, q & ::= & x \mid \mathsf{lam}\, x.\, p \mid m_1\ m_2 \mid \mathsf{fix}\ x.\ m \mid \langle\rangle \\
 & & & \mid \mathsf{nil} \mid \mathsf{cons}\ m_h\ m_t \mid \mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p\} \\
\text{Values} & v & ::= & \langle\rangle \mid \mathsf{lam}\, x.\, p \mid \mathsf{nil} \mid \mathsf{cons}\ m_h\ m_t
\end{array}
$$

The typing rules for PCFL and the big step lazy operational semantics denoted by $m \Downarrow v$ are standard and we omit them here. In particular, lists are only evaluated lazily, as the definition of values shows. The interested reader can skip ahead to their encoding in LF in the Section 4.2 or consult Pitts [1997].

Proving bisimilarity a congruence using Howe's method

Suppose we want to say when two programs (two closed terms) have the same behavior. A well known characterization is Morris-style *contextual equivalence*: occurrences of the first expression in any program can be replaced by the second without affecting the *observable* results of executing the program

While this notion of program equivalence is intuitive, it is indeed difficult to reason about it, mainly due to the quantification on every possible context.[1] Many techniques have been proposed through the years, ranging from domain theory [Abramsky, 1991], game semantics [Ghica and McCusker, 2000] to logical relations [Ahmed, 2006]. The idea of *bisimilarity* has usefully been adapted from concurrency theory to provide yet another characterization of contextual equivalence. Bisimilarity is, similarly to contextual equivalence, parametrized by the notion of *observable* we select: roughly, m and n are *bisimilar* if whenever m evaluates to an observable so does n, and all the subprograms of those are also bisimilar. In the case of *applicative* bisimilarity, evaluation at function type is pushed until values are reached.

[1] This notion can and has been simplified, starting from Milner's context lemma [Milner, 1977] and going through the CIU theorem [Mason and Talcott, 1991]. Some mechanizitions are also available [Ambler and Crole, 1999, Ford and Mason, 2003, McLaughlin et al., 2018], as we discuss further in Section 4.3.

To simplify the presentation, we will concentrate on the notion of *similarity*, from which bisimilarity can be obtained by symmetry, that is taking the conjunction of similarity and its inverse; this is possible thanks to determinism of evaluation.

Definition 1 (Applicative simulation)**.** An *applicative simulation* is a family of typed relations R_τ on closed terms satisfying the following conditions:

- if $m\ \mathcal{R}_\top\ n$ then $m \Downarrow \langle\rangle$ entails $n \Downarrow \langle\rangle$.
- if $m\ \mathcal{R}_{\mathsf{list}(\tau)}\ n$ then $m \Downarrow \mathsf{nil}$ entails $n \Downarrow \mathsf{nil}$.
- if $m\ \mathcal{R}_{\mathsf{list}(\tau)}\ n$ then $m \Downarrow \mathsf{cons}\ m_h\ m_t$ entails that there are terms n_h and n_t such that $n \Downarrow \mathsf{cons}\ n_h\ n_t$ for which $m_h\ \mathcal{R}_\tau\ n_h$ and $m_t\ \mathcal{R}_{\mathsf{list}(\tau)}\ n_t$.
- if $m\ \mathcal{R}_{\tau\to\tau'}\ n$ then $m \Downarrow \mathsf{lam}\, x.\, m'$ entails that there is a term n' such that $n \Downarrow \mathsf{lam}\, y.\, n'$ for which $m'[r/x]\ \mathcal{R}_{\tau'}\ n'[r/y]$ for every term r of type τ.

We can make sense of the non-well-founded nature of the last two conditions by noting that there are *non-empty* simulations, e.g. the identity relation and that the *union* of two applicative simulations is still a simulation. Hence there exists the *largest* one, which we call applicative *similarity*. This relation can also be characterized using the Knaster-Tarski fixed point theorem, as the greatest fixed point of an appropriate endofunction Φ on families of typed relations. For a detailed explanation we refer again to Pitts [1997], and we just mention that the definition of the function follows the simulation relation, and has, for example at $\tau \to \tau'$, $m\ \Phi(R_{\tau\to\tau'})\ n$ just in case whenever $m \Downarrow \mathsf{lam}\, x.\, m'$ for any m', there exists a term n' such that $n \Downarrow \mathsf{lam}\, y.\, n'$ and for every r of type τ, $m'[r/x]$ is $R_{\tau'}$-related to $n'[r/y]$; hence, similarity is the set *coinductively* defined by Φ, a relation we write as $m \preccurlyeq_\tau n$. This yields a co-induction principle that we describe first in its generality and below we show it instantiated to applicative similarity.

$$\frac{\exists S \text{ s.t. } a \in S \qquad S \subseteq \Phi(S)}{a \in \mathrm{gfp}(\Phi)}\, CI$$

$$\frac{\exists S_\tau \text{ s.t. } m\ S_\tau\ n \qquad S_\tau \text{ is an applicative simulation}}{m \preccurlyeq_\tau n}\, CI{-}\preceq$$

It is not difficult to show that similarity is a pre-order and we detail the proof of reflexivity using rule $CI-\preceq$ to highlight the similarities with the type-theoretic definition based instead on the notion of observation, on which our mechanization relies.

Theorem 4.1 (Reflexivity of applicative similarity)**.** $\forall m\ \tau, m \preccurlyeq_\tau m$.

Proof. To show the result we need to provide an appropriate simulation S and check the simulation conditions. Just choose S_τ to be the family $\{(m, m) \mid \cdot \vdash m : \tau\}$ where we use the judgment $m : \tau$ to say that term m has type σ.

We then consider each case in the applicative simulation definition.

- if $m\ \mathcal{S}_\top\ m$, then $m \Downarrow \langle\rangle$ entails $m \Downarrow \langle\rangle$: immediate;
- if $m\ \mathcal{S}_{\mathsf{list}(\tau)}\ m$, then $m \Downarrow \mathsf{nil}$ entails $m \Downarrow \mathsf{nil}$: immediate;
- assume $m\ \mathcal{S}_{\mathsf{list}(\tau)}\ m$ and $m \Downarrow \mathsf{cons}\ m_h\ m_t$; pick n_h, m_t to be m_h, m_t and by the definition of the simulation, it holds that $m_h\ \mathcal{S}_\tau\ n_h$ and $m_t\ \mathcal{S}_{\mathsf{list}(\tau)}\ n_t$;
- assume $m\ \mathcal{S}_{\tau\to\tau'}\ m$ and $m \Downarrow \mathsf{lam}\, x.\, m'$; again by picking m' for n' and by the definition of the simulation it is obvious that for every $r{:}\tau$, $[r/x]m'\ \mathcal{S}_{\tau'}\ [r/x]m'$.

□

In many cases, we do not have to look much further beyond the statement of the theorem to come up with an appropriate simulation, i.e. we can read off the definition of simulation from it — and this is indeed the case for all the coinductive proofs in the following development. However, to show the equivalence of specific programs we may have to come up with a complex bisimulation, possibly defined inductively and/or "up to". This phenomenon is well-known in inductive theorem proving, where sometimes the induction hypothesis coincides with the statement of the theorem, but in other cases it needs to be generalized in an appropriate lemma. The fixed point rules conflate those two aspects, generalization and lemma application, in one go. With an abuse of language, we will say that we prove a statement by coinduction and say that we appeal to the use of the "coinductive hypothesis" when the simulation corresponds to the statement of the theorem.

When dealing with program equivalence, *equational* (in addition to coinductive) reasoning would be helpful and this is why it is crucial to establish bisimilarity to be a *congruence*, i.e. a relation respecting the way terms are constructed. Since in this paper we restrict ourselves to similarity, we target *pre*-congruence. Given the presence of variable-binding operators, we need to consider relations over *open* terms, that is families of relations over terms indexed by a typing context Γ in addition to a type τ, which we write as $\Gamma \vdash m\ \mathcal{R}_\tau\ n$.

Definition 2 (Compatible relation)**.** A relation $\Gamma \vdash m\ \mathcal{R}_\tau\ n$ is *compatible* when:

(C0) $\Gamma \vdash \langle\rangle\ \mathcal{R}_\top\ \langle\rangle$;

(C1) $\Gamma, x{:}\tau \vdash x\ \mathcal{R}_\tau\ x$;

(C2) $\Gamma, x{:}\tau \vdash m\ \mathcal{R}_{\tau'}\ n$ entails $\Gamma \vdash (\mathsf{lam}\, x.\, m)\ \mathcal{R}_{\tau\to\tau'}\ (\mathsf{lam}\, x.\, n)$;

(C3) $\Gamma \vdash m_1\ \mathcal{R}_{\tau\to\tau'}\ n_1$ and $\Gamma \vdash m_2\ \mathcal{R}_\tau\ n_2$ entail $\Gamma \vdash (m_1\ m_2)\ \mathcal{R}_{\tau'}\ (n_1\ n_2)$;

(C4) $\Gamma, x{:}\tau \vdash m\ \mathcal{R}_\tau\ n$ entails $\Gamma \vdash (\mathsf{fix}\ x.\ m)\ \mathcal{R}_\tau\ (\mathsf{fix}\ x.\ n)$;

(C5a) $\Gamma \vdash m_h\ \mathcal{R}_\tau\ n_h$ and $\Gamma \vdash m_t\ \mathcal{R}_{\mathsf{list}(\tau)}\ n_t$ entail $\Gamma \vdash (\mathsf{cons}\ m_h\ m_t)\ \mathcal{R}_{\mathsf{list}(\tau)}\ (\mathsf{cons}\ n_h\ n_t)$;

(C5b) $\Gamma \vdash \mathsf{nil}\ \mathcal{R}_{\mathsf{list}(\tau)}\ \mathsf{nil}$;

(C6) $\Gamma \vdash m_1\ \mathcal{R}_{\mathsf{list}(\tau)}\ m_2$, $\Gamma \vdash n_1\ \mathcal{R}_{\tau'}\ n_2$ and $\Gamma, h_d{:}\tau, t_l{:}\mathsf{list}(\tau) \vdash p_1\ \mathcal{R}_{\tau'}\ p_2$ entail $\Gamma \vdash (\mathsf{lcase}\ m_1\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n_1 \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p_1\})\ \mathcal{R}_{\tau'}\ (\mathsf{lcase}\ m_2\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n_2 \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p_2\})$.

Definition 3 (Pre-congruence)**.** A *pre-congruence* is a compatible transitive relation.

By the very definition of simulation at arrow type it is clear that a key property for our development is for a relation to be preserved by pairwise substitution:

$$\Gamma, y{:}\tau \vdash m_1\ \mathcal{R}_{\tau'}\ m_2 \text{ and } \Gamma \vdash n_1\ \mathcal{R}_\tau\ n_2 \text{ entails } \Gamma \vdash [n_1/y]m_1\ \mathcal{R}_{\tau'}\ [n_2/y]m_2.$$

We generalize this property here using *simultaneous* substitutions, streamlining our formal development, see Section 4.2. Figure 4.1 gives rules for well-typed simultaneous substitutions $\Psi \vdash \sigma : \Gamma$, where σ replaces variables in Γ with terms typable in Ψ; then, we state when two such substitutions are $\mathcal{R}$-related:

Well-Typed Simultaneous Substitutions: $\Psi \vdash \sigma : \Gamma$

$$\frac{}{\Psi \vdash \cdot : \cdot} \qquad \frac{\Psi \vdash \sigma : \Gamma \qquad \Psi \vdash m : \tau}{\Psi \vdash \sigma, m/x : \Gamma, x : \tau}$$

Related Simultaneous Substitutions: $\Psi \vdash \sigma_1 \ \mathcal{R}_\Gamma \ \sigma_2$

$$\frac{}{\Psi \vdash \cdot \ \mathcal{R}_\cdot \ \cdot} \qquad \frac{\Psi \vdash \sigma_1 \ \mathcal{R}_\Gamma \ \sigma_2 \qquad \Psi \vdash m \ \mathcal{R}_\tau \ n}{\Psi \vdash (\sigma_1, m/x) \ \mathcal{R}_{\Gamma, x:\tau} \ (\sigma_2, n/x)}$$

Figure 4.1: (Related) simultaneous substitutions

Definition 4 (Substitutive relation)**.** A relation is *substitutive* **(Sub)** iff $\Gamma \vdash m_1 \ \mathcal{R}_\tau \ m_2$ and $\Psi \vdash \sigma_1 \ \mathcal{R}_\Gamma \ \sigma_2$ entails $\Psi \vdash [\sigma_1]m_1 \ \mathcal{R}_\tau \ [\sigma_2]m_2$.

Some other properties are admissible:

Lemma 4.2 (Elementary admissible properties)**. (Ref)** *If a relation is compatible, then it is* reflexive*; further, if* $\Gamma \vdash m \ \mathcal{R}_\tau \ m$*, then* $\Psi \vdash \sigma \ \mathcal{R}_\Gamma \ \sigma$*.*

(Cus) *If* $\mathcal{R}_\tau$ *is substitutive and reflexive, then it is also* closed under substitution*:* $\Gamma \vdash m_1 \ \mathcal{R}_\tau \ m_2$ *and* $\Psi \vdash \sigma : \Gamma$ *entails* $\Psi \vdash [\sigma]m_1 \ \mathcal{R}_\tau \ [\sigma]m_2$*.*

Weakening (**Wkn**), that is $\Gamma \vdash m \ \mathcal{R}_\tau \ n$ and $\Gamma \subseteq \Gamma'$ entailing $\Gamma' \vdash m \ \mathcal{R}_\tau \ n$, follows from (Cus) taking σ to be the identity substitution for Γ.

