TOWARDS FORMALIZING PARAMETRICITY FOR NESTED TYPES IN AGDA

A Thesis
by
DANIEL JEFFRIES

Submitted to the School of Graduate Studies
at Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

August 2021
Department of Computer Science

TOWARDS FORMALIZING PARAMETRICITY FOR NESTED TYPES IN AGDA

A Thesis
by
DANIEL JEFFRIES
AUGUST 2021

APPROVED BY:

____
Patricia Johann, Ph.D.
Chairperson, Thesis Committee

____
Raghuveer Mohan, Ph.D.
Member, Thesis Committee

____
Andrew Polonsky, Ph.D.
Member, Thesis Committee

____
Rahman Tashakkori, Ph.D.
Chairperson, Department of Computer Science

____
Mike McKenzie, Ph.D.
Dean, Cratis D. Williams School of Graduate Studies

Abstract

TOWARDS FORMALIZING PARAMETRICITY FOR
NESTED TYPES IN AGDA

Daniel Jeffries
B.S., Appalachian State University

Chairperson: Dr. Patricia Johann

Recent work by Johann, Ghiorzi, and Jeffries presents a Hindley-Milner-style calculus whose type system includes a primitive type formation rule for constructing nested types directly as fixpoints. The term calculus for this type system supports primitive pattern matching, functorial map functions, and fold combinators for nested types. A parametric model of the calculus is given in terms of category theory, using fixpoints of higher-order functors to interpret nested types. In this thesis, we formalize the syntax of the calculus and partially formalize its model in the proof assistant and functional programming language Agda. To prove a proposition in Agda, the user first gives a formal definition of the proposition as a type T. Then the user defines a function of type T, and Agda's type-checker checks whether the user-defined function has the correct type. Since the model is defined in terms of category theory, we use the agda-categories library to formalize the categorical constructs used in the model. We introduce the syntax and semantics of the calculus as they are being formalized.

Acknowledgments

I would like to thank several individuals for their support. First, I would like to thank my committee chairperson, Dr. Patricia Johann. Your guidance has been invaluable throughout my M.S. studies, and my research and writing skills have benefited greatly from your expertise. I am also grateful for the fact that you have exposed me to many new concepts in computer science and that you have provided a stimulating environment in which to study these concepts. I would also like to thank Dr. Enrico Ghiorzi for the knowledge he has imparted to me throughout our time working together, and I am grateful to both him and Dr. Johann for providing a welcoming environment when I was just starting to work on projects with them. I would also like to thank my committee members, Dr. Andrew Polonsky and Dr. Raghuveer Mohan, for their support and feedback. I am also grateful to Dr. Polonsky for introducing me to Agda and proof assistants. I thank Dr. Mohan for providing me with the opportunity to work on my first major undergraduate project, for helping me to develop my problem solving skills through competitive programming, and for always being available for stimulating conversations about algorithms. I thank the entire committee for their patience and for taking the time to read and evaluate this thesis. The work reported in this thesis was supported by NSF awards 1713389 and 1906388

Contents

Chapter 1

Introduction

Previous work [5] has yielded a calculus $\mathcal{N}$ of nested types and a parametric model for the calculus. In this thesis we present a formal implementation [4] of that calculus and model in the proof assistant Agda. The focus of this thesis is the implementation itself, but the syntax and the semantics of the calculus will be presented as it is being formalized.

Formalizing the calculus and model in Agda serves several purposes. The calculus and model are both nontrivial, so our formalization efforts should provide insights on common issues faced when formalizing languages in a proof assistant. It also serves a practical purpose because a complete formalization of the calculus and parametricity results will increase confidence in the correctness of our constructions and can potentially serve as a proof artifact alongside the original paper.

1.1 Proof Assistants

1.1.1 What is a proof assistant?

A *proof assistant*, or *interactive theorem prover*, is software that assists users in constructing and verifying the correctness of proofs in a particular logical system. The role of a proof assistant is to mechanize the aspects of proof development that are completely algorithmic, such as unwinding definitions and verifying that the inference rules of the logic are applied correctly. The act of constructing a proof is primarily left up to the user, although proof assistants can provide helpful hints and feedback. This justifies the term *interactive theorem prover*. This interaction with the user is in constrast with *automated theorem proving*, in which inference rules are applied algorithmically in search of a proof with no user interaction. In other words, an automated theorem prover mechanizes the construction of the proof while a proof assistant mechanizes the verification. Some proof assistants incorporate techniques from automated theorem proving, but automated theorem proving is beyond the scope of this thesis and receives no further mention.

Proof assistants can be used for many different applications, including verifying the correctness of a compiler, proving that an implementation of a data structure satisfies an abstract specification, and formalizing mathematics. Many proof assistants can also be viewed as programming languages in their own right and include libraries for standard constructions like any other programming language.

To prove something in a proof assistant, the user first specifies it as a proposition. As the user fills in the proof piece by piece, the proof assistant can automatically verify whether each step is valid according to the rules of the logic. Many proof assistants allow the user to place *holes* in a proof that act as placeholders to be filled in later. These holes allow the user to write a partial proof that can still be checked. If the partial proof is verified, then it might be possible to complete the proof. But if the partial proof is marked as invalid, then the user needs to back up and try a different approach. A proof assistant can also track and display the context — i.e., what has been proven so far — and the goal proposition — i.e., what remains to be proven — for each hole. These features allow the user to focus on the content of the proof rather than manually performing the careful bookkeeping required in proof development.

Although the construction of proofs is not automated like verification is, a proof assistant can often fill in some of the details. For example, if several cases are required for the next step of a proof, the proof assistant can fill in a template with a hole for each case.

1.1.2 Why use a proof assistant?

A reader might ask at this point what benefit there is to be gained from formalizing a proof in a proof assistant. As mentioned above, the most important benefit of formalizing a proof is increased confidence in its correctness, particularly the technical details. Having a machine-checked formalization of a proof increases the programmer's (and others') confidence that the proof is correct. In fact, it is becoming increasingly common for papers in programming languages (PL) research to be accompanied by a *proof artifact*, or an implementation of the main results in a proof assistant. A proof artifact also allows others to interact directly with the content of the paper in a familiar environment.

During the proof development phase, the user stands to benefit from existing libraries and reuse of proofs that have already been implemented in the proof assistant. Sometimes in a proof, the author cites "well-known" results without trotting out the details or even giving a proper citation. If the well-known result is also formalized in the proof assistant, then the user can simply import the required theorem and reuse it in their proof. This is also helpful for anyone reading the proof that might not be as familiar with the known results.

This leads to another benefit, which is that every part of the proof must be made explicit when formalized. This makes the formalized proofs self-contained and more approachable to those with less experience. Of course, some may view this as a detriment because it means even trivial proofs must be included when they might be elided on paper. However, if the elided proof really is trivial, then the proof assistant should be able to do most of the work automatically.

The greatest benefit of formalization during proof development is the bookkeeping the proof assistant provides for keeping track of the details of the proof. Rather than manually unwinding a complex definition to determine what must be proven next, the user can rely on the proof assistant to compute the next proof goal.

Once all the definitions and theorems of a system have been implemented in a proof assistant, the formalization can be helpful for generating examples. In a system with lots of details and complex syntax, this can be much more efficient than constructing nontrivial examples by hand.

While there are many reasons we might decide to formalize a proof, it should be said that a proof assistant is not a magic bullet. Even formalized proofs may contain mistakes, and although there have been efforts to prove the correctness of proof assistant implementations themselves, there can always be bugs in these implementations.

1.2 Agda and Dependent Types

Agda [8] is a functional programming language based on the theory of dependent types. Agda shares much of its syntax with, and is implemented in, Haskell. Like most functional languages, Agda allows the user to define their own types and write functions over these types.

Unlike Haskell, Agda programs can contain unicode characters. Since Agda is often used to formalize mathematics and logic, this feature allows us to use the standard notation for most logical and mathematical symbols.

To explore the basic syntax of Agda, let us first consider the polymorphic identity function:

```
id : ∀ {A : Set} → A → A
id x = x
```

The type of id says that for any type A, id takes an element of A as input and returns an element of A.

The curly braces around A indicate it is an *implicit argument*. In most cases, implicit arguments do not need to be provided by the programmer when calling a function, but they can be included by placing curly braces around the argument. For example, if we have defined a type MyType, then we can define the identity function on this type as a special case of the polymorphic identity function. The following definition type-checks with or without the "{MyType}"

```
idMyType : MyType → MyType
idMyType = id {MyType}
```

since the type-checker can determine the correct value for A based on the type signature of idMyType.

In Agda, types themselves are elements of *universe types*, with the base universe denoted by Set. All "base" types fit into Set, e.g., the type of booleans, the type of natural numbers, list types, function types (where domain and codomain are themselves "base" types), etc. Universe types fit into other universe types, and there is an entire hierarchy of universes with Set belonging to Set$_1$, Set$_1$ belonging to Set$_2$, and so on.

The data keyword is used to define new types. Data types are defined inductively by their *data constructors* (sometimes referred to simply as *constructors*). The simplest data types in Agda are the unit type and the empty type.

```
data ⊤ : Set where
  tt : ⊤

data ⊥ : Set where
```

The unit type represents a singleton set and has a single constructor tt representing its sole element. The unit type is called ⊤ ("top") in Agda. The empty type represents the empty set. The empty type has no constructors and is called ⊥ ("bottom") in Agda. The presence of Set between the colon and the keyword where indicates that the data type being defined has type Set, i.e., fits inside the base universe. Another data type is Bool, which has two constructors, true and false:

```
data Bool : Set where
  true : Bool
  false : Bool
```

Functions over data types are defined by *pattern matching*. When writing a function that takes a data type as input, pattern matching allows us to perform case analysis and do different computations for each data constructor. For example, to write a function from Bool to some other type, we can give a case for true and a case for false:

```
negate : Bool → Bool
negate true = false
negate false = true
```

We can also pattern match on multiple arguments, or choose not to match on an argument. If the value of the un-matched argument is needed, we can use a variable. Otherwise we can use an underscore to indicate the value is not used:

```
and : Bool → Bool → Bool
and true b = b
and false _ = false
```

Types with no elements, such as the empty type ⊥, are treated as special cases during pattern matching. Agda recognizes that if a type has no possible constructors, then there are no cases to consider when pattern matching on this type. Therefore, any function that takes an argument of type ⊥ can be defined trivially. This behavior is represented using the *absurd pattern* in Agda, denoted by (). For example, we can define a function exFalso from the empty type to any other type:

```
exFalso : ∀ {A : Set} → ⊥ → A
exFalso ()
```

We can also define *inductive* data types in Agda by including constructors that build up new elements from smaller ones. A canonical example is the type of natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

A natural number is either zero or a successor of another natural number. Functions over inductive data types can be defined using recursion:

```
_+n_ : ℕ → ℕ → ℕ
n +n zero = n
n +n suc m = suc (n +n m)
```

The underscores in the name of the addition operator indicate it is an infix operator and the arguments go where the underscores are. Another common inductive data type is the List data type:

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A
```

Unlike the other types we have defined so far, List is *parameterized* by a type A. This means each choice of A gives rise to a type List A whose elements are finite lists of elements of A. For example, List ℕ is the type of lists of natural numbers, with elements nil, (cons 2 nil), etc.

1.2.1 Dependent Types

Earlier we observed that Agda is based on the theory of dependent types. So far, we have only seen types that are constructed from other *types*, e.g., List ℕ. Agda's dependent type system allow us to define types that depend on *terms*, i.e., elements of some other type. The canonical example of a dependent type is the Vec data type:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n : ℕ} → A → Vec A n → Vec A (suc n)
```

The type Vec is parameterized by a type A and *indexed* by natural numbers. This means for each type A, Vec A is a function that takes a natural number n and produces the type Vec A n. The type parameter A gives the type of the elements and the natural number index gives the length of the vector.

An important distinction between type parameters and type indices is that fixing the indices and choosing different parameters generally produces types of the same "shape," while choosing different indices can produce types of very different "shape." For example, the elements of Vec ℕ n and Vec Bool n have the same index n and are essentially the same. These types both contain vectors of length n, and the only difference is in what kind of value fills each position in the vectors. On the other hand, the only element of type Vec A zero is the empty vector [], while the type Vec A (suc n) contains only nonempty vectors of the form (x :: xs). There is no way to express these constraints with the parameterized data type List since the type of a list provides no information about its structure.

Another canonical and very useful dependent data type is the type of propositional equality:

```
data _≡_ : ∀ {A : Set} → A → A → Set where
  refl : ∀ {A : Set} {a : A} → a ≡ a                    (1)
```

This type is called propositional equality to contrast it with *definitional* equality, which considers two terms equal because the computations that define them evaluate to the exact same result. For example, n +n zero is definitionally equal to n. The propositional equality type takes no parameters and is indexed by a type (implicitly) and two elements of this type. The type of the refl constructor (for "reflexive") can be read as "for any type A and an element a of A, a is equal to itself. This type may not seem particularly interesting, but it is very useful when used with pattern matching.

For instance, we can show that applying a function preserves equality. Let us write the signature with a hole on the right hand side.

```
cong : ∀ {A B : Set} {a1 a2 : A} → (f : A → B) → a1 ≡ a2 → f a1 ≡ f a2
cong f p = {!!} -- Goal:  f a1 ≡ f a2
```

We have a proof (p : a1 ≡ a2) and need to produce a proof of type (f a1 ≡ f a2). If we pattern match on p, then Agda will determine that a1 and a2 must be the same based on the type of the refl constructor. Now the goal reduces to (f a1 ≡ f a1), which we can simply prove using refl:

```
cong : ∀ {A B : Set} {a1 a2 : A} → (f : A → B) → a1 ≡ a2 → f a1 ≡ f a2
cong f refl = refl -- Goal:  f a1 ≡ f a1
```

Agda also includes *record* types, which can be used to bundle several types together as a single type. The primary advantage of record types is that they give names to the data elements in a dependent product. Consider the example record type:

```
record LengthAndVector (A : Set) : Set where
  constructor [_,_]
  field
    length : ℕ
    vector : Vec A length

  double-length : ℕ
  double-length = length +n length
```

Records can take parameters, and they must fit into some universe type. In this case, LengthAndVector takes a parameter A : Set and LengthAndVector itself belongs to Set. The contents of a record type are determined by its *fields*. The record type LengthAndVector has two fields: length, and vector (which depends on length). Record types can also include local definitions that use the data given by the fields. For example, LengthAndVector includes a local definition for double-length. To construct a value of a record type, we must construct a value for each of its fields. To do this, we use the record{...} syntax:

```
lv : LengthAndVector Bool
lv = record { length = 1 ; vector = true :: [] }
```

Or, since this record type includes a constructor declaration, we can use the name of the declared constructor, as in:

```
lv' : LengthAndVector Bool
lv' = [ 1 , (true :: []) ]
```

These two definitions are equivalent; the latter is just syntactic sugar for the former. To access the local definitions for a record, we can use the following syntax:

```
lv-double-length : ℕ
lv-double-length = LengthAndVector.double-length lv
```

1.3 Propositions As Types

The previous section introduces Agda as a functional programming language, but it is also a proof assistant. This brings us to a concept called the Curry-Howard correspondence or *propositions-as-types*. The propositions-as-types paradigm is the recognition of the direct correspondence between logical systems and typed functional programming languages. In the case of Agda, this means that a proposition P can be viewed as a type and vice versa. Thus we can use the phrases "terms of type P," "elements of P," and "proofs of P" interchangeably. Under this view, proofs of P are programs that produce elements of the corresponding type!

For example, the types ⊤ and ⊥ correspond to the true proposition and false proposition, respectively. A function between two types corresponds to an implication between two propositions. The function exFalso above can be viewed as a proof that, from falsehood, anything can be proven. An even simpler example from propositional logic is the rule *modus ponens* that says "if (A $\implies$ B) and A are provable, then B is provable," where $\implies$ is implication in propositional logic. This rule corresponds to function application in Agda:

```
app : ∀ {A B : Set} → (A → B) → A → B
app f x = f x
```

We can also represent logical conjunction and disjunction as data types:

```
data _×'_ (A B : Set) : Set where
  _,_ : A → B → (A ×' B)

data _⊎_ (A B : Set) : Set where
  inj₁ : A → (A ⊎ B)
  inj₂ : B → (A ⊎ B)
```

To prove the conjunction of A and B (i.e., A ×′ B)[1] we need a proof of A and a proof of B. To prove the disjunction of A and B (i.e., A ⊎ B) we need either a proof of A or a proof of B. The negation of a proposition is defined by:

```
¬_ : Set → Set
¬ P = P → ⊥
```

In terms of propositions, this definition says that P is considered false when P can be used to prove a false proposition. Not every type has an obvious translation as a proposition —e.g., natural numbers and list types do not —but there is a correspondence nevertheless.

[1] We rename the product type _ × _ from the standard library to _ ×′ _ and use _ × _ as the name of the type former in the calculus $\mathcal{N}$.

It is worth noting that proofs in Agda must be *constructive*. A constructive proof of proposition P is an algorithm that produces an object satisfying P (or an element of P when viewed as a type). In classical (i.e., non-constructive) logic, a proposition can be proven simply by showing that *there must exist* an object satisfying the proposition. It is not required in classical logic to actually *produce* such an object. Such a non-constructive proof will not do in Agda.

1.4 Nested Types and The Calculus $\mathcal{N}$

The calculus $\mathcal{N}$ that we are formalizing in this thesis is itself a functional programming language. It includes a type system with base types like $\top$ and $\bot$ and rules for constructing more complex types, including an inductive type former similar to the data declaration in Agda. In particular, the type system can express *nested types*. Nested types are less expressive than dependent types but generalize the commonly used *algebraic data types* (ADTs) like List A and $\mathbb{N}$.

An ADT must be defined only in terms of itself, in the following sense: if a constructor for the data type T A takes an argument whose type is some instance of T, the instance is required to be T A. For example, in the declaration of List A, cons cannot take an argument of type List (A ×′ ⊤) or List B for any other type B different from A: only List A is allowed.

Nested types are a more general class of types that can, for example, express the type of perfectly balanced binary trees with data at the leaves. This nested type is defined in Agda as:

```
data PTree (A : Set) : Set where
  pleaf : A → PTree A                                  (2)
  pnode : PTree (A ×' A) → PTree A
```

Nested types are more general than ADTs because nested types impose fewer restrictions on the types of constructors. In particular, the constructors for a nested type T A can take arguments whose types are different instances of T. For example, in the declaration of PTree A, the pnode constructor takes an argument of type PTree (A ×′ A). This additional structure in the data type declaration of PTree enforces the constraint that the inhabitants of PTree A are perfectly balanced binary trees with elements of A at the leaves. Thus PTree can be seen as a version of the List ADT with an additional constraint that the lists must have lengths that are powers of two.

The calculus $\mathcal{N}$ also includes term syntax and inference rules for constructing programs over nested types. The term syntax includes built-in operators for expressing pattern matching, a particular stylized form of recursion, and the uniform mapping of a function over a data type.

Once the syntax of types and terms has been specified for a language, we can consider models of the language. A model assigns meaning, or semantics, to the syntax of a language. Models allow us to reason about the constructs in a language based on their semantic properties rather than their syntax. For example, a particular model may allow us to prove that a language is logically consistent, or that all programs written in the language are terminating.

Our model, defined in terms of category theory, gives a semantics based on sets and a semantics based on relations over these sets. The set semantics maps types to sets and terms to elements of these sets. The relation semantics maps types to relations and terms to pairs of related elements, exactly in parallel to the set semantics. Our model is *parametric*, meaning we have an Identity Extension Lemma, which expresses a uniformity property of the interpretations of types, and an Abstraction Theorem, which expresses that the set interpretations of terms in related environments are related in the relational interpretation of the term's type. Parametricity allows us to prove properties of programs based solely on their types, yielding so-called *free theorems*. Parametricity and its consequences will be explained further after we formalize the relation semantics.

The structure of the thesis is as follows. In Chapter 2 we present the syntax of types in the calculus $\mathcal{N}$ and formalize the syntax and inference rules. In Chapter 3 we define some basic concepts from category theory and show how these concepts are formalized in Agda. In Chapter 4 we present and formalize the set semantics for our type system, and in Chapter 5 we formalize the term syntax. In Chapter 6 we give a partial formalization of the set semantics for terms and describe some limitations of our formalization. In Chapter 7 we formalize the relation semantics of types. In Chapter 8 we give a partial formalization of parametricity for our model. Chapter 9 concludes and discusses future work.

Chapter 2

Syntax of Types

In this chapter we describe the syntax of types for the calculus $\mathcal{N}$ and its formalization in Agda. To do this, we must formalize the notions of type variables, type contexts, type formation rules, and other constructs used to define the syntax of $\mathcal{N}$. We will also define weakening and substitution functions for types and prove that these operations preserve the type formation rules, i.e., if a type is well-formed, then it is still well-formed after substituting or weakening.

2.1 Type Variables

The type expressions of $\mathcal{N}$ are built up from type variables, constant types $\mathbb{0}$ and $\mathbb{1}$, and type formers such as $+$ and $\times$. Type variables are split into two classes: *type constructor* variables and *functorial* variables. Functorial variables play a special role in the formation of types, indicating that the resulting type should act as a functor with respect to these variables. For a type to be considered functorial, it must support a map operation that uniformly applies a function to the data structure without modifying its shape. For example, the List data type is functorial. Given a list

```
cons x1 (cons x2 (cons x3 nil))
```

we can apply a function f to each element without modifying the structure of the list to get:

```
cons (f x1) (cons (f x2) (cons (f x3) nil))
```

When constucting the List type in the calculus $\mathcal{N}$, we can use either a functorial variable or a type constructor variable for the type of the list elements, depending upon whether we need a map operation for lists of elements of that type. A type constructed with functorial variables should support a map operation for those variables, but we do not necessarily expect this for type constructor variables. Type constructor variables are also referred to as *non-functorial variables.* Types can contain both functorial variables and type constructor variables.

There are many different approaches to representing variables in a formalization. Variables can be assigned user-selected names, or they can be assigned numeric values based on their positions in an expression. The latter approach is called *De Bruijn indexing* [2] and avoids the issue of *alpha equivalence* that arises when there are two expressions that differ only in the names of their bound variables. For example, $\lambda x.x$ and $\lambda y.y$ are alpha equivalent expressions. However, we are not really concerned with alpha equivalence in this formalization, so we have chosen to formalize variables using names. The exact representation of variable names is not important as long as the type for variable names supports decidable equality for distinguishing between different variables. A proposition P is *decidable* if there exists a function that always produces either a "yes" answer indicating P is true or a "no" answer indicating P is false. The type of variable names is called Id in the formalization:

```
Id : Set
Id = String
```

In this formalization we use Agda's built-in String type for variable names, but other types such as the type of natural numbers would also work. To distinguish our two sets of variables, we can simply define two different data types, one for each kind of type variable:

```
data TCVar : ℕ → Set where
  _^T_ : Id → (k : ℕ) → TCVar k

data FVar : ℕ → Set where
  _^F_ : Id → (k : ℕ) → FVar k
```

Each of these data types is indexed by a natural number k, giving the arity of the variables included in the type. This means an element of TCVar k (resp., FVar k) is a k-ary type constructor (resp., functorial) variable.

2.2 Type Expressions

Now that we have defined type variables, we can define the grammar of type expressions for $\mathcal{N}$. The grammar of type expressions for $\mathcal{N}$ is given in [5] as:

$$\begin{aligned}\mathcal{F}^P(V) \; ::= \quad & \mathbb{0} \mid \mathbb{1} \mid \mathsf{Nat}^P\,\mathcal{F}^P(V)\,\mathcal{F}^P(V) \mid \mathcal{F}^P(V)+\mathcal{F}^P(V) \mid \mathcal{F}^P(V)\times\mathcal{F}^P(V) \\ & \mid V\,\overline{\mathcal{F}^P(V)} \mid P\,\overline{\mathcal{F}^P(V)} \;\mid \big(\mu\varphi^{\;k}.\lambda\alpha_1...\alpha_k.\mathcal{F}^{P,\alpha_1,\ldots,\alpha_k,\varphi}(V)\big)\,\overline{\mathcal{F}^P(V)}\end{aligned}$$

A *type expression* over P and V is any element of $\mathcal{F}^P(V)$, where P and V are disjoint sets of functorial variables and type constructor variables, respectively. We use the terms *type* and *type expression* interchangeably, and use the latter when we want to distinguish type expressions from Agda types, type variables, or typing judgments (see Section 2.4). The syntax $F\,\overline{X}$ indicates that F is either a type variable or a subexpression of the form $(\mu\varphi^{\;k}.\lambda\alpha_1...\alpha_k.\mathcal{F}^{P,\alpha_1,\ldots,\alpha_k,\varphi}(V))$, and the arity of F matches the length of the vector of type expressions $\overline{X}$. The notation ψ^k indicates that ψ is a variable of arity k, and the arity of a type expression $(\mu\varphi^{\;k}.\lambda\alpha_1...\alpha_k\mathcal{F}^{P,\alpha_1,\ldots,\alpha_k,\varphi}(V))$ is the arity of its *recursion variable* φ. As a matter of convention, we use letters from the beginning of the Greek alphabet (e.g., α, β, γ) for variables of arity 0 and letters from later in the alphabet (e.g., φ and ψ) for variables of arity greater than 0.

The two types $\mathbb{0}$ and $\mathbb{1}$ represent the empty type ($\bot$) and unit type ($\top$), respectively. The type $\mathsf{Nat}^{\overline{\alpha}}F\,G$ represents the type of natural transformations between types F and G. We call F and G the *domain* and *codomain*, respectively, of $\mathsf{Nat}^{\overline{\alpha}}F\,G$ and say that $\mathsf{Nat}^{\overline{\alpha}}F\,G$ is *natural in* the arity 0 variables in $\overline{\alpha}$. To a non-category-theorist, a natural transformation can be thought of as a uniformly defined polymorphic function between functorial types F and G that satisfies some commutativity conditions with the map operations for F and G. A natural transformation that is natural in zero variables is simply a function, so the Nat type $\mathsf{Nat}^{\emptyset}F\,G$ is the type of functions between F and G. The types $F+G$ and $F\times G$ represent the standard sum ($\uplus$) and product ($\times'$) types. The types of the form $V\,\overline{\mathcal{F}^P(V)}$ are applications of type constructor variables, and those of the form $P\,\overline{\mathcal{F}^P(V)}$ are applications of functorial variables. Note that a separate case for standalone variables is not needed, since this can be represented as a 0-ary variable applied to a vector of length zero. The μ-type $(\mu\,\varphi.\lambda\alpha\,F)\overline{G}$ represents the application to the types $\overline{G}$ of the least fixed point of its *body* F, with respect to φ and $\overline{\alpha}$. Note that φ and $\overline{\alpha}$ must be functorial variables and the variables in $\overline{\alpha}$ must have arity 0. The μ-type is analogous to Agda's `data` declaration because it allows us to define inductive data types such as List, PTree, etc. However, the type system of $\mathcal{N}$ is limited to nested types, so dependent data types such as Vec cannot be represented in $\mathcal{N}$.

We can express the grammar of type expressions as a data type in Agda, with a data constructor for each kind of expression:

```
data TypeExpr : Set where
  𝟘 : TypeExpr
  𝟙 : TypeExpr
  Nat^_[_,_] : ∀ {k : ℕ} → Vec (FVar 0) k → TypeExpr → TypeExpr → TypeExpr
  _+_ : TypeExpr → TypeExpr → TypeExpr
  _×_ : TypeExpr → TypeExpr → TypeExpr
  AppT_[_] : ∀ {k : ℕ} → TCVar k → Vec TypeExpr k → TypeExpr
  AppF_[_] : ∀ {k : ℕ} → FVar k → Vec TypeExpr k → TypeExpr
  μ_[λ_,_]_ : ∀ {k : ℕ} → FVar  k → Vec (FVar 0) k → TypeExpr → Vec TypeExpr k → TypeExpr
```

The data constructors for this type correspond directly to the rules of the grammar $\mathcal{F}^P(V)$. We use the type indices for Vec, FVar, and TCVar to enforce the aforementioned constraints on arities and vector lengths in applications and μ-types.

Let us consider some example types. The type PTree α is represented in the syntax of $\mathcal{N}$ as

$$(\mu\varphi.\lambda\beta.\beta + \varphi(\beta \times \beta))\alpha$$

This corresponds to the Agda definition of PTree in (2). The body of the μ-type is a sum with a summand for each constructor. The pleaf constructor just takes an argument of type A, which corresponds to the type variable β in this example. The pnode constructor takes an argument of type PTree (A ×′ A). To refer to the type PTree being defined in the syntax of $\mathcal{N}$, we use the recursion variable bound by μ, so that PTree (A ×′ A) corresponds to $\varphi(\beta \times \beta)$. The type PTree α would be represented as the following element of TypeExpr:

(3)

```
PTree-α : TypeExpr
PTree-α = μ φ [λ [ β ] , AppF β [ [] ] + AppF φ [ [ AppF β [ [] ] × AppF β [ [] ] ] ] ] [ AppF α [ [] ] ]
```

Here β and α are functorial variables of arity 0 and φ is a functorial variable of arity 1. There is a lot going on syntactically, so let's break PTree-α up into its components.

First note that in order to consider β as a type expression rather than a type variable, we must apply it to a vector of zero arguments, i.e., we must write it as AppF β[[]]. It is somewhat tedious to write AppF β[[]] every time we want to use β as a type *expression* as opposed to a type *variable*, so to avoid this, we introduce an auxiliary function VarExpr that turns a (functorial) type variable of arity 0 into a type expression:

```
VarExpr : FVar 0 → TypeExpr
VarExpr β = AppF β [ [] ]
```

Note that the empty vector is denoted by [], and a singleton (i.e., length 1) vector containing x can be written as [x] in Agda. The body of the μ-type is written as $\beta + \varphi(\beta \times \beta)$ in $\mathcal{N}$. This expression is represented in Agda as the following element of TypeExpr:

```
PTree-body : TypeExpr
PTree-body = VarExpr β + AppF φ [ [ VarExpr β × VarExpr β ] ]
```

PTree-body is a sum type whose first component is β and whose second component is φ applied to (the singleton vector containing) $\beta \times \beta$. The argument to PTree is a singleton vector containing the type expression α. The vector must have length 1 to match the arity of φ. Again, we must apply α to a vector of zero arguments for it to be an element of TypeExpr:

```
PTree-args : Vec TypeExpr 1
PTree-args = [ VarExpr α ]
```

Given the type expressions for the body and the vector of type expressions for the argument, we can give a more concise definition of PTree-α:

```
PTree-α : TypeExpr
PTree-α = μ φ [λ [ β ] , PTree-body ] PTree-args
```

This gives precisely the same definition as (3). Note that we have β appearing immediately after the λ rather than AppF β[[]] because the μ constructor

```
μ_[λ_,_]_ : ∀ {k : ℕ} → FVar k → Vec (FVar 0) k → TypeExpr → Vec TypeExpr k → TypeExpr
```

expects a vector of 0-ary type *variables* rather than type *expressions* in that position.

2.3 Type Contexts

A type is formed with respect to a *type context*. A type context indicates which variables are in scope, i.e., which variables can appear in the type. The calculus $\mathcal{N}$ includes two kinds of contexts: *functorial* contexts and *type constructor*, or *non-functorial*, contexts. A type constructor context contains type constructor variables, while a functorial context contains functorial variables.

In [5], a context is defined as a set of type variables, using the standard mathematical definition of sets. In other words, contexts in [5] contain at most one occurrence of each variable. It turns out that this uniqueness of variables in a context is not necessary to define the syntax of types, and it is somewhat simpler to formalize contexts as structures which allow repetition rather than sets (which do not).

We can represent both functorial and non-functorial contexts with a single Agda data type parameterized over a function V : ℕ → Set that takes a natural number and returns a type. This parameter is intended to be instantiated with TCVar or FVar.

```
infixl 20 _,,_
data TypeContext (V : ℕ → Set) : Set where
  ∅ : TypeContext V
  _,,_ : ∀ {k : ℕ} → TypeContext V → V k → TypeContext V
```

A type context is either empty or constructed from a new variable and another type context. New variables are added on the right, as in (Γ,,v), rather than on the left (which is standard for lists and vectors). Note that a type context can contain variables of different arities. This is one reason why we can't simply define type contexts as List (TCVar k) or Vec (TCVar k) n. Indeed, the type TCVar k only includes variables of a specific arity k, so a list of type List (TCVar k) could only contain variables of arity k.

Since the variable k in the _,,_ constructor of TypeContext does not appear in its return type, a TypeContext may contain variables of different arities, as desired. Given this generic definition of type contexts, we can define type constructor contexts and functorial contexts as instances of TypeContext.

```
TCCtx : Set
TCCtx = TypeContext TCVar

FunCtx : Set
FunCtx = TypeContext FVar
```

For convenience, we give names to the empty contexts of both varieties:

```
∅tc : TCCtx
∅tc = ∅

∅fv : FunCtx
∅fv = ∅
```

We will also need a function for adding a specific number of variables to a context. Since some types (e.g., type variable application) require a specific number of variables, this operation is needed to ensure that the context contains the correct number of variables. This function is parameterized over a function of type ℕ → Set just as TypeContext is, so it can be instantiated with TCVar or FVar as necessary:

```
_,++_ : ∀ {V : ℕ → Set} {k n : ℕ} → TypeContext V → Vec (V k) n → TypeContext V
Γ ,++ [] = Γ
Γ ,++ (α :: αs) = (Γ ,++ αs) ,, α
```

Note that this function _, + + _ appends a vector of variables *of a specific arity* k to a context. We could define a function that simply concatenates two contexts, regardless of their content. However, in all situations when we need to extend a context with multiple variables, the variables all have the same arity, so the _, + + _ operation will suffice.

To form some types, we need to know that a certain variable is present in the context. For this purpose, we define the proposition "context Γ contains variable v" with the following data type:

```
data _∋_ : ∀ {V : ℕ → Set} {k : ℕ} → TypeContext V → V k → Set where
  lookupZ : ∀ {V : ℕ → Set} {k : ℕ} {Γ : TypeContext V} {v : V k}
               → (Γ ,, v) ∋ v

  lookupDiffId : ∀ {V : ℕ → Set} {k : ℕ} {Γ : TypeContext V} {v v' : V k}
               → v ≢ v'
               → Γ ∋ v
               → (Γ ,, v') ∋ v

  lookupDiffArity : ∀ {V : ℕ → Set} {k j : ℕ} {Γ : TypeContext V} {v : V k} {v' : V j}
                  → k ≢ j
                  → Γ ∋ v
                  → (Γ ,, v') ∋ v
```

This data type is also referred to as *context lookup* or the context lookup relation. The constructor lookupZ gives a proof that, for any nonempty context, the rightmost element v appears in the context. The lookupDiffId and lookupDiffArity constructors take proofs that a variable v appears in a context Γ and prove that v appears in a larger context Γ,,v′ These constructors also require that v and v′ are different, either in name, in arity, or both. These proofs of inequality enforce the constraint that there is at most one proof that a variable is in a context. The implementation above ensures that although a variable may appear in a context multiple times, a proof that Γ ∋ v will always be the proof for the rightmost occurrence of v.

2.4 Type Formation Rules

Now that we have defined type variables, type expressions, contexts, and the context lookup relation, we can define the set of well-formed types as a data type in Agda. A typing judgment $\Gamma; \Phi \vdash F$ asserts that the type F is well-formed with respect to the non-functorial context Γ and the functorial context Φ. The set of well-formed types is constructed inductively from *type formation* rules. Type formation rules consist of 0 or more preconditions and a conclusion (a typing judgment) with a horizontal line separating the preconditions and the conclusion.

$$\frac{}{\Gamma;\Phi\vdash \mathbb{0}} \qquad \frac{}{\Gamma;\Phi\vdash \mathbb{1}}$$

$$\frac{\Gamma;\overline{\alpha^0}\vdash F \qquad \Gamma;\overline{\alpha^0}\vdash G}{\Gamma;\emptyset\vdash \mathsf{Nat}^{\overline{\alpha^0}} F\,G} \qquad \frac{\varphi^k\in\Gamma\cup\Phi \qquad \overline{\Gamma;\Phi\vdash F}}{\Gamma;\Phi\vdash \varphi^k\overline{F}}$$

$$\frac{\Gamma;\Phi\vdash F \qquad \Gamma;\Phi\vdash G}{\Gamma;\Phi\vdash F+G} \qquad \frac{\Gamma;\Phi\vdash F \qquad \Gamma;\Phi\vdash G}{\Gamma;\Phi\vdash F\times G}$$

$$\frac{\Gamma;\overline{\alpha^0},\varphi^k\vdash F \qquad \overline{\Gamma;\Phi\vdash G}}{\Gamma;\Phi\vdash (\mu\varphi^k.\lambda\overline{\alpha^0}.\,F)\,\overline{G}}$$

Figure 2.1: Type formation rules

In the calculus $\mathcal{N}$, there is one formation rule for each kind of type expression. These formation rules are displayed in Figure 2.1. The overbar in $\overline{\Gamma;\Phi\vdash F}$ (e.g., in the μ-type formation rule) indicates a vector of typing judgments.

The formation rules for $\mathbb{0}$ and $\mathbb{1}$ have no preconditions, so these types can be formed in any context. The type $\mathsf{Nat}^{\overline{\alpha^0}} F\,G$ can be formed in contexts Γ and $\emptyset$ if F and G only have $\overline{\alpha^0}$ in their functorial contexts, i.e., $\Gamma;\overline{\alpha^0}\vdash F$ and $\Gamma;\overline{\alpha^0}\vdash G$. The conclusion for $\mathsf{Nat}^{\overline{\alpha^0}} F\,G$ has an empty functorial context because the Nat type *binds* all of the variables in $\overline{\alpha^0}$, meaning those variables are no longer in scope outside of the Nat type. While it is technically possible to have a type such as $\mathsf{Nat}^{\overline{\alpha^0}}\,F\,(\mathsf{Nat}^{\overline{\alpha^0}} F\,G)$, the $\overline{\alpha^0}$ variables bound by the outer Nat type are necessarily different variables from those bound by the inner Nat type, despite having the same names. However, we are not really interested in such nested Nat types, and we avoid such naming conflicts by always using distinct names for distinct variables. The type $\varphi^k\overline{F}$ of variable application can be formed in contexts Γ and Φ if φ^k appears in one of the contexts and, for each F_i in $\overline{F}$, we have $\Gamma;\Phi\vdash F_i$. Given two types F and G with $\Gamma;\Phi\vdash F$ and $\Gamma;\Phi\vdash G$, we can form both the sum type $F+G$ and the product type $F\times G$ in contexts Γ and Φ. The type $(\mu\,\varphi.\lambda\alpha\,F)\overline{G}$ can be formed if $\Gamma;\Phi\vdash G_i$ for each G_i in $\overline{G}$ and $\Gamma;\overline{\alpha^0},\varphi^k\vdash F$. The latter entails that the only functorial variables allowed to appear in F are φ^k and the variables in $\overline{\alpha^0}$. This restriction on F is necessary to ensure that the Identity Extension Lemma holds for μ-types.