The definition of similarity applies only to closed terms. It is therefore customary to *extend* similarity to *open terms* via substitution. We do this using *grounding* substitutions:

Definition 5 (Open similarity)**.** $\Gamma \vdash m \preccurlyeq^\circ_\tau m'$ iff $[\sigma]m \preccurlyeq_\tau [\sigma]m'$ for any $\cdot \vdash \sigma : \Gamma$.

Now, it is immediate that open similarity is a pre-order and hence (C1) and transitivity hold. Further, (C2) also holds, since similarity satisfies

$$\mathsf{lam}\, x.\, m \preccurlyeq_{\tau\to\tau'} \mathsf{lam}\, x.\, n \text{ iff for all } p{:}\tau,\ [p/x]m \preccurlyeq_{\tau'} [p/x]n$$

$$\frac{\Gamma \vdash \langle\rangle \preccurlyeq^{\circ}_{\top} n}{\Gamma \vdash \langle\rangle \preccurlyeq^{\mathcal{H}}_{\top} n}\ hu \qquad \frac{\Gamma \vdash \mathsf{nil} \preccurlyeq^{\circ}_{\mathsf{list}(\tau)} n}{\Gamma \vdash \mathsf{nil} \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} n}\ hnil \qquad \frac{\Gamma, x{:}\tau \vdash x \preccurlyeq^{\circ}_{\tau} n}{\Gamma, x{:}\tau \vdash x \preccurlyeq^{\mathcal{H}}_{\tau} n}\ hvar$$

$$\frac{\Gamma, x{:}\tau \vdash m \preccurlyeq^{\mathcal{H}}_{\tau'} m' \qquad \Gamma \vdash \mathsf{lam}\, x.\, m' \preccurlyeq^{\circ}_{\tau\to\tau'} n}{\Gamma \vdash \mathsf{lam}\, x.\, m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} n}\ hlam$$

$$\frac{\Gamma, x{:}\tau \vdash m \preccurlyeq^{\mathcal{H}}_{\tau} m' \qquad \Gamma \vdash \mathsf{fix}\ x.\ m' \preccurlyeq^{\circ}_{\tau} n}{\Gamma \vdash \mathsf{fix}\ x.\ m \preccurlyeq^{\mathcal{H}}_{\tau} n}\ hfix$$

$$\frac{\Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} m_1' \quad \Gamma \vdash m_2 \preccurlyeq^{\mathcal{H}}_{\tau} m_2' \qquad \Gamma \vdash m_1'\ m_2' \preccurlyeq^{\circ}_{\tau'} n}{\Gamma \vdash m_1\ m_2 \preccurlyeq^{\mathcal{H}}_{\tau'} n}\ happ$$

$$\frac{\Gamma \vdash m_h \preccurlyeq^{\mathcal{H}}_{\tau} m_h' \quad \Gamma \vdash m_t \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} m_t' \qquad \Gamma \vdash \mathsf{cons}\ m_h'\ m_t' \preccurlyeq^{\circ}_{\mathsf{list}(\tau)} n}{\Gamma \vdash \mathsf{cons}\ m_h\ m_t \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} n}\ hcons$$

$$\frac{\begin{array}{c}\Gamma \vdash m \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} m' \quad \Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\tau'} m_1' \quad \Gamma, h_d{:}\tau, t_l{:}\mathsf{list}(\tau) \vdash m_2 \preccurlyeq^{\mathcal{H}}_{\tau'} m_2' \\ \Gamma \vdash \mathsf{lcase}\ m'\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow m_1' \mid \mathsf{cons}\ h_d\ t_l \Rightarrow m_2'\} \preccurlyeq^{\circ}_{\tau'} n\end{array}}{\Gamma \vdash \mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow m_1 \mid \mathsf{cons}\ h_d\ t_l \Rightarrow m_2\} \preccurlyeq^{\mathcal{H}}_{\tau'} n}\ hlcase$$

Figure 4.2: Definition of the Howe relation

However, a direct attempt to prove pre-congruence of open similarity breaks down when dealing with (C3) and (C6). Moreover, while it follows simply by construction that open similarity is closed under substitution, it is not obvious that it is substitutive.

Howe's idea [Howe, 1996] was to introduce a *candidate* relation $\preccurlyeq^{\mathcal{H}}_{\tau}$ (see Figure 4.2), which contains (open) similarity and can be shown to be almost a substitutive pre-congruence, and then to prove that it does coincide with similarity.

The informal proof consists of several lemmata:

(1) Semi-transitivity: the composition of the Howe relation with open similarity is contained in the former. The proof goes by case analysis on the derivation of the Howe relation using transitivity of open similarity.

(2) The Howe relation is reflexive. Induction on typing, using reflexivity of open similarity.

(3) Compatibility: (C0)—(C6) hold, an easy consequence of (2).

(4) Open similarity is contained in Howe, which follows immediately from (1) and (2).

(5) The Howe relation is substitutive, see Lemma 4.7.

(6) The Howe relation "mimics" the simulation conditions:

(6.1) If $\langle\rangle \preccurlyeq^{\mathcal{H}}_{\top} n$, then $n \Downarrow \langle\rangle$.

(6.2) If $\mathsf{nil} \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} n$, then $n \Downarrow \mathsf{nil}$.

(6.3) If $\mathsf{lam}\, x.\, m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} n$, then $n \Downarrow \mathsf{lam}\, x.\, m'$ and for every $q{:}\tau$ we have $[q/x]m \preccurlyeq^{\mathcal{H}}_{\tau'} [q/x]m'$.

(6.4) If $\mathsf{cons}\ m_h\ m_t \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} n$, then $n \Downarrow \mathsf{cons}\ p_h\ p_t$, with $m_h \preccurlyeq^{\mathcal{H}}_{\tau} p_h$ and $m_t \preccurlyeq^{\mathcal{H}}_{\mathsf{list}(\tau)} p_t$.

By inversion on the Howe relation and definition of similarity, using semi-transitivity and, in the lambda-case, substitutivity of the Howe relation.

(7) Downward closure: if $p \preccurlyeq^{\mathcal{H}}_{\tau} q$ and $p \Downarrow v$, then $v \preccurlyeq^{\mathcal{H}}_{\tau} q$. Induction on evaluation, and inversion on the Howe relation and similarity, with an additional case analysis on v.

(8) $p \preccurlyeq^{\mathcal{H}}_{\tau} q$ entails $p \preccurlyeq_{\tau} q$. By coinduction, using the coinductive hypothesis, point (6) and (7).

Once all of these properties have been proved, we are ready for the main result, stating that the Howe relation coincides with applicative similarity, and hence the pre-congruence of the latter follows as a corollary:

Theorem 4.3. $\Gamma \vdash p \preccurlyeq^{\mathcal{H}}_{\tau} q$ *iff* $\Gamma \vdash p \preccurlyeq^{\circ}_{\tau} q$

Proof. Right to left is point (4) above. Conversely, proceed by induction on Γ using (8) for the base case and closure under substitution for the step. □

Corollary 4.4. *Open similarity is a pre-congruence.*

On the role of substitutions in Howe's method

Substitutions play a central role in the overall proof that similarity is a pre-congruence. In the informal proof, we silently exploit equational laws about substitution; however, they can cause significant trouble during mechanization. We summarize the definition of substitution for our term language together with its equational theory in Figure 4.3. To illustrate how we rely on these substitution properties in proofs, we show here in more detail the proof of substitutivity and pay particular attention to the properties in Figure 4.3. Recall that the definition of $\Psi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$ is just an instance of the definition of related simultaneous substitutions.

Lemma 4.7 (Substitutivity of the Howe relation)**.** *Suppose we have* $\Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\tau} m_2$ *and* $\Psi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$*; then* $\Psi \vdash [\sigma_1]m_1 \preccurlyeq^{\mathcal{H}}_{\tau} [\sigma_2]m_2$*.*

Proof. By induction on the derivations of $\Gamma \vdash m_1 \preccurlyeq^{\mathcal{H}}_{\tau} m_2$.

Case $\dfrac{\Gamma \vdash m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} m' \quad \Gamma \vdash n \preccurlyeq^{\mathcal{H}}_{\tau} n' \qquad \Gamma \vdash m'\ n' \preccurlyeq^{\circ}_{\tau'} r}{\Gamma \vdash m\ n \preccurlyeq^{\mathcal{H}}_{\tau'} r}\ \mathit{happ}$

$\Psi \vdash [\sigma_1]m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} [\sigma_2]m'$	by IH on first subderivation
$\Psi \vdash [\sigma_1]n \preccurlyeq^{\mathcal{H}}_{\tau} [\sigma_2]n'$	by IH on second subderivation
$\Psi \vdash [\sigma_2](m'\ n') \preccurlyeq^{\circ}_{\tau'} [\sigma_2]r$	by cus on third subderivation
$[\sigma_2](m'\ n') = [\sigma_2]m'\ [\sigma_2]n'$	by def. of substitution (see Figure 4.3)
$\Psi \vdash [\sigma_1]m\ [\sigma_1]n \preccurlyeq^{\mathcal{H}}_{\tau'} [\sigma_2]r$	by rule *happ*
$\Psi \vdash [\sigma_1](m\ n) \preccurlyeq^{\mathcal{H}}_{\tau'} [\sigma_2]r$	by above line

Case $\dfrac{\Gamma, x{:}\tau \vdash m \preccurlyeq^{\mathcal{H}}_{\tau'} m' \qquad \Gamma \vdash \mathsf{lam}\, x.\, m' \preccurlyeq^{\circ}_{\tau\to\tau'} r}{\Gamma \vdash \mathsf{lam}\, x.\, m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} r}\ \mathit{hlam}$

$\Psi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$	by assumption
$\Psi, x : \tau \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$	by weakening (Lemma 4.5.2)
$[\sigma]x \preccurlyeq_{\tau} [\sigma]x$ for any σ where $\cdot \vdash \sigma : \Psi, x{:}\tau$	by reflexivity of similarity (Theorem 4.1
$\Psi, x{:}\tau \vdash x \preccurlyeq^{\circ}_{\tau} x$	by def. of open similarity

Equational Theory of Simultaneous Substitution

For $\Gamma' \vdash \sigma : \Gamma$ and $\Gamma \vdash m : \tau$, we define $[\sigma]m$ as follows

$$
\begin{array}{ll}
[\sigma]x & = \sigma(x) \\
[\sigma](\mathsf{lam}\, x.\, m) & = \mathsf{lam}\, x.\, [\sigma, x/x]m \\
[\sigma](m\ n) & = [\sigma]m\ [\sigma]n \\
[\sigma](\langle\rangle) & = \langle\rangle \\
[\sigma](\mathsf{nil}) & = \mathsf{nil} \\
[\sigma](\mathsf{cons}\ m\ n) & = \mathsf{cons}\ [\sigma]m\ [\sigma]n \\
[\sigma](\mathsf{fix}\ x.\ m) & = \mathsf{fix}\ x.\ [\sigma, x/x]m \\
[\sigma](\mathsf{lcase}\ m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow n \mid \mathsf{cons}\ h_d\ t_l \Rightarrow p\}) & \\
\qquad = \mathsf{lcase}\ [\sigma]m\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow [\sigma]n \mid \mathsf{cons}\ h_d\ t_l \Rightarrow [\sigma, h_d/h_d, t_l/t_l]p\} & \\
\\
[\sigma_2](\cdot) & = \cdot \\
[\sigma_2](\sigma_1, m/x) & = [\sigma_2]\sigma_1,\ [\sigma_2]m/x
\end{array}
$$

Lemma 4.5 (Substitution lemma and weakening property)**.**

1. *If* $\Gamma' \vdash \sigma : \Gamma$ *and* $\Gamma \vdash m : \tau$ *then* $\Gamma' \vdash [\sigma]m : \tau$.
2. *If* $\Gamma' \vdash \sigma : \Gamma$ *then* $\Gamma', y{:}\tau \vdash \sigma : \Gamma$.

Lemma 4.6 (Substitution properties)**.**

1. $[\sigma, n/x]m = [n/x]([\sigma, x/x]m)$
2. $[\sigma', n/x]\sigma = [n/x]([\sigma', x/x]\sigma)$
3. $[\sigma_2]([\sigma_1]m_1) = [[\sigma_2]\sigma_1]m$
4. $[\sigma_2]([\sigma_1]\sigma) = [[\sigma_2]\sigma_1]\sigma$
5. *Let* $\mathsf{id} = x_1/x_1, \ldots, x_n/x_n$ *be the identity substitutions for* $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$*, then* $[\mathsf{id}]m = m$ *and* $[\mathsf{id}]\sigma = \sigma$*. Moreover,* $[\sigma]\mathsf{id} = \sigma$*. A special case is when* $\Gamma = \cdot$*. In this case we have* $\mathsf{id} = \cdot$*. Moreover,* $[\cdot]m = m$*,* $[\cdot]\sigma = \sigma$*, and* $[\sigma]\cdot = \cdot$.

Figure 4.3: Properties of simultaneous substitutions

$\Psi, x{:}\tau \vdash x \preccurlyeq^{\mathcal{H}}_{\tau} x$ by rule *hvar*

$\Psi, x{:}\tau \vdash \sigma_1, x/x \preccurlyeq^{\mathcal{H}}_{\Gamma, x:\tau} \sigma_2, x/x$ by def. of Howe related substitutions

$\Psi, x{:}\tau \vdash [\sigma_1, x/x]m \preccurlyeq^{\mathcal{H}}_{\tau'} [\sigma_2, x/x]m'$	by IH on first subderivation
$\Psi \vdash [\sigma_2](\mathsf{lam}\, x.\, m') \preccurlyeq^{\circ}_{\tau\to\tau'} [\sigma_2]r$	by cus on second subderivation
$[\sigma_2](\mathsf{lam}\, x.\, m') = \mathsf{lam}\, x.\, [\sigma_2, x/x]m'$	by def. of substitution (see Figure 4.3)
$\Psi \vdash (\mathsf{lam}\, x.\, [\sigma_1, x/x]m) \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} [\sigma_2]r$	by rule *hlam*
$[\sigma_1](\mathsf{lam}\, x.\, m) = \mathsf{lam}\, x.\, [\sigma_1, x/x]m$	by def. of substitution (see Figure 4.3)
$\Psi \vdash [\sigma_1](\mathsf{lam}\, x.\, m) \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} [\sigma_2]r$	by above line

The other cases are analogous. □

4.2 Mechanizing Howe's method in Beluga

We discuss in this section the proof that similarity in PCFL is a pre-congruence using Howe's method in Beluga.

Beluga is a programming environment that supports both specifying formal systems and reasoning about them. To specify formal systems such as PCFL we use the logical framework LF. This allows us to take advantage of higher-order abstract syntax. A key challenge when reasoning about LF objects is that we must consider potentially open objects. In Beluga, this need is met by viewing all LF objects together with the contexts in which they are meaningful [Nanevski et al., 2008] as *contextual* LF objects and by abstracting not only over LF objects but also over contexts. We then view contextual objects and contexts as the particular index domain about which we can reason using the methodology described in Chapter 2. Under the propositions-as-types principle this logic corresponds to a functional language with indexed (co)inductive types that supports (co)pattern matching. Meta-theoretic proofs about formal systems are implemented as (co)recursive functions in Beluga.

An Overview of Beluga

A Beluga signature consists of LF declarations, inductive and coinductive definitions, and programs. For each LF type family `a` we declare its LF kind together with the constants that

allow us to form objects that inhabit the type family. We also support mutual LF definitions that we do not capture in our grammar below not to complicate matters further.

$$
\begin{array}{rl}
\text{Signature Decl. } \mathcal{D} ::= & \texttt{LF a : } \mathcal{K}_{\texttt{LF}} \texttt{ = c}_1 \texttt{ : } \mathcal{A}_1 \mid \ldots \mid \texttt{c}_k \texttt{ : } \mathcal{A}_k\texttt{;} \\
& \texttt{inductive a : } \mathcal{K} \texttt{ = c}_1\texttt{:}\mathcal{T}_1 \mid \ldots \mid \texttt{c}_n\texttt{:}\mathcal{T}_n\texttt{;} \\
& \texttt{coinductive a : } \mathcal{K} \texttt{ = (c}_1\texttt{:}\mathcal{T}_1\texttt{) :: } \mathcal{T}_1' \mid \ldots \mid \texttt{(c}_n\texttt{:}\mathcal{T}_n\texttt{) :: } \mathcal{T}_n'\texttt{;} \\
& \texttt{rec c : } \mathcal{T} \texttt{ = } \mathcal{E}\texttt{;}
\end{array}
$$

(Co)inductive definitions correspond to (co)indexed recursive types and semantically are interpreted as (greatest) least fixed points. We define an indexed recursive type family by defining constructors $\texttt{c}_i$ that can be used to construct its elements. We define a corecursive type by the observations we can make about it using indexed records, where we write the field $\texttt{c}_i$ together with the type $\mathcal{T}_i$ from which we can project the result $\mathcal{T}_i'$. Given a Beluga term of $\mathcal{E}$ type $\mathcal{T}_i$ we may use the projection $\texttt{c}_i$ and the result of $\mathcal{E}\,\texttt{.c}_i$ is then of type $\mathcal{T}_i'$.

We describe Beluga's type and terms more precisely in Figure 4.4. The syntax for LF kinds, types and terms is close to the syntax x in the Twelf system. We write curly braces `{ }` for the dependent LF function space and allow users to write simply $\rightarrow$, if there is no dependency. LF kinds, types, and LF terms used in a signature declaration must be pure, i.e. they cannot refer to closures highlighted in blue and written as `u[`σ`]` and `#p[`σ`]`. Here `u` and `#p` are meta-variables that are bound and introduced in Beluga types and patterns.

Substitutions in closures are either empty, written as $\_$ here, or a weakening substitution, denoted by …, which we use in practice to transition from a context to a possible extension. Substitutions can also be built by extending a substitution σ with a LF term $\mathcal{M}$.

LF contexts may be empty, consist of a context variable, or are built by concatenating to a LF context a LF variable declaration.

Contextual types and terms pair a LF context together with a LF type or LF term respectively. As LF contexts are first-class in Beluga, they form valid contextual objects. LF contexts are in general classified by a schema that allows us to state an invariant the LF context satisfies; here we only add the one schema we have used in this development (see page 162), namely `ctx`.

LF Kinds	$\mathcal{K}_{\mathsf{LF}}$	::=	$\mathtt{type} \mid \mathtt{\{X{:}}\mathcal{A}\mathtt{\}}\mathcal{K}_{\mathsf{LF}} \mid \mathcal{A} \to \mathcal{K}_{\mathsf{LF}}$
LF Types	$\mathcal{A}$	::=	$\mathtt{a}\ \mathcal{M}_1 \ldots \mathcal{M}_n \mid \mathtt{\{u{:}}\mathcal{A}\mathtt{\}}\mathcal{A}' \mid \mathcal{A} \to \mathcal{A}'$
LF Terms	$\mathcal{M}$	::=	$\mathtt{x} \mid \lambda\mathtt{x}.\mathcal{M} \mid \mathtt{c}\ \mathcal{M}_1 \ldots \mathcal{M}_n \mid \mathtt{u}[\sigma] \mid \mathtt{\#p}[\sigma]$
LF Subst.	σ	::=	$\_ \mid \ldots \mid \sigma, \mathcal{M}$
LF Context	$\Psi,\ \Phi$	::=	$\_ \mid \psi \mid \Psi\mathtt{,x{:}}\mathcal{A}$
Contextual Type	$\mathcal{U}$	::=	$[\Psi \vdash \mathtt{a}\ \mathcal{M}_1 \ldots \mathcal{M}_n] \mid \mathtt{[ctx]} \mid [\Psi \vdash \Phi] \mid \ldots$
Contextual Object	$\mathcal{C}$	::=	$[\Psi \vdash \mathcal{M}] \mid [\Psi] \mid \ldots$

Beluga Kinds	$\mathcal{K}$	::=	$\mathtt{ctype} \mid \mathtt{\{u{:}}\mathcal{U}\mathtt{\}}\mathcal{K} \mid \mathcal{U} \to \mathcal{K}$
Beluga Types	$\mathcal{T}$	::=	$\mathtt{\{u{:}}\mathcal{U}\mathtt{\}}\mathcal{T} \mid \mathtt{(u{:}ctx)}\mathcal{T} \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{U} \mid \mathtt{a}\ \mathcal{C}_1 \ldots \mathcal{C}_n$
Beluga Terms	$\mathcal{E}$	::=	$\mathcal{E}\ \mathcal{E} \mid \mathcal{C} \mid \mathtt{x} \mid \mathtt{c} \mid \mathcal{E}.\mathtt{c} \mid \mathtt{fun}\ \mathcal{B}_{\mathcal{R}} \mid \mathtt{let}\ \mathcal{P}\ \mathtt{=}\ \mathcal{E}\ \mathtt{in}\ \mathcal{E}$ $\mid \mathtt{mlam\ u} \Rightarrow \mathcal{E} \mid \mathtt{case}\ \mathcal{E}\ \mathtt{of}\ \mathcal{B}_{\mathcal{P}}$
Beluga Copattern Branches	$\mathcal{B}_{\mathcal{R}}$	::=	$\_ \mid \mathtt{(}\mathcal{B}_{\mathcal{R}}\ \mathtt{|}\ \mathcal{R}_1 \ldots \mathcal{R}_n \Rightarrow \mathcal{E}\mathtt{)}$
Beluga Pattern Branches	$\mathcal{B}_{\mathcal{P}}$	::=	$\_ \mid \mathtt{(}\mathcal{B}_{\mathcal{P}}\ \mathtt{|}\ \mathcal{P} \Rightarrow \mathcal{E}\mathtt{)}$
Beluga Pattern	$\mathcal{P}$	::=	$\mathtt{x} \mid \mathcal{C} \mid \mathtt{c}\ \mathcal{P}_1 \ldots \mathcal{P}_n$
Beluga Copatterns	$\mathcal{R}$	::=	$\cdot \mid \mathcal{P}\ \mathcal{R} \mid \mathtt{.c}\ \mathcal{R}$

Figure 4.4: Grammar of Beluga

Beluga's type language supports indexed dependent function space, $\mathtt{\{u{:}}\mathcal{U}\mathtt{\}}\mathcal{T}$, non-dependent functions, embedding contextual types, and (co)inductive definitions, described as $\mathtt{a}\ \mathcal{C}_1 \ldots \mathcal{C}_n$. Note that in $\mathtt{\{u{:}}\mathcal{U}\mathtt{\}}\mathcal{T}$ we make u explicit and hence any computation-level expression of that type expects first an object of type $\mathcal{U}$. We also permit a limited form of implicit type annotation for context variables; by writing $\mathtt{(u{:}ctx)}\mathcal{T}$ we can abstract over the context variable u and declare its context schema, while keeping u implicit.

Beluga's computation language is an indexed language in the sense of Chapter 2 which allows us to make statements about contextual types and objects and we highlight them in blue as well. Recursive definitions are split between the top level `rec` definition that handles recursion and the `fun` term that deals with (co)pattern matching. Beluga also supports case analysis using pattern matching independently for convenience. Further, as in OCaml, we not only allow functions to be defined via (co)pattern matching, but to be formed by abstracting

over their arguments. For example functions defined using `mlam` abstract over contextual variables occurring in the function body. Further, our language features let-expressions as a degenerate case of case analysis, together with applications, computation-level variables, constructors, constants, and projections. This is by no means a complete account of the computation language of Beluga— we have only described here the part relevant for our case study.

Metatheory of Beluga

An important question, when discussing formalizations, regards *trust* in mechanized proofs. There are two aspects that play a role: the first concerns the theoretical foundations underlying a proof environment and the second the implementation of that foundation in a concrete system. Beluga's theoretical foundation provides the justification for reading its programs as proofs. For reasoning inductively directly over contextual LF objects, we exploit the subterm ordering on LF terms, see [Pientka, 2005, Pientka and Abel, 2015]. Our (co)inductive programs give a high-level surface language to the calculi described in Chapters 2 and 3.

The implementation of these theoretical ideas in Beluga (status: August 2019) — as it is often the case — lags slightly behind; in particular, the totality checker does not yet support simultaneous (co) copattern matching following. Such extension would allow us not only to certify inductive proofs, but also coinductive ones in practice.

Encoding syntax in LF

We adopt the usual HOAS encoding for binding operators in the object language, and make essential use of LF's dependent types (see Figure 4.5). In particular the type family `term` encodes *intrinsically*-typed terms. This will make our overall mechanization more compact, as all terms are well-typed 'by construction' which is enforced by Beluga's type checker.

Variables such as `T` and `S` that are used in declaring the type of the LF constants are abstracted over at the front of the type of the constructor and we rely on type reconstruction to infer their types [Pientka, 2013]. These variables are treated as implicit and we subsequently omit passing them when forming `term` objects.

```
LF tp : type =
| top : tp
| arr : tp → tp → tp
| list: tp → tp;

LF term : tp → type =
| app  : term (arr S T) → term S → term T
| lam  : (term S → term T) → term (arr S T)
| fix  : (term T → term T) → term T
| unit : term top
| nil  : term (list T)
| cons : term T → term (list T) → term (list T)
| lcase: term (list S) → term T → (term S → term (list S) → term T)
      → term T;
```

Figure 4.5: LF definition of intrinsically typed terms

Encoding the operational semantics with indexed inductive types

To illustrate how we can use inductive types in Beluga, we encode the *value* and *evaluation* judgment as computation-level type families indexed by closed well-typed terms. This is demonstrably equivalent to encoding the same judgments at the LF level.

How do we enforce that a LF object is closed? This is accomplished by a contextual type `[ ⊢ term T]`, where the context that appears to the left hand side of the turnstile is empty; to improve readability we simply write `[term T]`. Note that we can embed contextual types into Beluga types, but not vice-versa. There is a strict separation between LF definitions that form our index objects and Beluga types that talk about LF definitions.

The inductive type family `Value` defines a subset of closed well-typed expressions, namely those that are the observables of our object language. Similarly, the inductive type family `Eval` relates two closed expressions of the same type, where the first big-step evaluates to the second (see Figure 4.6).