We can formalize the set of well-formed types as a data type, giving one constructor for each of the formation rules in Figure 2.1:

```
data _≀_⊢_ : TCCtx → FunCtx → TypeExpr → Set where
  𝟘-I : ∀ {Γ : TCCtx} {Φ : FunCtx} → Γ ≀ Φ ⊢ 𝟘
  𝟙-I : ∀ {Γ : TCCtx} {Φ : FunCtx} → Γ ≀ Φ ⊢ 𝟙
```

```
Nat-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {k : ℕ} {αs : Vec (FVar 0) k}
          {F G : TypeExpr}
        → (⊢F : Γ ≀ ( ∅ ,++ αs ) ⊢ F)
        → (⊢G : Γ ≀ ( ∅ ,++ αs ) ⊢ G)
        → Γ ≀ Φ ⊢ Nat^ αs [ F , G ]

AppT-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {k : ℕ} {φ : TCVar k}
           → (Γ∋φ : Γ ∋ φ)
           → (Fs : Vec TypeExpr k)
           → (⊢Fs : foreach (λ F → Γ ≀ Φ ⊢ F) Fs)
           → Γ ≀ Φ ⊢ AppT φ [ Fs ]

AppF-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {k : ℕ} {φ : FVar k}
           → (Φ∋φ : Φ ∋ φ)
           → (Fs : Vec TypeExpr k)
           → (⊢Fs : foreach (λ F → Γ ≀ Φ ⊢ F) Fs)
           → Γ ≀ Φ ⊢ AppF φ [ Fs ]

+-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {F G : TypeExpr}
      → (⊢F : Γ ≀ Φ ⊢ F)
      → (⊢G : Γ ≀ Φ ⊢ G)
      → Γ ≀ Φ ⊢ F + G

×-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {F G : TypeExpr}
      → (⊢F : Γ ≀ Φ ⊢ F)
      → (⊢G : Γ ≀ Φ ⊢ G)
      → Γ ≀ Φ ⊢ F × G

μ-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
        {k : ℕ} {φ : FVar k}
        {αs : Vec (FVar 0) k} {F : TypeExpr}
        → (⊢F : Γ ≀ (∅ ,++ αs) ,, φ ⊢ F)
        → (Gs : Vec TypeExpr k)
        → (⊢Gs : foreach (λ G → Γ ≀ Φ ⊢ G) Gs)
        → Γ ≀ Φ ⊢ μ φ [λ αs , F ] Gs
```

The typing judgment data type _ ≀ _ ⊢ _ is indexed by a non-functorial context, a functorial context, and a type expression. The elements of this data type can be thought of as derivation trees that show the steps taken to prove that the $\mathcal{N}$ type represented by the indexing type expression is well-formed with respect to the two indexing contexts.

The 𝟘-I and 𝟙-I data constructors express that 𝟘 and 𝟙 can be formed in any context. The Nat-I data constructor expresses that F and G must be formed with a functorial context of αs, as expected[1]. However, unlike the conclusion of the Nat formation rule in Figure 2.1, the return type of Nat-I is $\Gamma \wr \Phi \vdash$ Nat^αs[F, G], with Φ as the functorial context rather than $\emptyset$. Initially we used $\emptyset$ for the functorial context here, but this makes it impossible to pattern match on a term of type $\Gamma \wr \Phi \vdash$ F with the Nat-I constructor unless $\Phi = \emptyset$. Also, despite the fact that Nat types are always *formed* with an empty context, we may want to introduce functorial variables to the context of a Nat type by weakening (see Section 2.5). For these reasons, the data constructor Nat-I allows for a Nat type to be formed in an arbitrary functorial context Φ.

The formation rule for type variable application is split into two data constructors: AppT-I is for application of type constructor variables and AppF-I is for application of functorial variables. The AppT-I and AppF-I data constructors differ only in the type of the variable φ. The AppT-I data constructor expresses that, in order to form the type of variable application, we must choose a non-functorial context Γ, a functorial context Φ, a natural number k, and a type constructor variable φ of arity k. We must also give a proof that φ appears in Γ, a vector of type expressions of length k, and a proof that each of the type expressions in this vector is well-formed with respect to Γ and Φ. The assertion that each type expression in the vector is well-formed is formalized in terms of the foreach function:

```
foreach : ∀ {a b : Level} {A : Set a} {n : ℕ} → (P : A → Set b)
        → Vec A n → Set b
foreach P [] = big⊤
foreach P (x :: xs) = P x ×' foreach P xs
```

The foreach function takes a predicate P : A → Set b and a vector xs : Vec A n and returns a proposition

[1] Since overbars cannot be displayed in Agda code, we append an 's' to indicate that αs is a vector of variables. This convention is also used for other vectors throughout the implementation.

(an element of Set b) that asserts P holds for every element of xs. The base case for an empty vector returns a trivially true proposition (big⊤ is a universe-polymorphic version[2] of the unit type ⊤, which can be thought of as a trivially true proposition). The inductive case returns a proposition asserting that P holds for the first element x and P also holds for the tail of the list xs. In other words, to construct a proof of foreach P xs, we must construct a proof of P $\mathsf{x_i}$ for each $\mathsf{x_i}$ in the vector xs. Getting back to the definition of the typing judgment, a term of type foreach $(\lambda \mathsf{G} \to \Gamma \wr \Phi \vdash \mathsf{G})$ Gs consists of a proof of $\Gamma \wr \Phi \vdash \mathsf{G_i}$ for each $\mathsf{G_i}$ in Gs, corresponding to $\overline{\Gamma; \Phi \vdash \mathsf{G}}$ in Figure 2.1.

The data constructors for sum and product types express that $\Gamma \wr \Phi \vdash \mathsf{F}$ and $\Gamma \wr \Phi \vdash \mathsf{G}$ are the only preconditions required to form sum and product types.

The μ-I data constructor expresses that the body F must be formed with a functorial context containing only αs and φ. The μ-I data constructor also uses the foreach function to require that we have a proof of $\Gamma \wr \Phi \vdash \mathsf{G_i}$ for each $\mathsf{G_i}$ in Gs.

Let us consider some example typing judgment derivations. Earlier in Section 2.2 we showed how to construct the term corresponding to the type expression

$$(\mu\varphi.\lambda\beta.\beta + \varphi(\beta \times \beta))\alpha$$

in Agda. We now show how to construct the term corresponding to the *typing judgment*

$$\emptyset; \alpha \vdash (\mu\varphi.\lambda\beta.\beta + \varphi(\beta \times \beta))\alpha$$

in Agda:

```
⊢PTree-α : ∅tc ≀ ∅fv ,, α ⊢ PTree-α
⊢PTree-α = μ-I ⊢body PTree-args (⊢args , bigtt)
  where 0≢1 : 0 ≢ 1
        0≢1 = λ ()

        β,φ∋β : (∅fv ,, β ,, φ) ∋ β
        β,φ∋β = lookupDiffArity 0≢1 lookupZ
```

[2] The unit type ⊤ from the standard library is defined as an element of Set. To keep the type of foreach as general as possible, we use big⊤ : $\forall\{\ell\} \to$ Set ℓ, which belongs to Set ℓ for any universe level ℓ we choose. The single constructor for big⊤ is bigtt : big⊤. We need this generality in some cases to use foreach on a predicate P : A $\to$ Set b valued in a higher universe b.

```
    ⊢β : ∅tc ≀ ∅ ,, β ,, φ ⊢ VarExpr β
    ⊢β = AppF-I β,φ∋β [] bigtt

    β×β : TypeExpr
    β×β = VarExpr β × VarExpr β

    ⊢β×β : ∅tc ≀ ∅ ,, β ,, φ ⊢ β×β
    ⊢β×β = ×-I ⊢β ⊢β

    ⊢φβ×β : ∅tc ≀ ∅ ,, β ,, φ ⊢ AppF φ [ [ β×β ] ]
    ⊢φβ×β = AppF-I lookupZ [ β×β ] (⊢β×β , bigtt)

    ⊢body : ∅tc ≀ ∅ ,, β ,, φ ⊢ PTree-body
    ⊢body = +-I ⊢β ⊢φβ×β

    ⊢args : ∅tc ≀ ∅fv ,, α ⊢ VarExpr α
    ⊢args = AppF-I lookupZ [] bigtt
```

This proof is constructed using a where block in Agda, which allows us to introduce local definitions that are in scope for the clause containing the where block. The type of $\vdash$PTree-α indicates that the type expression PTree-α is formed with an empty non-functorial context and with α in the functorial context. At top-level, PTree-α is a μ-type, so $\vdash$PTree-α is constructed using μ-I. To use μ-I, we must give a proof that the body of the μ-type is well-formed and we must also give a proof that the argument type is well-formed. These proofs are called $\vdash$body and $\vdash$args, respectively. The term for $\vdash$body is constructed using $+$-I and proofs that β and $\varphi(\beta \times \beta)$ are well-formed. The term for $\vdash$args is a simple application of AppF-I, where lookupZ : $\emptyset$fv,,$\alpha \ni \alpha$ is the proof that α appears in the context $\emptyset$fv,,α. The only other context lookup proof that is needed is a proof that β appears in $\emptyset$fv,,β,,φ, which is proved using lookupDiffArity, a proof that zero is not equal to one, and a proof that β appears in the context $\emptyset$fv,,β. The type A $\not\equiv$ B is a synonym for A $\equiv$ B $\to \bot$, so it asserts that two types are not equal according to the propositional equality type from (1). The proof that 0 is not equal to 1 is trivial, and is proven in Agda using the absurd pattern $\lambda()$. The absurd pattern can be used here because Agda can deduce automatically that there are no terms of type 0 $\equiv$ 1.

2.5 Weakening

In addition to formalizing the set of well-formed types, we would like to prove some properties about it. The first property we would like to prove is that the type formation rules respect weakening. Given a typing judgment $\Gamma; \Phi \vdash \mathsf{F}$, we can *weaken* the typing judgment by adding extra variables that do not appear in F to Γ or Φ. To prove that our type formation rules respect weakening, we must show that for any Γ, Φ, F, and φ, if $\Gamma; \Phi \vdash \mathsf{F}$ is well-formed, then so is $\Gamma; \Phi, \varphi \vdash \mathsf{F}$ (or $\Gamma, \varphi; \Phi \vdash \mathsf{F}$, for non-functorial variables). Since we have two contexts, there are two notions of weakening for the calculus $\mathcal{N}$. The proofs that our type formation rules respect these two notions of weakening are very similar, so we will only describe the proof for functorial variables.

To show that the type formation rules respect weakening of functorial variables, we must define a function of the following type:

```
weakenFunCtx : ∀ {k : ℕ} { Γ : TCCtx } {Φ : FunCtx} {F : TypeExpr} (φ : FVar k)
                    → Γ ≀ Φ ⊢ F
                    → Γ ≀ Φ ,, φ ⊢ F
```

This function is defined mutually[3] with an analogous function for vectors of type expressions:

```
foreach-preserves-weakening-FV : ∀ {k n : ℕ} {Γ : TCCtx } {Φ : FunCtx} {φ : FVar k}
                                      → (Gs : Vec TypeExpr n)
                                      → foreach (λ G → Γ ≀ Φ ⊢ G) Gs
                                      → foreach (λ G → Γ ≀ Φ ,, φ ⊢ G) Gs
```

The function weakenTCCtx is defined by pattern matching on the argument of type $\Gamma \wr \Phi \vdash \mathsf{F}$. For the cases of 𝟘-I, 𝟙-I, and Nat-I, the proofs are trivial because these data constructors can be used to form a type in an arbitrary functorial context. The cases for +-I , ×-I , AppT-I, and μ-I are proved by straightforward applications of induction, i.e., by recursively calling weakenFunCtx. The interesting case is for AppF-I, which we show step by step:

```
weakenFunCtx (φ ^F k) (AppF-I {φ = ψ ^F j} Φ∋ψ Gs ⊢Gs) = {!!}
-- Goal:  Γ ≀ Φ ,, (φ ^F k) ⊢ AppF (ψ ^F j) [ Gs ]
```

[3] We could just define weakenFunCtx and then use the map function for Vec defined in the standard library (that applies a function to every element in a vector) to apply weakenFunCtx to vectors, but this definition using the map function for Vec is not accepted by Agda's termination checker. To satisfy the termination checker, we define foreach-preserves-weakening-FV mutually with weakenFunCtx.

First, we pattern match on the variable argument to get $(\varphi \mathbin{\hat{}}\mathsf{F}\ \mathsf{k}) : \mathsf{FVar}\,\mathsf{k}$, and we also pattern match on the (implicit) variable argument to AppF-I, using the syntax $\{\varphi = \psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j}\}$. Although we normally use the letters φ and ψ for terms of type $\mathsf{FVar}\,\mathsf{k}$ or $\mathsf{TCVar}\,\mathsf{k}$, i.e., to represent a variable tagged with its arity, we abuse notation here and use the letters φ and ψ for the variable names themselves. So in this case, we have $\varphi, \psi : \mathsf{String}$. To construct a term of the goal type, we need to use AppF-I, which requires a context lookup of type $(\Phi ,, \varphi \mathbin{\hat{}}\mathsf{F}\ \mathsf{k}) \ni \psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j}$ asserting that $\psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j}$ appears in the weakened context. We already have a proof $\Phi{\ni}\psi : (\Phi \ni \psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j})$ In order to prove that $\psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j}$ appears in the weakened context, we need to know whether $\varphi \mathbin{\hat{}}\mathsf{F}\ \mathsf{k}$ and $\psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j}$ are the same variable. They may differ in their names, their arities, or both.

In order to compare $\varphi \mathbin{\hat{}}\mathsf{F}\ \mathsf{k}$ and $\psi \mathbin{\hat{}}\mathsf{F}\ \mathsf{j}$ for equality, we need a function that computes whether two natural numbers are equal, and we also need a function for comparing strings for equality. Propositional equality $(\_ \equiv \_)$ for natural numbers and strings in Agda is decidable. Decidable propositions are formalized in Agda using the Dec type from the standard library:

```
record Dec {p} (P : Set p) : Set p where
  constructor _because_
  field
    does : Bool
    proof : Reflects P does
```

A term of type Dec P consists of a boolean value, does, and a term of type Reflects P does, which corresponds to a proof of P if does is true or a proof of the negation of P if does is false:

```
data Reflects {p} (P : Set p) : Bool → Set p where
  ofʸ : ( p : P) → Reflects P true
  ofⁿ : (¬p : ¬ P) → Reflects P false
```

The standard library also provides *pattern synonyms* to make pattern matching on terms of type Dec P more concise:

```
pattern yes p = true because ofʸ p
pattern no ¬p = false because ofⁿ ¬p
```

These pattern synonym declarations mean that we can write yes p instead of true because of^y p when pattern matching on terms of type Dec P, and similarly for no p. If we define functions that implement decidable equality for natural numbers and strings as

```
eqNat : ∀ (x y : ℕ) → Dec (x ≡ y)
_≟_ : ∀ (s t : String) → Dec (s ≡ t)
```

then we can perform the case analysis needed for the AppF-I case of weakenFunCtx. The function eqNat is defined by induction on its natural number arguments, and _ ≟ _ is provided in the standard library. To perform the case analysis needed for weakenFunCtx, we use a with clause, which, borrowing a phrase from the Agda documentation [7], allows us to pattern match on the result of an intermediate computation. In this case, we want to compute whether j and k are equal and whether ψ and φ are equal, so we pattern match on the results of eqNat j k and $\psi \overset{?}{=} \varphi$:

```
weakenFunCtx (φ ^F k) (AppF-I {φ = ψ ^F j} Φ∋ψ Gs ⊢Gs) with eqNat j k | ψ ≟ φ
... | yes refl | yes refl = {!!} -- Goal:  Γ ≀ Φ ,, (φ ^F k) ⊢ AppF (φ ^F k) [ Gs ]
... | yes refl | no ψ≢φ  = {!!} -- Goal:  Γ ≀ Φ ,, (φ ^F k) ⊢ AppF (ψ ^F k) [ Gs ]
... | no j≢k | _         = {!!} -- Goal:  Γ ≀ Φ ,, (φ ^F k) ⊢ AppF (ψ ^F j) [ Gs ]
```

There are four possible cases, but the (no/yes) and (no/no) cases can be proven with the same definition, which brings us down to the three cases in weakenFunCtx. Either the variables are identical (names and arity match), the arities match and the names do not, or the arities do not match.

In the first case, we can use lookupZ : Φ,,φ ^F k ∋ φ ^F k for the context lookup, since pattern matching on refl twice tells Agda that j can be replaced with k and ψ can be replaced with φ in the goal.

In the second case, the arities match, so both variables have arity k in the goal, and we also have a proof that ψ is not equal to φ. In this case we can use lookupDiffId and Φ∋ψ to prove that Φ,,φ ^F k ∋ ψ ^F k. Recall that lookupDiffId needs an argument of type ψ ^F k ≢ φ ^F k indicating that the variables (including arity) are not equal, and note that we have a proof ψ≢φ that the variable *names* are not equal. To prove that the *variables* (including arity) are not equal, we can use an auxiliary function

```
≢-FVar : ∀ {k : ℕ} → (v v' : Id) → v ≢ v' → (v ^F k) ≢ (v' ^F k)
```

to get (≢-FVar ψ φ ψ≢φ) : (ψ ^F k) ≢ (φ ^F k). The function ≢-FVar says that if two variable names are not equal, then they are still not equal after being tagged with their arities, and this function is defined by straightforward reasoning about proofs of equality.

In the last case above, the arities j and k do not match. We need a proof that ψ ^F j appears in the context Φ,,φ ^F k, which we can prove using lookupDiffArity with arguments j≢k and Φ∋ψ. We can now complete the AppF-I case for weakenFunCtx,

```
weakenFunCtx (φ ^F k) (AppF-I {φ = ψ ^F j} Φ∋ψ Gs ⊢Gs) with eqNat k j | φ ≟ ψ
... | yes refl | yes refl = AppF-I lookupZ Gs (foreach-preserves-weakening-FV Gs ⊢Gs)
... | yes refl | no φ≢ψ = AppF-I (lookupDiffId (≢-FVar ψ φ (≢-sym φ≢ψ)) Φ∋ψ) Gs
                                   (foreach-preserves-weakening-FV Gs ⊢Gs)            (4)
... | no k≢j | _          = AppF-I (lookupDiffArity (≢-sym k≢j) Φ∋ψ) Gs
                                   (foreach-preserves-weakening-FV Gs ⊢Gs)
```

using foreach-preserves-weakening-FV in each case to prove that the type arguments Gs are well-formed in the weakened context, i.e., foreach (λG → Γ ≀ Φ,,(φ ^F k) ⊢ G) Gs. The proof of foreach-preserves-weakening-FV is given by induction on the length of the vector, calling weakenFunCtx to prove that each individual type in the vector is well-formed in the weakened context. The proof that the type formation rules respect weakening of non-functorial variables is nearly identical to the proof of weakenFunCtx except that it is the AppT-I that requires a case analysis of variable equality.

2.6 Substitution

The calculus $\mathcal{N}$ also supports substitution of functorial variables. Given a type expression F that contains a 0-ary functorial variable α, we can replace α with another type expression H. This substitution is denoted $F[\alpha := H]$. We can apply this substitution to a vector of type expressions, denoted by $\overline{F[\alpha := H]}$, and we can also substitute a vector of variables/type expressions in a single type expression, denoted by $F[\overline{\alpha := H}]$. We refer to the substitution $F[\alpha := H]$ as *first-order* substitution, to contrast it with *second-order* substitution of variables with arity greater than 0. Second-order substitution is discussed in more detail at the end of this section.

We can define first-order substitution of type expressions as a function in Agda:

```
_[_:=_] : TypeExpr → FVar 0 → TypeExpr → TypeExpr
𝟘 [ α := H ] = 𝟘
𝟙 [ α := H ] = 𝟙
Nat^ βs [ F , G ] [ α := H ] = Nat^ βs [ F , G ]
(F + G) [ α := H ] = (F [ α := H ]) + (G [ α := H ])
(F × G) [ α := H ] = (F [ α := H ]) × (G [ α := H ])                    (5)
AppT φ [ Gs ] [ α := H ] = AppT φ [ fo-substVec Gs α H ]
AppF φ ^F k [ Gs ] [ (α ^F j) := H ] with eqNat k j | φ ≟ α
... | yes refl | yes refl = H
... | _ | _ = AppF (φ ^F k) [ fo-substVec Gs (α ^F j) H ]
(μ φ [λ βs , G ] Ks) [ α := H ] = μ φ [λ βs , G ] (fo-substVec Ks α H)
```

The cases for 𝟘 and 𝟙 are trivial because these types contain no variables. For the Nat case, since the only functorial variables that should appear in F and G are the βs, and since these βs are not in the context for the Nat type (i.e., the βs are bound by the Nat type), no substitution is applied. For the sum, product, AppT-I, and μ-I cases, the substitution is applied recursively. For the AppF-I case, we perform case analysis to check whether the variable being applied (φ ^F k) is equal to the variable being substituted (α ^F j). If the variables are identical, then φ ^F k, which necessarily has arity 0 (and thus no arguments), is replaced with the type H. Otherwise, the substitution is just applied to the argument types Gs. For the μ-I case, the substitution is not applied to the body G because the only functorial variables in the body are bound. The first-order substitution function in (5) is defined mutually with an analogous function for applying substitutions to vectors of type expressions:

```
fo-substVec : ∀ {n : ℕ} → Vec TypeExpr n → FVar 0 → TypeExpr → Vec TypeExpr n
```

The function fo-substVec simply applies the substitution $F_i[\alpha := H]$ to each type expression F_i in a vector of type expressions.

Given these definitions, we would like to prove that first-order substitution preserves well-formed types. In other words, if a type is well-formed, then it should still be well-formed after an appropriately typed substitution. We can formalize this in Agda as a function that takes two typing judgments and returns a typing judgment for a type expression involving a substitution:

```
fo-subst-preserves-typing : ∀ {Γ : TCCtx} {Φ : FunCtx} {α : FVar 0} {F H : TypeExpr}
                            → Γ ≀ (Φ ,, α) ⊢ F
                            → Γ ≀ Φ ⊢ H
                            → Γ ≀ Φ ⊢ F [ α := H ]
```

This function is defined mutually with an analogous function for vectors of type expressions,

```
foreach-preserves-fo-substVec : ∀ {k : ℕ} {Γ : TCCtx} {Φ : FunCtx} {α : FVar 0} {H : TypeExpr}
  → (Gs : Vec TypeExpr k)
  → Γ ≀ Φ ⊢ H
  → foreach (λ G → Γ ≀ Φ ,, α ⊢ G) Gs
  → foreach (λ G → Γ ≀ Φ ⊢ G) (fo-substVec Gs α H)
```

which is defined by induction on the length of the vector Gs, calling fo-subst-preserves-typing on each type expression in Gs.

Since all cases in (5) except AppF-I just recursively apply the substitution, the corresponding cases for fo-subst-preserves-typing can be proven using straightforward induction. Just as in (5), the interesting case is for the AppF-I constructor:

```
fo-subst-preserves-typing {α = α ^F j} (AppF-I {φ = φ ^F k} Φ,α∋φ Gs ⊢Gs) ⊢H = {!!}
-- Goal:  Γ ≀ Φ ⊢ ((AppF (φ ^F k) [ Gs ]) [ α ^F j := H ] | eqNat k j | φ ≟ α)
```

In this case, we must prove that the substitution (AppF (φ ˆF k) [Gs]) [α ˆF j := H] is well-formed with respect to contexts Γ and Φ. But notice that the goal type has some vertical bars, indicating the goal type involves a with clause. In order to make goal to reduce further, we must also use a with clause here to match on the results of eqNat k j and $\varphi \overset{?}{=} \alpha$.

```
fo-subst-preserves-typing {α = α ^F j} (AppF-I {φ = φ ^F k} Φ,α∋φ Gs ⊢Gs) ⊢H with eqNat k j | φ ≟ α
... | yes refl | yes refl = {!!} -- Goal:  Γ ≀ Φ ⊢ H
... | yes refl | no φ≢α = {!!} -- Goal:  Γ ≀ Φ ⊢ AppF (φ ^F j) [ fo-substVec Gs (α ^F j) H ]
... | no k≢j | _         = {!!} -- Goal:  Γ ≀ Φ ⊢ AppF (φ ^F k) [ fo-substVec Gs (α ^F j) H ]
```

In the first case, the goal reduces to Γ ≀ Φ ⊢ H, and we can simply use ⊢H to fill this hole. In the second and third cases, φ ˆF k is not equal to α ˆF j, so the substitution is only applied to the arguments Gs. These two cases are proven by straightforward induction on the vector of arguments Gs, similar to the induction used in (4). The only notable difference between these proofs and the proofs in (4) is that, in the latter proofs, we used lookupDiffId and lookupDiffArity to take a proof of $\Phi \ni \mathsf{v}$ that v appears in Φ and produce a proof of $\Phi,,\mathsf{v}' \ni \mathsf{v}$ that v appears in a larger context $\Phi,,\mathsf{v}'$. But to prove the cases above, we need to take a proof of $\Phi,,(\alpha\ \hat{}\,\mathsf{F}\ \mathsf{j}) \ni \varphi\ \hat{}\,\mathsf{F}\ \mathsf{k}$ and produce a proof of $\Phi \ni \varphi\ \hat{}\,\mathsf{F}\ \mathsf{k}$ for a *smaller* context. To do this, we must know that $\varphi\ \hat{}\,\mathsf{F}\ \mathsf{k}$ and $\alpha\ \hat{}\,\mathsf{F}\ \mathsf{j}$ differ in name or in arity, which we have in the code above as $\varphi \not\equiv \alpha$ and $\mathsf{k} \not\equiv \mathsf{j}$.

We can also define the substitution of multiple 0-ary variables as a function in Agda

```
_[_:=_]Vec : ∀ {k : ℕ} → TypeExpr → (Vec (FVar 0) k) → (Vec TypeExpr k) → TypeExpr
```

by straightforward induction on the vector of variables. Given this definition, we can prove by induction on the vector of type expressions Gs and the vector of variables αs that it preserves well-formed types. That is:

```
[:=]Vec-preserves-typing : ∀ {k : ℕ} {Γ : TCCtx} {Φ : FunCtx} (αs : Vec (FVar 0) k) {H : TypeExpr}
                           → (Gs : Vec TypeExpr k)
                           → Γ ≀ (Φ ,++ αs) ⊢ H
                           → foreach (λ G → Γ ≀ Φ ⊢ G) Gs
                           → Γ ≀ Φ ⊢ H [ αs := Gs ]Vec
```

$$
\begin{array}{lcl}
\mathbb{0}[\varphi :=_{\overline{\alpha}} H] & = & \mathbb{0} \\
\mathbb{1}[\varphi :=_{\overline{\alpha}} H] & = & \mathbb{1} \\
(\mathsf{Nat}^{\overline{\beta}} F\, G)[\varphi :=_{\overline{\alpha}} H] & = & \mathsf{Nat}^{\overline{\beta}} F\, G \\
(\psi \overline{F})[\varphi :=_{\overline{\alpha}} H] & = & \begin{cases} \psi\, \overline{F[\varphi :=_{\overline{\alpha}} H]} & \text{if } \psi \neq \varphi \\ H[\alpha := \overline{F[\varphi :=_{\overline{\alpha}} H]}] & \text{if } \psi = \varphi \end{cases} \\
(F + G)[\varphi :=_{\overline{\alpha}} H] & = & F[\varphi :=_{\overline{\alpha}} H] + G[\varphi :=_{\overline{\alpha}} H] \\
(F \times G)[\varphi :=_{\overline{\alpha}} H] & = & F[\varphi :=_{\overline{\alpha}} H] \times G[\varphi :=_{\overline{\alpha}} H] \\
((\mu\psi.\lambda\overline{\beta}.F)\overline{G})[\varphi :=_{\overline{\alpha}} H] & = & (\mu\psi.\lambda\overline{\beta}.\, F)\, \overline{G[\varphi :=_{\overline{\alpha}} H]}
\end{array}
$$

Figure 2.2: Second-order substitution of functorial variables.

2.6.1 Second-order substitution

It was mentioned at the beginning of Section 2.6 that we also have a notion of substitution for variables of arity greater than 0. Given our definition of first-order substitution, we can think of 0-ary variables as holes that can be filled with a single type. Now consider a type expression containing the variable φ of arity k (which is always applied to k arguments) and a type H containing a vector $\overline{\alpha}$ containing

k 0-ary variables. We can replace φ with H, using the arguments of φ in place of the variables in $\overline{\alpha}$. We write this as $F[\varphi :=_{\overline{\alpha}} H]$ and call this type expression the substitution of φ by H *along* the variables in $\overline{\alpha}$. For example, given the type expressions $\varphi\mathbb{1}$ and $\alpha + \alpha$, the second-order substitution $(\varphi\mathbb{1})[\varphi :=_{\alpha} (\alpha + \alpha)]$ yields $\mathbb{1} + \mathbb{1}$.

Like first-order substitution, second-order substitution is defined by induction on the structure of the type expression. Before we get into the Agda definition of second-order substitution, let us consider the definition of second-order substitution given in [5], shown in Figure 2.2.

As with first-order substitution, the cases for $\mathbb{0}$, $\mathbb{1}$, and Nat are trivial, and the cases for sums, products, and μ-types just recursively apply the substitution to subexpressions. In the variable application case, we must check whether the variable ψ being applied is equal to the variable φ being substituted. This is only possible if ψ is a functorial variable. If φ and ψ are equal in name and arity, then ψ is replaced by H, and the variables in $\overline{\alpha}$ are replaced by the recursively substituted arguments of ψ, i.e., the vector of type expressions $\overline{F[\varphi :=_{\overline{\alpha}} H]}$. If φ and ψ are not equal, then the second-order substitution is recursively applied to the arguments of ψ. When the arity of φ is 0, second-order substitution coincides with first-order substitution.

This definition of second-order substitution is implemented in Agda as:

```
_[_:=[_]_] : ∀ {k : ℕ} → TypeExpr → (FVar k) → Vec (FVar 0) k → TypeExpr → TypeExpr

𝟘 [ φ :=[ αs ] F ] = 𝟘
𝟙 [ φ :=[ αs ] F ] = 𝟙
Nat^ βs [ G , K ] [ φ :=[ αs ] F ] = Nat^ βs [ G , K ]
(G + K) [ φ :=[ αs ] F ] = (G [ φ :=[ αs ] F ]) + (K [ φ :=[ αs ] F ])
(G × K) [ φ :=[ αs ] F ] = (G [ φ :=[ αs ] F ]) × (K [ φ :=[ αs ] F ])
AppT (ψ ^T j) [ Gs ] [ φ :=[ αs ] F ] = AppT (ψ ^T j) [ so-substVec Gs φ αs F ]                  (6)

AppF ψ ^F j [ Gs ] [ φ ^F k :=[ αs ] F ] with ψ ≟ φ | eqNat k j
... | yes refl | yes refl = F [ αs := (so-substVec Gs (φ ^F k) αs F) ]Vec
... | yes refl | no k≢j = AppF (ψ ^F j) [ so-substVec Gs (φ ^F k) αs F ]
... | no ψ≢φ | _        = AppF (ψ ^F j) [ so-substVec Gs (φ ^F k) αs F ]

(μ ψ [λ βs , G ] Ks ) [ φ :=[ αs ] F ] = μ ψ [λ βs , G ] (so-substVec Ks φ αs F)
```

As with first-order substitution, the second-order substitution function is defined mutually with a function that applies the second-order substitution to a vector of type expressions:

```
so-substVec : ∀ {n k : ℕ} → Vec TypeExpr n → FVar k → Vec (FVar 0) k → TypeExpr → Vec TypeExpr n
```

The function so-substVec takes a vector of type expressions of length n, a functorial variable of arity k, a vector of 0-ary functorial variables of length k, and a type expression, and applies a second-order substitution to every type expression in the vector of length n. The "so" in so-substVec stands for "second-order."

Given these definitions for second-order substitution, we can prove that second-order substitution preserves well-formed types:

```
so-subst-preserves-typing : ∀ {k : ℕ} {Γ : TCCtx} {Φ : FunCtx} {φ : FVar k} {αs : Vec (FVar 0) k}
                          → (F H : TypeExpr)
                          → Γ ≀ (Φ ,, φ) ⊢ F
                          → Γ ≀ (Φ ,++ αs) ⊢ H
                          → Γ ≀ Φ ⊢ F [ φ :=[ αs ] H ]
```

This function takes typing judgments for F and H and produces a typing judgment for the substitution F[φ := [αs] H]. It is defined by induction on the structure of the typing judgment for F and is defined mutually with an analogous function for vectors of type expressions:

```
so-substVec-preserves-typing : ∀ {n k : ℕ} {Γ : TCCtx} {Φ : FunCtx} {φ : FVar k}
  {αs : Vec (FVar 0) k} {H : TypeExpr}
  → (Gs : Vec TypeExpr n)
  → Γ ≀ (Φ ,++ αs) ⊢ H
  → foreach (λ G → Γ ≀ Φ ,, φ ⊢ G) Gs
  → foreach (λ G → Γ ≀ Φ ⊢ G) (so-substVec Gs φ αs H)
```

The proof of so-subst-preserves-typing is similar to the proof for fo-subst-preserves-typing, so we elide the precise details. As usual, all cases except the AppF-I case are proved by routine applications of induction. In the AppF-I case, we do a case analysis on the equality of φ and the applied variable ψ, and there are three cases, in parallel with the AppF-I case of (6). If φ and ψ match in name and arity, a first-order vector substitution is triggered (see Figure 2.2), and so we must use [:=]Vec-preserves-typing to prove that the vector substitution of the form F[αs := ...]Vec is well-formed. If φ and ψ differ in name or arity, then the second-order substitution is only applied to the arguments Gs, so we use so-substVec-preserves-typing to prove that the type arguments Gs are well-formed after applying a second-order substitution to each type expression in Gs.

Chapter 3

Category Theory

3.1 Category Theory Crash Course

For readers unfamiliar with the subject, category theory [1] can be thought of as a generalization of the theory of sets and functions. Category theory is used to formalize various mathematical structures and the relationships between these structures. The primary entity used in category theory is called a *category.* A category consists of objects, which are somewhat analogous to sets, and morphisms, which are somewhat analogous to functions. In contrast with sets, the objects of a category may be "opaque" in the sense that the objects do not support any notion of containing elements. For example, the objects of a category may be natural numbers, or bitstrings, or other non-set-like objects.

Categories are usually depicted as directed graphs called *diagrams* where the nodes represent objects and the arrows represent morphisms. For example, the diagram

$$\begin{tikzcd}
A \arrow[loop left, "id_A"] \arrow[r, "f"] & B \arrow[loop above, "id_B"] \arrow[r, "g"] & C \arrow[loop right, "id_C"]
\end{tikzcd}$$

represents a category with three objects, A , B, and C, and five morphisms, id_A, f, id_B, g and id_C. A category must have an *identity morphism* id_A for every object A, so sometimes these identity morphisms are left implicit in the diagram. Thus the category above may also be depicted as:

$$\begin{tikzcd}
A \arrow[r, "f"] & B \arrow[r, "g"] & C
\end{tikzcd}$$

We refer to the endpoints of a morphism as the *source object* and *target object* of the morphism. For example, the source object of f is A and the target object of f is B. This is denoted as $f : A \to B$. To denote that A is an object in the category $\mathcal{C}$, we write $A \in ob(\mathcal{C})$ or $A : \mathcal{C}$.

3.1.1 Formal Definition

Definition 3.1.1. Formally, a category $\mathcal{C}$ consists of the following data: a class (i.e., a collection that may not necessarily be a set) $ob(\mathcal{C})$ of objects, a class of morphisms $\forall X, Y : \mathcal{C}.\ \mathcal{C}(X, Y)$ where each morphism is assigned a source object and a target object, and a composition operator $\circ$ that composes two morphisms where the source of the first morphism matches the target of the second morphism, and the resulting morphism takes the source of the second morphism to the target of the first. The type of the composition operator is therefore $\forall X, Y, Z : \mathcal{C}. \mathcal{C}(Y, Z) \to \mathcal{C}(X, Y) \to \mathcal{C}(X, Z)$. For example, in the diagram above, the composition of $g : B \to C$ and $f : A \to B$ is $g \circ f : A \to C$, which can be pronounced as "g after f" or "g compose f." To form a category, these data must must also satisfy the following axioms:

- identity: for every morphism $f : A \to B$, it holds that $f \circ id_A = f = id_B \circ f$.
- associativity: for every triple of morphisms $f : C \to D$, $g : B \to C$, and $h : A \to B$, it holds that $(f \circ g) \circ h = f \circ (g \circ h)$.

The identity axiom states that composing a morphism f with an identity morphism simply yields f. The associativity axiom states that the composition of morphisms is associative, so the order in which we perform a chain of compositions does not matter. Notice that the definition of a category depends on some notion of equality between morphisms. This notion of equality between morphisms is often taken for granted in category theory textbooks, but we will have to address it directly in our formalization.

3.1.2 Examples of Categories

One of the most commonly used categories is the category Sets[1] of sets and functions. An object of Sets is a set and a morphism between two sets A and B is a function from A to B. The identity morphisms in Sets are identity functions, and composition of morphisms is simply function composition. Unlike objects of arbitrary categories, sets have elements, and we can reason about morphisms in Sets

[1] This category is sometimes denoted simply as Set, but we call it Sets in this thesis to avoid conflicts with the "Set" (universe) keyword in Agda

(functions) pointwise, i.e., by considering how a morphism $f : A \to B$ acts on each element $x \in A$. In fact, composition of morphisms in Sets is defined pointwise: the composition of $g : B \to C$ and $f : A \to C$ is defined $(g \circ f)\, x = g\,(f\,x)$ for each $x \in A$. It is straightforward to show that this pointwise composition satisfies the identity and associativity axioms.

Another commonly used category is the category of natural numbers, which we denote as $\mathbb{N}$. An object of $\mathbb{N}$ is a natural number, and a morphism between natural numbers n and m exists if and only if $n \leq m$. The category of natural numbers is depicted by the following diagram:

$$\begin{tikzcd} 0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & ... \end{tikzcd}$$

Since the relation $\leq$ is a *preorder*, i.e., a reflexive and transitive relation, the identity and compositions come from reflexivity and transitivity, while the axioms are trivial because there is at most one morphism $n \leq m$ for any objects n and m. By the same reasoning, any preorder can be considered a category, where the objects are given by the underlying set of the preorder, and the morphisms are given by the underlying relation of the preorder.

There are also many mechanisms in category theory for combining categories to produce new categories. For example, given two categories $\mathcal{C}$ and $\mathcal{D}$, we can take the product category $\mathcal{C} \times \mathcal{D}$ whose objects are pairs (A, B) where $A : \mathcal{C}$ and $B : \mathcal{D}$ and whose morphisms $(f, g) : (A, B) \to (A', B')$ are pairs of morphisms where $f : A \to A'$ in $\mathcal{C}$ and $g : B \to B'$ in $\mathcal{D}$. Composition of morphisms in the product category is done componentwise, i.e., $(f_1, f_2) \circ (g_1, g_2) = (f_1 \circ g_1, f_2 \circ g_2)$. It is also possible to define a k-ary product category $\mathcal{C}^k$ whose objects (resp. morphisms) are k-tuples of objects (resp. morphisms) from $\mathcal{C}$. Composition of morphisms in $\mathcal{C}^k$ is also done componentwise, i.e., $(f_1, ..., f_k) \circ (g_1, ..., g_k) = (f_1 \circ g_1, ..., f_n \circ g_n)$.

3.1.3 Functors

Another fundamental notion in category theory is that of a *functor*. Informally, a functor between two categories $\mathcal{C}$ and $\mathcal{D}$ is a mapping from $\mathcal{C}$ to $\mathcal{D}$ that preserves the categorical structure. In particular, a functor from $\mathcal{C}$ to $\mathcal{D}$ maps objects of $\mathcal{C}$ to objects of $\mathcal{D}$, and it also maps morphisms of $\mathcal{C}$ to morphisms of $\mathcal{D}$. The structure of a category is determined by its morphisms, and so for a functor to “preserve the structure” of a category, it must map morphisms in a particular way.

Definition 3.1.2. Formally, a functor F from $\mathcal{C}$ to $\mathcal{D}$, denoted as $F : \mathcal{C} \to \mathcal{D}$, consists of a mapping of objects, F_0, and a mapping of morphisms, F_1, and some properties. For each object $A : \mathcal{C}$, there is a corresponding object $F_0 A : \mathcal{D}$. For each morphism $f : A \to B$ in $\mathcal{C}$, there is a corresponding morphism $F_1 f : F_0 A \to F_0 B$. Sometimes F_0 and F_1 are referred to as the "action on objects" and the "action on morphisms," respectively. To be considered a functor, F_0 and F_1 must satisfy the following *functor laws*:

- identity: for every object $A : \mathcal{C}$, it holds that $F_1(id_A) = id_{F_0 A}$.
- homomorphism: for pair of morphisms $g : B \to C, f : A \to B$ in $\mathcal{C}$, it holds that $F_1(g \circ f) = F_1 g \circ F_1 f$.

In other words, F_1 must preserve identities and compositions of morphisms. Just as we saw in the axioms for the definition of a category, the functor laws require some notion of equality between morphisms (the morphisms being compared for equality are in the category $\mathcal{D}$ in the identity and homomorphism conditions above). A functor from $\mathcal{C}$ to $\mathcal{C}$ is called an *endofunctor* on $\mathcal{C}$.

Given two functors $G : \mathcal{D} \to \mathcal{E}$ and $F : \mathcal{C} \to \mathcal{D}$, we can compose them to get the functor $G \circ F : \mathcal{C} \to \mathcal{E}$ whose action on an object $A : \mathcal{C}$ is given by $(G \circ F)_0\, A = G_0(F_0 A)$. The action of $G \circ F$ on a morphism $f : A \to B$ is given by $(G \circ F)_1 f = G_1(F_1 f)$.

3.1.4 Examples of Functors

The simplest example of a functor is the identity functor, I. Given a category $\mathcal{C}$, the identity functor on $\mathcal{C}$ maps every object to itself and does the same for morphisms. Clearly, these mappings satisfy the functor laws, since $I_1(id_A) = id_A = id_{I_0 A}$ and $I_1(g \circ f) = g \circ f = I_1 g \circ I_1 f$, where I_0 and I_1 are the mappings on objects and morphisms, respectively, of I.

Another simple functor is the constant functor. Given categories $\mathcal{C}$ and $\mathcal{D}$ and an object $d : \mathcal{D}$, the constant functor $K_d : \mathcal{C} \to \mathcal{D}$ sends every object to d and sends every morphism to id_d. The functor laws are trivially satisfied because every morphism is mapped to id_d.

Another example of a functor is the *product functor* $- \times - : \mathsf{Sets} \times \mathsf{Sets} \to \mathsf{Sets}$ from the product category of sets to the category of sets. Here we abuse notation, writing $- \times -$ to denote the product functor, the product of categories, and the cartesian product of sets, depending on the types of the arguments given. The functor $- \times - : \mathsf{Sets} \times \mathsf{Sets} \to \mathsf{Sets}$ maps each pair of sets $(A, B) \in \mathsf{Sets} \times \mathsf{Sets}$ to their cartesian product $A \times B$. It maps each pair of morphisms $(f, g) : (A, B) \to (A', B')$ to the function $(f \times g) : (A \times B) \to (A' \times B')$ such that $(f \times g)(x, y) = (fx, gy)$. Identities and composition of morphisms are given componentwise, i.e., $(g1 \times g2) \circ (f1 \times f2) = (g1 \circ f1) \times (g2 \circ f2)$. This ensures the functor laws are satisfied for $(f \times g)$, since they are satisfied for the components f and g in Sets.

Functors are a very useful concept for reasoning about data types in functional programming languages. Given a programming language whose types can be thought of as sets, we can model the types and functions of this language in the category Sets. Let us denote the set associated to a type T as $[\![T]\!]$, called the *interpretation* of T. If we think of the types in the language as sets, then the terms of a type T can be thought of as the elements of $[\![T]\!]$. Functions in the language of type $f : T \to T'$ can be thought of as functions $[\![f]\!] : [\![T]\!] \to [\![T']\!]$ in the category Sets. If we have built-in notions of lists and trees in our programming language, we can interpret both the list and tree data type constructors as endofunctors on Sets. The *List* functor maps each set $[\![T]\!]$ to the set $\mathit{List_0}\ [\![T]\!]$ of lists of elements of $[\![T]\!]$. Given a function $[\![f]\!] : [\![T]\!] \to [\![T']\!]$, the *List* functor produces a function from $\mathit{List_0}\ [\![T]\!] \to \mathit{List_0}\ [\![T']\!]$ that applies f to each element of the input list without modifying the structure of the list. Like with the product functor, the identities and composition of morphisms are given "componentwise," except that a list may have 0 or more components rather than having exactly two components like a pair. Similarly, the *Tree* functor maps each set $[\![T]\!]$ to the set $\mathit{Tree_0}\ [\![T]\!]$ of trees with elements of $[\![T]\!]$ at the nodes, and maps a function $[\![f]\!] : [\![T]\!] \to [\![T']\!]$ to a function $\mathit{Tree_1}\ [\![f]\!] : \mathit{Tree_0}\ [\![T]\!] \to \mathit{Tree_0}\ [\![T']\!]$ that applies $[\![f]\!]$ to each node in a tree without modifying the shape of the tree. Once again, the functor laws will be satisfied because composition of morphisms are given "componentwise," i.e., the composed morphisms are applied one after another to each node in the tree. Many other data type constructors can be viewed as functors, as we will see once we have implemented the set interpretation of types for the calculus $\mathcal{N}$.

It turns out that functors themselves can be used as the objects in a category. Such a category is called a *functor category*, and functors between functor categories are called *higher-order functors*. Natural transformations between higher-order functors are called *higher-order natural transformations*. Functor categories and higher-order functors are used frequently in our formalization, and we will formalize these notions in the next chapter.

3.1.5 Natural Transformations

The last categoryical concept introduced in this section is that of a *natural transformation*. Just as a functor is a mapping between two categories, a natural transformation is a mapping between functors.

Definition 3.1.3. Given two categories $\mathcal{C}$ and $\mathcal{D}$ and two functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation η between F and G is denoted with a double-arrow, i.e., $\eta : F \Rightarrow G$, and consists of the following data:

- components: for every object $A : \mathcal{C}$, we have a *component of* η *at* A, $\eta_A : F_0 A \to G_0 A$
- naturality: for every morphism $f : A \to B$ in $\mathcal{C}$, we have a *commuting square*

$$\begin{tikzcd}
F_0 A \arrow[r, "\eta_A"] \arrow[d, "F_1 f"] & G_0 A \arrow[d, "G_1 f"] \\
F_0 B \arrow[r, "\eta_B"] & G_0 B
\end{tikzcd}$$

In order to have a commuting square, the paths in the diagram from top-left to bottom-right must be equivalent, according to some specified notion of morphism equivalence. In the case shown above, this means it must hold that $\eta_B \circ F_1 f = G_1 f \circ \eta_A$.

Consider the previous examples of *List* and *Tree* as functors on a category of types. One example of a natural transformation between *Tree* and *List* is the interpretation of a polymorphic flattening function that takes a tree (of any type) as input and produces a list containing the same data. The naturality condition for *flatten* and a function $[\![f]\!] : [\![T]\!] \to [\![T']\!]$ says that

$$\begin{tikzcd}
\mathit{Tree}_0\, [\![T]\!] \arrow[r, "\mathit{flatten}_{[\![T]\!]}"] \arrow[d, "\mathit{Tree}_1 [\![f]\!]"] & \mathit{List}_0\, [\![T]\!] \arrow[d, "\mathit{List}_1 [\![f]\!]"] \\
\mathit{Tree}_0\, [\![T']\!] \arrow[r, "\mathit{flatten}_{[\![T']\!]}"] & \mathit{List}_0\, [\![T']\!]
\end{tikzcd}$$

must commute, i.e., $\mathit{flatten}_{[\![T']\!]} \circ \mathit{Tree}_1 [\![f]\!] = \mathit{List}_1 [\![f]\!] \circ \mathit{flatten}_{[\![T]\!]}$. In other words, first mapping a function over a tree and then flattening is the same as flattening the tree and then mapping the function over the resulting list. This naturality condition is satisfied because the flattening function (natural transformation) is defined uniformly for every type (object), i.e., it repackages the data in a tree in a type-independent way. The naturality condition for natural transformations between functorial data types (i.e., data types that can be interpreted as functors) can informally be stated as follows: modifying the data in a shape-preserving way and then repackaging the data into a different shape is the same as first repackaging the data into a different shape and then modifying the data in a shape-preserving way.

The fact that polymorphic functions in a functional programming language are easily modeled as natural transformations is part of the reason the calculus $\mathcal{N}$ includes a built-in type of natural transformations.

Given functors $F, G, H : \mathcal{C} \to \mathcal{D}$ and two natural transformations $\eta : G \Rightarrow H$ and $\delta : F \Rightarrow G$, we can compose η and δ by defining a natural transformation $\eta \circ \delta$ where each component $(\eta \circ \delta)_X$ is defined as a composition of components, i.e., $(\eta \circ \delta)_X = \eta_X \circ \delta_X$. This notion of composition of natural transformations is called *vertical* composition. There is another notion of composition called *horizontal* composition, but we do not use horizontal composition in this thesis. We can also define an identity natural transformation id_F for any functor F where each component $(id_F)_X$ is given by the identity morphism on $F_0\,X$. Using these notions of composition and identity for natural transformations, we can define the *functor category* for two categories $\mathcal{C}$ and $\mathcal{D}$, whose objects are functors from $\mathcal{C}$ to $\mathcal{D}$.

Definition 3.1.4. Given two categories $\mathcal{C}$ and $\mathcal{D}$, the functor category $[\mathcal{C}, \mathcal{D}]$[2] is defined by the following data:

- objects: functors from $\mathcal{C}$ to $\mathcal{D}$
- morphisms between objects F and G: natural transformations between F and G
- composition: vertical composition of natural transformations
- identity: identity natural transformations

The fact that composition is defined componentwise for natural transformations ensures that the axioms for identities and compositions are satisfied, since they are satisfied in the category $\mathcal{D}$ in which the components of these natural transformations live.

Given functors $F : \mathcal{C} \to \mathcal{D}$, $G, H : \mathcal{D} \to \mathcal{E}$, and a natural transformation $\eta : G \Rightarrow H$, we can construct a new natural transformation $\eta F : G \circ F \Rightarrow H \circ F$ whose components are defined by $(\eta F)_X = \eta_{FX}$. This natural transformation ηF is called the *whiskering* of η by F. We can also whisker in the other direction. Given functors $F : \mathcal{D} \to \mathcal{E}$, $G, H : \mathcal{C} \to \mathcal{D}$ and a natural transformation $\eta : G \Rightarrow H$, we can construct $F\eta : F \circ G \Rightarrow F \circ H$ whose components are defined by $(F\eta)_X = F_1(\eta_X)$.

[2] The notation $[\mathcal{C}, \mathcal{D}]$ is somewhat nonstandard: the functor category for $\mathcal{C}$ and $\mathcal{D}$ is usually written as $\mathcal{D}^{\mathcal{C}}$, but this exponent notation does not translate easily to Agda syntax. Also, the notation $[\mathcal{C}, \mathcal{D}]$ is used in [5] to denote a special kind of functor category, but we are not concerned with this special class in our formalization (see the comments about ω-cocontinuity in Chapter 4). For these reasons, we use the bracket notation $[\mathcal{C}, \mathcal{D}]$ for all functor categories throughout this thesis.

Another useful notion based on natural transformations is that of a *natural isomorphism* between two functors:

Definition 3.1.5. Given functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural isomorphism* between F and G is a pair of natural transformations $\eta : F \Rightarrow G$ and $\eta^{-1} : G \Rightarrow F$ such that, for every object $A : \mathcal{C}$, the components $\eta_A : F_0 A \to G_0 A$ and $\eta_A^{-1} : G_0 A \to F_0 A$ form an isomorphism, i.e., $\eta_A \circ \eta_A^{-1} = id_{G_0 A}$ and $\eta_A^{-1} \circ \eta_A = id_{F_0 A}$.

3.2 Category Theory in Agda

In this section we formalize the notions of categories, functors, and natural transformations in Agda. All of these notions are already formalized in the category theory library used in this thesis[3], but we present the formalizations step-by-step here to aid with readability and keep this chapter self-contained.

3.2.1 Categories

Recall from Definition 3.1.1 that a category consists of objects, morphisms, a composition operator, and some axioms. Recall that objects and morphisms are given as *collections* rather than sets, so they might be sets, but they could also be larger collections that are not sets (e.g., the collection of all sets). To account for this, our definition of a category will be parameterized by a universe level for the collection of objects and a universe level for the collection of morphisms. These universe level parameters allow users of the category theory library to specify the size of each collection when declaring a new category. A first attempt at defining the type of categories in Agda might look like the following:

```
record Category₀ (o ℓ : Level) : Set (lsuc (o ⊔ ℓ)) where
  infix 4 _⇒_
  infixr 9 _∘_

  field
    Obj : Set o
    _⇒_ : Rel Obj ℓ
    id : ∀ {A} → (A ⇒ A)
    _∘_ : ∀ {A B C} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)
```

[3] All of the Agda code in this chapter is from the agda-categories library introduced in [3].

This (incomplete) definition of a category is defined as a record type and takes two explicit parameters: the universe level o for the collection of objects and the universe level ℓ for the collection of morphisms. The "$\mathsf{Set}\,(\mathsf{lsuc}(\mathsf{o} \sqcup \ell))$" after the colon on the first line indicates the universe in which the $\mathsf{Category}_0$ type lives, which is one level above the least upper bound[4] of o and ℓ. The next two lines set the fixity and operator precedence of the infix operators for the type of morphisms and the type of composition in this category.

The first field required in this definition is Obj, the type of objects, which can be any Agda type that fits in the universe $\mathsf{Set}\,\mathsf{o}$ determined by the o parameter. The next field $\_ \Rightarrow \_$ is the type of morphisms in the category being defined. The Agda type $\mathsf{Rel}\,\mathsf{A}\,\ell$ is a type from the standard library with the following definition:

```
Rel : ∀ {a} (A : Set a) (ℓ : Level) → Set (a ⊔ lsuc ℓ)
Rel A ℓ = A → A → Set ℓ
```

That is, $\mathsf{Rel}\,\mathsf{A}\,\ell$ is the type of homogeneous relations on the type A. If $\mathsf{x}, \mathsf{y} : \mathsf{A}$ and $\mathsf{R} : \mathsf{Rel}\,\mathsf{A}\,\ell$, then $\mathsf{R}\,\mathsf{x}\,\mathsf{y}$ is the type of proofs that x and y are related by R. So in the definition of $\mathsf{Category}_0$, the type of $\_ \Rightarrow \_$ expands to $\mathsf{Obj} \to \mathsf{Obj} \to \mathsf{Set}\,\ell$. So given objects A and B, $\mathsf{A} \Rightarrow \mathsf{B}$ is the type of morphisms from A to B.

The last two fields in the definition of $\mathsf{Category}_0$ are for identity morphisms and composition of morphisms. For the id field, we must provide a function that, given any object A, produces a morphism from A to itself. For the composition field ($\_ \circ \_$), we must provide a function that composes morphisms of the appropriate types.

The careful reader will notice that this definition ($\mathsf{Category}_0$) is incomplete. In particular, this definition of a category is missing the axioms required for identity morphisms and the composition operator, so we will need to add additional fields to account for these axioms.

We also noted previously that the axioms for composition are given with respect to some notion of morphism equality. Given that Agda has a built-in notion of propositional equality, it might seem sensible to use propositional equality for comparing morphisms in all categories. But as noted in [3], there are several issues with this. The principal issue is that in many categories (e.g., Sets), a morphism is (some kind of) a function, and extra steps and axioms are needed in Agda to reason about equality of

[4] The function lsuc takes a universe level and returns the next universe level in the hierarchy. The function $\_ \sqcup \_$ takes two universe levels and returns the least upper bound of the two universe levels, i.e., the smallest universe level that is greater than or equal to the input universe levels.

functions in an *extensional* way, as we would like to do in some categories. For example, in the category Sets, we want to consider two functions to be equal iff they are equal on all inputs. This notion of function equality is called *extensional equality.* However, the default notion (propositional equality) of equality in Agda, written f ≡ g, compares f and g for *intensional equality*, which means f and g must be textually identical functions for the type f ≡ g to be inhabited. Another issue with using propositional equality to compare morphisms in all categories is that we may want to use different notions of equality in different categories. We may even want to define two categories that are the same except in their notion of morphism equality.

For these reasons, the notion of equality for morphisms must be given along with the other fields when declaring a new category. We will also add a third universe level parameter to the definition of a category that is used to specify the universe in which proofs of morphism equality should live. This notion of morphism equality is formalized in Agda as an *equivalence relation* on the type of morphisms. An equivalence relation is a relation that is reflexive, transitive, and symmetric. So in addition to providing the relation used for morphism equality, users of the category theory library must also provide a proof that this relation is an equivalence relation when declaring a new category. Given a notion of morphism equivalence, the library definition also adds fields to account for the axioms about identity morphisms and composition. The full definition of a category is given in Agda as:

```
record Category (o ℓ e : Level) : Set (lsuc (o ⊔ ℓ ⊔ e)) where
  infix 4 _≈_ _⇒_
  infixr 9 _∘_

  field
    Obj : Set o
    _⇒_ : Rel Obj ℓ
    id : ∀ {A} → (A ⇒ A)
    _∘_ : ∀ {A B C} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)

    _≈_ : ∀ {A B} → Rel (A ⇒ B) e

    assoc     : ∀ {A B C D} {f : A ⇒ B} {g : B ⇒ C} {h : C ⇒ D} → (h ∘ g) ∘ f ≈ h ∘ (g ∘ f)
    sym-assoc : ∀ {A B C D} {f : A ⇒ B} {g : B ⇒ C} {h : C ⇒ D} → h ∘ (g ∘ f) ≈ (h ∘ g) ∘ f
    identityˡ : ∀ {A B} {f : A ⇒ B} → id ∘ f ≈ f
```

```
identityʳ : ∀ {A B} {f : A ⇒ B} → f ∘ id ≈ f
identity² : ∀ {A} → id ∘ id {A} ≈ id {A}
equiv     : ∀ {A B} → IsEquivalence (_≈_ {A} {B})
∘-resp-≈ : ∀ {A B C} {f h : B ⇒ C} {g i : A ⇒ B} → f ≈ h → g ≈ i → f ∘ g ≈ h ∘ i
```

Notice the record type declaration includes a new universe parameter, e, for the universe level of proofs of morphisms equivalence. This definition also includes several additional fields that were not present in the previous definition Category$_0$. The first is _ ≈ _, the type of morphism equivalence. By the definition of Rel, the type of _ ≈ _ expands to (∀{A B} → (A ⇒ B) → (A ⇒ B) → Set e), so given morphisms f, g : A ⇒ B, f ≈ g is the type of proofs that f and g are equivalent morphisms.

The assoc and sym-assoc fields express that composition of morphisms is associative. Although sym-assoc is somewhat redundant, it is included because the library authors assert that this redundancy makes other concepts easier to express in the library. The next three identity fields express some axioms about composition of identity morphisms. The first two identity axioms, identity^{l} and identity^{r}, are standard, and identity^{2}, although somewhat redundant, is included as a design decision, similarly to sym-assoc.

The equiv field expresses that the relation given for morphism equality is indeed an equivalence relation. The type IsEquivalence is itself a record type with three fields. To provide a term of type IsEquivalence ≈ for some relation ≈, we must provide proofs that ≈ is reflexive, symmetric, and transitive:

```
refl : ∀ {f : A ⇒ B} → f ≈ f
sym : ∀ {f g : A ⇒ B} → f ≈ g → g ≈ f
trans : ∀ {f g h : A ⇒ B} → f ≈ g → g ≈ h → f ≈ h
```

The last field in the definition of a category is ∘-resp-≈, which expresses that morphism composition respects the equivalence relation given for morphism equality. That is, if two pairs of morphisms are equivalent, then their compositions should be equivalent. This axiom is non-standard and is a result of the design decision to use a local (user-defined) version of morphism equality in the definition of a category.

3.2.2 Examples of Categories in Agda

As a first example, let us consider the definition of Sets in Agda:

```
Sets : Category (lsuc lzero) lzero lzero
Sets = record
  { Obj       = Set
  ; _⇒_       = λ A B → (A → B)
  ; _≈_       = λ f g → f ≡-ext g
  ; id        = λ x → x
  ; _∘_       = λ f g x → f (g x)
  ; assoc     = ≡.refl
  ; sym-assoc = ≡.refl
  ; identityˡ = ≡.refl
  ; identityʳ = ≡.refl
  ; identity² = ≡.refl
  ; equiv     = record                                            (7)
    { refl  = ≡.refl
    ; sym   = λ eq → ≡.sym eq
    ; trans = λ eq₁ eq₂ → ≡.trans eq₁ eq₂
    }
  ; ∘-resp-≈ = resp
  }
  where resp : ∀ {A B C : Set} {f h : B → C} {g i : A → B}
             → ({y : B} → f y ≡ h y)
             → ({x : A} → g x ≡ i x)
             → {x : A} → f (g x) ≡ h (i x)
        resp {h = h} f≡h g≡i {x} = ≡.trans f≡h (≡.cong h g≡i)
```

The collection of objects is the base universe Set, which is really shorthand for Set lzero, where lzero is the base universe level. Since Set is an element of the universe Set (lsuc lzero) (also written Set_1), the first universe level parameter to Category in the type of Sets is lsuc lzero. The collection of morphisms between objects A and B is the set of functions A → B, and since A → B : Set, the second universe level parameter is lzero. The type of morphism equality is defined by an auxiliary function

```
_≡-ext_ : ∀ {ℓ} {A B : Set ℓ} → (f g : A → B) → Set ℓ
f ≡-ext g = ∀ {x} → f x ≡ g x
```

that asserts two functions are extensionally equal, i.e., for every element x in type A, f x is equal to g x. The identity morphism for a set A is the identity function on A, and composition of morphisms is given by function composition. Since the two compositions (h ∘ g) ∘ f and h ∘ (g ∘ f) in Sets both compute to λx → h(g(fx)), i.e., the compositions are definitionally equal, the associativity proofs for this category are trivial and can be proved using ≡.refl. The same reasoning applies to the identity proofs, since, for any function f, the composition of f with the identity function is definitionally equal to f. For example, f ∘ id = λx → f ((λy → y) x) = λx → f x = f. To satisfy the ≡ field, we must give a proof of

```
{A B : Set} → IsEquivalence (λ (f g : A → B) → f ≡-ext g)
```

to show that ≡-ext is an equivalence relation. For this we must produce a term of the form

```
  record
  { refl = {!!} -- Goal:  {f :  A → B} {x :  A} → f x ≡ f x
  ; sym = {!!} -- Goal:  {f g :  A → B} → f ≡-ext g → g ≡-ext f
  ; trans = {!!} -- Goal:  {f g h :  A → B} → f ≡-ext g → g ≡-ext h → f ≡-ext h
  }
```

which we define in (7) using the ≡.refl constructor of the propositional equality type and the functions

```
sym : ∀ {a : Level} {A : Set a} {x y : A} → x ≡ y → y ≡ x
sym ≡.refl = ≡.refl

trans : ∀ {a : Level} {A : Set a} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans ≡.refl y≡z = y≡z
```

for symmetry and transitivity of _ ≡ _ from the standard library. Note that we use the *qualified names* of refl, sym, and trans in (7), i.e., we prepend each definition with the name of its parent module and a period. This is enabled by the import statement

```
import Relation.Binary.PropositionalEquality as ≡
```

that imports the module for propositional equality from the standard library and renames the module to ≡. Using these qualified names makes it easier to distinguish between the refl, sym, and trans functions associated with the propositional equality type and the refl, sym, and trans fields of the IsEquivalence type.

Let us also consider the definition of a product category in Agda:

```
Product : ∀ {o ℓ e o′ ℓ′ e′ : Level}
          → (C1 : Category o ℓ e) (C2 : Category o′ ℓ′ e′)
          → Category (o ⊔ o′) (ℓ ⊔ ℓ′) (e ⊔ e′)
Product C1 C2 = record
  { Obj       = C1.Obj ×' C2.Obj
  ; _⇒_       = λ { (A1 , A2) (B1 , B2) → (A1 C1.⇒ B1) ×' (A2 C2.⇒ B2) }
  ; _≈_       = λ { (f1 , f2) (g1 , g2) → (f1 C1.≈ g1) ×' (f2 C2.≈ g2) }
  ; _∘_       = λ { (f1 , f2) (g1 , g2) → (f1 C1.∘ g1) , (f2 C2.∘ g2) }
  ; id        = C1.id , C2.id
  ; assoc     = C1.assoc , C2.assoc
  ; sym-assoc = C1.sym-assoc , C2.sym-assoc
  ; identityˡ = C1.identityˡ , C2.identityˡ
  ; identityʳ = C1.identityʳ , C2.identityʳ
  ; identity² = C1.identity² , C2.identity²
  ; equiv     = record
    { refl    = C1.Equiv.refl , C2.Equiv.refl
    ; sym     = λ { (f1≈g1 , f2≈g2) → (C1.Equiv.sym f1≈g1) , (C2.Equiv.sym f2≈g2) }
    ; trans = λ { (f1≈g1 , f2≈g2) (g1≈h1 , g2≈h2)
                → (C1.Equiv.trans f1≈g1 g1≈h1) , (C2.Equiv.trans f2≈g2 g2≈h2) }
    }
  ; ∘-resp-≈ =
      λ { (f1≈h1 , f2≈h2) (g1≈i1 , g2≈i2) → (C1.∘-resp-≈ f1≈h1 g1≈i1) , (C2.∘-resp-≈ f2≈h2 g2≈i2) }
  }
  where module C1 = Category C1
        module C2 = Category C2
```

This definition of Product takes two arbitrary categories C1 and C2 with potentially different universe levels and returns the product category Product C1 C2. An object of the product category is a pair of objects (A1, A2) where A1 is an object of C1 and A2 is an object of C2. A morphism from (A1, A2) to (B1, B2) is a pair of morphisms (f1, f2) where f1 : A1 → B1 and f2 : A2 → B2. Equality and composition of morphisms are defined componentwise, so all of the remaining fields can be defined using the corresponding fields from C1 and C2. The syntax $\lambda\{x \to ...\}$ allows us to pattern match on the argument x, and the module statements module C1 = Category C1 and module C2 = Category C2 allow us to refer to the fields of C1 and C2 using their qualified names.

3.2.3 Functors

The definition of a functor in Agda is also given as a record type. Recall from Definition 3.1.2 that a functor consists of a mapping of objects, a mapping of morphisms, and some axioms. These data are captured by the following record type:

```
record Functor {o ℓ e o′ ℓ′ e′ : Level}
               (C : Category o ℓ e)
               (D : Category o′ ℓ′ e′) : Set (o ⊔ ℓ ⊔ e ⊔ o′ ⊔ ℓ′ ⊔ e′) where
  private module C = Category C
  private module D = Category D

  field
    F₀ : C.Obj → D.Obj
    F₁ : ∀ {A B} (f : C [ A , B ]) → D [ F₀ A , F₀ B ]

    identity     : ∀ {A} → D [ F₁ (C.id {A}) ≈ D.id ]
    homomorphism : ∀ {X Y Z} {f : C [ X , Y ]} {g : C [ Y , Z ]} →
                     D [ F₁ (C [ g ∘ f ]) ≈ D [ F₁ g ∘ F₁ f ] ]
    F-resp-≈ : ∀ {A B} {f g : C [ A , B ]} → C [ f ≈ g ] → D [ F₁ f ≈ F₁ g ]
```

First notice that the universe level parameters for the Functor type are implicit rather than explicit. There are also two groups of universe level parameters since C and D may differ in these parameters. The F_0 and F_1 fields are the functor's actions on objects and morphisms, respectively. The Agda syntax $C[A, B]$ is shorthand for the type of morphisms from A to B in the category C. The identity field expresses that F_1 must preserve identity morphisms. The syntax $D[f \approx g]$ is shorthand for the type of morphism equality in the category D. The homomorphism field expresses that F_1 must preserve compositions. The syntax D[g ∘ f] is shorthand for the composition of g and f in the category D. The last field, F-resp-≈, expresses that F_1 must preserve morphism equivalence. Similar to the ∘-resp-≈ field in the definition of a category, this axiom is non-standard, but it is necessary because each category comes with its own notion of morphism equivalence.

To construct a value of type Functor C D for some categories C and D, we must provide a mapping of objects, a mapping of morphisms, and three proofs that these mappings interact with composition and morphism equality as expected.

3.2.4 Natural Transformations

Finally, we can define the type of natural transformations in Agda. The type of natural transformations is also given as a record type. Recall from Definition 3.1.3 that a natural transformation consists of a collection of components and a naturality (commuting square) condition. These data are captured by the following definition in Agda:

```
record NaturalTransformation {o ℓ e o′ ℓ′ e′ : Level} {C : Category o ℓ e}
                             {D : Category o′ ℓ′ e′}
                             (F G : Functor C D) : Set (o ⊔ ℓ ⊔ ℓ′ ⊔ e′) where
  private
    module F = Functor F
    module G = Functor G
  open F using (F₀; F₁)
  open Category D

  field
    η           : ∀ X → D [ F₀ X , G.F₀ X ]
    commute : ∀ {X Y} (f : C [ X , Y ]) → η Y ∘ F₁ f ≈ G.F₁ f ∘ η X
    sym-commute : ∀ {X Y} (f : C [ X , Y ]) → G.F₁ f ∘ η X ≈ η Y ∘ F₁ f
```

Again, the universe level parameters are implicit, along with the category parameters C and D. The module and open statements bring different definitions into scope to make the definition of natural transformations more concise. In particular, open F brings F_0 and F_1 into scope so we can write F_0 rather than $F.F_0$. The open Category D statement brings all of the fields of D into scope, e.g., _∘_ and

$\_ \approx \_$. The first field, η, is the collection of components of the natural transformation, indexed by the objects of C. The commute and sym-commute fields require the user to give a proof of a commuting square for every morphism in C. Similar to sym-assoc from the definition of Category, the redundancy of sym-commute is a design decision by the library authors.

Chapter 4

Set Semantics of Types

In this chapter we present the set semantics from [5] for the calculus $\mathcal{N}$ and formalize this semantics in Agda[1]. To interpret the variables appearing in a type, we introduce the notion of set environments. We show that set environments form a category and then interpret the types of $\mathcal{N}$ as functors from the category of set environments to the category of sets.

Some of the types from $\mathcal{N}$ (e.g., the product type) can be interpreted using standard categorical constructs, some of which have already been introduced (e.g., the product functor for sets). The Nat type is interpreted as a set of natural transformations that satisfy an additional property needed to ensure parametricity. To interpret a μ-type, we first interpret its body as a higher-order functor $\mathcal{H}$ and then interpret the μ-type as the fixpoint $\mu\mathcal{H}$ of this higher-order functor, applied to the interpretation of the arguments of the μ-type. Informally, the fixpoint is obtained by repeatedly applying the higher-order functor $\mathcal{H}$ until this process converges, i.e., until we reach the fixpont $\mu\mathcal{H}$, which satisfies $\mu\mathcal{H} \cong \mathcal{H}\,(\mu\mathcal{H})$.

It is worth noting that in [5], the types of $\mathcal{N}$ are interpreted specifically as ω-*cocontinuous* functors on *locally finitely presentable categories*. Local finite presentability and ω-cocontinuity are technical properties of categories and functors, respectively, that ensure the fixpoints used to interpret μ-types exist. In this formalization, we use a concrete construction (a data type in Agda) to interpret μ-types, and so we do not need to formalize ω-cocontinuity and local finite presentability. For these reasons, we make no further mention of ω-cocontinuity and local finite presentability in this thesis.

[1] Unless otherwise specified, all of the Agda code in this chapter is from our implementation.

4.1 Set Environments

A type is interpreted with respect to an environment that assigns meaning to each of the variables in the type.

Definition 4.1.1. A *set environment* maps each variable of arity k to a a functor from the k-ary product category Sets^k to Sets. A morphism of set environments $f : \rho \to \rho'$ only exists when ρ and ρ' are equal on all non-functorial variables. A morphism of set environments $f : \rho \to \rho'$ sends each non-functorial variable ψ^k to the identity natural transformation on $\rho\psi^k = \rho'\psi^k$ and sends each functorial variable φ^k to a natural transformation from the functor $\rho\varphi^k : \mathsf{Sets}^k \to \mathsf{Sets}$ to the functor $\rho'\varphi^k : \mathsf{Sets}^k \to \mathsf{Sets}$. Composition of morphisms is given pointwise, i.e., $(g \circ f)\varphi^k = (g\varphi^k) \circ (f\varphi^k)$, where the second composition is a (vertical) composition of natural transformations. An identity morphism of set environments maps a set environment ρ to itself by sending every variable φ^k to the identity natural transformation on $\rho\varphi^k$. These data yield a category of set environments.

Note that a 0-ary functor of sets, i.e, a functor of type $\mathsf{Sets}^0 \to \mathsf{Sets}$, is equivalent to a set, because Sets^0 is a category with a single object (the empty tuple of sets) and a single identity morphism. Thus a functor from Sets^0 "picks out" a set X by sending the single object of Sets^0 to X, and it sends the single morphism of Sets^0 to the identity morphism id_X. By the same reasoning, a natural transformation between functors of type $\mathsf{Sets}^0 \to \mathsf{Sets}$ is just a function between sets. Thus, a set environment maps each variable α of arity 0 to a set, and a morphism $f : \rho \to \rho'$ maps α to a function $f\,\alpha : \rho\,\alpha \to \rho'\alpha$.

Definition 4.1.2. Given a set environment ρ, a functorial variable of arity k, and a functor $F : \mathsf{Sets}^k \to \mathsf{Sets}$, we define the *environment extension* of ρ as $\rho[\varphi := F] = \rho'$ where $\rho'\psi = F$ if $\psi = \varphi$ and $\rho'\psi = \rho\psi$ otherwise. Environment extension can also be defined for a vector of variables $\overline{\varphi}$ and a vector of functors $\overline{F}$ of the appropriate arities and types. It is denoted $\rho[\overline{\varphi := F}]$ and is defined by induction on the (equal) lengths of the vectors $\overline{\varphi}$ and $\overline{F}$.

This notion of environment extension can be extended to *morphisms* of environments:

Definition 4.1.3. Given a functorial variable φ, set environments ρ and ρ', a morphism $f : \rho \to \rho'$, functors $F, G : \mathsf{Sets}^k \to \mathsf{Sets}$, and a natural transformation $\eta : F \Rightarrow G$, we define the extension of f as $f[\varphi := \eta] = f'$ where $f'\psi = \eta$ if $\psi = \varphi$ and $f'\psi = f\psi$ otherwise. Environment extension for morphisms can also be defined for a vector of variables $\overline{\varphi}$ and a vector of natural transformations $\overline{\eta}$ of the appropriate arities and types. It is denoted $f[\overline{\varphi := \eta}]$ and is defined by induction on the (equal) lengths of the vectors $\overline{\varphi}$ and $\overline{\eta}$.