Beluga has a sophisticated notion of built-in simultaneous substitution. Consider the rule `Ev_app`, where to build the evaluation derivation for `Eval [app M1 M2] [V]` we have to supply

```
inductive Value : [term T] → ctype =
  | Val_lam  : Value [lam λx.N]
  | Val_unit : Value [unit]
  | Val_nil  : Value [nil]
  | Val_cons : Value [cons M1 M2];

inductive Eval : [term T] → [term T] → ctype =
  | Ev_app      : Eval [M1] [lam λx.N] → Eval [N[M2]]  [V]
                → Eval [app M1 M2] [V]
  | Ev_val      : Value [V]
                → Eval [V] [V]
  | Ev_fix      : Eval [M[fix λx.M]] [V]
                → Eval [fix λx.M] [V]
  | Ev_case_nil : Eval [M] [nil] → Eval [M1] [V]
                → Eval [lcase M M1 (λh.λt.M2)] [V]
  | Ev_case_cons: Eval [M]  [cons H L] →  Eval [M2[H, L]] [V]
                → Eval [lcase M M1 (λh.λt.M2)] [V];
```

Figure 4.6: Inductive definition of values and evaluation

an evaluation derivation for `Eval [M1] [lam λx.N]` and `Eval [N[M2]] [V]` where `N` stands for a term of type `S` that may refer to `x:term T`. The substitution that in standard LF would be represented as meta-level application `N M2`, here consists of the singleton simultaneous substitution `N[M2]` that keeps its domain, namely `x`, implicit. In general, all capitalized variables denote LF objects that may depend on LF declarations. Given an LF term `N` that depends on a context γ, we can use `N` in a context ψ by associating `N` with a simultaneous substitution σ with domain ψ and co-domain γ, with type `[`$\psi \vdash \gamma$`]`. This *closure* is written in post-fix notation as `N[`σ`]`. If σ is the identity substitution, we may drop the closure. We make use of two kinds of *weakening substitutions*: 1) The weakening substitution, written as `[]`, moves a closed object from the empty context to a context γ. For example, the type `T` is closed in `[term T]`. To use the closed type `T` in a context γ, we need to associate it with the weakening substitution `[]`. Hence, `[`$\gamma \vdash$ `term T[]]` describes a `term` object of type `T` in a context γ and enforces that the type `T` is closed. 2) The weakening substitution, written

μValue.$\lambda T, M.$
<Val_lam : $\Sigma S_1, S_2{:}[\texttt{tp}].T = (\texttt{arr}\ S_1\ S_2) \times \Sigma N{:}[x{:}\texttt{term}\ S_1 \vdash \texttt{term}\ S_2[]].M = \texttt{lam}\ \lambda x.N \times 1$
Val_unit : $T = \texttt{top} \times M = \texttt{unit} \times 1$
Val_nil : $\Sigma S{:}[\texttt{tp}].T = \texttt{list}\ S \times M = \texttt{nil} \times 1$
Val_cons : $\Sigma S{:}[\texttt{tp}].T = \texttt{list}\ S \times \Sigma M_1{:}[\texttt{term}\ S].\Sigma M_2{:}[\texttt{term}\ (\texttt{list}\ S)].$
$M = \texttt{cons}\ M_1\ M_2 \times 1$ >

μEval.$\lambda T, M, W.$
<Ev_app : $\Sigma S{:}[\texttt{tp}],\ M_1{:}[\texttt{term}\ (\texttt{arr}\ S\ T)],\ M_2{:}[\texttt{term}\ S],\ N{:}[x{:}\texttt{term}\ T \vdash \texttt{term}\ S[]].$
$M = (\texttt{app}\ M_1\ M_2) \times \texttt{Eval}\ [M_1]\ [\texttt{lam}\ \lambda x.N] \times \texttt{Eval}\ [N[M_2]]\ [W]$
Ev_fix : $\Sigma N{:}[x{:}\texttt{term}\ T \vdash \texttt{term}\ T[]].\ M = \texttt{fix}\ \lambda x.N \times \texttt{Eval}\ [N[\texttt{fix}\ \lambda x.N]]\ [W]$
Ev_val : $M = W \times \texttt{Value}\ [W]$
... >

Figure 4.7: Core language definitions of `Value` and `Eval`

as `[…]`, moves an object `M` that is defined in a context γ to an extension of γ, for example γ`, x:term T[]`. Finally, we note that `[lam λx.N]` can be expanded to `[lam λx.N[x]]` where the substitution that maps `x` to itself is simply written as `[x]`.

The top level definitions of inductive types can be translated into recursive types in our core language, as presented in Chapter 2. As such, `Value` and `Eval` take on the familiar forms which appear in Figure 4.7. As we can see, the main difference with respect to the previously defined recursive types is that Beluga expresses the index domain explicitely using square brackets, but the definition is otherwise translated in the same was as before.

Encoding similarity using indexed coinductive types

In Beluga, we also can state coinductive type families and in particular similarity as a coinductive definition that relates closed well-typed terms.

To define the coinductive type `Sim [T] [M] [N]`, we declare observations `Sim_unit`, `Sim_nil`, `Sim_cons`, and `Sim_lam`; each one corresponds to a case in our definition of applicative simulation — compare Def. 1 to Figure 4.8. We write the observation together

```
coinductive Sim : {T:[tp]} [term T] → [term T] → ctype =
| (Sim_unit : Sim [top] [M] [N])
              :: Eval [M] [unit] → Eval [N] [unit]
| (Sim_nil  : Sim [list T] [M] [N])
              :: Eval [M] [nil] → Eval [N] [nil]
| (Sim_cons : Sim [list T] [M] [N])
              :: Eval [M] [cons H L] → Ex_sim_cons [H] [L] [N]
| (Sim_lam  : Sim [arr S T] [M] [N])
              :: Eval [M] [lam λx.M'] → Ex_sim_lam [x:term S ⊢ M'] [N]

and inductive Ex_sim_cons : [term T] → [term (list T)] → [term (list T)]
                            → ctype =
| ESim_cons: Eval [N] [cons H' L']
              → Sim [T] [H] [H'] → Sim [list T] [L] [L']
              → Ex_sim_cons [H] [L] [N]

and inductive Ex_sim_lam : [x:term S ⊢ term T[]] → [term (arr S T)]
                           → ctype =
| ESim_lam: Eval [N] [lam λx.N']
             → ({R:[term S]} Sim [T] [ M'[R] ] [ N'[R] ])
             → Ex_sim_lam [x:term S ⊢ M'] [N]
```

Figure 4.8: Coinductive definition of applicative similarity

with its type on the left side of `::` and on the right side we give the result type of the observation that describes our proof obligation. For example, we can make the observation `Sim_unit:Sim [top] [M] [N]`, if we can show that `Eval [M] [unit]` → `Eval [N] [unit]`. It corresponds directly to "$m \Downarrow \langle\rangle$ entails $n \Downarrow \langle\rangle$" in Def. 1. Note that `M` and `N` are implicitly quantified at the outside, as it becomes apparent in the desugared syntax on page 153. In addition equalities are handled implicitly by providing a term at the appropriate position. In the case of `Sim_unit`, we will have the equality guard $T = \mathsf{top}$. The definition of the observation `Sim_nil` follows a similar schema.

The result of the observation `Sim_cons` on `Sim [list T] [M] [N]` requires that if $m \Downarrow \mathsf{cons}\ h\ t$ then there are h' and t' such that $n \Downarrow \mathsf{cons}\ h'\ t'$ for which $h\ \mathcal{R}_\tau\ h'$ and $t\ \mathcal{R}_{\mathsf{list}(\tau)}\ t'$.

We hence need a way to encode an *existential* property. Although existentials (i.e. Σ-types) were found in our calculus, the implementation of Beluga does not support them at the top level, as they always can be realized using indexed inductive types. We therefore define an indexed inductive type `Ex_sim_cons` that relates h, t and n.

Last, we need to represent the result of observing `Sim_lam` that encodes the corresponding part from the definition:

> $m \Downarrow \mathsf{lam}\, x.\, m'$ for any $x{:}\tau \vdash m'{:}\tau'$ entails that there exists a $y{:}\tau \vdash n'{:}\tau'$ such that $n \Downarrow \mathsf{lam}\, y.\, n'$ for which $m'[r/x]\ \mathcal{R}_{\tau'}\ n'[r/y]$ for every term r of type τ.

We again resort to defining an inductive type `Ex_sim_lam` that relates the term `M'` with type `[x:term S ⊢ term T[]]`, i.e. `M'` has type `term T[]` under the assumption of the variable `x` having type `term S`. Hence we can simply write `[x:term S ⊢ M']`, as we interpret `M'` within the context `x:term S`. As `T` denotes a closed type, we associate it with a weakening substitution, since it is used in a non-empty context. The relation `Ex_sim_lam` exists if `Eval [N] [lam λx.N']` and for all `R:[term S]` we know `Sim [T] [M'[R]] [N'[R]]`. Finally, we remark that the coinductive type `Sim` and inductive types `Ex_sim_cons` and `Ex_sim_lam` are defined mutually.

Once again, the coinductive type family `Sim` is encoded using a greatest fixed point that is defined using records, universals, and implications.

$$
\begin{array}{l}
\nu\texttt{Sim}.\lambda T.\lambda M.\lambda N. \\
\{\ \texttt{Sim\_unit} : T = \texttt{top} \to \texttt{Eval}\ [M]\ [\texttt{unit}] \to \texttt{Eval}\ [N]\ [\texttt{unit}] \\
\quad \texttt{Sim\_lam}\ \ : \Pi S_1{:}[\texttt{tp}].\Pi S_2{:}[\texttt{tp}].\Pi M'{:}[x{:}\texttt{term}\ S_1 \vdash \texttt{term}\ S_2[]].\ T = \texttt{arr}\ S_1\ S_2 \\
\qquad\qquad \to \texttt{Eval}\ [M]\ [\texttt{lam}\ \lambda x.M'] \\
\qquad\qquad \to \Sigma N'{:}[x{:}\texttt{term}\ S_1 \vdash \texttt{term}\ S_2[]].\ \texttt{Eval}\ [N]\ [\texttt{lam}\ \lambda x.N'] \\
\qquad\qquad\quad \times \Pi R{:}[\texttt{term}\ S_1].\texttt{Sim}\ [S_2]\ [M'[R]]\ [N'[R]] \\
\quad \ldots \\
\}
\end{array}
$$

We only show the encoding for lambda-expressions and omit the observations we can make on lists to keep it readable. We further in-lined the definition of `Ex_sim_lam` to keep the

definition compact.

Encoding Simple Proofs about Simulation

Let us do some simple proofs about simulation. The underlying understanding allowing us to consider those programs as proofs has been discussed extensively in the previous chapters so we will not go into too much details to justify them. Let us reconsider the proof that similarity is reflexive: for all `T`, `M`, we have `Sim [T] [M] [M]`. The type of the function `sim_refl` encodes this statement directly. We leave `T` and `M` implicit, as these arguments can be reconstructed by Beluga.

```
rec sim_refl : Sim [T] [M] [M] =
fun .Sim_unit d ⇒ d
  | .Sim_nil  d ⇒ d
  | .Sim_cons d ⇒ ESim_cons d sim_refl sim_refl
  | .Sim_lam  d ⇒ ESim_lam d (mlam R ⇒ sim_refl)
```

As mentioned above, recursive functions are defined via the top-level definition `rec` which then makes use of `fun` as a term to define copattern definitions. There are four cases justified by the four observations. The first two simply prove that if the left-hand side evaluates to the correct value, then so does the right-hand side as they are the same. The last two cases make recursive calls to prove reflexivity of the subcases, instantiating the (hidden) existential by the result of the evaluation of the left-hand side. The recursive calls are deemed guarded because an observation appear on the left-hand side of the arrow. The case for `Sim_lam` makes use of the variable abstraction `mlam R ⇒ sim_refl` to abstract over the term `R` which appears in the type of the recursive call (`Sim [T] [M'[R]] [M'[R]]`). While it is not passed explicitly here, the reconstructed program would pass it to the recursive call to `sim_refl`.

Next, we prove that the simulation is transitive:

```
rec sim_trans : Sim [T] [M] [N] → Sim [T] [N] [R] → Sim [T] [M] [R] =
fun s1 s2 .Sim_unit  d ⇒ s2.Sim_unit (s1.Sim_unit d)
  | s1 s2 .Sim_nil   d ⇒ s2.Sim_nil (s1.Sim_nil d)
  | s1 s2 .Sim_cons  d ⇒
```

```
    let ESim_cons d1 s1' s1'' = s1.Sim_cons d in
    let ESim_cons d2 s2' s2'' = s2.Sim_cons d1 in
    ESim_cons d2 (sim_trans s1' s2') (sim_trans s1'' s2'')
  | s1 s2 .Sim_lam   d ⇒
    let ESim_lam d1 s1' = s1.Sim_lam d in
    let ESim_lam d2 s2' = s2.Sim_lam d1 in
    ESim_lam d2 (mlam P ⇒ sim_trans (s1' [P]) (s2' [P]))
```

The proof takes two arguments, and then requires us to build a simulation. In the two first cases, we simply chain together the fact that the arguments are each a simulation to obtain the result. For example, assuming `M` evaluates to `unit`, then we know by `s1` (by calling `s1.Sim_unit d`) that `N` evaluates to `unit` and, using that fact, we know by `s2` that so does `R`, which is what we needed to prove.

Now, for the last two cases, we first need to chain together the observation of the input and make recursive calls.. For example, if `M` evaluated to `cons HM LM` by `d`, then the first let-statement allows us to conclude that `N` evaluates to `cons HN LN` by `d1` and we have `s1' : Sim [T] [HM] [HN]` and `s1'' : Sim [list T] [LM] [LN]`. The second let-statement similarly lets us know that `R` evaluates to `cons HR LR` by `d2` and we have `s2' : Sim [T] [HN] [HR]` and `s2'' : Sim [list T] [LN] [LR]` Now, by use of a recursive call, we are able to show `Sim [T] [HM] [HR]` and `Sim [list T] [LM] [LR]` and thus prove which is what was needed to conclude that `M` is similar to `R`.