4.2 Set Environments in Agda

To formalize the category of set environments in Agda, we must first define the generalized product category. We define a function Cat^ that takes a natural number k and returns the k-ary product category for any category C:

```
module VecCat {o ℓ e : Level} (C : Category o ℓ e) where

  private module C = Category C

  foreach2 : ∀ {ℓ} {r} {k : ℕ} {A B : Set ℓ}
    → (P : A → B → Set r)
    → Vec A k → Vec B k → Set r

  componentwise-Vec-≈ : ∀ {k} {Xs Ys : Vec C.Obj k} → (fs gs : foreach2 C._⇒_ Xs Ys) → Set e

  compose-Vec-morphs : ∀ {k : ℕ} {Xs Ys Zs : Vec C.Obj k}
                      → foreach2 C._⇒_ Ys Zs → foreach2 C._⇒_ Xs Ys
                      → foreach2 C._⇒_ Xs Zs

  idVec : ∀ {k : ℕ} → (Cs : Vec C.Obj k) → foreach2 C._⇒_ Cs Cs

  Cat^ : ℕ → Category o ℓ e
  Cat^ k = record
    { Obj = Vec C.Obj k
    ; _⇒_ = λ Xs Ys → foreach2 C._⇒_ Xs Ys
    ; _≈_ = λ fs gs → componentwise-Vec-≈ fs gs
    ; id = λ {Xs} → idVec Xs
    ; _∘_ = λ fs gs → compose-Vec-morphs fs gs
    ...
    }
```

We define this function Cat^ in a module[2] parameterized by an arbitrary category C, since the k-ary product category can be defined for any category. An object of Cat^ k is a length k vector of objects from C. A morphism in Cat^ k from Xs to Ys is a k-tuple of functions $fs = (f_1 : Xs_1 \to Ys_1, ..., f_k : Xs_k \to Ys_k)$, formalized in terms of the function foreach2. The function foreach2 is the analogue of foreach for a binary relation $P : A \to B \to \mathrm{Set}\, r$, so a term of type foreach2 P Xs Ys is a k-tuple $(p_1, .., p_k)$ where $p_i : P\, Xs_i\, Ys_i$. Morphism equality and composition are defined compontentwise, using the corresponding fields from C on each pair of morphisms in the k-tuples fs and gs. Identity morphisms are defined as k-tuples of identity morphisms in C. We elide the remaining fields[3] because they all have the form of applying the corresponding fields of C componentwise to vectors of objects and k-tuples of morphisms. Using the definition of Cat^ k , we define the k-ary product of Sets as the instance of Cat^ k with C instantiated to Sets:

```
Sets^ : ℕ → Category (lsuc lzero) lzero lzero
Sets^ k = Cat^ Sets k
```

Now that we have defined Sets^ k, we can define set environments as a record type:

```
SetEnvTC : Set₁
SetEnvTC = ∀ {k : ℕ} → TCVar k → Functor (Sets^ k) Sets

SetEnvFV : Set₁
SetEnvFV = ∀ {k : ℕ} → FVar k → Functor (Sets^ k) Sets

record SetEnv : Set₁ where
  constructor SetEnv[_,_]
  field
    tc : SetEnvTC
    fv : SetEnvFV
```

A set environment consists of two fields: a mapping for type constructor variables, tc, and a mapping for functorial variables, fv. The mapping for type constructor (resp., functorial) variables sends a type constructor (resp., functorial) variable of arity k to a functor from Sets^ k to Sets, as desired.

[2] We only give the types of the functions defined in this module and elide their definitions. The definitions are all of the same form and are given by induction on the lengths of their input vectors.

[3] We use an ellipsis "..." in the definition of Cat^k to indicate that the record has additional fields in the code, but that we are not showing them here. We use this notation throughout this thesis.

Morphisms of set environments are also defined as a record type:

```
record SetEnvMorph (ρ ρ' : SetEnv) : Set₁ where
  constructor SetEnvM[_,_]
  field
    eqTC : ρ ≡TC ρ'
    fv : ∀ {k : ℕ} → (φ : FVar k) → NaturalTransformation (SetEnv.fv ρ φ) (SetEnv.fv ρ' φ)
```

To define a morphism of set environments from ρ to ρ', we need a proof eqTC that the tc fields of ρ and ρ' are equal. This proof of equality is defined in terms of the function _ ≡TC_:

```
_≡TC_ : ∀ (ρ ρ' : SetEnv) → Set₁
ρ ≡TC ρ' = (λ {k} → SetEnv.tc ρ {k}) ≡ (λ {k} → SetEnv.tc ρ' {k})
```

We might expect the definition of _ ≡TC_ to simply be SetEnv.tc ρ ≡ SetEnv.tc ρ', but we actually must use the expanded version because of the implicit argument k in the type of the tc field. Normally in functional programming languages, the functions λx → f x and f are considered identical. This property is called *eta-equality*, and, indeed, eta-equality holds in Agda for functions and their *explicit* arguments. However, eta-equality does not hold for implicit arguments because if we just write f for some function f with implicit arguments, Agda will try to infer the implicit arguments. For this reason, we must write the fully expanded λ{k} → SetEnv.tc ρ {k} in the definition of _ ≡TC_. The other field fv maps each functorial variable of arity k to a natural transformation from SetEnv.fv $\rho\,\varphi$ to SetEnv.fv $\rho'\,\varphi$. The identity natural transformation associated to a type constructor variable ψ can be defined by using the equality proof eqTC to deduce that SetEnv.tc $\rho\,\psi$ ≡ SetEnv.tc $\rho'\,\psi$. Composition of morphisms of set environments is defined by:

```
_∘SetEnv_ : ∀ {ρ ρ' ρ''} → SetEnvMorph ρ' ρ'' → SetEnvMorph ρ ρ' → SetEnvMorph ρ ρ''
g ∘SetEnv f = record { eqTC = ≡.trans (SetEnvMorph.eqTC f) (SetEnvMorph.eqTC g)
                     ; fv = λ {k} φ → (SetEnvMorph.fv g φ) ∘v (SetEnvMorph.fv f φ) }
```

To define the composite morphism g ∘ SetEnv f from ρ to ρ'', we compose the equality proofs using transitivity of equality (≡.trans). To compose the natural transformations associated with each functorial variable φ, we use the composition function for natural transformations from the agda-categories library:

```
_∘v_ : ∀ {o ℓ e o' ℓ' e' : Level} {C : Category o ℓ e} {D : Category o' ℓ' e'}
      → {F G H : Functor C D}
      → NaturalTransformation G H → NaturalTransformation F G
      → NaturalTransformation F H
```

The function _∘v_ implements vertical composition of natural transformations as it was defined in Section 3.1.5. An identity morphism of set environments is defined by:

```
idSetEnv : ∀ {ρ : SetEnv} → SetEnvMorph ρ ρ
idSetEnv = record { eqTC = ≡.refl ; fv = λ _ → idnat }
```

The equality proof for the eqTC field is given by the reflexivity proof ≡.refl, and the fv field sends every functorial variable to the appropriate identity natural transformation, which is defined in the agda-categories library as:

```
idnat : ∀ {F : Functor C D} → NaturalTransformation F F
```

Given these definitions, we can define the category SetEnvCat of set environments in Agda:

```
SetEnvCat : Category (lsuc lzero) (lsuc lzero) (lsuc lzero)
SetEnvCat = record
  { Obj = SetEnv
  ; _⇒_ = SetEnvMorph
  ; _≈_ = λ f g → ∀ {k : ℕ} {φ : FVar k}
                → ∀ {Xs : Vec Set k}
                → NaturalTransformation.η (SetEnvMorph.fv f φ) Xs
                  ≈ NaturalTransformation.η (SetEnvMorph.fv g φ) Xs
  ; id = idSetEnv
  ; _∘_ = _∘SetEnv_
  ...
  }
  where open Category (Sets) using (_≈_)
```

An object of SetEnvCat is an element of SetEnv, morphisms are given by SetEnvMorph, and identity and composition are given by idSetEnv and _∘SetEnv_, respectively. Two morphisms f and g are considered equivalent, written $\mathsf{f} \approx \mathsf{g}$, if, for every functorial variable φ of arity k and vector of sets Xs, the components

$$\mathsf{NaturalTransformation}.\eta\ (\mathsf{SetEnvMorph.fv}\,\mathsf{f}\,\varphi)\,\mathsf{Xs}$$

and

$$\mathsf{NaturalTransformation}.\eta\ (\mathsf{SetEnvMorph.fv}\,\mathsf{g}\,\varphi)\,\mathsf{Xs}$$

are equivalent in Sets. The remaining fields for associativity, etc., are straightforward to define, since composition and equality of morphisms are defined componentwise, and the components of the natural transformations here are morphisms in Sets.

The definition of environment extension in Agda is a direct translation of Definition 4.1.2:

```
extendfv : ∀ {k} → SetEnvFV
          → FVar k → Functor (Sets^ k ) (Sets )
          → SetEnvFV
extendfv fv (φ ^F k) F (ψ ^F j) with eqNat k j | φ ≟ ψ
... | yes ≡.refl | yes ≡.refl = F
... | _ | _ = fv (ψ ^F j)

_[_:fv=_] : ∀ {k : ℕ} → SetEnv → FVar k → Functor (Sets^ k) Sets → SetEnv
ρ [ φ :fv= F ] = record { tc = SetEnv.tc ρ ; fv = extendfv (SetEnv.fv ρ) φ F }
```

The tc field of the environment ρ is unaffected, and the fv field is defined in terms of extendfv. Recall that SetEnvFV is a type synonym for:

$$\forall\{k : \mathbb{N}\} \to \mathsf{FVar}\,\mathsf{k} \to \mathsf{Functor}\,(\mathsf{Sets}\hat{}\ \mathsf{k})\,\mathsf{Sets}$$

The extendfv function is defined by case analysis. It says that, when applied to ψ ^F j, the extension of fv with φ ^F k and F gives F if φ ^F k and ψ ^F j are equal and fv (ψ ^F j) otherwise. We define the environment extension function for vectors of variables as:

```
_[_:fvs=_] : ∀ {n k : ℕ} → SetEnv → Vec (FVar k) n → Vec (Functor (Sets^ k ) (Sets )) n → SetEnv
ρ [ Vec.[] :fvs= Vec.[] ] = ρ
ρ [ φ :: φs :fvs= F :: Fs ] = record { tc = SetEnv.tc ρ ; fv = extendfv (SetEnv.fv (ρ [ φs :fvs= Fs ])) φ F }
```

The base case for an empty vector leaves the environment unchanged. The inductive case is defined by recursively extending ρ with φs and Fs and then extending the functorial part of the recursive result with φ and F. The tc field is not modified, since no type constructor variables are involved here. Note that we could use the following, slightly more concise definition:

```
_[_:fvs=_]' : ∀ {n k : ℕ} → SetEnv → Vec (FVar k) n → Vec (Functor (Sets^ k ) (Sets )) n → SetEnv
ρ [ Vec.[] :fvs= Vec.[] ]' = ρ
ρ [ φ :: φs :fvs= F :: Fs ]' = (ρ [ φ :fv= F ]) [ φs :fvs= Fs ]'
```

But with this version, it requires extra steps to show that the tc field of the resulting environment is unchanged, whereas this fact is immediate using the former definition. For this reason, we use the former definition.

Environment extension for morphisms of environments is also defined by case analysis on variable equality, and its Agda definition is a direction translation of Definition 4.1.3:

```
extendfv-morph : ∀ {k} (φ : FVar k)
                    {F G : Functor (Sets^ k) Sets}
  → (ρ ρ' : SetEnv)
  → SetEnvMorph ρ ρ'
  → NaturalTransformation F G
  → SetEnvMorph (ρ [ φ :fv= F ])
                    (ρ' [ φ :fv= G ])
extendfv-morph (φ ^F k) {F} {G} ρ ρ' f η = record { eqTC = SetEnvMorph.eqTC f ; fv = fvmap }
  where fvmap : ∀ {j} (ψ : FVar j)
                  → NaturalTransformation (SetEnv.fv (ρ [ (φ ^F k) :fv= F ]) ψ)
                                          (SetEnv.fv (ρ' [ (φ ^F k) :fv= G ]) ψ)
        fvmap (ψ ^F j) with eqNat k j | φ ≟ ψ
        ... | yes ≡.refl | yes ≡.refl = η
        ... | yes ≡.refl | no _ = SetEnvMorph.fv f (ψ ^F j)
        ... | no _ | _ = SetEnvMorph.fv f (ψ ^F j)
```

The equality proof for the eqTC field is just given by the tc field of f, since the definition of $\rho[\varphi : \mathsf{fv} = \mathsf{F}]$ leaves the tc field unchanged. The fv field is given by fvmap, which sends a variable (ψ ^F j) to η if (φ ^F k) and (ψ ^F j) are equal and sends (ψ ^F j) to SetEnvMorph.fv f (ψ ^F j) otherwise. The analogous function for vectors of variables is defined by:

```
extendfv-morph-vec : ∀ {k n } (φs : Vec (FVar k) n)
  (Fs Gs : Vec (Functor (Sets^ k) Sets) n)
  → (ρ ρ' : SetEnv)
  → SetEnvMorph ρ ρ'
  → foreach2 (NaturalTransformation) Fs Gs
  → SetEnvMorph (ρ [ φs :fvs= Fs ])
                (ρ' [ φs :fvs= Gs ])
extendfv-morph-vec [] [] [] ρ ρ' f bigtt = f
extendfv-morph-vec (φ :: φs) (F :: Fs) (G :: Gs) ρ ρ' f (η , ηs) =
  record { eqTC = SetEnvMorph.eqTC f
         ; fv = SetEnvMorph.fv
                (extendfv-morph φ (ρ [ φs :fvs= Fs ]) (ρ' [ φs :fvs= Gs ])
                  (extendfv-morph-vec φs Fs Gs ρ ρ' f ηs) η) }
```

This function is defined by pattern matching on the the input vectors. The collection of natural transformations ηs must be given in terms of foreach2 rather than Vec to ensure that each natural transformation has the correct type, i.e., that η_i is a natural transformation from Fs_i to Gs_i. Indeed, we must use foreach2 here because the Vec type does not allow us to specify that each element in the vector should have a different type.

4.3 Interpreting Types as Sets

In this section, we present the set semantics of $\mathcal{N}$'s types as it appears in [5]. It should be noted at this point that the set semantics of types is actually defined mutually with the relation semantics of types. Specifically, the relational interpretation depends on the set interpretation (in many places), and the set interpretation of Nat types in Definition 4.3.1 depends on the relational interpretation of Nat types. We show where this mutual dependence appears in this section, but we skip formalizing it in Agda until Section 8.2 after we have defined the relation semantics of types in Chapter 7.

Given a non-functorial context Γ, a functorial context Φ, and a typing judgment $\Gamma; \Phi \vdash F$, we define set interpretation of $\Gamma; \Phi \vdash F$ as a functor from the category of set environments to the category of sets.

Definition 4.3.1. Given a typing judgment $\Gamma; \Phi \vdash F$, we define its set interpretation $[\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}$ as a functor from the category of set environments to the category of sets. The action of the set interpretation on objects is defined (in non-Agda syntax) by:

$$
\begin{aligned}
[\![\Gamma; \Phi \vdash \mathbb{0}]\!]^{\mathsf{Set}}\rho &= 0 \\
[\![\Gamma; \Phi \vdash \mathbb{1}]\!]^{\mathsf{Set}}\rho &= 1 \\
[\![\Gamma; \emptyset \vdash \mathsf{Nat}^{\overline{\alpha}}\, F\, G]\!]^{\mathsf{Set}}\rho &= \{\eta : \lambda\overline{A}.\, [\![\Gamma; \overline{\alpha} \vdash F]\!]^{\mathsf{Set}}\rho[\overline{\alpha := A}] \Rightarrow \lambda\overline{A}.\, [\![\Gamma; \overline{\alpha} \vdash G]\!]^{\mathsf{Set}}\rho[\overline{\alpha := A}] \\
&\qquad\qquad \mid \forall\overline{A}, \overline{B} : \mathsf{Set}.\forall\overline{R : \mathsf{Rel}(A, B)}. \\
&\qquad\qquad\quad (\eta_{\overline{A}}, \eta_{\overline{B}}) : [\![\Gamma; \overline{\alpha} \vdash F]\!]^{\mathsf{Rel}}\mathsf{Eq}_{\rho}[\overline{\alpha := R}] \to [\![\Gamma; \overline{\alpha} \vdash G]\!]^{\mathsf{Rel}}\mathsf{Eq}_{\rho}[\overline{\alpha := R}]\} \\
[\![\Gamma; \Phi \vdash \varphi\overline{F}]\!]^{\mathsf{Set}}\rho &= (\rho\varphi)\, \overline{[\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}\rho} \\
[\![\Gamma; \Phi \vdash F + G]\!]^{\mathsf{Set}}\rho &= [\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}\rho + [\![\Gamma; \Phi \vdash G]\!]^{\mathsf{Set}}\rho \\
[\![\Gamma; \Phi \vdash F \times G]\!]^{\mathsf{Set}}\rho &= [\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}\rho \times [\![\Gamma; \Phi \vdash G]\!]^{\mathsf{Set}}\rho \\
[\![\Gamma; \Phi \vdash (\mu\varphi.\lambda\overline{\alpha}.H)\overline{G}]\!]^{\mathsf{Set}}\rho &= (\mu T^{\mathsf{Set}}_{H,\rho})\overline{[\![\Gamma; \Phi \vdash G]\!]^{\mathsf{Set}}\rho} \\
\text{where } T^{\mathsf{Set}}_{H,\rho}\, F &= \lambda\overline{A}.[\![\Gamma; \varphi, \overline{\alpha} \vdash H]\!]^{\mathsf{Set}}\rho[\varphi := F][\overline{\alpha := A}] \\
\text{and } T^{\mathsf{Set}}_{H,\rho}\, \eta &= \lambda\overline{A}.[\![\Gamma; \varphi, \overline{\alpha} \vdash H]\!]^{\mathsf{Set}} id_{\rho}[\varphi := \eta][\overline{\alpha := id_A}]
\end{aligned}
$$

The types $\mathbb{0}$ and $\mathbb{1}$ are interpreted as the empty set and singleton set, respectively. The Nat type is interpreted as the set of natural transformations from

$$\lambda\overline{A}.\, [\![\Gamma; \overline{\alpha} \vdash F]\!]^{\mathsf{Set}}\rho[\overline{\alpha := A}]$$

to

$$\lambda\overline{A}.\, [\![\Gamma; \overline{\alpha} \vdash G]\!]^{\mathsf{Set}}\rho[\overline{\alpha := A}]$$

such that, for every vector of relations $\overline{R : \mathsf{Rel}(A, B)}$, the components of η at $\overline{A}$ and $\overline{B}$ give a morphism of relations from $[\![\Gamma; \overline{\alpha} \vdash F]\!]^{\mathsf{Rel}}\mathsf{Eq}_{\rho}[\overline{\alpha := R}]$ to $[\![\Gamma; \overline{\alpha} \vdash G]\!]^{\mathsf{Rel}}\mathsf{Eq}_{\rho}[\overline{\alpha := R}]$. A morphism of relations from $R : \mathsf{Rel}(A, B)$ to $S : \mathsf{Rel}(C, D)$ is a pair of functions A morphism of relations from $R : \mathsf{Rel}(A, B)$ to $S : \mathsf{Rel}(C, D)$ is a pair of functions $(f1 : A \to C, f2 : B \to D)$ such that, for every pair (x, y) related in R, the pair $(f1\, x, f2\, y)$ is related in S. In other words, $f1$ and $f2$ must preserve related elements. Therefore, the requirement for Nat types expresses that $\eta_{\overline{A}}$ and $\eta_{\overline{B}}$ must preserve related elements.

This restriction on the set interpretation of Nat types is required to ensure that the Identity Extension Lemma holds for $\mathcal{N}$. The set interpretation of type application is given by applying the functor $\rho\varphi$ to the vector of sets $\overline{[\![\Gamma;\Phi\vdash F]\!]^{\mathsf{Set}}\rho}$ given by interpreting the the arguments of φ. The set interpretation of a sum type is the sum (disjoint union) $\{inl\,x \,|\, x \in [\![\Gamma;\Phi\vdash F]\!]^{\mathsf{Set}}\rho\} \cup \{inr\,x \,|\, x \in [\![\Gamma;\Phi\vdash G]\!]^{\mathsf{Set}}\rho\}$. The set interpretation of a product type is the cartesian product $\{(x,y) \,|\, x \in [\![\Gamma;\Phi\vdash F]\!]^{\mathsf{Set}}\rho, y \in [\![\Gamma;\Phi\vdash G]\!]^{\mathsf{Set}}\rho\}$.

To interpret a μ-type, we first interpret its body $\Gamma;\varphi,\overline{\alpha}\vdash H$ and use environment extension to get a higher-order functor $T^{\mathsf{Set}}_{H,\rho} : [\mathsf{Sets}^k,\mathsf{Sets}] \to [\mathsf{Sets}^k,\mathsf{Sets}]$. That is, $T^{\mathsf{Set}}_{H,\rho}$ is a functor whose action on objects takes a functor $F : \mathsf{Sets}^k \to \mathsf{Sets}$ and returns the functor:

$$\lambda\overline{A}.[\![\Gamma;\varphi,\overline{\alpha}\vdash H]\!]^{\mathsf{Set}}\rho[\varphi := F][\overline{\alpha := A}]$$

The action of $T^{\mathsf{Set}}_{H,\rho}$ on morphisms takes a natural transformation $\eta : F \Rightarrow G$ and returns the natural transformation:

$$\lambda\overline{A}.[\![\Gamma;\varphi,\overline{\alpha}\vdash H]\!]^{\mathsf{Set}}id_\rho[\varphi := \eta][\overline{\alpha := id_A}]$$

of type

$$\lambda\overline{A}.[\![\Gamma;\varphi,\overline{\alpha}\vdash H]\!]^{\mathsf{Set}}\rho[\varphi := F][\overline{\alpha := A}] \Rightarrow \lambda\overline{A}.[\![\Gamma;\varphi,\overline{\alpha}\vdash H]\!]^{\mathsf{Set}}\rho[\varphi := G][\overline{\alpha := A}]$$

Given this higher-order functor $T^{\mathsf{Set}}_{H,\rho}$, we can take its fixpoint $\mu T^{\mathsf{Set}}_{H,\rho}$. Intuitively, the fixpoint of a higher-order functor $T : [\mathsf{Sets}^k,\mathsf{Sets}] \to [\mathsf{Sets}^k,\mathsf{Sets}]$ can be thought of as given by a repeated application of T, starting with the constant functor $K_0 : \mathsf{Sets}^k \to \mathsf{Sets}$ that maps every object to the empty set 0 and maps every morphism to id_0:

$$K_0 \longrightarrow T\,K_0 \longrightarrow T\,(T\,K_0) \longrightarrow \ldots \longrightarrow \mu T \cong T\,(\mu T)$$

The fixpoint μT is obtained when this repeated application converges, meaning that applying T to μT gives a functor that is isomorphic to μT itself. As an example, let T be the underlying (higher-order) functor for the list type, i.e., $T\,F\,X = 1 + X \times F\,X$. Then $\mu T = \mathit{List}$ is the interpretation of the list type. If we apply T to List, we get

$$T\,(\mathit{List})X = 1 + X \times \mathit{List}\,X$$

Thus, an element of $T\,(\mathit{List})X$ is either the single element of 1 (representing the empty list) or an element of X paired with a list of elements of X. This is precisely how the elements of $\mathit{List}\,X$ are defined, and the isomorphism between $T(\mu T)$ and μT is a consequence of the fixpoint computation converging.

Such fixpoints do not always exist for arbitrary functors, but the category of sets and the functors interpreting our types have sufficient structure for these fixpoints to exist in our setting. We can also take fixpoints of higher-order natural transformations [6]. That is, given higher-order functors $T, T' : [\mathsf{Sets}^k, \mathsf{Sets}] \to [\mathsf{Sets}^k, \mathsf{Sets}]$ and a higher-order natural transformation $\eta : T \Rightarrow T'$, there exists a natural transformation $\mu\eta : \mu T \Rightarrow \mu T'$.

After taking the fixpoint of $T^{\mathsf{Set}}_{H,\rho}$ to get $\mu T^{\mathsf{Set}}_{H,\rho} : \mathsf{Sets}^k \to \mathsf{Sets}$, to interpret the type $(\mu\varphi^k.\lambda\overline{\alpha^0}.\,F)\overline{G}$ we apply $\mu T^{\mathsf{Set}}_{H,\rho}$ to the vector of sets $\overline{[\![\Gamma;\Phi \vdash G]\!]^{\mathsf{Set}}\rho}$ given by interpreting the arguments $\overline{G}$.

The functor $T^{\mathsf{Set}}_{H,\rho} : [\mathsf{Sets}^k, \mathsf{Sets}] \to [\mathsf{Sets}^k, \mathsf{Sets}]$ can be extended to a functor $T^{\mathsf{Set}}_{H} : \mathsf{SetEnvCat} \to [[\mathsf{Sets}^k, \mathsf{Sets}], [\mathsf{Sets}^k, \mathsf{Sets}]]$ whose action on an object ρ gives $T^{\mathsf{Set}}_{H,\rho}$. The action of T^{Set}_{H} on a morphism $f : \rho \to \rho'$ gives the higher-order natural transformation $T^{\mathsf{Set}}_{H,f} : T^{\mathsf{Set}}_{H,\rho} \Rightarrow T^{\mathsf{Set}}_{H,\rho'}$ whose action on $F : \mathsf{Set}^k \to \mathsf{Set}$ is the natural transformation $T^{\mathsf{Set}}_{H,f}\,F : T^{\mathsf{Set}}_{H,\rho}\,F \Rightarrow T^{\mathsf{Set}}_{H,\rho'}\,F$ whose component at $\overline{A}$ is $(T^{\mathsf{Set}}_{H,f}\,F)_{\overline{A}} = [\![\Gamma;\varphi,\overline{\alpha} \vdash H]\!]^{\mathsf{Set}} f[\varphi := id_F][\overline{\alpha := id_A}]$.

4.4 Interpreting Types as Sets in Agda

The set interpretation of types is defined in Agda as the function SetSem:

```
SetSem : ∀ {Γ : TCCtx} → {Φ : FunCtx} → {F : TypeExpr}
             → Γ ≀ Φ ⊢ F
             → Functor SetEnvCat Sets
```

The set interpretation function SetSem takes a type F and a typing judgment Γ ≀ Φ ⊢ F and returns a functor from the category of set environments to the category of sets. It is defined by induction on the structure of the type formation rules, just as in Definition 4.3.1. We also have a function for interpreting vectors of type expressions and their typing judgments:

```
SetSemVec : ∀ {k : ℕ} {Γ : TCCtx} {Φ : FunCtx}
               → {Fs : Vec TypeExpr k}
               → foreach (λ F → Γ ≀ Φ ⊢ F) Fs
               → Functor SetEnvCat (Sets^ k)
```

SetSem and SetSemVec are defined by mutual induction, where SetSemVec applies SetSem to each type (and its typing judgment) in a vector of types.

The set interpretations for 𝟘-I and 𝟙-I are defined as constant functors, using the ConstF construction from the agda-categories library:

```
SetSem 𝟘-I = ConstF ⊥
SetSem 𝟙-I = ConstF ⊤
```

```
ConstF : ∀ {o ℓ e o' ℓ' e' : Level}
           → {C : Category o ℓ e} {D : Category o' ℓ' e'}
           → (d : Category.Obj D) → Functor C D
```

As described in Section 3.1.4, the constant functor ConstF d sends every object to d and sends every morphism to the identity morphism on d.

For the remaining types, we will go slightly out of order in order to present the simplest interpretations first. The set interpretation for a product type is given by:

```
SetSem (×-I ⊢F ⊢G) = SetProd ∘F (SetSem ⊢F ※ SetSem ⊢G)
```

SetProd is a functor from the product category Product Sets Sets to Sets. The function _∘F_ composes two functors where the source of the first matches the target of the second. It is defined in the agda-categories library by:

```
_∘F_ : ∀ {o ℓ e o' ℓ' e' o'' ℓ'' e''}
         → {C : Category o ℓ e} {D : Category o' ℓ' e'} {E : Category o'' ℓ'' e''}
         → Functor D E → Functor C D → Functor C E
```

The function _※_ from the agda-categories library

```
_※_ : ∀ {o ℓ e o' ℓ' e' o'' ℓ'' e''}
      → {C : Category o ℓ e}
      → {D₁ : Category o' ℓ' e'}
      → {D₂ : Category o'' ℓ'' e''}
      → (F : Functor C D₁) → (G : Functor C D₂) → Functor C (Product D₁ D₂)
```

takes two functors F and G with the same source C and potentially different targets D_1 and D_2 and returns a functor from C to Product D_1 D_2. The action of F※G on an object X is given by the pair of objects (Functor.F_0 F X , Functor.F_0 G X). The action on a morphism f is given by the pair of morphisms (Functor.F_1 F f , Functor.F_1 G f).

SetProd sends each pair of sets to their cartesian product and each pair of functions (f1, f2) to the function λ{(x, y) → (f1 x, f2 y)}:

```
SetProd : Functor (Product Sets Sets) Sets
SetProd = record
  { F₀ = λ { (A , B) → A ×' B }
  ; F₁ = λ { (f1 , f2) (x , y) → (f1 x , f2 y) }
  ; identity = ≡.refl
  ; homomorphism = ≡.refl
  ; F-resp-≈ = λ { (f1≈g1 , f2≈g2) → ≡.cong₂ _,_ f1≈g1 f2≈g2 }
  }
```

The identity and homomorphism fields are trivial for SetProd, and the F-resp-≈ field is defined using the standard library function

```
cong₂ : ∀ {a} {A B C : Set a} (f : A → B → C) {x y u v} → x ≡ y → u ≡ v → f x u ≡ f y v
cong₂ f ≡.refl ≡.refl = ≡.refl
```

to deduce that the pairs of morphisms (f1, f2) and (g1, g2) are equivalent from the facts that the components are equivalent. Given the definitions of _※_and SetProd, we can see that the set interpretation of a product type is computed by first interpreting ⊢F and ⊢G and then taking the cartesian product of the resulting pair of sets or morphisms, as expected.

The set interpretation of a sum type is given by:

```
SetSem (+-I ⊢F ⊢G) = SetSum ∘F (SetSem ⊢F ※ SetSem ⊢G)
```

Analogously to SetProd, SetSum sends each pair of sets to their disjoint union, and it sends each pair of functions (f, g) to the function λ{inj_1 x → inj_1 (f x); inj_2 y → inj_2 (g y)} that is defined by pattern matching:

```
SetSum : Functor (Product Sets Sets) Sets
SetSum = record
```

```
{ F₀ = λ { (A , B) → A ⊎ B }
; F₁ = λ { {A , B} (f , g) (inj₁ x) → inj₁ (f x)
         ; {A , B} (f , g) (inj₂ y) → inj₂ (g y) }
; identity = λ { {A , B} {inj₁ x} → ≡.refl
               ; {A , B} {inj₂ y} → ≡.refl }
; homomorphism = λ { {A , A'} {B , B'} {C , C'} {f , f'} {g , g'} {inj₁ x} → ≡.refl
                   ; {A , A'} {B , B'} {C , C'} {f , f'} {g , g'} {inj₂ y} → ≡.refl }
; F-resp-≈ = λ { (f≈h , g≈i) {inj₁ x} → ≡.cong inj₁ f≈h
               ; (f≈h , g≈i) {inj₂ y} → ≡.cong inj₂ g≈i }
}
```

The functor laws are proved by pattern matching and the proofs for identity and homomorphism are trivial. The proof for F-resp-≈ is defined using ≡.cong to deduce that composing a morphism with inj_1 or inj_2 (as the action of SetSum on morphisms does) preserves equality of morphisms. Given the definitions of _※_and SetSum, we can see that the set interpretation of a sum type is computed by first interpreting ⊢F and ⊢G and then taking the disjoint union of the resulting pair of sets or morphisms, as expected.

The set interpretations of type applications are defined by:

```
SetSem (AppT-I {φ = φ} Γ∋φ Gs ⊢Gs) = eval ∘F (VarSem-TC φ ※ SetSemVec ⊢Gs)
SetSem (AppF-I {φ = φ} Φ∋φ Gs ⊢Gs) = eval ∘F (VarSem-FV φ ※ SetSemVec ⊢Gs)
```

These set interpretations are defined in terms of the functions

```
VarSem-TC : ∀ {k} (φ : TCVar k) → Functor SetEnvCat ([Sets^ k ,Sets])
```

and

```
VarSem-FV : ∀ {k} (φ : FVar k) → Functor SetEnvCat ([Sets^ k ,Sets])
```

These functions each take a variable of arity k and return a functor from the category of set environments to the functor category [Sets^ k ,Sets] whose objects are functors from Sets^ k to Sets. The action of VarSem-TC on objects takes a set environment and returns the k-ary functor of sets given by applying ρ to φ. The action on morphisms takes a morphism f of set environments and returns the natural transformation given by applying f to φ. VarSem-FV is defined analogously. The syntax [Sets^ k ,Sets] comes from a function we define in terms of the functor category construction Functors defined in the agda-categories library:

```
[Sets^_,Sets] : ℕ → Category (lsuc lzero) (lsuc lzero) (lsuc lzero)
[Sets^_,Sets] k = Functors (Sets^ k) Sets
```

Functors takes two categories C and D as arguments. An object of Functors C D is a functor from C to D. A morphism in Functors C D is a natural transformation, and composition is given by the vertical composition function _∘v_. We sometimes use the following bracket syntax for Functors:

```
[[_,_]] : ∀ {o ℓ e o' ℓ' e'} → Category o ℓ e → Category o' ℓ' e' → Category _ _ _
[[ C , D ]] = Functors C D
```

The definitions of VarSem-TC and VarSem-FV are very similar, so we only describe the latter. Given a variable φ, the action of VarSem-FV φ on an object ρ is given by the functor SetEnv.fv ρ φ. The action of VarSem-FV φ on a morphism f : SetEnvMorph ρ ρ' is given by the natural transformation SetEnvMorph.fv f φ.

The other function needed to define the set interpretation of type application is the eval function from the standard library:

```
eval : ∀ {o ℓ e o' ℓ' e'}
      → {C : Category o ℓ e} {D : Category o' ℓ' e'}
      → Functor (Product (Functors C D) C) D
```

The eval function takes two categories C and D and returns a functor from the product category Product (Functors C D) C to the category D. The action of eval on a pair of objects (F, X) is defined as the application of F to X, i.e., $\mathsf{Functor.F}_0\,\mathsf{F}\,\mathsf{X}$. The action on a pair of morphisms (η, f) from (F, X) to (G, Y) is defined by

$$\mathsf{NaturalTransformation}.\eta\;\eta\;\mathsf{Y} \circ \mathsf{Functor.F}_1\;\mathsf{F}\;\mathsf{f}$$

which is equivalent to

$$\mathsf{Functor.F}_1\;\mathsf{G}\;\mathsf{f} \circ \mathsf{NaturalTransformation}.\eta\;\eta\;\mathsf{X}$$

by naturality of η:

$$\begin{tikzcd}
\mathsf{Functor.F_0\,F\,X} \arrow[r, "\mathsf{NaturalTransformation.}\eta\ \eta\ \mathsf{X}"] \arrow[d, "\mathsf{Functor.F_1\,F\,f}"] & \mathsf{Functor.F_0\,G\,X} \arrow[d, "\mathsf{Functor.F_1\,G\,f}"] \\
\mathsf{Functor.F_0\,F\,Y} \arrow[r, "\mathsf{NaturalTransformation.}\eta\ \eta\ \mathsf{Y}"] & \mathsf{Functor.F_0\,G\,Y}
\end{tikzcd}$$

Given this definition of eval, we can see that the set interpretation of type application is a functor whose action on objects is given by interpreting the variable φ as a functor from Sets^ k to Sets and interpreting ⊢Gs as a vector of sets, and then applying the functor given by φ to this vector of sets. The action on morphisms is given by interpreting the variable φ as a natural transformation η and interpreting ⊢Gs as a morphism f in Sets^ k and then following one of the paths through the naturality square associated with η and f.

The set interpretation of a Nat type is defined in Definition 4.3.1 as a set of natural transformations that satisfy a condition about preserving related elements. For now, we define the set interpretation of a Nat type simply as a set of natural transformations, and we will include the relation-preserving condition when we define the relation semantics of types. Note that in Definition 4.3.1, the value of ρ on functorial variables has no bearing on the set interpretation of a Nat type since the only functorial variables that are used are those in $\overline{\alpha}$, and these variables are all overwritten in the environment by an environment extension. Thus we can reassign the values of functorial variables in an environment prior to interpreting a Nat type, and the interpretation will be unchanged.

The set interpretation of a Nat type (Nat-I {αs = αs} ⊢F ⊢G) is defined by:

```
SetSem (Nat-I {αs = αs} ⊢F ⊢G) = NatTypeFunctor αs ⊢F ⊢G
```

The functor NatTypeFunctor is defined as:

```
module NatType-setsem
  {k : ℕ} {Γ : TCCtx} {F G : TypeExpr} (αs : Vec (FVar 0) k)
  (⊢F : Γ ≀ (∅ ,++ αs) ⊢ F) (⊢G : Γ ≀ (∅ ,++ αs) ⊢ G) where

  NatTypeSem-map : ∀ {ρ ρ' : SetEnv} (f : SetEnvMorph ρ ρ')
                   → NatTypeSem αs (NatEnv ρ ) ⊢F ⊢G
                   → NatTypeSem αs (NatEnv ρ') ⊢F ⊢G
  NatTypeSem-map SetEnvM[ ≡.refl , _ ] = idf
```

```
NatTypeFunctor : Functor SetEnvCat Sets
NatTypeFunctor = record
  { F₀ = λ ρ → NatTypeSem αs (NatEnv ρ) ⊢F ⊢G
  ; F₁ = λ f → NatTypeSem-map f
  ...
  }
```

The action on an object ρ is given by NatTypeSem αs (NatEnv ρ) ⊢F⊢G. Here NatEnv ρ is defined by:

```
NatEnv : SetEnv → SetEnv
NatEnv SetEnv[ tc , fv ] = SetEnv[ tc , (λ _ → ConstF ⊤) ]
```
(8)

That is, the set environment NatEnv ρ is the same as ρ on type constructor variables, but it sends every functorial variable to ConstF ⊤[4] . This makes it easy to show the set interpretation of Nat types is a functor, because whenever we have a morphism f from ρ to ρ', we are given a proof that ρ and ρ' have equal tc fields. If we apply NatEnv to both environments to get NatEnv ρ and NatEnv ρ', then these environments are also equal on their fv fields. Thus, if we have a morphism from ρ to ρ', then NatEnv ρ and NatEnv ρ' are identical. This allows us to define the action of NatTypeFunctor on morphisms as an identity morphism, as shown in the definition of NatTypeSem-map above[5].