On the adequacy of coinductive encodings

We next sketch the adequacy of the encoding of similarity; a full proof, such as those in the electronic appendix of Tiu and Miller [2010] would be too long to spell out. Instead, we rely on our intuitive understanding of (co)inductive types; to get started, we assume the adequacy of LF encodings, whereby we denote the mapping of terms m and types τ to their encodings as $\ulcorner m \urcorner$ and $\ulcorner \tau \urcorner$ respectively. Conversely, the decoding of an LF object `M` and `T` into terms and types is written $\llcorner \texttt{M} \lrcorner$ and $\llcorner \texttt{T} \lrcorner$ respectively.

Lemma 4.8. *For any term m and type τ, we have $\llcorner \ulcorner m \urcorner \lrcorner = m$ and $\llcorner \ulcorner \tau \urcorner \lrcorner = \tau$.*

Proof. Standard, following for example the work of Pfenning [1997]. □

We further build on the adequacy of the encoding of substitutions. In particular, the translation of $[\sigma]m$ is equivalent to first translating σ and the term m to their corresponding representations in LF and then relying on the built-in simultaneous LF substitution operation of applying $\ulcorner\sigma\urcorner$ to $\ulcorner m\urcorner$. The encoding $\ulcorner\sigma\urcorner$ is defined inductively on the substitution σ as expected: $\ulcorner\cdot\urcorner = \hat{}$ and $\ulcorner\sigma, m/x\urcorner = \ulcorner\sigma\urcorner,\ \ulcorner m\urcorner$. Further, recall that we write the application of a simultaneous LF substitution in prefix form, while we write the closure of an LF object together with an LF substitution in post-fix.

Lemma 4.9 (Compositionality)**.** $\ulcorner[\sigma]m\urcorner = [\ulcorner\sigma\urcorner]\ulcorner m\urcorner$.

Proof. Generalization of the compositionality lemma for LF. □

Lemma 4.10 (Soundness)**.** *If* $\texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]$ *then* $m \preccurlyeq_\tau n$.

Proof. (Sketch) We apply rule $CI-\preceq$ to unfold $m \preccurlyeq_\tau n$ selecting the family S_τ to be

$$\{(m, n) \mid \texttt{Sim}\ [\ulcorner\tau\urcorner]\ [\ulcorner m\urcorner]\ [\ulcorner n\urcorner]\}$$

We then show that S_τ satisfies the simulation conditions unfolding the definition of S_τ.

□

Before addressing the other direction, we briefly contrast the more familiar inductive reasoning with the coinductive reasoning we will use. To prove a conjecture inductively on an object, we consider all possible ways such an object can be constructed and we reason inductively about some notion of *size* of an object or a derivation. In an inductive proof, to show that the property holds for objects of size m, we may assume that it holds for objects of size k where $k < m$.

For example, to prove that for all terms M and V, if $\mathcal{D} : \texttt{Eval}\ [M]\ [V]$ then there exists $\mathcal{F} : \texttt{Value}\ [V]$, we proceed by induction on the height n of $\mathcal{D}$, the derivation of $\texttt{Eval}\ [M]\ [V]$. We therefore prove the following by considering all possible constructors that we can use to build such a derivation $\mathcal{D}$.

IH	For all $k < n$, for all terms M and V,
	if $\mathcal{D}' : \texttt{Eval}\ [M]\ [V]$ and $\mathsf{size}(\mathcal{D}') = k$ then there is $\mathcal{F} : \texttt{Value}\ [V]$
To show	For all term M and V, if $\mathcal{D} : \texttt{Eval}\ [M]\ [V]$ and $\mathsf{size}(\mathcal{D}) = n$ then
	there is $\mathcal{F} : \texttt{Value}\ [V]$

Dually, to prove a conjecture *coinductively*, we consider all possible observations we can make about an object and we *reason inductively on the number of observations*, which we refer to as *depth*. In a coinductive proof, we assume that the conjecture holds when we can make k observations about the object, and we show that the conjecture also holds when we make n observations about it where $k < n$. In essence, to prove a statement by coinduction we reason by complete induction on the number of observations. For example, if we want to prove reflexivity of simulation, i.e. for all terms M and types T, $\mathcal{D} : \texttt{Sim}\ [T]\ [M]\ [M]$, then we proceed by induction on the number of observation on $\mathcal{D}$ and consider all possible observations we can make about $\mathcal{D}$.

IH	For all $k < n$, for all terms M and types T,
	$\mathcal{D}' : \texttt{Sim}\ [T]\ [M]\ [M]$ and $\mathsf{depth}(\mathcal{D}') = k$
To show	For all term M and types T, $\mathcal{D} : \texttt{Sim}\ [T]\ [M]\ [M]$ and $\mathsf{depth}(\mathcal{D}) = n$

We are now ready to address the other direction of the adequacy statement. For a more formal justification of reasoning about inductive data via sizes and coinductive data via observations we refer the reader to [Abel and Pientka, 2013, 2016].

Lemma 4.11 (Completeness)**.** *If* $m \preccurlyeq_\tau n$*, then* $\texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$.

Proof. We proceed by complete induction on the number of observations we can make on $\texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$.

IH	For all $k < j$, for all terms m and types τ,
	If $\mathcal{S} : m \preccurlyeq_\tau n$, then $\mathcal{D}' : \texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$ and $\mathsf{depth}(\mathcal{D}') = k$
To show	for all terms m and types τ,
	If $\mathcal{S} : m \preccurlyeq_\tau n$, then $\mathcal{D} : \texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$ and $\mathsf{depth}(\mathcal{D}) = j$

Observation `Sim_unit`.

To show: If $m \preccurlyeq_\tau n$ then $(\ulcorner \tau \urcorner = \texttt{top}) \to \texttt{Eval}\ [\ulcorner m \urcorner]\ [\texttt{unit}] \to \texttt{Eval}\ [\ulcorner n \urcorner]\ [\texttt{unit}]$.

Assume $m \preccurlyeq_\tau n$, $\ulcorner \tau \urcorner = \texttt{top}$, and $\texttt{Eval}\ [\ulcorner m \urcorner]\ [\texttt{unit}]$

$\tau = \top$	by definition of $\ulcorner \tau \urcorner$
$m \Downarrow \langle\rangle$ entails $n \Downarrow \langle\rangle$	by Def. 1 using the assumption $m \preccurlyeq_\tau n$
$\llcorner\ \texttt{Eval}\ [\ulcorner m \urcorner]\ [\texttt{unit}]\ \lrcorner = m \Downarrow \langle\rangle$	by decoding of `Eval`
$n \Downarrow \langle\rangle$	by previous lines
$\ulcorner n \Downarrow \langle\rangle \urcorner = \texttt{Eval}\ [\ulcorner n \urcorner]\ [\texttt{unit}]$	by encoding of `Eval`

Observation `Sim_lam`.

IH	For all $k < j$, if $m \preccurlyeq_\tau n$ then $\mathcal{D} : \texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$ and $\mathsf{depth}(\mathcal{D}) = k$
To show	If $m \preccurlyeq_\tau n$ then $\mathcal{D}\ .\texttt{S\_lam} : \texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$ and $\mathsf{depth}(\mathcal{D}\ .\texttt{Sim\_lam}) = j$

Making an observation corresponds to projecting with the dot notation the field `Sim_lam` of the record. We further note that $\mathsf{depth}(\mathcal{D}\ .\texttt{Sim\_lam}) = \mathsf{depth}(\mathcal{D}) + 1$. Since we are making the observation `Sim_lam`, we can unfold the definitions. Hence, it suffices to show

if $m \preccurlyeq_\tau n$ then
for all $\texttt{S}_1, \texttt{S}_2, \texttt{M}'$. $(\ulcorner \tau \urcorner = \texttt{arr}\ \texttt{S}_1\ \texttt{S}_2) \to \texttt{Eval}\ [\ulcorner m \urcorner]\ [\texttt{lam}\ \lambda x.\texttt{M}']$
$\to$ there exists $\texttt{N}'$ s.t. $\texttt{Eval}\ [\texttt{N}]\ [\texttt{lam}\ \lambda x.\texttt{N}']$
and (for all $\texttt{R}$, $\mathcal{D} : \texttt{Sim}\ [\texttt{S}_2]\ [\ \texttt{M}'[\texttt{R}]\ ]\ [\ \texttt{N}'[\texttt{R}]\ ]$)

Moreover, $\mathsf{depth}(\mathcal{D})$ is clearly less than $\mathsf{depth}(\mathcal{D}\ .\texttt{Sim\_lam})$ and we may appeal to the induction hypothesis, which can be specialized to the following statement:

if $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$ then $\texttt{Sim}\ [\texttt{S}_2]\ [\ \texttt{M}'[\texttt{R}]\ ]\ [\ \texttt{N}'[\texttt{R}]\ ]$ where $\ulcorner m' \urcorner = \texttt{M}'$, $\ulcorner r \urcorner = \texttt{R}$, $\ulcorner n' \urcorner = \texttt{N}'$.

Assume $m \preccurlyeq_\tau n$, $\ulcorner \tau \urcorner = (\texttt{arr}\ \texttt{S}_1\ \texttt{S}_2)$, and $\texttt{Eval}\ [\ulcorner m \urcorner]\ [\texttt{lam}\ \lambda x.\texttt{M}']$

$\tau = s_1 \to s_2$ since $\ulcorner s_1 \to s_2 \urcorner = \texttt{arr}\ \texttt{S}_1\ \texttt{S}_2$ where $s_1 = \llcorner \texttt{S}_1 \lrcorner$ and $s_2 = \llcorner \texttt{S}_2 \lrcorner$.

for any $x{:}s_1 \vdash m'{:}s_2$. $m \Downarrow \mathsf{lam}\, x.\, m'$ entails that
there exists a $y{:}s_1 \vdash n'{:}s_2$ such that $n \Downarrow \mathsf{lam}\, y.\, n'$
and for every $r{:}s_1$, $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$; by definition of $m \preccurlyeq_\tau n$

$\texttt{Eval}\ [\ulcorner m \urcorner]\ [\texttt{lam}\ \lambda x.\texttt{M}'] = \ulcorner m \Downarrow \llcorner \texttt{lam}\ \lambda \texttt{x}.\texttt{M}' \lrcorner \urcorner$ by encoding of Eval

$\texttt{lam}\ \lambda \texttt{x}.\texttt{M}' = \ulcorner \mathsf{lam}\, x.\, m' \urcorner$ by encoding of terms

there exists a $y{:}s_1 \vdash n'{:}s_2$ such that $n \Downarrow \mathsf{lam}\, y.\, n'$
and for every $r{:}s_1$, $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$ by previous lines

$\ulcorner n \Downarrow \mathsf{lam}\, y.\, n' \urcorner = \texttt{Eval}\ \ulcorner n \urcorner\ (\texttt{lam}\ \lambda y. \ulcorner n' \urcorner)$ by encoding of Eval

Assume $\texttt{R} : \texttt{term}\ \texttt{S}_1$.
$\ulcorner [r/x]m' \urcorner = \texttt{M}'[\texttt{R}]$ and $\ulcorner [r/y]n' \urcorner = \texttt{N}'[\texttt{R}]$ by Theorem 4.9 (Compositionality)
$\texttt{Sim}\ [\texttt{S}_2]\ [\texttt{M}'[\texttt{R}]]\ [\texttt{N}'[\texttt{R}]]$ by the specialized induction hypothesis
using $[r/x]m' \preccurlyeq_{s_2} [r/y]n'$ from the previous line

Therefore, there exists $\texttt{N}'$, namely $\ulcorner n' \urcorner$, and $\texttt{Eval}\ [\ulcorner n \urcorner]\ [\texttt{lam}\ \lambda y. \ulcorner n' \urcorner]$ and for all $\texttt{R}{:}\texttt{term}\ \texttt{S}_1$, we have $\texttt{Sim}\ [\texttt{S}_2]\ [\texttt{M}'[\texttt{R}]]\ [\texttt{N}'[\texttt{R}]]$. Hence, $\texttt{Sim}\ [\ulcorner s_1 \to s_2 \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$. This concludes this case. □

Theorem 4.12 (Adequacy of encoding of similarity as a coinductive type)**.** *$m \preccurlyeq_\tau n$ iff* $\texttt{Sim}\ [\ulcorner \tau \urcorner]\ [\ulcorner m \urcorner]\ [\ulcorner n \urcorner]$.

We remark that while soundness follows the same structure of [Honsell et al., 2001] and [Tiu and Miller, 2010], the possibility to induct on the number of observation allows one to establishes completeness in a novel and easier way with respect to an analogous result in [Tiu and Miller, 2010], which had to resort to a complex induction on the structure of the arguments of the coinductively defined predicate/type.

Subsequently, we will not make the number of observations explicit in coinductive arguments, but simply permit corecursive calls when they are guarded by an observation.

A concrete example of similarity

In this section we show how we can interactively build an actual simulation between two terms, namely that `two` is simulated by `suc one`, following the example in [Pitts, 2011]. We represent the numbers via Church encodings, where `one` $\equiv \mathsf{lam}\, f.\, \mathsf{lam}\, x.\, f\ x$, `two` $\equiv \mathsf{lam}\, f.\, \mathsf{lam}\, x.\, f(f\ x)$, and `suc` $\equiv \mathsf{lam}\, n.\, \mathsf{lam}\, x.\, \mathsf{lam}\, y.\, x\ (n\ x\ y)$. We thus want to prove the following theorem:

```
rec sim_two_suc_one :
       Sim [_] [lam λf.lam λx.app f (app f x)]
               [app (lam λn. lam λx. lam λy. app x (app (app n x) y))
                    (lam λf.lam λx.app f x)]
```

We will build the proof incrementally, by inserting *holes*, denoted by `?` and refining them, analogously to Agda's or Epigram's methodology [McBride, 2004]. We start with the following program body:

```
fun .Sim_lam (Ev_val Val_lam) ⇒
        ESim_lam (Ev_app (Ev_val Val_lam) (Ev_val Val_lam)) sim_lemma1
```

where `sim_lemma1` is used to abstract over the nested copattern matching:

```
rec sim_lemma1 : {M:[exp (arr T T)]}
   Sim [arr T T]
       [lam (λx. app M[] (app M[] x))]
       [lam (λy. app M[] (app (app (lam (λf. lam (λw. app f w))) M[]) y))] =
fun [M] .Sim_lam (Ev_val Val_lam) ⇒
     ESim_lam (Ev_val Val_lam)
              (mlam N ⇒ ?)
```

Here, the goal has type `Sim [_] [app M (app M N)] [app M (app (app (lam (λf. lam (λw. app f w))) M) N)]`, which we cannot prove directly using our definition of similarity: since our evaluation strategy is call-by-name and the metavariable `M` is not a concrete term, the right-hand side will not reduce. Instead, we use the fact that similarity is a pre-congruence, the main result of this work. We only need property (C3) of Definition 2, which translates to the following lemma.

```
rec sim_cong_app : Sim [arr S T] [M1] [M2] → Sim [S] [N1] [N2]
                → Sim [T] [app M1 N1] [app M2 N2]
```

Using this result and reflexivity of similarity, we can thus refine the body of `sim_lemma1`:

```
fun [M] .Sim_lam (Ev_val Val_lam) ⇒
     ExSimlam (Ev_val Val_lam)
              (mlam N ⇒ sim_cong_app sim_refl ?)
```

where the current hole has type:

```
Sim [T] [app M N] [app (app (lam (λu. lam (λw. app u w))) M) N]
```

Now, we can easily use a derivation of the evaluation of the left-hand side to derive the evaluation of the right-hand side of this similarity as follows:

```
rec ev1 : Eval [app M N] [V] →
              Eval [app (app (lam (λu. lam (λw. app u w))) M) N] [V] =
fun d ⇒ Ev_app (Ev_app (Ev_val Val_lam) (Ev_val Val_lam)) d;
```

Constructing the above simulation requires us to match on the possible values that `app M N` can take through the possible observations — in fact all of them as the type `T` is abstract. We then use `ev1` on the derivations of `Eval [app M N] [V]` for the given `V` and reflexivity when needed.

```
rec sim_lemma2 : Sim [T] [app M N]
                         [app (app (lam (λu. lam (λw. app u w))) M) N] =
fun .Sim_lam  d ⇒ ESim_lam (ev1 d) (fun [V] ⇒ sim_refl)
  | .Sim_unit d ⇒ ev1 d
  | .Sim_nil  d ⇒ ev1 d
  | .Sim_cons d ⇒ ESim_cons (ev1 d) sim_refl sim_refl
```

Using this lemma, we can complete the body of `sim_lemma1`:

```
fun [M] .Sim_lam (Ev_val Val_lam) ⇒
     ExSimlam (Ev_val Val_lam)
              (mlam N ⇒ sim_cong_app sim_refl sim_lemma2)
```

This concludes the proof.

Defining open similarity using first-class contexts and substitutions

Similarity only relates closed terms. However, in general, we want to be able to reason about similarity of open terms, i.e. terms that depend on a context γ. In Beluga, we can declare schemas of contexts that classify contexts in the same way that types classify terms and kinds classify types, describing the shape of each declaration in a context. Moreover, we can take advantage of built-in substitutions to *relate* two contexts. In particular, we can describe *grounding* substitutions with the type `[ ⊢ γ]`, where the range of the substitution is empty.

We begin by defining the schema of contexts that can occur in our development:

```
schema ctx = term T;
```

Here we declare the schema `ctx` that states that each declaration of a context γ of schema `ctx` can only contain variable declarations of type `term T` for some type `T`. For example, the context `x:term top, y:term (list top)` is a valid context of schema `ctx`. On the other hand, a context `x:term unit, a:tp` is not.

We can now state open similarity as an inductive type relating well-typed terms in the context γ. In the kind of the inductive type `OSim`, we make the type `T` explicit, but leave γ implicit. This distinction is reflected in Beluga's source syntax. We use curly braces `{ }` to describe explicit index arguments and round ones `( )` to give type annotations implicitly.

We can now define open similarity: two terms `[γ ⊢ M]` and `[γ ⊢ N]` are openly similar if they are similar for all grounding substitutions σ. Here, we pass γ explicitly to `OSimC` so that Beluga knows the type of σ.

```
inductive OSim:(γ:ctx){T:[tp]} [γ ⊢ term T[]] → [γ ⊢ term T[]] → ctype =
| OSimC : {γ:ctx}({σ:[ ⊢ γ]} Sim [T] [M[σ]] [N[σ]])
           → OSim [T] [γ ⊢ M] [γ ⊢ N]
```

We can easily show that open similarity is closed under substitutions by simply composing the input substitution σ with the grounding substitution σ'.

```
rec osim_cus : (γ:ctx) (ψ:ctx) {σ:[ψ ⊢ γ]} OSim [T] [γ ⊢ M] [γ ⊢ N]
               → OSim [T] [ψ ⊢ M[σ]] [ψ ⊢ N[σ]] =
```

```
inductive Howe: (γ:ctx){T:[tp]}[γ ⊢ term T[] ] → [γ ⊢ term T[] ] → ctype =
  | Howe_unit: OSim [top] [γ ⊢ unit] [γ ⊢ M]
               → Howe [top] [γ ⊢ unit] [γ ⊢ M]
  | Howe_var : {#p:[γ ⊢ term T[]]} OSim [T] [γ ⊢ #p] [γ ⊢ M]
               → Howe [T] [γ ⊢ #p] [γ ⊢ M]
  | Howe_lam : Howe [T] [γ,x:term S[] ⊢ M] [γ,x:term S[] ⊢ N]
               → OSim [arr S T] [γ ⊢ lam λx.N] [γ ⊢ R]
               → Howe [arr S T] [γ ⊢ lam λx.M] [γ ⊢ R]
  | Howe_app : Howe [arr S T] [γ ⊢ M] [γ ⊢ M'] 
               → Howe [S] [γ ⊢ N] [γ ⊢ N']
               → OSim [T] [γ ⊢ app M' N'] [γ ⊢ R]
               → Howe [T] [γ ⊢ app M N] [γ ⊢ R]
…
```

Figure 4.9: The Howe relation

```
fun [ψ ⊢ σ] (OSimC [γ] f) ⇒ OSimC [ψ] (fun [σ'] ⇒ f [σ[σ']])
```

Defining the Howe relation

The encoding of the Howe relation (see Figure 4.9) is, in our view, one of the high points of the formalization: it follows very closely its mathematical formulation, while retaining all the powerful abstractions that Beluga offers. This is apparent in the variable case where Beluga's *parameter* variables, denoted `#p`, range over elements from the context γ. They permit us to precisely characterize when a variable is Howe related to a term `M` in the given context, while looking remarkably similar to the informal version. The same applies to lambda-abstractions case, where one notes the correct scoping of `M`, `N` and `R` with respect to γ. The cases for `fix` and `lcase` follow the same principle and we omit them to save space.

The structure of the Howe relation makes it trivial to prove it is a precongruence. We show here the proof in the application case, which is indirectly used for the example of simulation we presented above. The proof of precongruence of similarity in fact follows immediately once we prove equivalent similarity and the Howe relation, see Section 4.2.

```
rec howe_cong_app : Howe [arr S T] [M1] [M2] → Howe [S] [N1] [N2]
                    → Howe [T] [app M1 N1] [app M2 N2] =
fun h1 h2 ⇒ Howe_app h1 h2 osim_refl;
```

Using reflexivity and transitivity of open similarity, respectively, we can show reflexivity and semi-transitivity of the candidate relation. We only show the types.

```
rec howe_refl : (γ:ctx) {M:[γ ⊢ term T[] ]} Howe [T] [γ ⊢ M] [γ ⊢ M]

rec howe_osim_trans : (γ:ctx) Howe [T] [γ ⊢ M] [γ ⊢ N]
                      → OSim [T] [γ ⊢ N] [γ ⊢ R]
                      → Howe [T] [γ ⊢ M] [γ ⊢ R]
```

From this it immediately follows that open similarity is a Howe relation.

```
rec osim_howe:(γ:ctx) OSim [T] [γ ⊢ M] [γ ⊢ N]
                    → Howe [T] [γ ⊢ M] [γ ⊢ N] =
fun s ⇒ howe_osim_trans (howe_refl [_ ⊢ _]) s
```

Substitutivity of the Howe relation

As remarked in Section 4.1, a crucial point of the proof is showing that the Howe relation is substitutive. Traditionally, substitution properties tend to be tedious to prove in proof assistants due to the necessity to reason manually about contexts. Here, Beluga's contextual abstractions significantly reduces the amount of boilerplate work needed for that proof.

We first encode (Figure 4.10) $\Phi \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma_2$ using an inductive type that relates two simultaneous substitutions . The base case relates empty substitutions, written as `[`ψ ⊢ `]`. In the inductive case, the substitution `[`ψ ⊢ σ_1`, M]` and `[`ψ ⊢ σ_2`, N]` are related, if so are `[`ψ ⊢ σ_1`]` and `[`ψ ⊢ σ_2`]` and `[`ψ ⊢ `M]` is Howe related to `[`ψ ⊢ `N]`.

In the subsequent proofs, we rely on the weakening property of simultaneous substitutions: namely, that weakening preserves Howe-relatedness, see function `howe_subst_wkn` in Figure 4.10. In Beluga, weakening a substitution is simply achieved by composing it with the weakening substitution `[…]`, which has here domain ψ and range ψ`, x:term S[]`. This is

```
rec howe_ren : {γ:ctx}{ψ:ctx}{σ: [ψ ⊢# γ]} Howe [T] [γ ⊢ M] [γ ⊢ N]
             → Howe [T] [ψ ⊢ M[σ]] [ψ ⊢ N[σ]]

inductive Howe_subst:
              {γ:ctx} (ψ:ctx) {σ1 : [ψ ⊢ γ]} {σ2 : [ψ ⊢ γ]} ctype =
  | HNil  : Howe_subst [] [ψ ⊢ ] [ψ ⊢ ]
  | HCons : Howe_subst [γ] [ψ ⊢ σ1] [ψ ⊢ σ2]
            → Howe [T] [ψ ⊢ M] [ψ ⊢ N]
            → Howe_subst [γ,x:term T[]] [ψ ⊢ σ1, M] [ψ ⊢ σ2, N]

rec howe_subst_wkn : Howe_subst [γ] [ψ ⊢ σ1] [ψ ⊢ σ2]
         → Howe_subst [γ] [ψ,x:term S[] ⊢ σ1[…]] [ψ,x:term S[] ⊢ σ2[…]] =
fun [S] HNil ⇒ HNil
| [S] HCons hs' h' ⇒
           HCons (howe_subst_wkn [⊢ S] hs')
                 (howe_ren [_] [_, x:term S[] ] [_, x:term S[] ⊢ … ] h')
```

Figure 4.10: Howe related substitutions

supported in Beluga's theory of simultaneous substitutions [Cave and Pientka, 2013], which internalizes the notions in Figure 4.3. The proof of `howe_subst_wkn` is done by induction over the predicate `Howe_subst` and by appealing to `howe_ren`, a special case of substitutivity of Howe on renamings. In the following, namely in the proof of `howe_osim`, see Figure 4.12, we will also need the following reflexivity property of `Howe_subst`, which holds by a simple induction on substitutions:

```
rec howe_subst_refl: (γ:ctx)(ψ:ctx){σ:[ψ ⊢γ]}
                        Howe_subst [γ] [ψ ⊢ σ] [ψ ⊢ σ]
```

A fragment of the proof of substitutivity in Beluga appears in Figure 4.11. We only show here the same two cases we described in the informal proof, together with the variable case, but the remaining cases follow a similar pattern. We make use of the lemmas described above, together with the following additional lemma for the variable case:

```
rec howe_subst_var : (γ:ctx) (ψ:ctx) Howe [T] [γ ⊢ #p] [γ ⊢ M]
```

```
                → Howe_subst [γ] [ψ ⊢ σ1] [ψ ⊢ σ2]
                → Howe [T] [ψ ⊢ #p[σ1]] [ψ ⊢ M[σ2]] =
```

This is proven by simple induction on the position of the variable in the context which follows the inductive definition of `Howe_subst`.

We can see that `howe_subst` straightforwardly represents the proof of Lemma 4.7. What is remarkable in this program with respect to the informal proof is that there are no explicit references to the substitution properties outside of the weakening of the Howe relation on substitutions. The encoding is very concise and captures the essential steps in the proof.

```
rec howe_subst : Howe [T] [γ ⊢ M] [γ ⊢ N]
              → Howe_subst [γ] [ψ ⊢ σ1] [ψ ⊢ σ2]
              → Howe [T] [ψ ⊢ M[σ1]] [ψ ⊢ N[σ2]] =
fun h (hs:Howe_subst [γ] [ψ ⊢ σ1] [ψ ⊢ σ2]) ⇒
case h of
…
| Howe_var [γ ⊢ #p] s ⇒ howe_subst_var h hs
| Howe_lam h' s        ⇒
  Howe_lam (howe_subst h' (HCons (howe_subst_wkn hs)
                                 (howe_refl [ψ,x:term _] [ψ,x:term _ ⊢x])))
           (osim_cus [ψ ⊢ σ2] s)
| Howe_app h1' h2' s   ⇒
  Howe_app (howe_subst h1' hs) (howe_subst h2' hs) (osim_cus [ψ ⊢ σ2] s);
```

Figure 4.11: Substitutivity property of the Howe relation

Main theorem

The key lemma in our main theorem is the proof that the Howe relation is *downward closed*:

```
rec down_closed : Eval [M] [V] → Howe [T] [M] [N] → Howe [T] [V] [N]
```

The proof of this lemma (point 4.1 at page 141) relies on several previous lemmas such as transitivity of closed and open similarity, semi-transitivity and substitutivity of the Howe

relation, together with the unfolding of similarity using the observations. The proof is otherwise straightforward but long, and we omit it.