The definition of NatTypeSem αs (NatEnv ρ) ⊢F⊢G, itself is given by the record type:

```
{-# NO_POSITIVITY_CHECK #-}
{-# NO_UNIVERSE_CHECK #-}
record NatTypeSem {k : ℕ} {Γ : TCCtx} {F G : TypeExpr} (αs : Vec (FVar 0) k) (ρ : SetEnv)
  (⊢F : Γ ≀ (∅ ,++ αs) ⊢ F) (⊢G : Γ ≀ (∅ ,++ αs) ⊢ G)
    : Set where
  constructor NatT2[_]

  field
    nat : NaturalTransformation (extendSetSem-αs αs ρ (SetSem ⊢F)) (extendSetSem-αs αs ρ (SetSem ⊢G))
```

This record type consists of a single field nat that is a natural transformation between two functors of type [Sets^ k ,Sets]. The function

[4] There is nothing special about the functor ConstF ⊤. We just want to send all functorial variables to the same functor, and we chose ConstF ⊤ for this purpose.

[5] In this part of the implementation, the identity function in Agda is renamed to idf to avoid confusion with the identity morphisms of categories.

```
extendSetSem-αs : ∀ {k} → (αs : Vec (FVar 0) k)
  → (ρ : SetEnv)
  → Functor SetEnvCat Sets
  → Functor (Sets^ k) Sets
```

takes a functor from SetEnvCat to Sets and returns a functor from Sets^ k to Sets. It is defined in terms of environment extension. The Agda functor

```
extendSetSem-αs αs ρ (SetSem ⊢F)
```

corresponds to

$$\lambda\overline{A}.\,[\![\Gamma;\overline{\alpha}\vdash F]\!]^{\mathsf{Set}}\rho[\overline{\alpha := A}]$$

from Definition 4.3.1. The universe Agda computes for this record type is Set$_1$ rather than Set because of the way universe levels are computed for the type NaturalTransformation. To make this record type fit into the universe Set, as is required for the set interpretation to be an object of Sets, we can use the NO_UNIVERSE_CHECK flag to make Agda turn off its universe checker for this definition. Generally, turning off the universe checker can lead to inconstitency of Agda's type system, but in this instance it is justified because, as shown in [5], some categorical reasoning (beyond the scope of this thesis) shows that the set of natural transformations given in Definition 4.3.1 is indeed a set. We must also use the NO_POSITIVITY_CHECK flag because the data of NatTypeSem refer to SetSem, which is defined in terms of NatTypeSem. The remaining fields for the functor NatTypeFunctor are straightforward to prove because the action on morphisms is the identity morphism.

The set interpretation of a μ-type μ-I ⊢F Ks⊢Ks is defined by:

```
SetSem (μ-I ⊢F Ks ⊢Ks) = MuSem ⊢F (SetSemVec ⊢Ks)
```

The function MuSem is defined by:

```
MuSem : ∀ {k : ℕ} {Γ : TCCtx} {F : TypeExpr}
            {φ : FVar k} {αs : Vec (FVar 0) k}
          → Γ ≀ (∅ ,++ αs) ,, φ ⊢ F
          → Functor SetEnvCat (Sets^ k) → Functor SetEnvCat Sets
MuSem {k} ⊢F SemKs =
let T : Functor SetEnvCat [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]]
    T = TSet ⊢F


    fixT : Functor SetEnvCat [Sets^ k ,Sets]
    fixT = fixH ∘F T
in eval ∘F (fixT ※ SemKs)
```

Like the set interpretation of type applications, the set interpretation of the μ-type is given in terms of eval. This means the action on an object ρ is computed by first taking the fixpoint fixT of the higher-order functor obtained by applying T to ρ, and then applying this fixpoint to the vector of sets given by applying the functor SemKs interpreting the arguments to ρ. The functor fixH

```
fixH : ∀ {k} → Functor [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]] [Sets^ k ,Sets]
```

takes a higher-order functor and returns its fixpoint, which is a first-order functor, i.e., an object of [Sets^ k ,Sets]. The action of fixH on morphisms takes a higher-order natural transformation from H1 to H2 and returns a natural transformation from the fixpoint of H1 to the fixpoint of H2. We will discuss how fixH is defined in Section 4.4.1. The function TSet

```
TSet : ∀ {k : ℕ} {Γ : TCCtx} {H : TypeExpr}
             {φ : FVar k} {αs : Vec (FVar 0) k}
           → Γ ≀ (∅ ,++ αs) ,, φ ⊢ H
           → Functor (SetEnvCat) ([[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]])
```

takes a typing judgment Γ ≀ (∅ ,++ αs),,φ ⊢ H for the body of a μ-type and returns a functor from the category of set environments to the category [[[Sets^ k ,Sets] , [Sets^ k ,Sets]]] whose objects are higher-order functors and whose morphisms are higher-order natural transformations. TSet⊢H corresponds to the functor T_H^{Set} described in Section 4.3. In this formalization, TSet is defined in terms of another function TSetProd:

```
TSetProd : ∀ {k : ℕ} {Γ : TCCtx} {H : TypeExpr}
               {φ : FVar k} {αs : Vec (FVar 0) k}
  → Γ ≀ (∅ ,++ αs) ,, φ ⊢ H
  → Functor (Product (Product SetEnvCat [Sets^ k ,Sets]) (Sets^ k)) Sets
```

TSetProd takes a typing judgment for the body of a μ-type and returns a functor whose source category is

$$\mathsf{Product}\,(\mathsf{Product}\,\mathsf{SetEnvCat}\,[\mathsf{Sets\hat{}\ k}\ ,\mathsf{Sets}])\,(\mathsf{Sets\hat{}\ k}))\,\mathsf{Sets}$$

Its objects are triples $((\rho, \mathsf{F}), \mathsf{As})$, where ρ : SetEnv, F : Functor (Sets^ k) Sets, and As : Vec Set k. TSetProd is defined in terms of environment extension, and its action on objects takes a triple $((\rho, \mathsf{F}), \mathsf{As})$ and returns the set given by interpreting the typing judgment $\Gamma \wr (\emptyset\ ,\!{+}{+}\ \alpha\mathsf{s}),,\varphi \vdash \mathsf{H}$ in the environment given by extendeding ρ to send φ to F and αs to As, i.e.,

$$(\mathsf{Functor.F_0}\,\mathsf{SetSem} \vdash \mathsf{H})((\rho[\varphi : \mathsf{fv} = \mathsf{F}])[\alpha\mathsf{s} : \mathsf{fvs} = \mathsf{As}])$$

The action of TSetProd ⊢H on morphisms is analogous and takes a triple of morphisms $((\mathsf{f}, \eta), \mathsf{gs})$ to a function given by applying $\mathsf{Functor.F_1}$ SetSem ⊢H to the morphism of set environments given by extending f to send φ to η and αs to gs.

Given this definition of TSetProd, we can use *currying* to define TSet. In the category of sets, we can take a function

$$\mathsf{f} : (\mathsf{A} \times' \mathsf{B}) \to \mathsf{C}$$

whose domain is a product and *curry* it to get a new function:

$$\mathsf{f}' : \mathsf{A} \to (\mathsf{B} \to \mathsf{C})$$

The curried function f′ is defined taking its arguments and feeding them to f as a pair, i.e.,

$$\mathsf{f}'\,\mathsf{x}\,\mathsf{y}\ =\ \mathsf{f}\,(\mathsf{x}, \mathsf{y})$$

Currying can be extended from sets and functions to functors and natural transformations. Currying at the level of functors is defined in the agda-categories library by the function:

```
curry₀ : ∀ {o ℓ e o' ℓ' e' o'' ℓ'' e'' : Level}
        → {C₁ : Category o ℓ e} {C₂ : Category o' ℓ' e'} {D : Category o'' ℓ'' e''}
        → Functor (Product C₁ C₂) D
        → Functor C₁ (Functors C₂ D)
```

Using curry$_0$, we can define TSet as:

```
TSet {k} {Γ} {H} {φ} {αs} ⊢H = curry₀ (curry₀ (TSetProd ⊢H))
```

Note that we have to apply curry$_0$ twice. The first application gives a functor of type

```
  Functor (Product SetEnvCat [Sets^ k ,Sets]) (Functors (Sets^ k) Sets)
= Functor (Product SetEnvCat [Sets^ k ,Sets]) [Sets^ k ,Sets]
```

and the second application gives a functor of type

```
  Functor SetEnvCat (Functors [Sets^ k ,Sets] [Sets^ k ,Sets])
= Functor SetEnvCat [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]]
```

The fact that we define TSet in terms of TSetProd and curry$_0$ is a design decision. We could have defined TSet directly, but this would require manipulating higher-order functors and natural transformations. Since the target category of TSetProd is just the category of sets, it is somewhat easier to define than TSet, since the action of TSetProd on morphisms just gives functions rather than higher-order natural transformations.

4.4.1 Fixpoints of Functors in Agda

In this section we present our Agda formalization of fixpoints of higher-order functors over the category of sets. The goal of this development is to define the function fixH

```
fixH : ∀ {k} → Functor [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]] [Sets^ k ,Sets]
```

that is used to interpret μ-types. The action of fixH on objects is defined by fixH$_0$:

```
module object-fixpoint {k : ℕ} where
  HFunc : Set₁
  HFunc = Functor ([Sets^ k ,Sets]) ([Sets^ k ,Sets])

  {-# NO_POSITIVITY_CHECK #-}
  data HFixObj (H : HFunc) : Vec Set k → Set


  {-# TERMINATING #-}
  HFix-fmap : ∀ (H : HFunc) → {As Bs : Vec Set k} → VecMorph As Bs
              → HFixObj H As → HFixObj H Bs


  {-# TERMINATING #-}
  HFix-id : ∀ (H : HFunc) {As} {xs} → HFix-fmap H {As} {As} (Category.id (Sets^ k )) xs ≡ xs

  {-# TERMINATING #-}
  HFix-homo : ∀ (H : HFunc) {As Bs Cs} → {f : VecMorph As Bs} → {g : VecMorph Bs Cs}
              → ∀ {x} → HFix-fmap H (Category._∘_ (Sets^ k ) g f) x ≡ HFix-fmap H g (HFix-fmap H f x)

  {-# TERMINATING #-}
  HFix-resp : ∀ (H : HFunc) {As Bs} → {fs gs : VecMorph As Bs}
              → (Sets^ k) [ fs ≈ gs ]
              → ∀ {x : HFixObj H As} → HFix-fmap H fs x ≡ HFix-fmap H gs x

  fixH₀ : ∀ (H : HFunc) → Functor (Sets^ k) Sets
  fixH₀ H = record
              { F₀ = HFixObj H
              ; F₁ = HFix-fmap H
              ; identity = HFix-id H
              ; homomorphism = HFix-homo H
              ; F-resp-≈ = HFix-resp H
              }
```

The function fixH$_0$ takes a higher-order functor H and returns the fixpoint of H, which is itself a (first-order) functor. The action of the fixpoint fixH$_0$ H on objects is given by the data type HFixObj H, which takes a vector of sets As and returns a set HFixObj H As:

```
data HFixObj H where
  hin : ∀ {As : Vec Set k}
        → Functor.F₀ (Functor.F₀ H (fixH₀ H)) As
        → HFixObj H As
```

HFixObj is defined by a single constructor hin that, for any vector of types As, takes an argument of type

$$\mathsf{Functor.F}_0\,(\mathsf{Functor.F}_0\,\mathsf{H}\,(\mathsf{fixH}_0\,\mathsf{H}))\,\mathsf{As}$$

and produces an element of HFixObj H As. The type (and name) of hin is motivated by the fact that in a categorical semantics such as that of [5], the elements of a fixpoint $(\mu T)\,X$ can be defined by applying a morphism $in_X : (T\,\mu T)\,X \to (\mu T)\,X$ to the elements of $(T\,\mu T)\,X$, where T is a higher-order functor on sets. The morphisms $in_X : (T\,\mu T)\,X \to (\mu T)\,X$ form a natural transformation $in : (T\,\mu T) \Rightarrow \mu T$.

The action of the fixpoint fixH$_0$ H on morphisms is given by HFix-fmap:

```
HFix-fmap H {As} {Bs} gs (hin {As} x) = hin {!!} -- Goal:  Functor.F₀ (Functor.F₀ H (fixH₀ H)) Bs
```

HFix-hmap is defined by pattern matching on the constructor hin, and the definition must be given using hin, since this is the only constructor for HFixObj. The goal type is

$$\mathsf{Functor.F}_0\,(\mathsf{Funtor.F}_0\,\mathsf{H}\,(\mathsf{fixH}_0\,\mathsf{H}))\,\mathsf{Bs}$$

and we have

$$\mathsf{x} : \mathsf{Functor.F}_0\,(\mathsf{Funtor.F}_0\,\mathsf{H}\,(\mathsf{fixH}_0\,\mathsf{H}))\,\mathsf{As}$$

We complete the definition using the action on morphisms of the functor Funtor.F$_0$ H (fixH$_0$ H) applied to the morphism gs:

```
HFix-fmap H {As} {Bs} gs (hin {As} x) = hin (Functor.F₁ (Functor.F₀ H (fixH₀ H)) gs x)
```

The NO_POSITIVITY_CHECK flag is required for HFixObj because fixH$_0$, which depends on HFixObj, appears in a *non-positive* position in the type of hin, i.e., it is used as the argument to some function. This mutual dependence and non-positivity means that Agda cannot determine on its own that HFixObj is well-defined.

The TERMINATING flag must be used to turn off the termination checker for HFix-fmap because, like HFixObj, it is mutually dependent with fixH$_0$, and Agda cannot determine that this mutual recursion is well-defined. This may seem worrying at first, but for the higher-order functors we are interested in (given by the interpretations of types) these functions will always terminate for an input constructed from a finite number of applications of hin. The other functions HFix-id, etc., are defined similarly to HFix-fmap in terms of pattern matching, so they also require the TERMINATING flag.

We can now define the action of fixH on morphisms as:

```
{-# TERMINATING #-}
fixH₁ : ∀ {k} (H1 H2 : Functor ([Sets^ k ,Sets]) ([Sets^ k ,Sets]))
         → NaturalTransformation H1 H2
         → NaturalTransformation (fixH₀ H1) (fixH₀ H2)
fixH₁ {k} H1 H2 η = record { η = μη ; commute = commutes ; sym-commute = λ f → ≡.sym (commutes f) }
  where μη : (Xs : Vec Set k) → HFixObj H1 Xs → HFixObj H2 Xs                                    (9)
        μη = fixH₁-component η

        commutes : ∀ {Xs Ys : Vec Set k} → (f : VecMorph Xs Ys)
                      → ∀ {x : HFixObj H1 Xs}
                      → μη Ys (Functor.F₁ (fixH₀ H1) f x) ≡ Functor.F₁ (fixH₀ H2) f (μη Xs x)
```

The function fixH$_1$ takes a higher-order natural transformation η and returns a natural transformation between fixpoints. The components of fixH$_1$ η are defined by:

```
{-# TERMINATING #-}
fixH₁-component : ∀ {k} {H1 H2 : Functor ([Sets^ k ,Sets]) ([Sets^ k ,Sets])}
                  → NaturalTransformation H1 H2
                  → (Xs : Vec Set k) → HFixObj H1 Xs → HFixObj H2 Xs
fixH₁-component {k} {H1} {H2} η Xs (hin x) =
  hin (NaturalTransformation.η ημH2 Xs (NaturalTransformation.η H1μη Xs x))
     where open Functor H1 renaming (F₀ to H1₀)
           open Functor H2 renaming (F₀ to H2₀)
           ημH2 : NaturalTransformation (H1₀ (fixH₀ H2)) (H2₀ (fixH₀ H2))
           ημH2 = NaturalTransformation.η η (fixH₀ H2)
           H1μη : NaturalTransformation (H1₀ (fixH₀ H1)) (H1₀ (fixH₀ H2))
           H1μη = Functor.F₁ H1 (fixH₁ H1 H2 η)
```
(10)

To define a component from HFixObj H1 Xs to HFixObj H2 Xs by pattern matching, we have two possible paths to choose from, shown in Figure 4.1 below. The natural transformations μηH2 and H1μη are defined in (10), and the other natural transformations in Figure 4.1 are defined by:

```
ημH1 : NaturalTransformation (H1₀ (fixH₀ H1)) (H2₀ (fixH₀ H1))
ημH1 = NaturalTransformation.η η (fixH₀ H1)
H2μη : NaturalTransformation (H2₀ (fixH₀ H1)) (H2₀ (fixH₀ H2))
H2μη = Functor.F₁ H2 (fixH₁ H1 H2 η)
```

$$\begin{tikzcd}
\mathsf{H1}_0\,(\mathsf{fixH}_0\ \mathsf{H1}) \arrow[r, "\eta\mu\mathsf{H1}"] \arrow[d, "\mathsf{H1}\mu\eta"'] & \mathsf{H2}_0\,(\mathsf{fixH}_0\ \mathsf{H1}) \arrow[d, "\mathsf{H2}\mu\eta"] \\
\mathsf{H1}_0\,(\mathsf{fixH}_0\ \mathsf{H2}) \arrow[r, "\eta\mu\mathsf{H2}"'] & \mathsf{H2}_0\,(\mathsf{fixH}_0\ \mathsf{H2})
\end{tikzcd}$$

Figure 4.1: Naturality square for fixH₁ H1 H2 η in the functor category [Sets^ k ,Sets].

It turns out that both paths of Figure 4.1 are equivalent, by naturality, so we choose the path given by the left and bottom morphisms in Figure 4.1 to define fixH₁-component. We elide the definition of commutes in (9), but commutes is defined by pattern matching on the hin constructor, using naturality (the commute field) of $\eta\mu$H2 and H1$\mu\eta$. Since the definitions of fixH$_1$ and fixH$_1$-component are self-referential, these definitions also require the TERMINATING flag.

We must also prove the functor laws for fixH$_1$. That is, we must define the functions

```
fixH₁-identity : ∀ {k} (H : Functor ([Sets^ k ,Sets]) ([Sets^ k ,Sets]))
                            → [Sets^ k ,Sets] Categories.Category.[
                               fixH₁ H H (Category.id [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]] {H})
                               ≈ Category.id [Sets^ k ,Sets] {fixH₀ H}
                            ]

fixH₁-homomorphism : ∀ {k} (H1 H2 H3 : Functor ([Sets^ k ,Sets]) ([Sets^ k ,Sets]))
                               → (f : NaturalTransformation H1 H2) → (g : NaturalTransformation H2 H3)
                               → [Sets^ k ,Sets] Categories.Category.[
                                  fixH₁ H1 H3 (g ∘v f)
                                  ≈ fixH₁ H2 H3 g ∘v fixH₁ H1 H2 f
                               ]

fixH₁-F-resp : ∀ {k} (H1 H2 : Functor ([Sets^ k ,Sets]) ([Sets^ k ,Sets]))
                          → (f g : NaturalTransformation H1 H2)
                          → [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]] Categories.Category.[ f ≈ g ]
                          → [Sets^ k ,Sets] Categories.Category.[
                             fixH₁ H1 H2 f ≈ fixH₁ H1 H2 g
                          ]
```

The function fixH$_1$-identity proves that the fixpoint of an identity natural transformation on H is an identity natural transformation the fixpoint fixH$_0$ H. The function fixH$_1$-homomorphism proves that the fixpoint of a composition of a pair of natural transformations is equivalent to the composition of their fixpoints. The function fixH$_1$-F-resp proves that fixH$_1$ preserves morphism equality. We elide the definitions of these functions because the definitions are somewhat long and technical.

With these functions in hand, we can define the functor fixH:

```
fixH : ∀ {k} → Functor [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]] [Sets^ k ,Sets]
fixH = record
```

```
    { F₀ = λ H → fixH₀ H
    ; F₁ = λ {H1} {H2} η → fixH₁ H1 H2 η
    ; identity = λ {H} → fixH₁-identity H
    ; homomorphism = λ {H1} {H2} {H3} {f} {g} → fixH₁-homomorphism H1 H2 H3 f g
    ; F-resp-≈ = λ {H1} {H2} {f} {g} f≈g → fixH₁-F-resp H1 H2 f g f≈g
    }
```

We can also prove some properties about this fixpoint construction. In particular, we want to prove the application of H to the fixpoint fixH$_0$ H is isomorphic to the fixpoint itself. Since the objects we are comparing are functors, we want to show there exists a natural isomorphism between these functors. First, we define the two natural transformations that make up the natural isomorphism:

```
in-nat : ∀ {k} → (H : Functor [Sets^ k ,Sets] [Sets^ k ,Sets])
       → NaturalTransformation (Functor.F₀ H (fixH₀ H)) (fixH₀ H)
in-nat H = record { η = λ { Xs x → hin x }
                  ; commute = λ f → ≡.refl
                  ; sym-commute = λ f → ≡.refl }


in-inv-nat : ∀ {k} → (H : Functor [Sets^ k ,Sets] [Sets^ k ,Sets])
           → NaturalTransformation (fixH₀ H) (Functor.F₀ H (fixH₀ H))
in-inv-nat H =
  record { η = λ { X (hin x) → x }
         ; commute = λ { {Xs} {Ys} fs {hin x} → ≡.refl }
         ; sym-commute = λ { {Xs} {Ys} fs {hin x} → ≡.refl } }
```

The components of in-nat are given by hin, and the components of the inverse natural transformation in-inv-nat are given by pattern matching on hin and returning its argument x. The naturality proofs for these components are trivial because of the way HFix-fmap is defined by pattern matching on hin.

Now we can use the definition of NaturalIsomorphism defined in the agda-categories library by:

```
record NaturalIsomorphism
    {o ℓ e o' ℓ' e'}
    {C : Category o ℓ e}
    {D : Category o' ℓ' e'}
    (F G : Functor C D) : Set (o ⊔ ℓ ⊔ ℓ' ⊔ e') where

  field
    F⇒G : NaturalTransformation F G
    F⇐G : NaturalTransformation G F

  module ⇒ = NaturalTransformation F⇒G
  module ⇐ = NaturalTransformation F⇐G

  field
    iso : ∀ (X : Category.Obj C) → Categories.Morphism.Iso D (⇒.η X) (⇐.η X)
```

According to this definition, a natural isomorphism consists of a pair of natural transformations where every pair of components forms an isomorphism. Isomorphism is defined in the agda-categories library by:

```
record Iso {o ℓ e} (C : Category o ℓ e)
           {A B : Category.Obj C}
           (from : C [ A , B ]) (to : C [ B , A ]) : Set e where
  open Category C


  field
    isoˡ : to ∘ from ≈ id
    isoʳ : from ∘ to ≈ id
```

That is, an isomorphism is a pair of morphisms from and to and a pair of proofs that the compositions of from and to are identity morphisms.

Using these definitions, we can prove that in-nat and in-inv-nat form a natural isomorphism:

```
in-iso : ∀ {k} (H : Functor [Sets^ k ,Sets] [Sets^ k ,Sets])
       → (Xs : Vec Set k)
       → Categories.Morphism.Iso Sets
         (NaturalTransformation.η (in-inv-nat H) Xs)
         (NaturalTransformation.η (in-nat H) Xs)
```

```
in-iso H Xs = record { iso^l = λ { {hin x} → ≡.refl }
                     ; iso^r = ≡.refl }
```

The proofs that the compositions of the components of in-nat and in-inv-nat are identity morphisms are given by pattern matching. It is easy to see that the compositions are indeed identity morphisms: one component applies the hin constructor to its input, and the other component strips away the hin constructor, so both compositions of these components leave their inputs unchanged. We combine all of these data into a term of type $\mathsf{NaturalIsomorphism}\,(\mathsf{fixH}_0\,\mathsf{H})\,(\mathsf{Functor.F}_0\,\mathsf{H}\,(\mathsf{fixH}_0\,\mathsf{H}))$:

```
in-nat-iso : ∀ {k} → (H : Functor [Sets^ k ,Sets] [Sets^ k ,Sets])
           → NaturalIsomorphism (fixH₀ H) (Functor.F₀ H (fixH₀ H))
in-nat-iso H = record { F⇒G = in-inv-nat H ; F⇐G = in-nat H ; iso = in-iso H }
```

This natural isomorphism shows that $\mathsf{fixH}_0\,\mathsf{H}$ is indeed the fixpoint of the higher-order functor H.

Chapter 5

Syntax of Terms

In this chapter we present the syntax of terms for the calculus $\mathcal{N}$ and formalize this syntax in Agda. To do this, we must introduce the notions of term variables, term contexts, and term formation rules.

5.1 Term Variables and Term Contexts

We assume an infinite set V of *term variables*, disjoint from the sets of type constructor variables and functorial variables. Given a non-functorial context Γ and a functorial context Φ, a *term context* for Γ and Φ is a set of bindings of the form $x : F$ where $x \in V$ and $\Gamma; \Phi \vdash F$. We write $\Gamma; \Phi \vdash \Delta$ to denote that Δ is a term context for Γ and Φ. In contrast with type variables and type contexts, it is necessary to ensure that a particular term variable appears at most once in a term context, because it doesn't make sense to have both $x : F$ and $x : G$ in the same term context for different types F and G.

To formalize this notion of term contexts, we must first choose how to represent term variables in Agda. The type of term variables is called TermId in the formalization:

```
TermId : Set
TermId = String
```

We use the built-in String type from the standard library for term variables, just as we did for type variables.

5.1.1 Term Contexts in Agda

Term contexts are defined in Agda by:

```
data TermCtx (Γ : TCCtx) (Φ : FunCtx) : Set where
  Δ∅ : TermCtx Γ Φ
  _,-_:_⟨_⟩ : ∀ (Δ : TermCtx Γ Φ)
          → (x : TermId)
          → {F : TypeExpr} → (⊢F : Γ ≀ Φ ⊢ F)
          → (isYes (Δ-lookup x Δ) ≡ false)
          → TermCtx Γ Φ
```

A term context is either empty, or is constructed from a context Δ and a variable x of type F, where x does not appear in Δ. The fact that x does not appear in Δ is formalized by the argument of type isYes (Δ-lookup x Δ) ≡ false, where isYes P is a boolean that is true iff there is a proof of the proposition P. It is defined in the standard library as:

```
isYes : ∀ {a} {P : Set a} → Dec P → Bool
isYes (yes _) = true
isYes (no _) = false
```

We must also define a data type that asserts a variable appears in a context:

```
data _:∋_ : ∀ {Γ : TCCtx} {Φ : FunCtx} → TermCtx Γ Φ → TermId → Set where
  z∋ : ∀ {Γ : TCCtx} {Φ : FunCtx} {Δ : TermCtx Γ Φ} {x : TermId}
      → {p : isYes (Δ-lookup x Δ) ≡ false}
      → {F : TypeExpr} → {⊢F : Γ ≀ Φ ⊢ F}
      → (Δ ,- x : ⊢F ⟨ p ⟩) :∋ x

  s∋ : ∀ {Γ : TCCtx} {Φ : FunCtx} {Δ : TermCtx Γ Φ} {x y : TermId}
      → {p : isYes (Δ-lookup x Δ) ≡ false}
      → {F : TypeExpr} → {⊢F : Γ ≀ Φ ⊢ F}
      → Δ :∋ y
      → (Δ ,- x : ⊢F ⟨ p ⟩) :∋ y
```

A variable cannot appear in an empty context, and there are two ways for a variable to appear in a nonempty context Δ ,- x : ⊢F ⟨ p⟩. The constructor z∋ is used to prove that a variable x is the rightmost variable in a context, and the constructor s∋ is used to prove that a variable y appears in Δ ,- x : ⊢F ⟨ p⟩. when y appears in Δ. Given this definition of _:∋_, we can define a function that decides whether a variable appears in a context:

```
Δ-lookup : ∀ {Γ : TCCtx} {Φ : FunCtx} → (x : TermId) (Δ : TermCtx Γ Φ) → Dec (Δ :∋ x)
```

It is defined by pattern matching on the term context Δ. The type isYes (Δ-lookup x Δ) ≡ false asserts that the result of isYes (Δ-lookup x Δ) is equal to false, i.e., that x does not appear in Δ. There are other ways to formalize that x does not appear in Δ, but this definition using isYes is particularly useful when we need to construct an actual term context. By using isYes and Δ-lookup that decides whether x appears in Δ, we force Agda to evaluate Δ-lookup x Δ while it is type-checking elements of TermCtx Γ Φ. Since Δ-lookup x Δ is evaluated during type-checking, Agda can produce a term of type isYes (Δ-lookup x Δ) ≡ false automatically, if x is indeed absent from Δ. For example, if we have two term variables x : TermId and y : TermId that are defined by different strings, then we can define the term context containing only x of type 𝟙[1] as:

```
ctx-x : TermCtx ∅ ∅
ctx-x = Δ∅ ,- x : 𝟙-I ⟨ ≡.refl ⟩
```

The proof for isYes ... ≡ false is given trivially by ≡.refl since there are no constructors of _:∋_ for empty contexts. To add y of type 𝟙 to this context, we define another term context:

```
ctx-x-y : TermCtx ∅ ∅
ctx-x-y = ctx-x ,- y : 𝟙-I ⟨ {!!} ⟩ -- Goal:  isYes (Δ-lookup y ctx-x) ≡ false
```

Now we must give a proof of isYes (Δ-lookup y ctx-x) ≡ false. But if we ask Agda to reduce the goal further, we get:

```
ctx-x-y' : TermCtx ∅ ∅
ctx-x-y' = ctx-x ,- y : 𝟙-I ⟨ {!!} ⟩ -- Goal:  false ≡ false
```

Now the goal is simply false ≡ false, which can be proven by ≡.refl:

```
ctx-x-y'' : TermCtx ∅ ∅
ctx-x-y'' = ctx-x ,- y : 𝟙-I ⟨ ≡.refl ⟩
```

[1] The $\mathcal{N}$ types used in this example are not important, so we use 𝟙 and the constructor 𝟙-I for its typing judgment.

$$\frac{\Gamma;\Phi\vdash F}{\Gamma;\Phi\,|\,\Delta,x:F\vdash x:F}\qquad\frac{\Gamma;\Phi\,|\,\Delta\vdash t:\mathbb{0}\qquad\Gamma;\Phi\vdash F}{\Gamma;\Phi\,|\,\Delta\vdash\bot_F t:F}\qquad\frac{}{\Gamma;\Phi\,|\,\Delta\vdash\top:\mathbb{1}}$$

$$\frac{\Gamma;\Phi\,|\,\Delta\vdash s:F}{\Gamma;\Phi\,|\,\Delta\vdash\mathsf{inl}\,s:F+G}\qquad\frac{\Gamma;\Phi\,|\,\Delta\vdash t:G}{\Gamma;\Phi\,|\,\Delta\vdash\mathsf{inr}\,t:F+G}$$

$$\frac{\Gamma;\Phi\vdash F,G\qquad\Gamma;\Phi\,|\,\Delta\vdash t:F+G\qquad\Gamma;\Phi\,|\,\Delta,x:F\vdash l:K\qquad\Gamma;\Phi\,|\,\Delta,y:G\vdash r:K}{\Gamma;\Phi\ |\ \Delta\vdash\mathsf{case}\,t\,\mathsf{of}\,\{x\mapsto l;\,y\mapsto r\}:K}$$

$$\frac{\Gamma;\Phi\,|\,\Delta\vdash s:F\qquad\Gamma;\Phi\,|\,\Delta\vdash t:G}{\Gamma;\Phi\,|\,\Delta\vdash(s,t):F\times G}\qquad\frac{\Gamma;\Phi\,|\,\Delta\vdash t:F\times G}{\Gamma;\Phi\,|\,\Delta\vdash\pi_1 t:F}\qquad\frac{\Gamma;\Phi\,|\,\Delta\vdash t:F\times G}{\Gamma;\Phi\,|\,\Delta\vdash\pi_2 t:G}$$

$$\frac{\Gamma;\overline{\alpha}\vdash F\qquad\Gamma;\overline{\alpha}\vdash G\qquad\Gamma;\overline{\alpha}\,|\,\Delta,x:F\vdash t:G}{\Gamma;\emptyset\,|\,\Delta\vdash L_{\overline{\alpha}}x.t:\mathsf{Nat}^{\overline{\alpha}}\,F\,G}$$

$$\frac{\overline{\Gamma;\Phi\vdash K}\qquad\Gamma;\emptyset\,|\,\Delta\vdash t:\mathsf{Nat}^{\overline{\alpha}}\,F\,G\qquad\Gamma;\Phi\,|\,\Delta\vdash s:F[\overline{\alpha:=K}]}{\Gamma;\Phi\,|\,\Delta\vdash t_{\overline{K}}s:G[\overline{\alpha:=K}]}$$

$$\frac{\Gamma;\overline{\varphi},\overline{\gamma}\vdash H\qquad\overline{\Gamma;\overline{\beta},\overline{\gamma}\vdash F}\qquad\overline{\Gamma;\overline{\beta},\overline{\gamma}\vdash G}}{\Gamma;\emptyset\ |\ \emptyset\vdash\mathsf{map}_H^{\overline{F},\overline{G}}:\mathsf{Nat}^{\emptyset}\,(\overline{\mathsf{Nat}^{\overline{\beta},\overline{\gamma}}\,F\,G})\,(\mathsf{Nat}^{\overline{\gamma}}\,H[\overline{\varphi:=_{\overline{\beta}}F}]\ H[\overline{\varphi:=_{\overline{\beta}}G}])}$$

$$\frac{\Gamma;\varphi,\overline{\alpha}\vdash H}{\Gamma;\emptyset\,|\,\emptyset\vdash\mathsf{in}_H:\mathsf{Nat}^{\overline{\beta}}H[\varphi:=_{\overline{\beta}}(\mu\varphi.\lambda\overline{\alpha}.H)\overline{\beta}][\overline{\alpha:=\beta}]\,(\mu\varphi.\lambda\overline{\alpha}.H)\overline{\beta}}$$

$$\frac{\Gamma;\varphi,\overline{\alpha}\vdash H\qquad\Gamma;\overline{\beta}\vdash F}{\Gamma;\emptyset\,|\,\emptyset\vdash\mathsf{fold}_H^F:\mathsf{Nat}^{\emptyset}\,(\mathsf{Nat}^{\overline{\beta}}\,H[\varphi:=_{\overline{\beta}}F][\overline{\alpha:=\beta}]\,F)\,(\mathsf{Nat}^{\overline{\beta}}\,(\mu\varphi.\lambda\overline{\alpha}.H)\overline{\beta}\,F)}$$

Figure 5.1: Well-formed terms of $\mathcal{N}$.

5.2 Well-formed Terms

A term judgment $\Gamma;\Phi\,|\,\Delta\vdash x:F$ is parameterized by a non-functorial context Γ, a functorial context Φ, a term context Δ, and a type F, where $\Gamma;\Phi\vdash\Delta$ and $\Gamma;\Phi\vdash F$. The well-formed terms of $\mathcal{N}$ are defined by the inference rules in Figure 5.1.

The calculus $\mathcal{N}$ includes the standard introduction and elimination rules for term variables, $\mathbb{0}$, $\mathbb{1}$, $+$, and $\times$. The term $L_{\overline{\alpha}}x.t$ is analogous to the abstraction term used to define functions in the lambda calculus. The L rule introduces a term of type $\mathsf{Nat}^{\overline{\alpha}}F\,G$, and it binds the type variables in $\overline{\alpha}$ and the term variable x. Note that F and G have to be typed in contexts with $\overline{\alpha}$, and the resulting term $L_{\overline{\alpha}}x.t$ is formed with respect to contexts Γ and $\emptyset$, so we must have that $\Gamma;\emptyset\vdash\Delta$. The term $t_{\overline{K}}s$ applies a term t of Nat type to a term s, and there is one type in $\overline{K}$ for every functorial variable in $\overline{\alpha}$. It represents the application of a natural transformation at its component for the types in $\overline{K}$. The term $\mathsf{map}_H^{\overline{F},\overline{G}}$ represents the functorial map function that maps a vector of natural transformations over the functorial variables in $\overline{\varphi}$ in a type H. In the rule for $\mathsf{map}_H^{\overline{F},\overline{G}}$, there is one type F and one type G for each functorial variable in $\overline{\varphi}$. Moreover, for each φ^k in $\overline{\varphi}$ the number of functorial variables in $\overline{\beta}$ in the judgments for its corresponding type F and G is k. The term in_H represents the natural transformation from $T(\mu T)$ to μT for a higher-order functor T and is used to define the terms of a μ-type. The term fold_H^F represents a stylized form of recursion that can be defined for semantic fixpoints, and it is used to define recursive functions that take μ-types as their inputs. In the rules for in_H and fold_H^F, the functorial variables in $\overline{\beta}$ do not appear in H, and there is one β for every α.

To simplify the formalization of terms, we modify the map rule from [5] slightly so that map is only parameterized by a single type F and a single type G. This yields the following inference rule:

$$\frac{\Gamma; \varphi, \overline{\gamma} \vdash H \qquad \Gamma; \overline{\beta}, \overline{\gamma} \vdash F \qquad \Gamma; \overline{\beta}, \overline{\gamma} \vdash G}{\Gamma; \emptyset \mid \emptyset \vdash \mathsf{map}_H^{F,G} : \mathsf{Nat}^{\emptyset}\; (\mathsf{Nat}^{\overline{\beta},\overline{\gamma}}\, F\, G)\; (\mathsf{Nat}^{\overline{\gamma}}\, H[\varphi :=_{\overline{\beta}} F]\; H[\varphi :=_{\overline{\beta}} G])}$$

This modified rule is less expressive than the one given in Figure 5.1, but it is still expressive enough to give a map term for types functorial in a single variable such as $\mathit{List}\,\alpha$ and $\mathit{PTree}\,\alpha$.