Moving on, we first establish lemmas that mimic the similarity conditions (previous point 4.1). For example: If $\mathsf{lam}\ x.\ m \preccurlyeq^{\mathcal{H}}_{\tau\to\tau'} n$, then $n \Downarrow \mathsf{lam}\ x.\ m'$ and for every $q{:}\tau$ we have $[q/x]m \preccurlyeq^{\mathcal{H}}_{\tau'} [q/x]m'$. Again, as we do not have existential types, we encode the existence of a term `N'` using the inductive types `Howe_abs`. A fragment of the type signature is as follows:

```
inductive Howe_abs: [x:term S ⊢ term T[]] → [term (arr S T)] → ctype =
  | Howe_absC : Eval [N] [lam λx.N']
                → ({R:[term T]} Howe [T'] [M'[R]] [N'[R]])
                → HoweAbs [x:term S ⊢ M'] [N];

rec howe_ev_abs : Howe [arr S T] [lam λx.M'] [N]
                  → HoweAbs [x:term S ⊢ M'] [N]
```

We are now ready to prove that the Howe relation coincides with open similarity. We do this by first proving that, in the empty context, the Howe relation is a similarity, then we embed the open version into an open similarity. To do so, we construct out of the input substitution σ for open similarity a derivation $\cdot \vdash \sigma \preccurlyeq^{\mathcal{H}}_{\Gamma} \sigma$ by reflexivity. The proofs appear in Figure 4.12.

4.3 Related work

The first HOAS-like formal verification of the congruence of a notion of bisimilarity concerned the π-calculus [Honsell et al., 2001] and was carried out in Coq using the weak HOAS approach and instantiating the *Theory of Contexts* to axiomatizing properties of names. As common in many coinductive developments in Coq, the authors soon ran afoul of the guardedness checker in `Cofix`-style proofs and had to resort to an explicit greatest-fixed point encoding for Strong Late Bisimilairy. Abella's take on the same issue [Tiu and Miller, 2010] seems preferable; that paper details, among so much more, a proof that similarity is a pre-congruence for the finite π-calculus. The encoding is rather elegant, where all issues involving

```
rec howe_sim : Howe [T] [M] [N] → Sim [T] [M] [N] =
fun h .Sim_unit e ⇒ howe_ev_unit (down_closed e h)
  | h .Sim_nil  e ⇒ howe_ev_nil (down_closed e h)
  | h .Sim_cons e ⇒
       let Howe_consC e' h1 h2 = howe_ev_cons (down_closed e h) in
           ESim_cons e' (howe_sim h1) (howe_sim h2)
  | h .Sim_lam  e ⇒
       let Howe_absC e' f = howe_ev_abs (down_closed e h) in
           ESim_lam e' (mlam R ⇒ howe_sim (f [R])

rec howe_osim : {γ:ctx} Howe [T] [γ ⊢ M] [γ ⊢ N]
                → OSim [T] [γ ⊢ M] [γ ⊢ N] =
fun [γ] h ⇒ OSimC (mlam σ ⇒ howe_sim (howe_subst h (howeSubst_refl [σ])));
```

Figure 4.12: The Howe relation is included in open similarity

bindings, names, and substitutions are handled declaratively without explicit side-conditions, thanks to the ∇-quantifier. This style of encoding has been extended in [Chaudhuri et al., 2015] to handle bisimilarity "up-to". This is achieved via a limited form of quantification on relations that does not have, to our knowledge, a consistency proof yet. The authors do not pursue a proof of congruence in the cited paper.

Recent years have seen much work regarding the formalization of process calculi, in particular using Nominal Isabelle. Among those we mention Bengtson and Parrow [2009], Parrow et al. [2014], Bengtson et al. [2016] which discuss various versions of the π/ψ-calculus and their congruence properties, without resorting to the Howe's proof strategy.

Encoding bisimilarity in the λ-calculus, in particular via Howe's method, brings in additional challenges, as we have seen. We are aware of several formalizations through the years:

1. In [Ambler and Crole, 1999] the authors verify in Isabelle/HOL 98 the same result of the present paper and a bit more (they also show that similarity coincides with contextual pre-order) for PCFL using de Bruijn indexes as an encoding techniques for binders.

The development, for the congruence part, consists of around 160 lemmas/theorems, and it confirms a common belief about (standard) concrete syntax approaches: doable, but very hard-going;

2. A partial improvement was presented in [Momigliano et al., 2002], which was based on the HOAS approach implemented in an early version of the *Hybrid* tool [Felty and Momigliano, 2012], but one crucial lemma was left unproven, tellingly: Howe's substitutivity. This was related to the difficulty of lifting substitution as β-conversion to substitution on judgments in one-level Hybrid term;

3. In Momigliano [2012] the author fixed this problem, giving a complete Abella proof for the simply typed calculus with unit. The proof consists of circa 45 theorems, 1/4 of which devoted to maintaining typing invariants in (open)similarity and in the candidate relation, 1/7 of which instead used to make sure that some ∇-quantified variable cannot occur in certain predicates. The main source of difficulty was again in the proof of substitutivity of the Howe relation, in particular while handling structural properties of explicit contexts.

4. Using the present work as a blue print, Chaudhuri [2018] gives a proof in Abella of the substitutivity of the Howe relation that is very close to the one discussed here; it is based on a theory of first class simultaneous substitutions encoded via the *copy* clauses as originally suggested by Miller: $m_1/x_1, \ldots, m_n/x_n$ is represented as the Abella context `copy nn mn :: ... :: copy m1 n1 :: nil`.The application of a substitution $[\sigma]m = n$ becomes the *derivability* of the judgment {⌜σ⌝ ⊢ `copy` ⌜m⌝ ⌜n⌝}. The theory underlying this formalization consists of 15 theorems, 4 of which are sensitive to the signature: in this sense the theory has to be stated and re-proven for each signature. The effort required to automate the infrastructure for simultaneous substitutions in Abella should be analogous to similar libraries in Coq [Kaiser et al., 2017]. Compared to the Beluga development, where substitutions and their equality theory are built-in, this is more labour-intense roughly adding a factor of 1.6. Moreover, Abella's proposed handling of simultaneous substitutions is, of course, relational and must be explicitly applied whenever needed. For example, the proof of `osim_ocus`, which in

Beluga is a one liner, requires here the appeal to five lemmas to ensure that substitutions and their compositions are functional, that types and well-formedness of contexts are preserved etc.

In another recent paper McLaughlin et al. [2018] give a formalization of the coincidence of observational and applicative approximation not going through the candidate relation, but triangulating with a notion of *logical* (as in logical relations) approximation. This is then extended to CIU approximation. The encoding uses first-order syntax for terms, but a form of weak HOAS for judgments following Allais et al. [2017], and it is therefore compatible with a standard proof assistant such as Agda. Similarly to us, it leverages the use of intrinsically well-typed and well-scoped terms and simultaneous substitutions, although the latter are not supported natively by the framework. Interestingly, it offers an elegant notion of concrete context (and thus of Morris approximation) that seems much easier to reason with than previous efforts [Ford and Mason, 2003].

Lenglet and Schmitt [2018] present a formalization of Sangiorgi's Higher-Order π-calculus in Coq, using the locally nameless approach to representing name restriction and well-scoped de Bruijn indices for process variables. The formalization includes a proof that strong context bisimilarity is a congruence, via an adaptation to the concurrent setting of Howe's method. The authors seem unaware of HOAS representations of the π-calculus and the representation technique adopted, albeit state of the art among the concrete ones, still requires a lot of boilerplate infrastructure to handle names and related notions.

4.4 Conclusions

We have outlined how to use Beluga to encode a significant example of reasoning about program equivalence using Howe's method for PCFL. This has reinforced several observations that have been done in other case studies involving Beluga, viz. [Cave and Pientka, 2015, 2018]:

- Using intrinsically typed terms instead of working with explicit typing invariants makes our encoding more compact and easier to deal with, since HOAS maintains our terms

well-scoped. The advantages of intrinsically typed representations have also been observed in non-HOAS setting [Benton et al., 2012, Allais et al., 2017].

- The support for built-in simultaneous substitutions and contexts lead us to generalize some statements, for example substitutivity; but this paid off in our mechanization, as many crucial lemmas became simpler to prove.
- Thanks to catering for both indexed inductive and coinductive data-types in Beluga, the encoding of similarity and of the candidate relation was concise and as close as the informal presentation as one can reasonably hope for.

While we hope that we have succeeded in showing that Beluga is an excellent environment for the meta-theory of program equivalence, this case study has shown that is not well suited (yet) to verifying the equivalence of concrete pieces of code. To approach this, we need better interactive tools. The interactive proof environment Harpoon [Errington, 2020] is being developed to that effect. It allows tactic based reasoning of Beluga programs. At the time of writing of this thesis, it does not support coinduction.

Last but not least, we take a look beyond mechanizing bisimilarity which has been the focus of this chapter. The attentive reader may have noticed that we have not proven that bisimilarity coincides with contextual equivalence, as e.g., in [Ambler and Crole, 1999]. The challenge lies in the encoding of the latter notion: we would rather avoid using a notion of concrete non-α-equivalent terms with holes, in favour of a context-less formulation: contextual equivalence can be seen as the largest *adequate and compatible* relation [Lassen, 1998, Pitts, 2011]. This requires extending Beluga to at least second-order quantification. This would also be useful in many other scenarios, e g., normalization for system $\mathcal{F}$.

Finally, we believe it would be interesting to explore other approaches to proving program equivalence such as *step-indexed logical relations* [Ahmed, 2006] and compare this approach with Howe's. This would give us a deeper understanding of program equivalence proofs and provide insights into how both approaches scale to more complex programming languages such as in [Pitts, 2005, Crary and Harper, 2007].

Chapter 5

Conclusion

This thesis described an indexed type theory with support for coinductive reasoning. Coinductive types were defined by observations while coinductive proofs are represented using copattern matching. We provided a simple framework to allowing us to use custom index domains ranging from simple like natural numbers or sophisticated like contextual LF. We proved subject reduction and progress for this language. We also designed a non-deterministic coverage algorithm mimicking interactive splitting, and criteria for termination. Those criteria are can be checked statically to accept safe programs. We showed them correct by defining a translation to a core calculus. We proved the core calculus to be normalizing, the translation to be evaluation preserving, and the criteria to be sufficient for the translation.

In addition, we implemented a prototype implementation for this language with contextual LF as index domain in the proof assistant Beluga by extending it with copattern matching. We showcase this prototype with a case study on Howe's method. The case study used coinductive types to define a simulation on a simply-typed lazy λ-calculus. It leverages Beluga's support for open terms and substitutions to define the auxiliary Howe relation on open terms and prove it to be substitutive. This property is key to show the equivalence and is often a challenge in proof assistants with limited support for meta reasoning for programming languages.

5.1 Future Work

Coinductive Indices

We believe our framework to be able to handle coinductive index domains in addition to inductive ones but we haven't spelled out such a domain. In particular, providing and adequate definition of equality and unification for coinductive definition can prove to be challenging. An example of streams as indices with a basic corecursor and observations `head` and `tail` might not have such a simple answer for equality. It might be needed to design a notion of bisimulation on top of it. Manipulating bisimulations in addition to equalities could prove to be challenging given the intensional nature of our theory. It is also possible that we can reuse some insights from the extensional type theories of NuPRL or Isabelle/HOL to obtain a domain of coinductive indices. Such development would also give insights towards a fully dependent copattern language.

Internal Encoding of Equality Constraints

The Agda language has support for copatterns and can explicitly spell out equality guards to simulate the description of our indexed coinductive types. On the other hand, Agda offers intrinsic support to handle equality constraints for inductive types. For example, we can define vectors of `A`'s as

```
data Vec : Nat → type =
| nil : Vec 0
| cons : A → Vec n → Vec (suc n)
```

which is equivalent to

```
data Vec (n : Nat) : type =
| nil : n = 0
| cons : Σm.n = (suc m) * A * Vec m
```

Defining such vectors coinductively can only be done as

```
codata Vec (n : Nat) : type =
```

```
| head : Πm.n = (suc m) → A
| tail : Πm.n = (suc m) → Vec m
```

Intrinsic support could lead to definitions like the following:

```
codata Vec : Nat → type =
| head : Vec (suc m) → A
| tail : Vec (suc m) → Vec m
```

Such addition leads us to ask how we can handle coinductive indices that cannot simply be pattern matched on. For example, if instead we were to use coinductive natural numbers that are defined as:

```
codata CoNat : type =
| out : CoNat'
and data CoNat' : type =
| zero : CoNat'
| suc : CoNat → CoNat'
```

In this case, we can only pattern match once we apply the observation `out`. It gets even more complicated if the codata has more than one observation. Moreover, as we mentioned above, the question of an adequate representation of equality for such type is not so simple.

Beluga Implementation

At the time of the writing of this thesis, Beluga's termination checking algorithm is lacking support for coinduction. The criteria and the translation from Chapter 3 does provide an answer as to how to extend it but the work remains to be done. Extending Beluga to second-order quantification would also gives us the opportunity to abstract over relations and thus encode adequate and compatible relations. This would allow us to prove that our bisimulation from Chapter 4 coincides with contextual equivalence (as done by Ambler and Crole [1999]), as contextual equivalence is the largest adquate and compatible relation [Lassen, 1998, Pitts, 2011].

Carrying out other case studies in Beluga would also be interesting. Step-indexed logical relation [Ahmed, 2006] have been used to show program equivalence. Beluga has already been used to show logical relation proofs [Cave and Pientka, 2015]. One could compare the two developments and see how they scales to more complex programming languages such as in [Pitts, 2005].

Bibliography

A. Abel. Termination checking with types. *RAIRO - Theoretical Informatics and Applications*, 38(4):277–319, 3 2004.

A. Abel and R. Matthes. Fixed points of type constructors and primitive recursion. In J. Marcinkowski and A. Tarlecki, editors, *18th International Workshop on Computer Science Logic (CSL'04)*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2004.

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)*, pages 185–196, 2013. doi: 10.1145/2500365.2500591.

A. Abel and B. Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2 (61 pages), 2016. ISSN 1469-7653. doi: 10.1017/S0956796816000022. URL `http://journals.cambridge.org/article_S0956796816000022`.

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

S. Abramsky. A domain equation for bisimulation. *Inf. Comput.*, 92(2):161–218, 1991. doi: 10.1006/inco.1991.9999. URL `https://doi.org/10.1006/inco.1991.9999`.

Agda team. The Agda Wiki, 2014.

A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *15th European Symposium on Programming (ESOP'06)*, pages 69–83. Springer, 2006. ISBN 978-3-540-33096-7. doi: doi:10.1007/11693024_6. URL `http://dx.doi.org/10.1007/11693024_6`.

K. Y. Ahn. *The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed Types*. PhD thesis, Portland State University, 2014.

K. Y. Ahn and T. Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034807. URL `http://doi.acm.org/10.1145/2034773.2034807`.

G. Allais, J. Chapman, C. McBride, and J. McKinna. Type-and-scope safe programs and their proofs. In Y. Bertot and V. Vafeiadis, editors, *6th Conference on Certified Programs and Proofs (CPP'17)*, pages 195–207. ACM, 2017. doi: 10.1145/3018610.3018613. URL `http://doi.acm.org/10.1145/3018610.3018613`.

S. Ambler and R. L. Crole. Mechanized operational semantics via (co)induction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, Lecture Notes in Computer Science (LNCS 1690), pages 221–238. Springer, 1999. ISBN 3-540-66463-7.

J. Andronick and A. P. Felty, editors. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, 2018. ACM. ISBN 978-1-4503-5586-5. URL `http://dl.acm.org/citation.cfm?id=3176245`.

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

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

D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7 (2):1–89, 2014a. ISSN 1972-5787. doi: 10.6092/issn.1972-5787/4650. URL `https://jfr.unibo.it/article/view/4650`.

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

H. Basold and H. Geuvers. Type theory based on dependent inductive and coinductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, page 327–336. Association for Computing Machinery, 2016.

J. Bengtson and J. Parrow. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science*, 5(2), 2009. URL `http://arxiv.org/abs/0809.3960`.

J. Bengtson, J. Parrow, and T. Weber. Psi-calculi in Isabelle. *J. Autom. Reasoning*, 56(1): 1–47, 2016. doi: 10.1007/s10817-015-9336-2.

N. Benton, C. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in coq. *J. Autom. Reasoning*, 49(2):141–159, 2012. doi: 10.1007/s10817-011-9219-0. URL `http://dx.doi.org/10.1007/s10817-011-9219-0`.

U. Berger and A. Setzer. Undecidability of equality for codata types. In *Coalgebraic Methods in Computer Science*, pages 34–55. Springer International Publishing, 2018.

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

J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for isabelle/hol. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving*, pages 93–110, Cham, 2014. Springer International Publishing.

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

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

A. Cave and B. Pientka. First-class substitutions in contextual type theory. In *8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM Press, 2013.

A. Cave and B. Pientka. A case study on logical relations using contextual types. In I. Cervesato and K.Chaudhuri, editors, *10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, pages 18–33. Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.

A. Cave and B. Pientka. Mechanizing Proofs with Logical Relations – Kripke-style. *Mathematical Structures in Computer Science*, 28(9):1606–1638, 2018. doi: 10.1017/S0960129518000154.

K. Chaudhuri. A two-level logic perspective on (simultaneous) substitutions. In Andronick and Felty [2018], pages 280–292. ISBN 978-1-4503-5586-5. doi: 10.1145/3167093. URL `http://doi.acm.org/10.1145/3167093`.

K. Chaudhuri, M. Cimini, and D. Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In *CPP*, pages 157–166. ACM, 2015.

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

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

J. Cockx and A. Abel. Elaborating dependent (co)pattern matching. *Proceedings of the ACM on Programming Languages*, 2(ICFP), July 2018.

J. Cockx and A. Abel. Elaborating dependent (co)pattern matching: No pattern left behind. *Journal of Functional Programming*, 30:e2, 2020.

T. Coquand. Pattern matching with dependent types. In *Informal Proceedings of Workshop on Types for Proofs and Programs*, pages 66–79. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992.

T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-58085-0.

K. Crary and R. Harper. Syntactic logical relations for polymorphic and recursive types. *Electr. Notes Theor. Comput. Sci.*, 172:259–299, 2007.

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

J. Errington. Mechanizing metatheory interactively. Master's thesis, Mcgill University, 2020.

A. P. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

J. Ford and I. A. Mason. Formal foundations of operational semantics. *Higher-Order and Symbolic Computation*, 16(3):161–202, 2003.

D. R. Ghica and G. McCusker. Reasoning about idealized ALGOL using regular languages. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Proceedings*, vol-

ume 1853 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 2000. doi: 10.1007/3-540-45022-X_10. URL `https://doi.org/10.1007/3-540-45022-X_10`.

E. Giménez. Codifying guarded definitions with recursive schemes. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '94, pages 39–59, 1995.

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.

J. Y. Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. These d'état, Université de Paris 7, 1972.

H. Goguen, C. Mcbride, and J. Mckinna. Eliminating dependent pattern matching. In *Joseph Goguen Festschrift*, volume 4060 of *Lecture Notes in Computer Science (LNCS)*, pages 521–540. Springer, 2006a.

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

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

T. Hagino. Codatatypes in ML. *Journal of Symbolic Logic*, 8(6):629–650, 1989.

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

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

D. Hausmann, T. Mossakowski, and L. Schröder. Iterative circular coinduction for cocasl in isabelle/hol. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering*, pages 341–356, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

D. Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In E. L. Gunter and A. P. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, Lecture Notes in Computer Science (LNCS 1275), pages 153–169. Springer, 1997. doi: 10.1007/BFb0028392. URL `http://dx.doi.org/10.1007/BFb0028392`.

F. Honsell, M. Miculan, and I. Scagnetto. Π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.12 edition, 2020.

R. Jacob-Rao, B. Pientka, and D. Thibodeau. Index-stratified types. In H. Kirchner, editor, *3rdd International Conference on Formal Structures for Computation and Deduction (FSCD'18)*, LIPIcs, pages 19:1–19:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, January 2018.

J. Kaiser, S. Schäfer, and K. Stark. Autosubst 2: Towards reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '17, pages 10–14, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5374-8. doi: 10.1145/3130261.3130263.

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

P. Laforgue and Y. Régis-Gianas. Copattern matching and first-class observations in ocaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, page 97–108, New York, NY, USA, 2017. Association for Computing Machinery.

S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Dept of Computer Science, Univ of Aarhus, 1998.

D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ml. In *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 173–184. ACM Press, 2007. ISBN 1-59593-575-4.

S. Lenglet and A. Schmitt. Ho(π) in Coq. In Andronick and Felty [2018], pages 252–265. ISBN 978-1-4503-5586-5. doi: 10.1145/3167083. URL `http://doi.acm.org/10.1145/3167083`.

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.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., USA, 1994.

I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(03):287–327, 1991.

C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

C. McBride. Let's see how things unfold: Reconciling the infinite with the intensional (extended abstract). In *Algebra and Coalgebra in Computer Science*, pages 113–126. Springer Berlin Heidelberg, 2009.

R. McDowell and D. Miller. A logic for reasoning with higher-order abstract syntax. In G. Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445. IEEE Computer Society Press, July 1997.

R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus: Preliminary report. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 3, 1996.

C. McLaughlin, J. McKinna, and I. Stark. Triangulating context lemmas. In Andronick and Felty [2018], pages 102–114. ISBN 978-1-4503-5586-5. doi: 10.1145/3167081. URL `http://doi.acm.org/10.1145/3167081`.

N. Mendler, R. L. Constable, and P. Panangaden. Infinite objects in type theory. In *Proceedings of First IEEE Symposium on Logic in Computer Science*, LICS, pages 249–257, 1986.

N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Symposium on Logic in Computer Science (LICS'87)*, pages 30–36. IEEE Computer Society, 1987.

N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991. ISSN 0168-0072. doi: http://dx.doi.org/10.1016/0168-0072(91)90069-X. URL `http://www.sciencedirect.com/science/article/pii/016800729190069X`.

D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1st edition, 2012. ISBN 052187940X, 9780521879408.

D. Miller and C. Palamidessi. Foundational aspects of syntax. *ACM Comput. Surv.*, 31(3es), Sept. 1999. ISSN 0360-0300. doi: 10.1145/333580.333590. URL `http://doi.acm.org/10.1145/333580.333590`.

D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Log.*, 6 (4):749–783, 2005.

R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1 – 22, 1977. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(77)90053-6. URL `http://www.sciencedirect.com/science/article/pii/0304397577900536`.

R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc, 1982.

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

A. Momigliano. A supposedly fun thing I may have to do again: A HOAS encoding of Howe's method. In *7nth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'12)*, pages 33–42. ACM, 2012. ISBN 978-1-4503-1578-4. URL `http://doi.acm.org/10.1145/2364406.2364411`.

A. Momigliano and A. Tiu. Induction and co-induction in sequent calculus. In M. Coppo, S. Berardi, and F. Damiani, editors, *Post-proceedings of TYPES 2003*, Lecture Notes in Computer Science (LNCS 3085), pages 293–308, Jan. 2003.

A. Momigliano, S. Ambler, and R. L. Crole. A Hybrid encoding of Howe's method for establishing congruence of bisimilarity. *Electr. Notes Theor. Comput. Sci.*, 70(2), 2002.

A. Momigliano, B. Pientka, and D. Thibodeau. A case study in programming coinductive proofs: Howe's method. *Mathematical Structures in Computer Science*, 29(8):1309–1343, 2019.

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

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

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.

D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, 1981.

D. M. R. Park. On the semantics of fair parallelism. In *Abstract Software Specifications*, pages 504–526, 1979.

J. Parrow, J. Borgström, P. Raabjerg, and J. Å. Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, 24(2), 2014. doi: 10.1017/S0960129513000170.

L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conference on Automated Deduction (CADE-12), Nancy, France*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 1994. ISBN 3-540-58156-1.

F. Pfenning. Computation and deduction, 1997.

F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), pages 202–206. Springer, 1999.

B. Pientka. Verifying termination and reduction properties about higher-order logic programs. *Journal of Automated Reasoning*, 34(2):179–207, 2005.

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

B. Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *Journal of Functional Programming*, 1(1–37), 2013.

B. Pientka and A. Abel. Structural recursion over contextual objects. In T. Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

B. Pientka and A. Cave. Inductive Beluga:Programming Proofs (System Description). In A. P. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer, 2015.

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

B. Pientka and J. Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In J. Giesl and R. Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer, 2010.

A. M. Pitts. Operationally Based Theories of Program Equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, 1997.

A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. Cambridge University Press, Nov. 2011. ISBN 978-1-107-00497-9.

A. Setzer, A. Abel, B. Pientka, and D. Thibodeau. Unnesting of copatterns. In G. Dowek, editor, *Joint International Conference on Rewriting and Typed Lambda Calculi (RTA-TLCA'14)*, Lecture Notes in Computer Science (LNCS 8560), pages 31–45. Springer, 2014. doi: 10.1007/978-3-319-08918-8_3.

W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2): 198–212, 1967.

D. Thibodeau, A. Cave, and B. Pientka. Indexed codata. In J. Garrigue, G. Keller, and E. Sumii, editors, *21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, pages 351–363. ACM, 2016.

A. Tiu and D. Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. Comput. Logic*, 11(2):1–35, 2010. ISSN 1529-3785. URL `http://doi.acm.org/10.1145/1656242.1656248`.

A. Tiu and A. Momigliano. Cut elimination for a logic with induction and co-induction. *J. Applied Logic*, 10(4):330–367, 2012. URL `https://doi.org/10.1016/j.jal.2012.07.007`.

D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *2012 27th Annual IEEE Symposium on Logic in Computer Science (LICS12)*, pages 596–605, 2012.

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

T. Uustalu and V. Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10:5–26, 1999.

H. Xi. Applied type system. In *TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2004.

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

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM Press, 2003. doi: 10.1145/604131.604150.

N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, to appear (draft available on Noam's webpage), 2007.

N. Zeilberger. Focusing and higher-order abstract syntax. In *Conference Record of the 35th Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, page to appear(draft available on Noam's webpage). ACM, 2008. URL `draftavailableonNoam'swebpage`.

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