The "extra" variables in $\overline{\gamma}$ in the map rule allow us to map polymorphic functions (i.e., terms of Nat type) over nested types. For example, if we have the types $\mathit{List}\,\gamma$ and $\mathit{PTree}\,\gamma$ then we can define a term of Nat type that flattens perfect trees into lists. To map the flattening function over the functorial variable α in the type $\mathit{List}\,\alpha$, we can use the following map term:

$$\frac{\Gamma; \alpha, \gamma \vdash \mathit{List}\,\alpha \qquad \Gamma; \gamma \vdash \mathit{PTree}\,\gamma \qquad \Gamma; \gamma \vdash \mathit{List}\,\gamma}{\Gamma; \emptyset \mid \emptyset \vdash \mathsf{map}_{\mathit{List}\,\alpha}^{\mathit{PTree}\,\gamma,\mathit{List}\,\gamma} : \mathsf{Nat}^{\emptyset}(\mathsf{Nat}^{\gamma}(\mathit{PTree}\,\gamma)\,(\mathit{List}\,\gamma))\,(\mathsf{Nat}^{\gamma}\,(\mathit{List}\,(\mathit{PTree}\,\gamma))\,(\mathit{List}\,(\mathit{List}\,\gamma)))}$$

5.3 Well-formed Terms in Agda

Before formalizing these inference rules, we define the grammar of term *expressions* in Agda by:

```
data TermExpr : Set where
  tmvar : TermId → TermExpr
  ⊥e : TermExpr → TermExpr
  ⊤i  : TermExpr
  inl : TermExpr → TermExpr
  inr : TermExpr → TermExpr
  case_of[_↦_≀_↦_] : (t : TermExpr) → (x : TermId)
```

```
                          → (l : TermExpr) → (y : TermId) → (r : TermExpr) → TermExpr
    _,,_ : TermExpr → TermExpr → TermExpr
    π₁ : TermExpr → TermExpr
    π₂ : TermExpr → TermExpr
    L[_]_,_ : {k : ℕ} → (αs : Vec (FVar 0) k) → (x : TermId) → (t : TermExpr) → TermExpr
    app_[_]_ : {k : ℕ} → (t : TermExpr) → (Ks : Vec TypeExpr k) → (s : TermExpr) → TermExpr
    map : (F : TypeExpr) → (G : TypeExpr) → (H : TypeExpr) → TermExpr
    inn  : (H : TypeExpr) → TermExpr
    fold  : (F : TypeExpr) → (H : TypeExpr) → TermExpr
```

A term expression is simply the name of a term (e.g., ⊤, map, etc.) without any reference to its type or contexts. The grammar of term expressions is not explicitly defined in [5], but we can infer it from Figure 5.1. TermExpr has one constructor for each of the inference rules there. The constructors of TermExpr can be thought of as "term constructors" that take some parameters and produce a term. The types of these constructors are derived from those inference rules, and some of their arguments are named to show the correspondence more clearly. We use inn as the name of the constructor for in because "in" is a keyword in Agda.

Using this definition of TermExpr, we can define the set of well-formed terms by:

```
  data _≀_|_⊢_:_ : ∀ (Γ : TCCtx) (Φ : FunCtx)
                       → TermCtx Γ Φ
                       → TermExpr
                       → {F : TypeExpr}
                       → Γ ≀ Φ ⊢ F
                       → Set where

    var-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
            → (Δ : TermCtx Γ Φ)
            → (x : TermId)
            → {F : TypeExpr}
            → (⊢F : Γ ≀ Φ ⊢ F)
            → (p : isYes (Δ-lookup x Δ) ≡ false)
            → Γ ≀ Φ | Δ ,- x : ⊢F ⟨ p ⟩ ⊢ tmvar x : ⊢F
```

```
⊥e-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
      → (Δ : TermCtx Γ Φ)
      → {F : TypeExpr}
      → (⊢F : Γ ≀ Φ ⊢ F)
      → (t : TermExpr)
      → Γ ≀ Φ | Δ ⊢ t : 𝟘-I
      → Γ ≀ Φ | Δ ⊢ ⊥e t : ⊢F

⊤-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
    → (Δ : TermCtx Γ Φ)
    → Γ ≀ Φ | Δ ⊢ ⊤i : 𝟙-I

inl-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
      → (Δ : TermCtx Γ Φ)
      → {F G : TypeExpr}
      → (⊢F : Γ ≀ Φ ⊢ F) (⊢G : Γ ≀ Φ ⊢ G)
      → (s : TermExpr)
      → Γ ≀ Φ | Δ ⊢ s : ⊢F
      → Γ ≀ Φ | Δ ⊢ inl s : +-I ⊢F ⊢G

inr-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
      → (Δ : TermCtx Γ Φ)
      → {F G : TypeExpr}
      → (⊢F : Γ ≀ Φ ⊢ F) (⊢G : Γ ≀ Φ ⊢ G)
      → (s : TermExpr)
      → Γ ≀ Φ | Δ ⊢ s : ⊢G
      → Γ ≀ Φ | Δ ⊢ inr s : +-I ⊢F ⊢G

case-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
      → (Δ : TermCtx Γ Φ)
      → {F G K : TypeExpr}
      → (⊢F : Γ ≀ Φ ⊢ F) → (⊢G : Γ ≀ Φ ⊢ G) → (⊢K : Γ ≀ Φ ⊢ K)
      → (t : TermExpr)
      → Γ ≀ Φ | Δ ⊢ t : +-I ⊢F ⊢G
      → (x : TermId)
      → (px : isYes (Δ-lookup x Δ) ≡ false)
      → (l : TermExpr)
      → Γ ≀ Φ | Δ ,- x : ⊢F ⟨ px ⟩ ⊢ l : ⊢K
      → (y : TermId)
      → (py : isYes (Δ-lookup y Δ) ≡ false)
      → (r : TermExpr)
      → Γ ≀ Φ | Δ ,- y : ⊢G ⟨ py ⟩ ⊢ r : ⊢K
      → Γ ≀ Φ | Δ ⊢ case t of[ x ↦ l ≀ y ↦ r ] : ⊢K

pair-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
     → (Δ : TermCtx Γ Φ)
     → {F G : TypeExpr}
     → (⊢F : Γ ≀ Φ ⊢ F) (⊢G : Γ ≀ Φ ⊢ G)
     → (s : TermExpr)
     → Γ ≀ Φ | Δ ⊢ s : ⊢F
     → (t : TermExpr)
     → Γ ≀ Φ | Δ ⊢ t : ⊢G
     → Γ ≀ Φ | Δ ⊢ (s ,, t) : ×-I ⊢F ⊢G

π₁-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
    → (Δ : TermCtx Γ Φ)
    → {F G : TypeExpr}
    → (⊢F : Γ ≀ Φ ⊢ F) (⊢G : Γ ≀ Φ ⊢ G)
    → (t : TermExpr)
    → Γ ≀ Φ | Δ ⊢ t : ×-I ⊢F ⊢G
    → Γ ≀ Φ | Δ ⊢ π₁ t : ⊢F

π₂-I : ∀ {Γ : TCCtx} {Φ : FunCtx}
    → (Δ : TermCtx Γ Φ)
    → {F G : TypeExpr}
    → (⊢F : Γ ≀ Φ ⊢ F) (⊢G : Γ ≀ Φ ⊢ G)
```

```
      → (t : TermExpr)
      → Γ ≀ Φ | Δ ⊢ t : ×-I ⊢F ⊢G
      → Γ ≀ Φ | Δ ⊢ π₂ t : ⊢G

L-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {k : ℕ} {αs : Vec (FVar 0) k}
     → {F G : TypeExpr}
     → (⊢F : Γ ≀ ( ∅ ,++ αs ) ⊢ F) (⊢G : Γ ≀ ( ∅ ,++ αs ) ⊢ G)
     → (Δ : TermCtx Γ ∅ )
     → (x : TermId)
     → (p : isYes (Δ-lookup x (weakenFunCtx-Δ-Vec αs Δ)) ≡ false)
     → (t : TermExpr)
     → Γ ≀ ( ∅ ,++ αs ) | (weakenFunCtx-Δ-Vec αs Δ) ,- x : ⊢F ⟨ p ⟩ ⊢ t : ⊢G
     → Γ ≀ ∅ | Δ ⊢ L[ αs ] x , t : Nat-I ⊢F ⊢G

app-I : ∀ {Γ : TCCtx} {Φ : FunCtx} {k : ℕ} {αs : Vec (FVar 0) k}
       → {F G : TypeExpr} {Ks : Vec TypeExpr k}
       → (⊢F : Γ ≀ ( ∅ ,++ αs ) ⊢ F) (⊢G : Γ ≀ ( ∅ ,++ αs ) ⊢ G)
       → (⊢Ks : foreach (λ K → Γ ≀ Φ ⊢ K) Ks)
       → (Δ : TermCtx Γ ∅ )
       → (t : TermExpr)
       → Γ ≀ ∅ | Δ ⊢ t : Nat-I ⊢F ⊢G
       → (s : TermExpr)
       → Γ ≀ Φ | (weakenFunCtx-Δ-∅ Φ Δ) ⊢ s :
                   ([:=]Vec-preserves-typing αs Ks (weakenFunCtx-∅-,++ αs ⊢F) ⊢Ks)
       → Γ ≀ Φ | (weakenFunCtx-Δ-∅ Φ Δ) ⊢ app t [ Ks ] s :
                   ([:=]Vec-preserves-typing αs Ks (weakenFunCtx-∅-,++ αs ⊢G) ⊢Ks)

map-I : ∀ {Γ : TCCtx} {g : ℕ} {k : ℕ}
       → {φ : FVar k}
       → {βs : Vec (FVar 0) k}
       → {γs : Vec (FVar 0) g}
       → {H F G : TypeExpr}
       → (⊢H : Γ ≀ (∅ ,++ γs) ,, φ ⊢ H)
      → (⊢F : Γ ≀ (∅ ,++ (γs ++ βs)) ⊢ F)
      → (⊢G : Γ ≀ (∅ ,++ (γs ++ βs)) ⊢ G)
      → Γ ≀ ∅ | Δ∅ ⊢ map F G H :
            Nat-I {αs = []}
              (Nat-I ⊢F ⊢G)
              (Nat-I (so-subst-preserves-typing {αs = βs} ⊢H (FunCtx-resp-++ γs βs ⊢F))
                     (so-subst-preserves-typing {αs = βs} ⊢H (FunCtx-resp-++ γs βs ⊢G)))

in-I : ∀ {Γ : TCCtx} {k : ℕ} {φ : FVar k} {αs βs : Vec (FVar 0) k}
    → {H : TypeExpr}
    → (⊢H : Γ ≀ ((∅ ,++ αs) ,, φ) ⊢ H)
    → Γ ≀ ∅ | Δ∅ ⊢ inn H :
          Nat-I {αs = βs} (in-I-helper ⊢H)
                          (μ-I ⊢H (VarExprVec βs) (VarTypeVec βs))

fold-I : ∀ {Γ : TCCtx} {k : ℕ} {φ : FVar k} {αs βs : Vec (FVar 0) k}
    → {H F : TypeExpr}
    → (⊢H : Γ ≀ ((∅ ,++ αs) ,, φ) ⊢ H)
    → (⊢F : Γ ≀ (∅ ,++ βs) ⊢ F)
    → Γ ≀ ∅ | Δ∅ ⊢ fold F H :
        Nat-I {αs = []}
          (Nat-I {αs = βs} (fold-I-helper ⊢H ⊢F) ⊢F)
          (Nat-I {αs = βs} (μ-I ⊢H (VarExprVec βs) (VarTypeVec βs)) ⊢F)
```

These constructors are given in the order of the inference rules in Figure 5.1 from top to bottom and left to right. Although some of these constructors take many arguments, most of them do not require much explanation. However, the types of the constructors **L-I**, **app-I**, **map-I**, **in-I**, and **fold-I** involve some proofs about substitution and weakening that are used to show that the substituted and weakened types appearing in the corresponding term inference rules are well-formed.

In particular, the conclusion of the inference rule for L implies that $\Gamma;\emptyset \vdash \Delta$, but in the term judgment for t in that rule we have $\Gamma;\overline{\alpha} \vdash \Delta$. This is justified by weakening each of the types in Δ by $\overline{\alpha}$, which is defined[2] in Agda by:

[2] We elide the actual definitions of the functions in this section and just give their types. They are all implemented by

```
weakenFunCtx-Δ-Vec : ∀ {k n : ℕ} {Γ : TCCtx} {Φ : FunCtx} (φs : Vec (FVar k) n)
                   → TermCtx Γ Φ
                   → TermCtx Γ (Φ ,++ φs)
```

In the inference rule for term application, we have that $\Gamma; \emptyset \vdash \Delta$ in the term judgment for t, but we have $\Gamma; \Phi \vdash \Delta$ in the conclusion, which is justified by weakening each of the types in Δ by Φ. This is defined in Agda by:

```
weakenFunCtx-Δ-∅ : ∀ { Γ : TCCtx } → (Φ : FunCtx)
                 → TermCtx Γ ∅
                 → TermCtx Γ Φ
```

The inference rule for term application also involves substitution of vectors of types, so we must also use the function [:=]Vec-preserves-typing defined in Section 2.6. We also use the function:

```
weakenFunCtx-∅-,++ : ∀ {k n} {Γ : TCCtx} {Φ : FunCtx} → (φs : Vec (FVar k) n)
                   → {F : TypeExpr} → Γ ≀ (∅ ,++ φs) ⊢ F
                   → Γ ≀ (Φ ,++ φs) ⊢ F
```

This function is used to weaken the typing judgment for F by the types in Φ in the type of app-I.

In the type of map-I, we use the vector concatenation _++_ function[3] to concatenate γs and βs. This concatenation function is defined in the standard library as[4]:

```
_++_ : ∀ {m n} {A : Set} → Vec A m → Vec A n → Vec A (m + n)
```

Note that we are forced to use the vector concatenation function in the typing judgments for F and G in the type of map-I. We cannot simply use the functorial context ((∅ ,++ γs) ,++ βs) because this context does not have the form of (∅ ,++ αs) required to form the type Nat-I ⊢F ⊢G. The type of map-I also uses the function so-subst-preserves-typing defined in Section 2.6 and the function

```
FunCtx-resp-++ : ∀ {Γ : TCCtx} {k j : ℕ} (αs : Vec (FVar 0) k) (βs : Vec (FVar 0) j) {F : TypeExpr}
               → Γ ≀ ( ∅ ,++ (αs ++ βs)) ⊢ F
               → Γ ≀ ( ∅ ,++ αs ) ,++ βs ⊢ F
```

pattern matching on their input typing judgment (or term context, for functions that take term contexts as their inputs) and applying the appropriate weakening functions recursively.

[3] Note that this function is distinct from the concatenation function _,++_ for type contexts and type variables, which has a comma in its name.

[4] Here, + is the addition function for natural numbers.

that is required to ensure that the second-order substitution in the type of map-I is well-typed.

The types of in-I and fold-I use the functions

```
VarExprVec : ∀ {k} → Vec (FVar 0) k → Vec TypeExpr k
```

and

```
VarTypeVec : ∀ {k} {Γ : TCCtx} {Φ : FunCtx} → (βs : Vec (FVar 0) k)
            → foreach (λ β → Γ ≀ Φ ,++ βs ⊢ β) (VarExprVec βs)
```

to prove that each of the types in a vector βs is well-formed with respect to contexts Γ and Φ ,++ βs. We also define the functions

```
in-I-helper : ∀ {Γ : TCCtx} {k : ℕ} {φ : FVar k} {αs βs : Vec (FVar 0) k} {H : TypeExpr}
          → (⊢H : Γ ≀ ((∅ ,++ αs) ,, φ) ⊢ H)
          → Γ ≀ (∅ ,++ βs) ⊢
              ((H [ φ :=[ βs ] (μ φ [λ αs , H ] VarExprVec βs) ]) [ αs := VarExprVec βs ]Vec)
```

and

```
fold-I-helper : ∀ {Γ : TCCtx} {k : ℕ} {φ : FVar k} {αs βs : Vec (FVar 0) k}
            → {H : TypeExpr} → (⊢H : Γ ≀ ((∅ ,++ αs) ,, φ) ⊢ H)
            → {F : TypeExpr} → (⊢F : Γ ≀ (∅ ,++ βs) ⊢ F)
            → Γ ≀ (∅ ,++ βs) ⊢ (H [ φ :=[ βs ] F ]) [ αs := VarExprVec βs ]Vec
```

to prove that the substitutions in the rules for inn and fold are well-typed. The functions in-I-helper and fold-I-helper are defined in terms of the weakening and substitution functions given here and in Chapter 2.

Chapter 6

Set Semantics of Terms

In this chapter we present the set semantics for the terms of $\mathcal{N}$. To define the term semantics, we must define the set interpretation of term contexts as a functor from the category of set environments to the category of sets. The set interpretation of a term is a natural transformation from the functor interpreting its term context to the functor interpreting its type.

Given a non-functorial context Γ, a functorial context Φ, and a term context $\Delta = x_1 : F_1, ..., x_n : F_n$ with $\Gamma; \Phi \vdash \Delta$, we define the set interpretation of Δ as the n-ary product of the functors interpreting the types in $F_1, ..., F_n$:

$$\begin{aligned}
&[\![\Gamma; \Phi \vdash \Delta]\!]^{\mathsf{Set}} : \mathsf{SetEnvCat} \to \mathsf{Sets} \\
&[\![\Gamma; \Phi \vdash \Delta]\!]^{\mathsf{Set}} = [\![\Gamma; \Phi \vdash F_1]\!]^{\mathsf{Set}} \times ... \times [\![\Gamma; \Phi \vdash F_n]\!]^{\mathsf{Set}}
\end{aligned}$$

Given a term judgment $\Gamma; \Phi \,|\, \Delta \vdash t : F$, we define its set interpretation as a natural transformation:

$$[\![\Gamma; \Phi \,|\, \Delta \vdash t : F]\!]^{\mathsf{Set}} : [\![\Gamma; \Phi \vdash \Delta]\!]^{\mathsf{Set}} \Rightarrow [\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}$$

Thus for each set environment ρ, the set interpretation of $\Gamma; \Phi \,|\, \Delta \vdash t : F$ in ρ is a function from $[\![\Gamma; \Phi \vdash \Delta]\!]^{\mathsf{Set}}\rho$ to $[\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}\rho$. The set interpretations for the terms of $\mathcal{N}$ are defined in Figure 6.1.

All of the term interpretations described here are given with respect to a set environment ρ. Term variables are interpreted by projecting out the set corresponding to $x : F$ in the product given by $[\![\Gamma; \Phi \vdash \Delta, x : F]\!]^{\mathsf{Set}}\rho$. A term of the form $\bot_F\, t$ is interpreted by composing the unique morphism

$$
\begin{array}{lcl}
\llbracket \Gamma;\Phi \,|\, \Delta, x : F \vdash x : F \rrbracket^{\mathsf{Set}}\rho & = & \pi_{|\Delta|+1} \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash \bot_F t : F \rrbracket^{\mathsf{Set}}\rho & = & !^{0}_{\llbracket \Gamma;\Phi \vdash F \rrbracket^{\mathsf{Set}}\rho} \circ \llbracket \Gamma;\Phi \mid \Delta \vdash t : \mathbb{0} \rrbracket^{\mathsf{Set}}\rho, \text{ where} \\
 & & \quad !^{0}_{\llbracket \Gamma;\Phi \vdash F \rrbracket^{\mathsf{Set}}\rho} \text{ is the unique morphism from } 0 \\
 & & \quad \text{to } \llbracket \Gamma;\Phi \vdash F \rrbracket^{\mathsf{Set}}\rho \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash \top : \mathbb{1} \rrbracket^{\mathsf{Set}}\rho & = & !^{\llbracket \Gamma;\Phi \vdash \Delta \rrbracket^{\mathsf{Set}}\rho}_{1}, \text{ where } !^{\llbracket \Gamma;\Phi \vdash \Delta \rrbracket^{\mathsf{Set}}\rho}_{1} \\
 & & \quad \text{is the unique morphism from } \llbracket \Gamma;\Phi \vdash \Delta \rrbracket^{\mathsf{Set}}\rho \text{ to } 1 \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash (s,t) : F \times G \rrbracket^{\mathsf{Set}}\rho & = & \llbracket \Gamma;\Phi \,|\, \Delta \vdash s : F \rrbracket^{\mathsf{Set}}\rho \times \llbracket \Gamma;\Phi \,|\, \Delta \vdash t : G \rrbracket^{\mathsf{Set}}\rho \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash \pi_1 t : F \rrbracket^{\mathsf{Set}}\rho & = & \pi_1 \circ \llbracket \Gamma;\Phi \,|\, \Delta \vdash t : F \times G \rrbracket^{\mathsf{Set}}\rho \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash \pi_2 t : G \rrbracket^{\mathsf{Set}}\rho & = & \pi_2 \circ \llbracket \Gamma;\Phi \,|\, \Delta \vdash t : F \times G \rrbracket^{\mathsf{Set}}\rho \\
\llbracket \Gamma;\Phi \mid \Delta \vdash \mathsf{case}\, t\, \mathsf{of}\, \{x \mapsto l;\; y \mapsto r\} : K \rrbracket^{\mathsf{Set}}\rho & = & \mathsf{eval} \circ \langle \mathsf{curry}\, [\llbracket \Gamma;\Phi \,|\, \Delta, x : F \vdash l : K \rrbracket^{\mathsf{Set}}\rho, \\
 & & \qquad\qquad \llbracket \Gamma;\Phi \,|\, \Delta, y : G \vdash r : K \rrbracket^{\mathsf{Set}}\rho], \\
 & & \qquad \llbracket \Gamma;\Phi \,|\, \Delta \vdash t : F + G \rrbracket^{\mathsf{Set}}\rho \rangle \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash \mathsf{inl}\, s : F + G \rrbracket^{\mathsf{Set}}\rho & = & \mathit{inl} \circ \llbracket \Gamma;\Phi \,|\, \Delta \vdash s : F \rrbracket^{\mathsf{Set}}\rho \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash \mathsf{inr}\, t : F + G \rrbracket^{\mathsf{Set}}\rho & = & \mathit{inr} \circ \llbracket \Gamma;\Phi \,|\, \Delta \vdash t : G \rrbracket^{\mathsf{Set}}\rho \\
\llbracket \Gamma;\emptyset \,|\, \Delta \vdash L_{\overline{\alpha}} x.t : \mathsf{Nat}^{\overline{\alpha}}\, F\, G \rrbracket^{\mathsf{Set}}\rho & = & \mathsf{curry}(\llbracket \Gamma;\overline{\alpha} \,|\, \Delta, x : F \vdash t : G \rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := _}]) \\
\llbracket \Gamma;\Phi \,|\, \Delta \vdash t_{\overline{K}} s : G[\overline{\alpha := K}] \rrbracket^{\mathsf{Set}}\rho & = & \mathsf{eval} \circ \langle \lambda d.\, (\llbracket \Gamma;\emptyset \,|\, \Delta \vdash t : \mathsf{Nat}^{\overline{\alpha}}\, F\, G \rrbracket^{\mathsf{Set}}\rho\; d)_{\overline{\llbracket \Gamma;\Phi \vdash K \rrbracket^{\mathsf{Set}}\rho}}, \\
 & & \qquad \llbracket \Gamma;\Phi \,|\, \Delta \vdash s : F[\overline{\alpha := K}] \rrbracket^{\mathsf{Set}}\rho \rangle \\
 & & \\
\llbracket \Gamma;\emptyset \,|\, \emptyset \vdash \mathsf{map}^{F,G}_{H} : \mathsf{Nat}^{\emptyset}(\mathsf{Nat}^{\overline{\beta},\overline{\gamma}} F\, G) & = & \lambda d\, \overline{\eta}\, \overline{C}.\, \llbracket \Gamma;\varphi,\overline{\gamma} \vdash H \rrbracket^{\mathsf{Set}} id_{\rho[\overline{\gamma := C}]}[\varphi := \lambda \overline{B}.\eta_{\overline{B}\,\overline{C}}] \\
\qquad (\mathsf{Nat}^{\overline{\gamma}}\, H[\varphi :=_{\overline{\beta}} F]\, H[\varphi :=_{\overline{\beta}} G]) \rrbracket^{\mathsf{Set}}\rho & & \\
\llbracket \Gamma;\emptyset \,|\, \emptyset \vdash \mathsf{in}_H : \mathsf{Nat}^{\overline{\beta}}\, H[\varphi := (\mu\varphi.\lambda\overline{\alpha}.H)\overline{\beta}][\overline{\alpha := \beta}] & = & \lambda d.\, in_{T^{Set}_{H,\rho}} \\
\qquad\qquad (\mu\varphi.\lambda\overline{\alpha}.H)\overline{\beta} \rrbracket^{\mathsf{Set}}\rho & & \\
\llbracket \Gamma;\emptyset \,|\, \emptyset \vdash \mathsf{fold}^{F}_{H} : \mathsf{Nat}^{\emptyset}\, (\mathsf{Nat}^{\overline{\beta}}\, H[\varphi :=_{\overline{\beta}} F][\overline{\alpha := \beta}]\, F) & = & \lambda d.\, \mathit{fold}_{T^{Set}_{H,\rho}} \\
\qquad\qquad (\mathsf{Nat}^{\overline{\beta}}\, (\mu\varphi.\lambda\overline{\alpha}.H)\overline{\beta}\; F) \rrbracket^{\mathsf{Set}}\rho & &
\end{array}
$$

Figure 6.1: Term semantics

$!^{0}_{\llbracket \Gamma;\Phi\vdash F\rrbracket^{\mathsf{Set}}\rho}$ from the empty set to the set interpretation of $\Gamma;\Phi\vdash F$ with respect to ρ with the term interpretation $\llbracket \Gamma;\Phi\,|\,\Delta\vdash t:\mathbb{0}\rrbracket^{\mathsf{Set}}\rho$. The term $\top$ is interpreted as the unique morphism $!^{\llbracket \Gamma;\Phi\vdash\Delta\rrbracket^{\mathsf{Set}}\rho}_{1}$ from the set interpretation $\llbracket \Gamma;\Phi\vdash\Delta\rrbracket^{\mathsf{Set}}\rho$ of the term context Δ into the singleton set 1. A term of the form (s,t) is interpreted as the product of the term interpretations $\llbracket \Gamma;\Phi\,|\,\Delta\vdash s:F\rrbracket^{\mathsf{Set}}\rho$ and $\llbracket \Gamma;\Phi\,|\,\Delta\vdash t:G\rrbracket^{\mathsf{Set}}\rho$. A term of the form $\pi_1 t$ is interpreted by composing the projection function $\pi_1 : A\times B\to A$, with the term interpretation $\llbracket \Gamma;\Phi\,|\,\Delta\vdash t:F\times G\rrbracket^{\mathsf{Set}}\rho$. The projection function π_1 takes a pair and projects out its first component. The intepretation for a term of the form $\pi_2\, t$ is defined analogously by the projection function $\pi_2 : A\times B\to B$. A term of the form $\mathsf{case}\, t\,\mathsf{of}\,\{x\mapsto l;\; y\mapsto r\}$ is interpreted in terms of the sum of two term interpretations. The sum of functions $f: A\to C$ and $g: B\to C$ is denoted $[f,g]:(A+B)\to C$, and it is defined by case analysis. If the input to $[f,g]$ comes from the left summand A, then f is applied to this input. Otherwise the input must come from the right summand B, in which case g is applied. A term of the form $\mathsf{inl}\, s$ is interpreted by composing the injection $inl : A\to A+B$ with the term interpretation $\llbracket \Gamma;\Phi\,|\,\Delta\vdash s:F\rrbracket^{\mathsf{Set}}\rho$. A term of the form $\mathsf{inr}\, t$ is interpreted analogously in terms of the injection $inr : B\to A+B$. A term of the form $L_{\overline{\alpha}}\,x.t$ is interpreted by applying a form of currying to the natural transformation

$$\llbracket \Gamma;\overline{\alpha}\,|\,\Delta, x:F\vdash t:G\rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := \_}] : \llbracket \Gamma;\overline{\alpha}\vdash\Delta\rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := \_}]\times\llbracket \Gamma;\overline{\alpha}\vdash F\rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := \_}] \Rightarrow \llbracket \Gamma;\overline{\alpha}\vdash G\rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := \_}]$$

to get a natural transformation of type

$$\llbracket \Gamma;\overline{\alpha}\vdash F\rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := \_}] \Rightarrow \llbracket \Gamma;\overline{\alpha}\vdash G\rrbracket^{\mathsf{Set}}\rho[\overline{\alpha := \_}] \tag{6.1}$$

i.e., an element of $\llbracket \Gamma;\emptyset\vdash \mathsf{Nat}^{\overline{\alpha}} F\, G\rrbracket^{\mathsf{Set}}\rho$. A term of the form $t_{\overline{K}}s$ is interpreted by first interpreting $\Gamma;\emptyset\,|\,\Delta\vdash t:\mathsf{Nat}^{\overline{\alpha}} F\, G$ with respect to ρ to get a natural transformation of type (6.1). Then the component of this natural transformation at $\overline{\llbracket \Gamma;\Phi\vdash K\rrbracket^{\mathsf{Set}}\rho}$ is applied to the term interpretation of $\Gamma;\Phi\,|\,\Delta\vdash s:F[\overline{\alpha := K}]$ with respect to ρ. A term of the form $\mathsf{map}^{F,G}_{H}$ is interpreted as the action of the set interpretation $\llbracket \Gamma;\varphi,\overline{\gamma}\vdash H\rrbracket^{\mathsf{Set}}$ on morphisms applied to the morphism of environments:

$$\begin{aligned} id_{\rho[\overline{\gamma := C}]}[\varphi := \lambda\overline{B}.\eta_{\overline{B}\,\overline{C}}] : \rho[\overline{\gamma := C}][\varphi := \lambda\overline{B}.\llbracket \Gamma;\overline{\beta},\overline{\gamma}\vdash F\rrbracket^{\mathsf{Set}}\rho[\overline{\beta := B}][\overline{\gamma := C}]] \\ \to\; \rho[\overline{\gamma := C}][\varphi := \lambda\overline{B}.\llbracket \Gamma;\overline{\beta},\overline{\gamma}\vdash G\rrbracket^{\mathsf{Set}}\rho[\overline{\beta := B}][\overline{\gamma := C}]] \end{aligned}$$

A term of the form in_H is interpreted as the natural transformation $in_{T^{\mathsf{Set}}_{H,\rho}} : T^{\mathsf{Set}}_{H,\rho}(\mu T^{\mathsf{Set}}_{H,\rho}) \Rightarrow \mu T^{\mathsf{Set}}_{H,\rho}$

associated with the higher-order functor $T^{\mathsf{Set}}_{H,\rho} : [\mathsf{Sets}^k, \mathsf{Sets}] \to [\mathsf{Sets}^k, \mathsf{Sets}]$. A term of the form fold^F_H is interpreted as the natural transformation $\mathit{fold}_{T^{\mathsf{Set}}_{H,\rho}}$ associated with $T^{\mathsf{Set}}_{H,\rho}$. It takes a natural transformation $\eta : T^{\mathsf{Set}}_{H,\rho} F \Rightarrow F$ and produces the natural transformation $\mathit{fold}_{T^{\mathsf{Set}}_{H,\rho}}\, \eta : \mu T^{\mathsf{Set}}_{H,\rho} \Rightarrow F$, where F is a functor of type $\mathsf{Sets}^k \to \mathsf{Sets}$. Note that terms of the form $\mathsf{map}^{F,G}_H$, in_H and fold^F_H are all formed with respect to empty term contexts (denoted by $\emptyset$), so they ignore their inputs d of type $[\![\Gamma; \emptyset \vdash \emptyset]\!]^{\mathsf{Set}} = 1$.

For the term interpretations in Figure 6.1 to be well-defined, we must prove that the set interpretation of types respects substitution and weakening, i.e., we must prove the following lemmas:

$$[\![\Gamma; \Phi \vdash G[\overline{\alpha := K}]]\!]^{\mathsf{Set}}\rho = [\![\Gamma; \Phi, \overline{\alpha} \vdash G]\!]^{\mathsf{Set}}\rho[\overline{\alpha := [\![\Gamma; \Phi \vdash K]\!]^{\mathsf{Set}}\rho}] \tag{6.2}$$

$$[\![\Gamma; \Phi \vdash F[\varphi := H]]\!]^{\mathsf{Set}}\rho = [\![\Gamma; \Phi, \varphi \vdash F]\!]^{\mathsf{Set}}\rho[\varphi := \lambda\overline{A}.\, [\![\Gamma; \Phi, \overline{\alpha} \vdash H]\!]^{\mathsf{Set}}\rho[\overline{\alpha := A}]] \tag{6.3}$$

These lemmas express that the interpretation of a type with a substitution is equal to the interpretation of the un-substituted type in an appropriately extended environment. In other words, if we have some variables that we would like to replace in the interpretation of a type, we can do this at either the syntactic level or at the semantic level and get the same result. We must also prove analogous lemmas for weakening, i.e., if $\Gamma; \Phi \vdash F$, then

$$[\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}} = [\![\Gamma; \Phi, \varphi \vdash F]\!]^{\mathsf{Set}} \tag{6.4}$$

$$[\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}} = [\![\Gamma, \varphi; \Phi \vdash F]\!]^{\mathsf{Set}} \tag{6.5}$$

To see why these lemmas are needed, consider the interpretation of a term $t_{\overline{K}} s$. The interpretation of $\Gamma; \emptyset \,|\, \Delta \vdash t : \mathsf{Nat}^{\overline{\alpha}} F\, G$ gives a natural transformation of the type in (6.1), but the interpretation of $\Gamma; \Phi \,|\, \Delta \vdash s : F[\overline{\alpha := K}]$ gives an element of $[\![\Gamma; \Phi \vdash F[\overline{\alpha := K}]]\!]^{\mathsf{Set}}$. In order to apply the component

$$([\![\Gamma; \emptyset \,|\, \Delta \vdash t : \mathsf{Nat}^{\overline{\alpha}} F\, G]\!]^{\mathsf{Set}}\rho\, d)_{\overline{[\![\Gamma; \Phi \vdash K]\!]^{\mathsf{Set}}\rho}}$$

of type

$$[\![\Gamma; \overline{\alpha} \vdash F]\!]^{\mathsf{Set}}\rho[\overline{\alpha := [\![\Gamma; \Phi \vdash K]\!]^{\mathsf{Set}}\rho}] \to [\![\Gamma; \overline{\alpha} \vdash G]\!]^{\mathsf{Set}}\rho[\overline{\alpha := [\![\Gamma; \Phi \vdash K]\!]^{\mathsf{Set}}\rho}]$$

to the term interpretation $[\![\Gamma; \Phi \,|\, \Delta \vdash s : F[\overline{\alpha := K}]]\!]^{\mathsf{Set}}\rho\, d$, we must have that

$$[\![\Gamma; \Phi \vdash F[\overline{\alpha := K}]]\!]^{\mathsf{Set}}\rho = [\![\Gamma; \overline{\alpha} \vdash F]\!]^{\mathsf{Set}}\rho[\overline{\alpha := [\![\Gamma; \Phi \vdash K]\!]^{\mathsf{Set}}\rho}]$$

6.1 Set Semantics of Terms in Agda

In this section, we give a partial definition of the set semantics of terms and discuss some limitations with our approach. We define the interpretation of term contexts in Agda by pattern matching:

```
ContextInterp : ∀ {Γ : TCCtx} {Φ : FunCtx} → TermCtx Γ Φ → Functor SetEnvCat Sets
ContextInterp Δ∅ = ConstF ⊤'
ContextInterp (Δ ,- _ : ⊢F ⟨ _ ⟩) = ContextInterp Δ ×Set SetSem ⊢F
```

An empty context is interpreted as a constant functor that sends every environment to the singleton set ⊤' [1] and every morphism of environments to the identity function on ⊤'. A non-empty context is interpreted as the product of ContextInterp Δ and SetSem ⊢F, where ×Set is defined by:

```
_×Set_ : ∀ {o ℓ e} {C : Category o ℓ e} → (F G : Functor C Sets) → Functor C Sets
F ×Set G = SetProd ∘F (F ※ G)
```

Given two functors F and G, (F ×Set G) takes an object (resp., morphism) of C and applies both F and G to get a pair of objects (resp., morphisms) and then takes the cartesian product of the pair.

The set interpretation of terms is defined by:

```
TermSetSem : ∀ {Γ : TCCtx} {Φ : FunCtx} {Δ : TermCtx Γ Φ}
                → {F : TypeExpr} → {⊢F : Γ ≀ Φ ⊢ F}
                → {t : TermExpr}
                → (⊢t : Γ ≀ Φ ∣ Δ ⊢ t : ⊢F)
                → NaturalTransformation (ContextInterp Δ) (SetSem ⊢F)
```

The set interpretation of term variables is defined by:

```
TermSetSem (var-I Δ x ⊢F p) = var-interp Δ {x} ⊢F {p = p}


var-interp : ∀ {Γ : TCCtx} {Φ : FunCtx}
             → (Δ : TermCtx Γ Φ) {x : TermId}
             → {F : TypeExpr} (⊢F : Γ ≀ Φ ⊢ F)
             → {p : isYes (Δ-lookup x Δ) ≡ false}
```

[1] Here we rename the unit type ⊤ in Agda to ⊤' to avoid confusion with the $\mathcal{N}$ term ⊤. For the same reason, we rename the empty type ⊥ to ⊥'.

```
                → NaturalTransformation (ContextInterp (Δ ,- x : ⊢F ⟨ p ⟩))
                                         (SetSem ⊢F)
var-interp Δ ⊢F = proj₂Nat {F = ContextInterp Δ} {G = SetSem ⊢F}
```

The function var-interp produces a natural transformation that projects out the type for the variable x from a context. It is defined in terms of proj$_2$Nat, which is defined by:

```
proj₂Nat : ∀ {F G : Functor C Sets} → NaturalTransformation (F ×Set G) G
```

A component of proj$_2$Nat {F} {G} takes a pair as input and returns the second component of the pair. The proofs of naturality are defined by straightforward pattern matching.

The set interpretation of ⊥e-I is defined by a composition of natural transformations:

```
TermSetSem (⊥e-I _ _ t ⊢t) = 𝟘! ∘v TermSetSem ⊢t
```

The natural transformation 𝟘! is defined by:

```
𝟘! : ∀ {F : Functor C Sets} → NaturalTransformation (ConstF ⊥') F
𝟘! = record { η = λ _ → exFalso ; commute = λ f → λ {} ; sym-commute = λ f → λ {} }
```

Each component of 𝟘! has ⊥' as its input type, so we define the components by exFalso. The naturality proofs are defined by pattern matching on ⊥' using the absurd lambda pattern[2].

The set interpretation of ⊤-I is defined by:

```
TermSetSem (⊤-I _) = 𝟙!
```

The natural transformation 𝟙! is defined by:

```
𝟙! : ∀ {F : Functor C Sets} → NaturalTransformation F (ConstF ⊤')
𝟙! = record { η = λ _ → const tt ; commute = λ f → ≡.refl ; sym-commute = λ f → ≡.refl }
```

Each component of 𝟙! is defined by sending its input to the single element tt of ⊤'.

The set interpretation of pair-I is defined by:

```
TermSetSem (pair-I _ ⊢F ⊢G _ ⊢s _ ⊢t) = prod-Nat (TermSetSem ⊢s) (TermSetSem ⊢t)
```

The function prod-Nat is defined by:

[2] The absurd lambda here is written $\lambda\{\}$ rather than $\lambda()$ because the argument of type ⊥' is implicit.

```
prod-Nat : ∀ {F G H : Functor C Sets}
            → NaturalTransformation F G
            → NaturalTransformation F H
            → NaturalTransformation F ((G ×Set H))
```

It takes a pair of natural transformations with the same domain F and produces a natural transformation from F into the product of G and H. Each component of prod-Nat is defined by applying the two input natural transformations to produce a pair. The naturality proofs are defined by applying the naturality proofs of the input natural transformations componentwise.

The set interpretations of π_1-I and π_2-I are defined by:

```
TermSetSem (π₁-I _ _ ⊢G t ⊢t) = proj₁Nat ∘v TermSetSem ⊢t

TermSetSem (π₂-I _ ⊢F _ t ⊢t) = proj₂Nat ∘v TermSetSem ⊢t
```

The natural transformation proj$_2$Nat is defined above, and the natural transformation proj$_1$Nat is defined analogously. That is, each of its components is defined by projecting out the first component of a pair.

The set interpretations of inl and inr are defined by compositions:

```
TermSetSem (inl-I _ ⊢F ⊢G t ⊢t) = inl-Nat ∘v TermSetSem ⊢t

TermSetSem (inr-I _ ⊢F ⊢G t ⊢t) = inr-Nat ∘v TermSetSem ⊢t

inl-Nat : ∀ {F G : Functor C Sets} → NaturalTransformation F ((F +Set G))

inr-Nat : ∀ {F G : Functor C Sets} → NaturalTransformation G ((F +Set G))
```

Here, the components of inl-Nat and inr-Nat are defined by the injections inj$_1$ and inj$_2$ of the sum type, and the naturality proofs are trivial. The function +Set

```
_+Set_ : ∀ {o ℓ e} {C : Category o ℓ e} → (F G : Functor C Sets) → Functor C Sets
F +Set G = SetSum ∘F (F ※ G)
```

is the analogue for sum types of ×Set defined above for product types.

The set interpretation of case-I is slightly more involved and is defined by:

```
TermSetSem (case-I Δ ⊢F ⊢G ⊢K _ ⊢t _ _ _ ⊢l _ _ _ ⊢r) =
  let ⟦Δ⟧ : Functor SetEnvCat Sets
  ⟦Δ⟧ = ContextInterp Δ
  ⟦F⟧ : Functor SetEnvCat Sets
  ⟦F⟧ = SetSem ⊢F
  ⟦G⟧ : Functor SetEnvCat Sets
  ⟦G⟧ = SetSem ⊢G

  [l,r] : NaturalTransformation ((⟦Δ⟧ ×Set ⟦F⟧) +Set (⟦Δ⟧ ×Set ⟦G⟧)) (SetSem ⊢K)
  [l,r] = sum-Nat (TermSetSem ⊢l) (TermSetSem ⊢r)

  distr : NaturalTransformation (⟦Δ⟧ ×Set (⟦F⟧ +Set ⟦G⟧))
                                ((⟦Δ⟧ ×Set ⟦F⟧) +Set (⟦Δ⟧ ×Set ⟦G⟧))
  distr = ×Set-distr ⟦Δ⟧ ⟦F⟧ ⟦G⟧

  semt : NaturalTransformation ⟦Δ⟧ (⟦F⟧ +Set ⟦G⟧)
  semt = TermSetSem ⊢t

  ⟦Δ⟧×tsem : NaturalTransformation ⟦Δ⟧ (⟦Δ⟧ ×Set (⟦F⟧ +Set ⟦G⟧))
  ⟦Δ⟧×tsem = prod-Nat idnat (TermSetSem ⊢t)

in [l,r] ∘v distr ∘v ⟦Δ⟧×tsem
```

It is defined by composing three natural transformation to produce a natural transformation of type NaturalTransformation (ContextInterp Δ) (SetSem ⊢K). We first use sum-Nat on the interpretations of ⊢l and ⊢r. The function sum-Nat

```
sum-Nat : ∀ {F G H : Functor C Sets}
          → NaturalTransformation F H
          → NaturalTransformation G H
          → NaturalTransformation ((F +Set G)) H
```

takes two natural transformations with a common codomain H and returns a natural transformation from the sum of functors F +Set G to the functor H. Given two natural transformations η_1 and η_2, each component of sum-Nat η_1 η_2 is defined by case analysis, applying η_1 if the input is from the left summand and applying η_2 is the input is from the right summand. We also use the function ×Set-distr defined by:

```
×Set-distr : ∀ {o ℓ e} {C : Category o ℓ e} → (F G H : Functor C Sets)
                 → NaturalTransformation (F ×Set (G +Set H)) ((F ×Set G) +Set (F ×Set H))
```

It produces a natural transformation that distributes ×Set over +Set. Each of its components is defined by pattern matching and repackaging its inputs into the appropriate shape. This natural transformation does not really change its input data. It is just required as an intermediate step to make the types line up. The last natural transformation $\llbracket\Delta\rrbracket$×tsem used in the intepretation of case-I is defined by the function prod-Nat inroduced above. Each of its components is defined by taking an element d of $\llbracket\Delta\rrbracket$ and returning a pair (d, t), where t is obtained by applying the interpretation of ⊢t to the input d.

The term intepretation for L-I is defined by:

```
TermSetSem (L-I ⊢F ⊢G Δ _ _ _ ⊢t) = curryNatType ⊢F ⊢G (TermSetSem ⊢t)


curryNatType : ∀ {Γ Φ} {Δ : TermCtx Γ ∅} {k} {αs : Vec (FVar 0) k} {F} {G}
                    → (⊢F : Γ ≀ ∅ ,++ αs ⊢ F) (⊢G : Γ ≀ ∅ ,++ αs ⊢ G)
                    → NaturalTransformation
                         (ContextInterp (weakenFunCtx-Δ-Vec αs Δ) ×Set SetSem ⊢F)
                         (SetSem ⊢G)
                    → NaturalTransformation (ContextInterp Δ) (SetSem (Nat-I ⊢F ⊢G))
```

The function curryNatType takes a natural transformation given by interpreting ⊢t and returns a natural transformation from ContextInterp Δ to SetSem (Nat-I ⊢F ⊢G) = NatTypeFunctor αs⊢F⊢G. To do so, it first uses a proof that the interpretation of term contexts respects weakening to change the domain of its input natural transformation from ContextInterp (weakenFunCtx-Δ-Vec αs Δ) ×Set (SetSem ⊢F) to ContextInterp Δ ×Set (SetSem ⊢F). Each component of the natural transformation returned by curryNatType takes a set environment ρ and an element of $\mathsf{Functor.F}_0$ (ContextInterp Δ) ρ and returns a natural transformation. Such a component is defined by whiskering the natural transformation obtained by weakening with the environment extension functor extendSetEnv-αs αs ρ : Functor (Sets^ k) SetEnvCat used in the interpretation of Nat types. The whiskering used here is defined in the agda-categories library by:

```
_∘ʳ_ : ∀ {o ℓ e o' ℓ' e' o'' ℓ'' e''}
        → {C : Category o ℓ e} {D : Category o' ℓ' e'} {E : Category o'' ℓ'' e''}
```

```
  → {G H : Functor D E} → NaturalTransformation G H → (F : Functor C D)
  → NaturalTransformation (G ∘F F) (H ∘F F)
```

The component of a natural transformation $\eta \circ^r \mathsf{F}$ at an object X is defined by the component of η at the object $\mathsf{Functor.F_0\,F\,X}$. This gives a natural transformation from

$$(\mathsf{ContextInterp}\,\Delta\ \times\mathsf{Set}\ (\mathsf{SetSem}\vdash\mathsf{F}))\circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\,\rho$$

to

$$(\mathsf{SetSem}\vdash\mathsf{G})\circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\,\rho$$

We also have a proof that functor composition distributes over $\times\mathsf{Set}$, so we can turn this into a natural transformation from

$$((\mathsf{ContextInterp}\,\Delta)\circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\,\rho)\ \times\mathsf{Set}\ ((\mathsf{SetSem}\vdash\mathsf{F})\circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\,\rho)$$

to

$$(\mathsf{SetSem}\vdash\mathsf{G})\circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\,\rho$$

Since Δ is formed with an empty functorial context, any environment extension of functorial variables has no effect on its interpretation. That is, for every $\rho : \mathsf{SetEnv}$,

$$\mathsf{Functor.F_0}\ (\mathsf{ContextInterp}\,\Delta)\rho \equiv \mathsf{Functor.F_0}\ (\mathsf{ContextInterp}\,\Delta)(\rho[\alpha\mathsf{s}\ \mathsf{:fvs=}\ \ldots])$$

no matter what the variables in αs are replaced by. Therefore the functor

$$(\mathsf{ContextInterp}\,\Delta)\circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\,\rho$$

from Setsˆ k to Sets is equivalent (naturally isomorphic) to

$$\mathsf{ConstF}\ \ (\mathsf{Functor.F_0}(\mathsf{ContextInterp}\,\Delta)\rho)$$

Since we have an element of $\mathsf{Functor.F_0}\ (\mathsf{ContextInterp}\,\Delta)\ \rho$ at hand, we can complete the definition using the function

```
makeNat : ∀ {o ℓ e} {C : Category o ℓ e}
          → (F G : Functor C Sets)
          → (D : Set)
          → (d : D)
          → NaturalTransformation (ConstF D ×Set F) G
          → NaturalTransformation F G
makeNat F G D d η = η ∘v apply-d
  where apply-d : NaturalTransformation F (ConstF D ×Set F)
        apply-d = prod-Nat (toConstF F D d) idnat
        toConstF : NaturalTransformation F (ConstF D)
        toConstF = record { η = λ X x → d ; commute = λ f → ≡.refl ; sym-commute = λ f → ≡.refl }
```

on our natural transformation from

$$(\mathsf{ConstF}\ (\mathsf{Functor.F}_0(\mathsf{ContextInterp}\ \Delta)\ \rho))\ \times\mathsf{Set}\ ((\mathsf{SetSem} \vdash \mathsf{F}) \circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\ \rho)$$

to

$$(\mathsf{SetSem} \vdash \mathsf{G}) \circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\ \rho$$

This yields a natural transformation from

$$(\mathsf{SetSem} \vdash \mathsf{F}) \circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\ \rho$$

to

$$(\mathsf{SetSem} \vdash \mathsf{G}) \circ\mathsf{F}\ \mathsf{extendSetEnv\text{-}}\alpha\mathsf{s}\ \alpha\mathsf{s}\ \rho$$

as desired. The naturality proofs for curryNatType are straightforward because the action of $\mathsf{NatTypeFunctor}\ \alpha\mathsf{s} \vdash \mathsf{F} \vdash \mathsf{G}$ on morphisms of environments always gives identity morphisms.

6.1.1 Interpreting the Remaining Terms

To define the set interpretations for app-I, map-I, in-I and fold-I, we need proofs that (syntactic) substitution and weakening interact nicely with (semantic) environment extension. In particular, we need to formalize Equations 6.2, 6.3, 6.4, and 6.5 in Agda. The proofs for weakening (Equations 6.4 and

6.5) are straightforward enough because the weakening functions we defined in Section 2.5 do not really change their input typing judgments in the sense that they only add new variables to the contexts. The contexts themselves have no bearing on the set interpretation of types[3] so weakening a typing judgment does not change its set interpretation. However, proving Equations 6.2 and 6.3 in Agda is not so straightforward because a substitution *can* change the interpretation of a type, and there are also environment extensions involved. We can prove Equation 6.2 for a substitution of a single variable, i.e.,

$$[\![\Gamma;\Phi \vdash G[\alpha := K]]\!]^{\mathsf{Set}}\rho = [\![\Gamma;\Phi,\alpha \vdash G]\!]^{\mathsf{Set}}\rho[\alpha := [\![\Gamma;\Phi \vdash K]\!]^{\mathsf{Set}}\rho]$$

but this does not suffice to define all of the remaining terms. The obstacle we run into with vectors of variables, e.g., as in

$$[\![\Gamma;\Phi \vdash G[\overline{\alpha := K}]]\!]^{\mathsf{Set}}\rho = [\![\Gamma;\Phi,\overline{\alpha} \vdash G]\!]^{\mathsf{Set}}\rho[\overline{\alpha := [\![\Gamma;\Phi \vdash K]\!]^{\mathsf{Set}}\rho}]$$

is that we need to know that the variables in $\overline{\alpha}$ are disjoint from those in Φ, i.e., that the types in $\overline{K}$ do not depend on the variables in $\overline{\alpha}$. This issue arises because vector substitution is defined by induction, so if we have variables α_1, α_2 and types K_1, K_2, then

$$G[\alpha_1, \alpha_2 := K_1, K_2] = (G[\alpha_2 := K_2])[\alpha_1 := K_1]$$

Semantically, this gives

$$\begin{aligned}
& [\![\Gamma;\Phi \vdash (G[\alpha_2 := K_2])[\alpha_1 := K_1]]\!]^{\mathsf{Set}}\rho \\
= \;& [\![\Gamma;\Phi,\alpha_1 \vdash (G[\alpha_2 := K_2])]\!]^{\mathsf{Set}}\rho[\alpha_1 := [\![\Gamma;\Phi \vdash K_1]\!]^{\mathsf{Set}}\rho] \\
= \;& [\![\Gamma;\Phi,\alpha_1,\alpha_2 \vdash G]\!]^{\mathsf{Set}}(\rho[\alpha_1 := [\![\Gamma;\Phi \vdash K_1]\!]^{\mathsf{Set}}\rho])[\alpha_2 := [\![\Gamma;\Phi \vdash K_2]\!]^{\mathsf{Set}}(\rho[\alpha_1 := [\![\Gamma;\Phi \vdash K_1]\!]^{\mathsf{Set}}\rho])]
\end{aligned}$$

The environment extension we really want however is

$$[\![\Gamma;\Phi,\alpha_1,\alpha_2 \vdash G]\!]^{\mathsf{Set}}(\rho[\alpha_1 := [\![\Gamma;\Phi \vdash K_1]\!]^{\mathsf{Set}}\rho])[\alpha_2 := [\![\Gamma;\Phi \vdash K_2]\!]^{\mathsf{Set}}\rho]$$

where α_2 is mapped to $[\![\Gamma;\Phi \vdash K_2]\!]^{\mathsf{Set}}\rho$ rather than $[\![\Gamma;\Phi \vdash K_2]\!]^{\mathsf{Set}}(\rho[\alpha_1 := [\![\Gamma;\Phi \vdash K_1]\!]^{\mathsf{Set}}\rho])$. If we know

[3] It is true that for type application, we must know that the variable φ being applied appears in one of the contexts. But the particular proof of φ appearing in a context is not important and does not affect the set interpretation of types.

that K_2 is really not using the variables in $\overline{\alpha}$, then these two set interpretation of K_2 are equivalent. But when we formalized type variable contexts in Agda, we did not have the foresight to consider this issue, and so we do not currently have a mechanism for proving

$$[\![\Gamma; \Phi \vdash K_2]\!]^{\mathsf{Set}}\rho = [\![\Gamma; \Phi \vdash K_2]\!]^{\mathsf{Set}}(\rho[\alpha_1 := [\![\Gamma; \Phi \vdash K_1]\!]^{\mathsf{Set}}\rho])$$

in Agda. For these reasons, we leave the formalization of the term interpretations for app-I, map-I, in-I and fold-I for future work.

Chapter 7

Relation Semantics of Types

In this chapter we present the relation semantics of types from [5] for the calculus $\mathcal{N}$ and formalize this semantics in Agda. The relation semantics of types is given in parallel to the set interpretation of types. That is, for each construct (e.g., set environments) used in the set semantics there is a corresponding construct (e.g., relation environments) in the relation semantics. The relation semantics of types is also defined mutually recursively with the set interpretation of types because of the relation-preserving condition in the set interpretation of Nat types.

7.1 Relations and Relation Transformers

A relation over sets A and B is a subset of the cartesian product $A \times B$. We write $R : \mathsf{Rel}(A, B)$ to denote that R is a relation over A and B and sometimes use the notation (A, B, R) to denote this relation. For a relation $R : \mathsf{Rel}(A, B)$, we refer to A as the *domain* of R and B as the *codomain* of R. We sometimes write $\pi_1 R$ and $\pi_2 R$ for the domain and codomain of R, respectively. Given a relation $R : \mathsf{Rel}(A, B)$ and elements $x \in A$ and $y \in B$, we say x and y are *related* if the pair (x, y) is included in the subset of $A \times B$ given by R. We denote this by $(x, y) \in R$. The *equality* relation $\mathsf{Eq}_X : \mathsf{Rel}(X, X)$ for a set X only relates elements of X to themselves, i.e, $\mathsf{Eq}_X = \{(x, x) \mid x \in X\}$. A morphism of relations from $R : \mathsf{Rel}(A, B)$ to $S : \mathsf{Rel}(C, D)$ is a pair of functions $(f1 : A \to C, f2 : B \to D)$ such that, for every pair (x, y) related in R, the pair $(f1\, x, f2\, y)$ is related in S. In other words, $f1$ and $f2$ must preserve related elements. Composition of morphisms is defined componentwise, i.e., $(g1, g2) \circ (f1, f2) = (g1 \circ f1, g2 \circ f2)$. An identity morphism of relations is given by a pair of identity functions. These data yield a category of relations, which we denote by Rels.

7.1.1 Relation Transformers

Before defining relation environments and the relational interpretation of types, some setup is required to define the relational analogues of set functors. Rather than simply using functors of relations in place of functors of sets, we require some extra structure on our functors of relations. This extra structure is defined in terms of *relation transformers*, and it is necessary to ensure that the relation semantics is "over" the set semantics in the sense that the relational interpretation of types relates elements of the set interpretation of types (see Lemma 7.4.1). The following definitions come directly from [5], with some minor changes in terminology.

Definition 7.1.1. A *k-ary relation transformer* F is a triple (F^1, F^2, F^*), where

- $F^1, F^2 : [\mathsf{Set}^k, \mathsf{Set}]$ and $F^* : [\mathsf{Rel}^k, \mathsf{Rel}]$ are functors
- If $R_1 : \mathsf{Rel}(A_1, B_1), ..., R_k : \mathsf{Rel}(A_k, B_k)$, then $F^*\overline{R} : \mathsf{Rel}(F^1\overline{A}, F^2\overline{B})$
- If $(\alpha_1, \beta_1) \in \mathsf{Rels}(R_1, S_1), ..., (\alpha_k, \beta_k) \in \mathsf{Rels}(R_k, S_k)$ then $F^*\overline{(\alpha, \beta)} = (F^1\overline{\alpha}, F^2\overline{\beta})$

Recall that the notation $m \in \mathsf{Rels}(R, S)$ indicates that m is a morphism in the category Rels from R to S. The last clause of Definition 7.1.1 expands to: if $\overline{(a, b) \in R}$ implies $\overline{(\alpha\, a, \beta\, b) \in S}$ then $(c, d) \in F^*\overline{R}$ implies $(F^1\overline{\alpha}\, c, F^2\overline{\beta}\, d) \in F^*\overline{S}$. When convenient we identify a 0-ary relation transformer (A, B, R) with $R : \mathsf{Rel}(A, B)$, and write $\pi_1 F$ for F^1 and $\pi_2 F$ for F^2.

A relation transformer can be thought of as a functor of relations F^* with extra structure defined by two functors of sets. The action of F^* on objects gives relations that are "over" the sets given by the actions of F^1 and F^2 on objects, and the action of F^* on morphisms is given precisely by the actions of F^1 and F^2 on morphisms. If F^* is the third component of a relation transformer, then F^1 and F^2 are uniquely determined, but not every functor of relations can be made into the third component of a relation transformer.

Definition 7.1.2. The category RT_k of k-ary relation transformers is given by the following data:

- An object of RT_k is a k-ary relation transformer
- A morphism $\delta : (G^1, G^2, G^*) \to (H^1, H^2, H^*)$ in RT_k is a pair of natural transformations (δ^1, δ^2), where $\delta^1 : G^1 \Rightarrow H^1$ and $\delta^2 : G^2 \Rightarrow H^2$, such that, for all $\overline{R : \mathsf{Rel}(A, B)}$, if $(x, y) \in G^*\overline{R}$ then $(\delta^1_{\overline{A}} x, \delta^2_{\overline{B}} y) \in H^*\overline{R}$

- Identity morphisms and composition are inherited from the category of functors on Sets

We also have a notion of higher-order relation transformers, which are analogous to higher-order functors of sets:

Definition 7.1.3. A *k-ary higher-order relation transformer* H is a triple $H = (H^1, H^2, H^*)$, where

- H^1 and H^2 are functors from $[\mathsf{Sets}^k, \mathsf{Sets}]$ to $[\mathsf{Sets}^k, \mathsf{Sets}]$
- H^* is a functor from RT_k to $[\mathsf{Rel}^k, \mathsf{Rel}]$
- For all $\overline{R : \mathsf{Rel}(A, B)}$, $\pi_1((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^1\delta^1)_{\overline{A}}$ and $\pi_2((H^*(\delta^1, \delta^2))_{\overline{R}}) = (H^2\delta^2)_{\overline{B}}$
- The action of H on relation transformers is given by
  $H\,(F^1, F^2, F^*) = (H^1F^1,\ H^2F^2,\ H^*(F^1, F^2, F^*))$
- The action of H on morphisms of relation transformers is given by $H\,(\delta^1, \delta^2) = (H^1\delta^1, H^2\delta^2)$ for $(\delta^1, \delta^2) : (F^1, F^2, F^*) \to (G^1, G^2, G^*)$

A higher-order relation transformer can be thought of as an endofunctor on RT_k with extra structure. Since the result of applying a higher-order relation transformer H to k-ary relation transformers and morphisms between them must again be k-ary relation transformers and morphisms between them, respectively, Definition 7.1.3 implicitly requires that the following three conditions hold:

- $H^*(F^1, F^2, F^*)\overline{R} : \mathsf{Rel}(H^1F^1\overline{A}, H^2F^2\overline{B})$ whenever $R_1 : \mathsf{Rel}(A_1, B_1), ..., R_k : \mathsf{Rel}(A_k, B_k)$
- $H^*(F^1, F^2, F^*)\,\overline{(\alpha, \beta)} = (H^1F^1\overline{\alpha}, H^2F^2\overline{\beta})$
  whenever $(\alpha_1, \beta_1) \in \mathsf{Rels}(R_1, S_1), ..., (\alpha_k, \beta_k) \in \mathsf{Rels}(R_k, S_k)$
- If $(\delta^1, \delta^2) : (F^1, F^2, F^*) \to (G^1, G^2, G^*)$ and $R_1 : \mathsf{Rel}(A_1, B_1), ..., R_k : \mathsf{Rel}(A_k, B_k)$,
  then $((H^1\delta^1)_{\overline{A}}x, (H^2\delta^2)_{\overline{B}}y) \in H^*(G^1, G^2, G^*)\overline{R}$ whenever $(x, y) \in H^*(F^1, F^2, F^*)\overline{R}$

Note, however, that this last condition is automatically satisfied because it is implied by the third bullet point of Definition 7.1.3. Higher-order relation transformers also form a category, denoted HRT_k.

Definition 7.1.4. For higher-order relation transformers H and K, a morphism $\sigma : H \to K$ is a pair $\sigma = (\sigma^1, \sigma^2)$, where $\sigma^1 : H^1 \Rightarrow K^1$ and $\sigma^2 : H^2 \Rightarrow K^2$ are natural transformations between endofunctors on $[\mathsf{Set}^k, \mathsf{Set}]$ and the component of σ at $F = (F^1, F^2, F^*) \in RT_k$ is given by $\sigma_F = (\sigma^1_{F^1}, \sigma^2_{F^2})$. This entails that $\sigma^i_{F^i}$ is natural in $F^i : [\mathsf{Set}^k, \mathsf{Set}]$, and, for every F, both $(\sigma^1_{F^1})_{\overline{A}}$ and $(\sigma^2_{F^2})_{\overline{A}}$ are natural in

$\overline{A}$. Moreover, since the results of applying σ to k-ary relation transformers must be morphisms of k-ary relation transformers, this definition implicitly requires that $(\sigma_F)_{\overline{R}} = ((\sigma^1_{F^1})_{\overline{A}}, (\sigma^2_{F^2})_{\overline{B}})$ is a morphism in Rel for any k-tuple of relations $\overline{R : \mathsf{Rel}(A,B)}$, i.e., that if $(x,y) \in H^* F \overline{R}$, then $((\sigma^1_{F^1})_{\overline{A}} x, (\sigma^2_{F^2})_{\overline{B}} y) \in K^* F \overline{R}$.

7.2 Relations and Relation Transformers in Agda

Relations as defined above are *proof-irrelevant*, i.e., two elements are either related (included in the subset) or they are not. In other words, relations as defined above do not support a notion of having multiple proofs of $(x,y) \in R$. A *proof-relevant* relation $R : \mathsf{Rel}(A,B)$ may have multiple proofs of $(x,y) \in R$ for a particular x and y. Note that proof-relevant relations cannot be defined simply as subsets of cartesian products because such subsets cannot contain multiple occurrences of the same pair (x,y).

In our Agda formalization, we use proof-relevant relations to define the relation semantics of types, as proof-relevant relations are much easier to manipulate in Agda than proof-irrelevant ones. A proof-relevant relation is defined in Agda as a binary function with a return type of Set:

```
REL : Set → Set → Set₁
REL A B = A → B → Set
```

The return type of the function REL[1] itself is Set$_1$ because the definition of REL involves the type Set. Given a relation R : REL A B and elements x : A and y : B, the type R x y gives the set of proofs that x and y are related. We define a shorthand for this type by:

```
_,_∈_ : ∀ {A B : Set} → A → B → REL A B → Set
x , y ∈ R = R x y
```

To implement proof-irrelevant relations in Agda, we could use functions of type A → B → Bool, where two elements x and y are considered related by R : A → B → Bool iff R x y ≡ true, but manipulating these equality proofs is cumbersome. There are other ways we could define proof-irrelevant relations as well.

[1] We use capital letters for REL because Rel is already defined in Agda as the type of homogeneous relations, i.e., Rel A = A → A → Set.

In order to package all of the data defining a relation into a single type, we define the record type RelObj:

```
record RelObj : Set₁ where
  constructor R[_,_,_]
  field
    fst : Set
    snd : Set
    rel : REL fst snd


open RelObj
```

An element R of RelObj is defined by a set fst R for the domain, a set snd R for the codomain, and a relation rel R on fst R and snd R. The statement open RelObj brings the fields fst : RelObj → Set, snd : RelObj → Set and rel : (R : RelObj) → REL (fst R) (snd R) into scope so they can be used in other definitions.

We define morphisms of relations by the record type:

```
record RelMorph (R S : RelObj) : Set₁ where
  constructor RM[_,_,_]
  field
    mfst : fst R → fst S
    msnd : snd R → snd S
    preserves : preservesRelObj R S mfst msnd
```

A morphism of relations consists of a pair of functions mfst and msnd and a proof that these functions preserve related elements, defined by the function preservesRelObj:

```
preservesRelObj : ∀ (R R' : RelObj)
                      → (f : fst R → fst R') → (f2 : snd R → snd R') → Set
preservesRelObj R R' f1 f2 = ∀ {x : fst R} {y : snd R} → (x , y ∈ rel R) → (f1 x , f2 y ∈ rel R')
```

The type preservesRelObj R R' f1 f2 asserts that for every x : fst R and y : snd R, a proof of x,y ∈ R implies (f1 x , f2 y) ∈ R'. Composition for morphisms of relations is defined by:

```
_∘RelM_ : ∀ {R S T : RelObj} → RelMorph S T → RelMorph R S → RelMorph R T
(RM[ g1 , g2 , pg ]) ∘RelM (RM[ f1 , f2 , pf ]) = RM[ (g1 ∘' f1) , (g2 ∘' f2) , (pg ∘' pf) ]
```

The functions for mfst and msnd are composed componentwise using function composition[2]. The proof that these compositions preserve related elements is defined by composing

```
pg : ∀{x : fst S}{y : snd S} → (x , y ∈ rel S) → (g1 x , g2y ∈ rel T)
```

and

```
pf : ∀{x : fst R}{y : snd R} → (x , y ∈ rel R) → (f1 x , f2 y ∈ rel S)
```

to get a proof of type:

```
∀{x : fst R}{y : snd R} → (x , y ∈ rel R) → (g1 (f1 x) , g2 (f2 y) ∈ rel T)
```

The identity morphisms of relations are defined by:

```
idRelMorph : ∀ {R} → RelMorph R R
idRelMorph = RM[ idf , idf , idf ]
```

The category of relations is defined in Agda by:

```
Rels : Category (lsuc lzero) (lsuc lzero) lzero
Rels = record
  { Obj = RelObj
  ; _⇒_ = RelMorph
  ; _≈_ = λ f g → (mfst f Sets.≈ mfst g) ×' (msnd f Sets.≈ msnd g)
  ; id = idRelMorph
  ; _∘_ = _∘RelM_
  ...
  }
```

Two morphisms of relations are considered equivalent if their component functions are equivalent in the category of sets. Note that the proofs of relation preservation are not considered when comparing

[2] We rename the composition function from the standard library from _∘_ to _∘'_ to distinguish it from the composition function associated with each category.

two morphisms for equality. The remaining fields are elided because they are given componentwise by the corresponding fields of Sets.

We give names to the k-ary product of Rels and the category of functors from the k-ary product of Rels to Rels :

```
Rels^ : ℕ → Category (lsuc lzero) (lsuc lzero) lzero
Rels^ k = Cat^ Rels k

[Rels^_,Rels] : ℕ → Category (lsuc lzero) (lsuc lzero) (lsuc lzero)
[Rels^ k ,Rels] = Functors (Rels^ k) Rels
```

We define the equality relation associated to each set by:

```
EqRelObj  : Set → RelObj
EqRelObj X = R[ X , X , _≡_ ]                                   (11)
```

The third component of EqRelObj is given by the propositional equality type _≡_, which means two elements of X are related iff they are identical, as desired.

7.2.1 Relation Transformers in Agda

In this section we formalize the definitions of relation transformers and higher-order relation transformers in Agda. We use slightly different definitions than those given in Section 7.1.1 to make relation transformers easier to work with in Agda. The alternative definitions we give are equivalent to those in Section 7.1.1. We first present the direct translations into Agda of the definitions in Section 7.1.1 to illustrate why we choose to give alternative definitions.

The type of relation transformers is defined in Agda by:

```
record RTObj (k : ℕ) : Set₁ where
  open Category Sets using (_≈_)
  field
    F1 : Functor (Sets^ k) Sets
    F2 : Functor (Sets^ k) Sets
    F* : Functor (Rels^ k) Rels

    over-rels1 : ∀ (Rs : Vec RelObj k) → fst (Functor.F₀ F* Rs) ≡ Functor.F₀ F1 (vecfst Rs)
    over-rels2 : ∀ (Rs : Vec RelObj k) → snd (Functor.F₀ F* Rs) ≡ Functor.F₀ F2 (vecsnd Rs)          (12)

    over-morphs1 : ∀ {Rs Ss : Vec RelObj k} (ms : (Rels^ k) [ Rs , Ss ])
                 → mfst (Functor.F₁ F* ms)
                   ≈ {!!} -- Goal:  fst (Functor.F₀ F* Rs) → fst (Functor.F₀ F* Ss)

    over-morphs2 : ∀ {Rs Ss : Vec RelObj k} (ms : (Rels^ k) [ Rs , Ss ])
                 → msnd (Functor.F₁ F* ms)
                   ≈ {!!} -- Goal:  snd (Functor.F₀ F* Rs) → snd (Functor.F₀ F* Ss)
```

Just as it was defined in Definition 7.1.1, a relation transformer consists of a triple of functors of the appropriate types and proofs that the actions of F* on objects and morphisms are over the corresponding actions of F1 and F2. The field over-rels1 expresses that, for every vector of relations Rs, the domain of $\mathsf{Functor.F}_0\,\mathsf{F*}\,\mathsf{Rs}$ is equal to the set given by $\mathsf{Functor.F}_0\,\mathsf{F1}\,(\mathsf{vecfst}\,\mathsf{Rs})$, where vecfst : ∀{k} → Vec RelObj k → Vec Set k applies fst : RelObj → Set to every RelObj in a vector. The field over-rels2 expresses an analogous proposition for vecsnd and snd. We also have the (incomplete) over-morphs1 and over-morphs2 fields that should correspond to the third bullet point in Definition 7.1.1. The over-morphs1 field should express the fact that the first component of the action of F* on morphisms in Rels^ k is equivalent to the action of F1 on the first components of these morphisms, i.e.,

$$\mathsf{mfst}\,(\mathsf{Functor.F}_1\,\mathsf{F*}\,\mathsf{ms}) \approx \mathsf{Functor.F}_1\,\mathsf{F1}\,(\mathsf{vecmfst}\,\mathsf{ms})$$

where vecmfst takes a morphism ms in Rels^ k and returns a morphism in Sets^ k by applying mfst componentwise to ms. But the equivalence of morphisms above does not type-check! As shown in a comment in (12), the expected type for the morphism being compared to $\mathsf{mfst}\,(\mathsf{Functor.F_1}\ \mathsf{F^*}\ \mathsf{ms})$ is

$$\mathsf{fst}\,(\mathsf{Functor.F_0}\ \mathsf{F^*}\ \mathsf{Rs}) \to \mathsf{fst}\,(\mathsf{Functor.F_0}\ \mathsf{F^*}\ \mathsf{Ss})$$

However, the morphism we would like to give for that hole is

$$\mathsf{Functor.F_1}\ \mathsf{F1}\,(\mathsf{vecmfst}\ \mathsf{ms}) : \mathsf{Functor.F_0}\ \mathsf{F1}\,(\mathsf{vecfst}\ \mathsf{Rs}) \to \mathsf{Functor.F_0}\ \mathsf{F1}\,(\mathsf{vecfst}\ \mathsf{Ss})$$

These types are *propositionally* equal because of the proof given by over-rels1, so it is possible to make this definition work. However, it would be much nicer if these types were *definitionally* equal because then we could compare these morphisms for equality directly. This would save us the trouble of manipulating equality proofs everywhere.

To sidestep this issue, we use the following definition of relation transformers that is equivalent to the one above:

```
record RTObj (k : ℕ) : Set₁  where
  constructor RT[_,_,_]
  open RelObj
  field
    F1 : Functor (Sets^ k) Sets
    F2 : Functor (Sets^ k) Sets
    F*Data : RTObj* F1 F2
```

This definition of RTObj consists of two set functors, F1 and F2, and some data defined by RTObj*:

```
record RTObj* {k : ℕ} (F1 F2 : Functor (Sets^ k) Sets) : Set₁ where
  open RelObj
  private module F1 = Functor F1
  private module F2 = Functor F2
  field
    F*Rel : ∀ (Rs : Vec RelObj k) → REL (F1.₀ (vecfst Rs)) (F2.₀ (vecsnd Rs))

    F*Rel-preserves : ∀ {Rs Ss : Vec RelObj k} → (ms : (Rels^ k) [ Rs , Ss ])
                      → preservesRel (F*Rel Rs) (F*Rel Ss) (F1.₁ (vecmfst ms)) (F2.₁ (vecmsnd ms))
```

The field F*Rel corresponds to the action on objects of F* from the original definition of RTObj, but it differs in that it gives a relation whose domain is F1$_0$ (vecfst Rs) and whose codomain is F2$_0$ (vecsnd Rs) *by definition*. Note that the return type of F*Rel is given by REL rather than RelObj because we need to give the domain and codomain explicitly. The field F*Rel-preserves gives a proof that the actions of F1 and F2 on morphisms preserve related elements between the relations given by F*Rel. F*Rel-preserves is given in terms of preservesRel, which is the analogue of preservesRelObj for elements of REL (as opposed to RelObj).

Given the data of RelObj*, we can define a functor of relations that corresponds to the F* field of the original definition of RTObj:

```
F*RelObj : ∀ (Rs : Vec RelObj k) → RelObj
F*RelObj Rs = R[ (F1._0 (vecfst Rs)) , (F2._0 (vecsnd Rs)) , (F*Rel Rs) ]

F*RelMap : ∀ {Rs Ss : Vec RelObj k} → (ms : (Rels^ k) [ Rs , Ss ]) → Rels [ F*RelObj Rs , F*RelObj Ss ]
F*RelMap {Rs} {Ss} ms = RM[ (F1._1 (vecmfst ms)) , (F2._1 (vecmsnd ms)) , F*Rel-preserves ms ]

F* : Functor (Rels^ k) Rels
F* =                                                                                    (13)
  record
    { F_0 = F*RelObj
    ; F_1 = F*RelMap
    ...
    }
```

The action of F* on objects is defined by F*RelObj, which applies the actions on objects of F1 and F2 for the set components and uses F*Rel for the relation component. The action on morphisms is defined by F*RelMap , which applies the actions on morphisms of F1 and F2 for the mfst and msnd fields and uses F*Rel-preserves for the relation-preserving proof.

It is not too difficult to see that these two definitions of relation transformers are equivalent. They both have fields F1 and F2, and for both definitions, the action of F* on morphisms is completely determined by F1 and F2. For the original definition, the action of F* on objects must satisfy over-rels1 and over-rels2. In our alternative definition, we must give the field F*Rel, but this is equivalent to satisfying over-rels1 and over-rels2 because it is defined directly in terms of F1 and F2. The field F*Rel-preserves is implicit in the original definition because the action of F* on morphisms returns a morphism of relations that must preserve related elements by definition. Since such a morphism of relations is (componentwise) equivalent (by over-morphs1 and over-morphs2) to those given by the actions of F1 and F2 on morphisms, it follows that the actions of F1 and F2 on morphisms preserve related elements.

The type of morphisms of relation transformers is defined by:

```
record RTMorph {k : ℕ} (H K : RTObj k) : Set₁ where
  constructor RTM[_,_,_]
  private module H = RTObj H
  private module K = RTObj K

  field
    d1 : NaturalTransformation H.F1 K.F1
    d2 : NaturalTransformation H.F2 K.F2

    d-preserves : ∀ {Rs : Vec RelObj k} → preservesRelObj (Functor.F₀ H.F* Rs) (Functor.F₀ K.F* Rs)
                                              (NaturalTransformation.η d1 (vecfst Rs))
                                              (NaturalTransformation.η d2 (vecsnd Rs))
```

This definition is a direct translation of Definition 7.1.2. With the original definition of RTObj, the d-preserves field would be another place that would require manipulating proofs of propositional equality. From the data of RTMorph, we can define a natural transformation from the functor of relations H* associated with H to the functor of relations K* associated with K:

```
    d* : NaturalTransformation (RTObj.F* H) (RTObj.F* K)
```

The components of d* are defined by the components of d1 and d2, and the naturality proofs are given componentwise in terms of the naturality proofs for d1 and d2.

Composition of morphisms of relation transformers is given componentwise:

```
compose-RTMorph : ∀ {k} {J K L : RTObj k} → RTMorph K L → RTMorph J K → RTMorph J L
compose-RTMorph RTM[ f1 , f2 , f-resp ] RTM[ g1 , g2 , g-resp ] = RTM[ f1 ∘v g1 , f2 ∘v g2 , f-resp ∘' g-resp ]
```

We use composition of natural transformations to compose the d1 and d2 fields and use regular function composition to compose the d-preserves fields. Identity morphisms of relation transformers are defined by:

```
idRTMorph : ∀ {k : ℕ} → (H : RTObj k) → RTMorph H H
idRTMorph {k} H = RTM[ idnat , idnat , idf ]
```

We have identity natural transformations for the d1 and d2 fields, and the identity function for the d-preserves field.

With these definitions in hand, we can define the category of k-ary relation transformers:

```
RTCat : ℕ → Category (lsuc lzero) (lsuc lzero) (lsuc lzero)
RTCat k = record
  { Obj = RTObj k
  ; _⇒_ = RTMorph
  ; _≈_ = λ m m' → [Sets^ k ,Sets] [ RTMorph.d1 m ≈ RTMorph.d1 m' ]
                ×' [Sets^ k ,Sets] [ RTMorph.d2 m ≈ RTMorph.d2 m' ]
  ; id = λ {H} → idRTMorph H
  ; _∘_ = compose-RTMorph
  ...
  }
```

Two morphisms of relation transformers are considered equivalent if the d1 and d2 are equivalent morphisms in the functor category [Sets^ k ,Sets]. The remaining fields are defined componentwise in terms of the corresponding fields in the definition of [Sets^ k ,Sets].

We also restructure the definition of higher-order relation transformers given in Definition 7.1.3 to reap the benefits of definitional equality. The type of higher-order relation transformers is defined in Agda by:

```
record HRTObj (k : ℕ) : Set₁ where
  constructor HRT[_,_,_]
  field
    H1 : Functor [Sets^ k ,Sets] [Sets^ k ,Sets]
    H2 : Functor [Sets^ k ,Sets] [Sets^ k ,Sets]

    H*Data : HRTObj* H1 H2
```

An element of HRTObj k consists of two higher-order set functors H1 and H2 and some data defined by the record type HRTObj*:

```
record HRTObj* {k : ℕ} (H1 H2 : Functor [Sets^ k ,Sets] [Sets^ k ,Sets]) : Set₁ where
  private module H1 = Functor H1
  private module H2 = Functor H2
  field

    H*RTRel : ∀ (Rt : RTObj k) (Rs : Vec RelObj k)
                → REL (Functor.F₀ (H1.₀ (RTObj.F1 Rt)) (vecfst Rs))
                      (Functor.F₀ (H2.₀ (RTObj.F2 Rt)) (vecsnd Rs))

    H*RTRel-preserves : ∀ (Rt : RTObj k) → {Rs Ss : Vec RelObj k}
                          → (ms : Rels^ k [ Rs , Ss ])
                          → preservesRel (H*RTRel Rt Rs) (H*RTRel Rt Ss)
                              (Functor.F₁ (H1.₀ (RTObj.F1 Rt)) (vecmfst ms))
                              (Functor.F₁ (H2.₀ (RTObj.F2 Rt)) (vecmsnd ms))

    H*RT-preserves : ∀ {Rt St : RTObj k} (m : RTMorph Rt St) → {Rs : Vec RelObj k}
                          → preservesRel (H*RTRel Rt Rs) (H*RTRel St Rs)
                              (NaturalTransformation.η (H1.₁ (RTMorph.d1 m)) (vecfst Rs))
                              (NaturalTransformation.η (H2.₁ (RTMorph.d2 m)) (vecsnd Rs))
```

Similarly to Definition 7.1.3, we are given functors H1 and H2. In contrast with Definition 7.1.3, we must *construct* the functor H* in terms of H1, H2, and the three fields of HRTObj*. The fields H*RTRel, H*RTRel-preserves, and H*RT-preserves correspond to the three implicit conditions listed in Definition 7.1.3.

From these data, we can construct a functor H* from the category of relation transformers to the functor category [Rels^ k ,Rels]:

```
H* : Functor (RTCat k) [Rels^ k ,Rels]
H* = record
        { F₀ = λ Rt → H*Obj Rt
        ; F₁ = λ {Rt} {St} d → H*-map d
        ; identity = (Functor.identity H1) , (Functor.identity H2)
        ; homomorphism = (Functor.homomorphism H1) , (Functor.homomorphism H2)
        ; F-resp-≈ = λ (f1≈g1 , f2≈g2) → (Functor.F-resp-≈ H1 f1≈g1) , (Functor.F-resp-≈ H2 f2≈g2)
        }
```

The action of H* on objects is defined by H*Obj

```
H*Obj : ∀ (Rt : RTObj k) → Functor (Rels^ k) Rels
H*Obj Rt@(RT[ F1 , F2 , F* ]) =
      RTObj.F* (RT[ H1.₀ F1
                  , H2.₀ F2
                  , record { F*Rel = H*RTRel Rt ; F*Rel-preserves = H*RTRel-preserves Rt } ])
```

H*Obj takes a relation transformer Rt[3] and produces a functor of relations by first applying H1 and H2 to the components of Rt and then using the F* construct defined in (13) to get a functor of relations. This definition of H*Obj ensures that the first implicit condition in Definition 7.1.3 is satisfied.

The action of H* on morphisms of relation transformers is defined by:

```
H*-map : ∀ {Rt St : RTObj k}
      → (d : RTMorph Rt St)
      → [Rels^ k ,Rels] [ H*Obj Rt , H*Obj St ]
H*-map {Rt} d =
  record { η          = λ Rs → H*-map-component d Rs
      ; commute = λ fs → H*-map-commutes d fs
      ; sym-commute = λ fs → Category.Equiv.sym Rels (H*-map-commutes d fs) }
```

H*-map takes a morphism of relation transformers from Rt to St and returns a natural transformation

[3] The Agda notation patternName@(pattern) gives a name to the pattern inside the parentheses. This allows us to refer to a term that we have pattern matched on by a shorter name rather than using the full pattern to refer to the term.

from H*Obj Rt to H*Obj St in the functor category [Rels^ k ,Rels]. The components of this natural transformation are defined by:

```
H*-map-component : ∀ {Rt St : RTObj k} → (d : RTMorph Rt St)
                     → (Rs : Vec RelObj k)
                     → RelMorph (Functor.F₀ (H*Obj Rt) Rs) (Functor.F₀ (H*Obj St) Rs)
H*-map-component d@(RTM[ d1 , d2 , d-preserves ]) Rs =
  RM[ (NaturalTransformation.η (H1.₁ d1) (vecfst Rs))
    , (NaturalTransformation.η (H2.₁ d2) (vecsnd Rs))
    , H*RT-preserves d ]
```

H*-map-commutes is defined by first applying the actions of H1 and H2 on morphisms to d1 and d2. This gives

```
H1.₁ d1 : NaturalTransformation (H1.₀ (RTObj.F1 Rt)) (H1.₀ (RTObj.F1 St))
```

and

```
H2.₁ d2 : NaturalTransformation (H2.₀ (RTObj.F2 Rt)) (H2.₀ (RTObj.F2 St))
```

We then take the component of H1.$_1$ d1 at vecfst Rs and the component of H2.$_1$ d2 at vecsnd Rs. The proof that these components (functions) preserve related elements is given by H*RT-preserves. The proofs of naturality for H*-map-commutes are defined by:

```
H*-map-commutes : ∀ {Rt St : RTObj k} → (d : RTMorph Rt St)
                    → {Rs Ss : Vec RelObj k}
                    → (fs : (Rels^ k) [ Rs , Ss ])
                    → Rels [ H*-map-component d Ss ∘RelM Functor.F₁ (H*Obj Rt) fs
                             ≈ Functor.F₁ (H*Obj St) fs ∘RelM H*-map-component d Rs ]
H*-map-commutes RTM[ d1 , d2 , _ ] fs =
                     NaturalTransformation.commute (H1.₁ d1) (vecmfst fs)
                   , NaturalTransformation.commute (H2.₁ d2) (vecmsnd fs)
```

The naturality proof is defined componentwise in terms of the naturality proofs for H1.$_1$ d1 on the morphism vecmfst fs and H2.$_1$ d2 on the morphism vecmsnd fs. The identity, homomorphism, and F-resp-≈ fields for H* are defined componentwise using the corresponding fields of H1 and H2.

The data of HRTObj* are also sufficient to define an endofunctor on the category of relation transformers.

```
HEndo : Functor (RTCat k) (RTCat k)
HEndo = record
             { F₀ = HEndo-obj
             ; F₁ = HEndo-map
             ; identity = (Functor.identity H1) , (Functor.identity H2)
             ; homomorphism = (Functor.homomorphism H1) , (Functor.homomorphism H2)
             ; F-resp-≈ = λ { (f1≈g1 , f2≈g2) → Functor.F-resp-≈ H1 f1≈g1 , Functor.F-resp-≈ H2 f2≈g2 }
             }
```

The action of HEndo on objects is defined by:

```
HEndo-obj : ∀ (Rt : RTObj k) → RTObj k
HEndo-obj Rt@(RT[ F1 , F2 , _ ]) = RT[ (H1.₀ F1) , (H2.₀ F2) , HEndo-rel* Rt ]
```

HEndo-obj takes a relation transformer and returns a relation transformer whose first two components are defined by applying the actions of H1 and H2 on objects componentwise. The third component is defined by:

```
HEndo-rel* : ∀ (Rt : RTObj k) → RTObj* (H1.₀ (RTObj.F1 Rt)) (H2.₀ (RTObj.F2 Rt))
HEndo-rel* Rt = record { F*Rel = H*RTRel Rt
                       ; F*Rel-preserves = H*RTRel-preserves Rt }
```

HEndo-rel* gives the action on relations for HEndo-obj and a proof that the actions of H1.$_0$ F1 and H2.$_0$ F2 on morphisms preserve related elements. These fields are defined by applying H*RTRel and H*RTRel-preserves to Rt.

The action of HEndo on morphisms is defined by:

```
HEndo-map : ∀ {Rt St : RTObj k} (d : RTMorph Rt St) → RTMorph (HEndo-obj Rt) (HEndo-obj St)
HEndo-map d@(RTM[ d1 , d2 , _ ]) = RTM[ (H1.₁ d1) , (H2.₁ d2) , H*RT-preserves d ]
```

HEndo-map takes a morphism of relation transformers and returns a morphism of relation transformers whose first two components are defined by applying the actions of H1 and H2 on morphisms componentwise to get the natural transformations H1.$_1$ d1 and H2.$_1$ d2. The third component is defined by

H*RT-preserves, which proves that, for every vector of relations Rs, the components of these natural transformations at vecfst Rs and vecsnd Rs preserve related elements.

The type of morphisms of higher-order relation transformers is defined by:

```
record HRTMorph {k : ℕ} (H K : HRTObj k) : Set₁ where
  constructor HRTM[_,_,_]
  private module H = HRTObj H
  private module K = HRTObj K

  open H using (H* ; H1 ; H2)
  open K renaming (H* to K* ; H1 to K1 ; H2 to K2)

  field
    σ1 : NaturalTransformation H1 K1
    σ2 : NaturalTransformation H2 K2

    σ-preserves : ∀ (Rt : RTObj k) (Rs : Vec RelObj k)
                → preservesRelObj (Functor.F₀ (Functor.F₀ H* Rt) Rs)
                                  (Functor.F₀ (Functor.F₀ K* Rt) Rs)
                   (NaturalTransformation.η
                     (NaturalTransformation.η σ1 (RTObj.F1 Rt)) (vecfst Rs))
                   (NaturalTransformation.η
                     (NaturalTransformation.η σ2 (RTObj.F2 Rt)) (vecsnd Rs))
```

This definition is a direct translation of 7.1.4. An element of HRTMorph consists of a pair of higher-order natural transformations and proof that appropriately typed components of these natural transformations preserve related elements. The field σ-preserves corresponds to the implicit condition of Definition 7.1.4.

Composition of morphisms of higher-order relation transformers is given componentwise:

```
composeHRTMorph : ∀ {k} {H K L : HRTObj k} → HRTMorph K L → HRTMorph H K
                 → HRTMorph H L
composeHRTMorph HRTM[ σ1 , σ2 , σ-preserves ] HRTM[ σ1' , σ2' , σ'-preserves ]
  = HRTM[ (σ1 ∘v σ1') , (σ2 ∘v σ2')
           , (λ Rt Rs xy∈HRtRs → σ-preserves Rt Rs (σ'-preserves Rt Rs xy∈HRtRs)) ]
```

The $\sigma 1$ and $\sigma 2$ fields are defined by composition of natural transformations, and the σ-preserves field is defined by function composition. Identity morphisms of higher-order relation transformers are defined by:

```
idHRTMorph : ∀ {k : ℕ} {H : HRTObj k} → HRTMorph H H
idHRTMorph {H} = HRTM[ idnat , idnat , (λ Rt Rs → idf) ]
```

The $\sigma 1$ and $\sigma 2$ fields are defined by identity natural transformations and the σ-preserves field is defined by the identity function.

With these definitions in hand, we can define the category of k-ary higher-order relation transformers:

```
HRTCat : ℕ → Category (lsuc lzero) (lsuc lzero) (lsuc lzero)
HRTCat k = record
                 { Obj = HRTObj k
                 ; _⇒_ = HRTMorph
                 ; _≈_ = λ {H} {K} m m' → (HRTMorph.σ1 m ≈ HRTMorph.σ1 m')
                                          ×'(HRTMorph.σ2 m ≈ HRTMorph.σ2 m')
                 ; id = idHRTMorph
                 ; _∘_ = composeHRTMorph
                 ...
                 }
                 where open Category ([[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]])
```

Two morphisms of higher-order relation transformers are considered equivalent if their components are equivalent in the higher-order functor category [[[Sets^ k ,Sets] , [Sets^ k ,Sets]]]. The remaining fields are defined componentwise in terms of the corresponding fields in the definition of [[[Sets^ k ,Sets] , [Sets^ k ,Sets]]].

7.2.2 Functors of Relation Transformers

Given the definitions of categories of relations and relation transformers in the previous sections, we can define several functors that go between these categories and their set analogues.

We have projection functors from the category of relations to the category of sets:

```
π₁Rels : Functor Rels Sets
π₁Rels = record
          { F₀ = fst
          ; F₁ = mfst
          ; identity = ≡.refl
          ; homomorphism = ≡.refl
          ; F-resp-≈ = λ { (f1≈g1 , _) → f1≈g1 }
          }

π₂Rels : Functor Rels Sets
π₂Rels = record
          { F₀ = snd
          ; F₁ = msnd
          ; identity = ≡.refl
          ; homomorphism = ≡.refl
          ; F-resp-≈ = λ { (_ , f2≈g2) → f2≈g2 }
          }
```

These projection functor π_1Rels (resp., π_2Rels) is defined by projecting out the fst and mfst (resp., snd and msnd) fields of relations and morphisms of relations. The remaining fields are straightforward to define because composition and identity for relations are defined componentwise by morphisms in Sets.

We also have a functor that takes a set and returns the corresponding equality relation:

```
Eq : Functor Sets Rels
Eq = record
      { F₀ = EqRelObj
      ; F₁ = EqRel-map
      ; identity = ≡.refl , ≡.refl
      ; homomorphism = ≡.refl , ≡.refl
      ; F-resp-≈ = λ f≈g → f≈g , f≈g
      }
```

The action on objects is defined above in (11). The action on morphisms is defined by:

```
EqRel-map : ∀ {X Y : Set} → (X → Y) → RelMorph (EqRelObj X) (EqRelObj Y)
EqRel-map f = RM[ f , f , (λ x≡y → ≡.cong f x≡y) ]
```

The action of Eq on a function f is given by pairing f with itself. The proof that this pair of functions preserves related (equal) elements is defined by applying ≡.cong to f and a proof of type x ≡ y to get a proof of type f x ≡ f y. Again, the remaining fields are straightforward to define because composition and identity for relations are defined componentwise by morphisms in Sets.

We also have the analogous functors for relation transformers:

```
π₁RT : ∀ {k} → Functor (RTCat k) [Setsˆ k ,Sets]
π₁RT {k} = record
                  { F₀ = RTObj.F1
                  ; F₁ = RTMorph.d1
                  ; identity = ≡.refl
                  ; homomorphism = ≡.refl
                  ; F-resp-≈ = λ { (f1≈g1 , _) → f1≈g1 }
                  }

π₂RT : ∀ {k} → Functor (RTCat k) [Setsˆ k ,Sets]
π₂RT {k} = record
                  { F₀ = RTObj.F2
                  ; F₁ = RTMorph.d2
                  ; identity = ≡.refl
                  ; homomorphism = ≡.refl
                  ; F-resp-≈ = λ { (_ , f2≈g2) → f2≈g2 }
                  }
```

These projection functors of relation transformers are defined by projecting out the appropriate fields of RTObj and RTMorph. The remaining fields are straightforward to define because composition and identities are given componentwise and because equality of morphisms in [Setsˆ k ,Sets] computes down to equality of morphisms in Sets.

We also have a functor that takes a functor of sets and produces the *equality relation transformer* associated to it. We elide the details of this construction because its definition is motivated by some abstract categorical reasoning, but it is a direct translation of the equality relation transformer construction given in [5].

```
EqRT : ∀ {k} → Functor [Sets^ k ,Sets] (RTCat k)
```

7.3 Relation Environments

In this section we define relation environments and discuss their formalization in Agda. Relation environments are analogous to set environments, except that instead of sending variables to functors in [Sets^ k ,Sets], relation environments send variables to relation transformers.

Definition 7.3.1. A *relation environment* maps each variable of arity k to a k-ary relation transformer. A morphism of relation environments $f : \rho \to \rho'$ only exists when ρ and ρ' are equal on all non-functorial variables. A morphism of relation environments $f : \rho \to \rho'$ sends each non-functorial variable ψ^k to the identity morphism on $\rho\psi^k = \rho'\psi^k$ and sends each functorial variable φ^k to a morphism of relation transformers from $\rho\varphi^k$ to $\rho'\varphi^k$. Composition of morphisms is given pointwise, i.e., $(g \circ f)\varphi^k = (g\varphi^k) \circ (f\varphi^k)$, where the second composition is a composition in the category of relation transformers. An identity morphism of relation environments maps a relation environment ρ to itself by sending every variable φ^k to the identity morphism on $\rho\varphi^k$. If ρ is a relation environment, we write $\pi_1\rho$ and $\pi_2\rho$ for the set environments mapping each type variable φ to the functors $(\rho\varphi)^1$ and $(\rho\varphi)^2$, respectively. If ρ is a set environment, we write Eq_ρ for the relation environment mapping each type variable φ to the equality relation transformer[4] $\mathsf{Eq}_{\rho\,\varphi}$. These data yield a category of relation environments, RelEnv.

7.3.1 Relation Environments in Agda

The definition of relation environments in Agda is exactly in parallel to the definition of set environments. In fact, we can generalize the constructions for set environments given in Section 4.2 to give a definition of environments such that each environment interprets variables in an arbitrary category. We then define set environments and relation environments by instantiating the generic construction with the appropriate categories. For set environments, we interpret variables of arity k as objects of [Sets^ k ,Sets]. For relation environments, we interpret variables of arity k as objects of RTCat k.

[4] We elide the formal definition of equality relation transformers from [5] in this thesis, but an equality relation transformer Eq_F can be thought of informally as the analogue of the equality relation on a set for a functor of Sets.

The definitions for relation environments are given the same names as their set analogues except "Set" is replaced with "Rel". For example the category of relation environments is called RelEnvCat. Its objects are called RelEnv and its morphisms are called RelEnvMorph.

We also have functors for going between set environments and relation environments. In particular, we have projection functors that take relation environments and return set environments:

π_1Env : Functor RelEnvCat SetEnvCat

π_2Env : Functor RelEnvCat SetEnvCat

Each of these functors is defined by applying the corresponding projection functor for relation transformers (π_1RT or π_2RT) pointwise to each variable in the relation environment. We also have a functor that takes a set environment and produces a relation environment defined in terms of equality relations and equality relation transformers:

EqEnv : Functor SetEnvCat RelEnvCat

This functor is defined by applying the functor EqRT pointwise to each variable in a set environment.

7.4 Interpreting Types as Relations

Before defining the relation semantics of types, let us recall the relation constructions we have defined so far and compare them to their set analogues. The following table shows the correspondence between the relation constructions and their analogues for sets:

In place of sets, the relation semantics uses relations to interpret types. In place of functions, we have morphisms of relations, i..e, pairs of functions that must preserve related elements. In place of k-ary functors of sets, we have k-ary relation transformers. In place of k-ary higher-order functors of sets, we have k-ary higher-order relation transformers. In place of set environments, we have relation environments that interpret variables of arity k as k-ary relation transformers. The fact that all of the relation constructions have the corresponding set constructions as their components is key to ensuring that the relation semantics is "over" the set semantics. Note that we use the letter ρ to denote both set environments and relation environments, but the type of ρ should be clear from context (otherwise we state its type explicitly). For example, if we write $\pi_1\rho$, then ρ must be a relation environment. If we write Eq_ρ, then ρ must be a set environment.

Set semantics	**Relation semantics**
$A : \mathsf{Sets}$	$(A, B, R) : \mathsf{Rels}$
$f : A \to A'$	$(f, g) : (A, B, R) \to (A', B', R')$
$F : [\mathsf{Sets}^k, \mathsf{Sets}]$	$(F^1, F^2, F^*) : RT_k$
$H : [[\mathsf{Sets}^k, \mathsf{Sets}], [\mathsf{Sets}^k, \mathsf{Sets}]]$	$(H^1, H^2, H^*) : HRT_k$
$\rho : \mathsf{SetEnv}$	$\rho : \mathsf{RelEnv}$

Table 7.1: Constructs used in the set semantics of types and their relational analogues.

Definition 7.4.1. Given a typing judgment $\Gamma; \Phi \vdash F$, we define its relational interpretation $\llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Rel}}$ as a functor from the category of relation environments to the category of relations. The action of the relational interpretation on a relation environment ρ is defined (in non-Agda syntax) by:

$$\begin{aligned}
\llbracket \Gamma; \Phi \vdash \mathbb{0} \rrbracket^{\mathsf{Rel}} \rho &= 0 \\
\llbracket \Gamma; \Phi \vdash \mathbb{1} \rrbracket^{\mathsf{Rel}} \rho &= 1 \\
\llbracket \Gamma; \emptyset \vdash \mathsf{Nat}^{\overline{\alpha}}\, F\, G \rrbracket^{\mathsf{Rel}} \rho &= \{ \eta : \lambda \overline{R}.\, \llbracket \Gamma; \overline{\alpha} \vdash F \rrbracket^{\mathsf{Rel}} \rho[\overline{\alpha := R}] \Rightarrow \lambda \overline{R}.\, \llbracket \Gamma; \overline{\alpha} \vdash G \rrbracket^{\mathsf{Rel}} \rho[\overline{\alpha := R}] \} \\
&= \{ (t, t') \in \llbracket \Gamma; \emptyset \vdash \mathsf{Nat}^{\overline{\alpha}}\, F\, G \rrbracket^{\mathsf{Set}} (\pi_1 \rho) \times \llbracket \Gamma; \emptyset \vdash \mathsf{Nat}^{\overline{\alpha}}\, F\, G \rrbracket^{\mathsf{Set}} (\pi_2 \rho) \mid \\
&\qquad \forall R_1 : \mathsf{Rel}(A_1, B_1) \ldots R_k : \mathsf{Rel}(A_k, B_k). \\
&\qquad\quad (t_{\overline{A}}, t'_{\overline{B}}) : \llbracket \Gamma; \overline{\alpha} \vdash F \rrbracket^{\mathsf{Rel}} \rho[\overline{\alpha := R}] \to \llbracket \Gamma; \overline{\alpha} \vdash G \rrbracket^{\mathsf{Rel}} \rho[\overline{\alpha := R}] \} \\
\llbracket \Gamma; \Phi \vdash \varphi \overline{F} \rrbracket^{\mathsf{Rel}} \rho &= (\rho \varphi) \overline{\llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Rel}} \rho} \\
\llbracket \Gamma; \Phi \vdash F + G \rrbracket^{\mathsf{Rel}} \rho &= \llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Rel}} \rho + \llbracket \Gamma; \Phi \vdash G \rrbracket^{\mathsf{Rel}} \rho \\
\llbracket \Gamma; \Phi \vdash F \times G \rrbracket^{\mathsf{Rel}} \rho &= \llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Rel}} \rho \times \llbracket \Gamma; \Phi \vdash G \rrbracket^{\mathsf{Rel}} \rho
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma;\Phi \vdash (\mu\varphi.\lambda\overline{\alpha}.H)\overline{G}]\!]^{\mathsf{Rel}}\rho &= (\mu T_{H,\rho})\overline{[\![\Gamma;\Phi \vdash G]\!]^{\mathsf{Rel}}\rho}\\
\text{where } T_{H,\rho} &= (T^{\mathsf{Set}}_{H,\pi_1\rho}, T^{\mathsf{Set}}_{H,\pi_2\rho}, T^{\mathsf{Rel}}_{H,\rho})\\
\text{and } T^{\mathsf{Rel}}_{H,\rho}\,F &= \lambda\overline{R}.[\![\Gamma;\varphi,\overline{\alpha} \vdash H]\!]^{\mathsf{Rel}}\rho[\varphi := F][\overline{\alpha := R}]\\
\text{and } T^{\mathsf{Rel}}_{H,\rho}\,\delta &= \lambda\overline{R}.[\![\Gamma;\varphi,\overline{\alpha} \vdash H]\!]^{\mathsf{Rel}} id_{\rho}[\varphi := \delta][\overline{\alpha := id_{\overline{R}}}]
\end{aligned}$$

The types $\mathbb{0}$ and $\mathbb{1}$ are interpreted as the empty relation 0 (the empty subset of the product of empty sets) and the singleton relation 1 (the singleton subset of the product of singleton sets), respectively. The Nat type is interpreted as a a subset of

$$[\![\Gamma;\emptyset \vdash \mathsf{Nat}^{\overline{\alpha}}\,F\,G]\!]^{\mathsf{Set}}(\pi_1\rho) \times [\![\Gamma;\emptyset \vdash \mathsf{Nat}^{\overline{\alpha}}\,F\,G]\!]^{\mathsf{Set}}(\pi_2\rho)$$

where natural transformations t and t' are related if, for every vector of relations $R_1 : \mathsf{Rel}(A_1, B_1)...R_k : \mathsf{Rel}(A_k, B_k)$ $t_{\overline{A}}$ and $t_{\overline{B}}$ preserve related elements from $[\![\Gamma;\overline{\alpha} \vdash F]\!]^{\mathsf{Rel}}\rho[\overline{\alpha := R}]$ to $[\![\Gamma;\overline{\alpha} \vdash G]\!]^{\mathsf{Rel}}\rho[\overline{\alpha := R}]$. The relational interpretation of type application is given by applying (the third component of) the relation transformer $\rho\varphi$ to the vector of relations $\overline{[\![\Gamma;\Phi \vdash f]\!]^{\mathsf{Rel}}\rho}$ given by interpreting the arguments of φ. The relational interpretation of a sum type is defined as the sum of the relations given by interpreting the summand types. Given relations $R : \mathsf{Rel}(A, B)$ and $R' : \mathsf{Rel}(A', B')$, the sum $R+R' : \mathsf{Rel}(A+A', B+B')$ is defined by:

$$R + R' = \{inl\,x, inl\,y \mid x, y \in R\} \cup \{inr\,x', inr\,y' \mid x', y' \in R'\}$$

That is, two elements are related by $R+R'$ if they come from the same summand (both *inl* or both *inr*) and are related by R or R'. The relational interpretation of a product type is defined as the product of the relations given by interpreting the components of the product. Given relations $R : \mathsf{Rel}(A, B)$ and $R' : \mathsf{Rel}(A', B')$, the product $R \times R' : \mathsf{Rel}(A \times A', B \times B')$ is defined by:

$$R \times R' = \{(x, x'), (y, y') \mid x, y \in R, x', y' \in R'\}$$

That is, two elements (pairs) are related by $R\times R'$ if their first components are related by R and their second components are related R'. The relational interpretation of μ types is defined by taking the fixpoint of the higher-order relation transformer $T_{H,\rho}$ and applying the fixpoint to the vector of relations given by interpreting the arguments $\overline{G}$ of the μ-type. The higher-order relation transformer $T_{H,\rho}$ is defined as a triple where first two components are given by the higher-order set functors $T^{\mathsf{Set}}_{H,\pi_1\rho}$ and $T^{\mathsf{Set}}_{H,\pi_2\rho}$. The third component is given by the higher-order functor of relations $T^{\mathsf{Rel}}_{H,\rho} : [[\mathsf{Rels}^k, \mathsf{Rels}], [\mathsf{Rels}^k, \mathsf{Rels}]]$, which is defined in terms of environment extension. We describe the fixpoint construction for higher-order relation transformers at the end of this chapter.

As desired, all of these relation constructions are “over” their set analogues. For example, the relational interpretation of natural transformations in relation environment ρ relates elements of the set interpretations of natural transformations in set environments $\pi_1\rho$ and $\pi_2\rho$. Since we have defined the relational interpretation of types this way, we can prove the following pair of lemmas, which define formally what it means for the relation interpretation to be “over” the set interpretation:

Lemma 7.4.1. *For every* $\Gamma; \Phi \vdash F$ *and relation environment* ρ*,*

$$\pi_1(\llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Rel}} \rho) = \llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Set}} \pi_1 \rho$$

$$\pi_2(\llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Rel}} \rho) = \llbracket \Gamma; \Phi \vdash F \rrbracket^{\mathsf{Set}} \pi_2 \rho$$

We formalize these “over-ness” lemmas in Agda in Chapter 8

7.4.1 Interpreting Types as Relations in Agda

The relational interpretation of types is defined in Agda by the function:

```
RelSem : ∀ {Γ : TCCtx} → {Φ : FunCtx} → {F : TypeExpr}
       → Γ ≀ Φ ⊢ F
       → Functor RelEnvCat Rels
```

The relational interpretation function RelSem takes a type F and a typing judgment $\Gamma \wr \Phi \vdash \mathsf{F}$ and returns a functor from the category of relation environments to the category of relations. It is defined by induction on the structure of the type formation rules, just as in Definition 7.4.1. We also have a function for interpreting vectors of type expressions and their typing judgments:

```
RelSemVec : ∀ {k : ℕ} {Γ : TCCtx} {Φ : FunCtx}
          → {Fs : Vec TypeExpr k}
          → foreach (λ F → Γ ≀ Φ ⊢ F) Fs
          → Functor RelEnvCat (Rels^ k)
```

RelSem and RelSemVec are defined by mutual induction, where RelSemVec applies RelSem to each type (and its typing judgment) in a vector of types.

The relational interpretations for 𝟘-I and 𝟙-I are defined as constant functors:

```
RelSem 𝟘-I = ConstF Rel⊥
RelSem 𝟙-I = ConstF Rel⊤
```

Rel⊥ and Rel⊤ are the empty relation and singleton relation, respectively. These relations are defined by:

```
Rel⊥ : RelObj
Rel⊥ = R[ ⊥ , ⊥ , (λ _ _ → ⊥) ]

Rel⊤ : RelObj
Rel⊤ = R[ ⊤ , ⊤ , (λ _ _ → ⊤) ]
```

The domain and codomain of the empty relation Rel⊥ are the empty set, and the set of proofs that two elements are related is also given by the empty set. The domain and codomain of the singleton relation Rel⊤ are the singleton set, and the set of proofs that the single element is related to itself is given by the singleton set. That is, there is exactly one proof that the singleton element is related to itself.

The relational interpretation for a product type is given by:

```
RelSem (×-I ⊢F ⊢G) = RelsProd ∘F (RelSem ⊢F ※ RelSem ⊢G)
```

This definition is analogous to the set interpretation for product types, but this definition uses the product functor

```
RelsProd : Functor (Product Rels Rels) Rels
```

for relations instead of the one for sets. The action of RelsProd on objects takes a pair of relations and returns their product. It is defined by:

```
_×REL_ : ∀ {A B A' B' : Set} → REL A B → REL A' B' → REL (A ×' A') (B ×' B')
(R0 ×REL S0) (x , x') (y , y') = R0 x y ×' S0 x' y'

_×Rel_ : RelObj → RelObj → RelObj
R ×Rel S = R[ (fst R ×' fst S) , (snd R ×' snd S) , (rel R ×REL rel S) ]
```

The domain and codomain of the product relation are given by componentwise products, and two pairs are related if their components are both related. The remaining fields for RelsProd are defined componentwise.

The relational interpretation for a sum type is given by:

```
RelSem (+-I ⊢F ⊢G) = RelsSum ∘F (RelSem ⊢F ※ RelSem ⊢G)
```

The sum functor for relations is defined by:

```
RelsSum : Functor (Product Rels Rels) Rels
```

The action of RelsSum on objects takes a pair of relations and returns their sum. It is defined by:

```
_+REL_ : ∀ {A B A' B' : Set} → REL A B → REL A' B' → REL (A ⊎ A') (B ⊎ B')
(R0 +REL S0) (inj₁ x) (inj₁ y) = R0 x y
(R0 +REL S0) (inj₂ x') (inj₂ y') = S0 x' y'
(R0 +REL S0) _ _ = ⊥

_+Rel_ : RelObj → RelObj → RelObj
R +Rel S = R[ (fst R ⊎ fst S) , (snd R ⊎ snd S) , (rel R +REL rel S) ]
```

The domain and codomain of the sum relation are given by componentwise sums (disjoint unions) of sets, and two elements are related if they come from the same summand (both inj$_1$ or both inj$_2$) and are related by the appropriate relation. If two elements come from different summands (i.e., inj$_1$ and inj$_2$ or vice versa), then they are not related. This is implemented by the third case of _+REL_, which says that the set of proofs that elements from different summands are related is empty. The remaining fields for RelsSum are defined componentwise.

The relational interpretations of type applications are defined by:

```
RelSem (AppT-I {φ = φ} Γ∋φ Gs ⊢Gs) = eval ∘F (VarRelSem-TC φ ※ RelSemVec ⊢Gs)
RelSem (AppF-I {φ = φ} Φ∋φ Gs ⊢Gs) = eval ∘F (VarRelSem-FV φ ※ RelSemVec ⊢Gs)
```

The functors VarRelSem-TC and VarRelSem-FV

```
VarRelSem-TC : ∀ {k : ℕ} (φ : TCVar k) → Functor RelEnvCat [Rels^ k ,Rels]
VarRelSem-FV : ∀ {k : ℕ} (φ : FVar k) → Functor RelEnvCat [Rels^ k ,Rels]
```

each take a type variable of arity k and return a functor from the category of relation environments to the category of k-ary functors of relations. The action of VarRelSem-TC on objects takes a relation environment and returns the k-ary functor of relations associated with the relation transformer given by applying ρ to φ. The action on morphisms takes a morphism f of relation environments and returns the natural transformation associated with the morphism of relation transformers given by applying f to φ. VarRelSem-FV is defined analogously.

The relational interpretation of a Nat type is defined by:

```
RelSem (Nat-I {αs = αs} ⊢F ⊢G) = NatTypeRelFunctor αs ⊢F ⊢G
```

NatTypeRelFunctor is defined analogously to the functor NatTypeFunctor introduced in Section 4.4. Its action on objects takes a relation environment and returns a relation defined by:

```
NatTypeRelObj : ∀ {k : ℕ} {Γ : TCCtx} {F G : TypeExpr}
                → (αs : Vec (FVar 0) k) (ρ : RelEnv)
                → (⊢F : Γ ≀ ∅ ,++ αs ⊢ F) → (⊢G : Γ ≀ ∅ ,++ αs ⊢ G)
                → RelObj
NatTypeRelObj {k} αs ρ ⊢F ⊢G =
  R[ NatTypeSem αs (Functor.F₀ π₁Env ρ) ⊢F ⊢G
   , NatTypeSem αs (Functor.F₀ π₂Env ρ) ⊢F ⊢G
   , (λ η1 η2 → NTs-preserve-rels αs ρ ⊢F ⊢G (NatTypeSem.nat η1) (NatTypeSem.nat η2)) ]
```

The domain and codomain of this relation are given by the set interpretation of Nat types in the set environments Functor.F$_0$ π_1Env ρ and Functor.F$_0$ π_2Env ρ. The third component is defined by:

```
{-# NO_UNIVERSE_CHECK #-}
record NTs-preserve-rels {k : ℕ} {Γ : TCCtx}
  {F G : TypeExpr} (αs : Vec (FVar 0) k) (ρ : RelEnv)
  (⊢F : Γ ≀ (∅ ,++ αs) ⊢ F) (⊢G : Γ ≀ (∅ ,++ αs) ⊢ G)
  (η1 : NaturalTransformation (extendSetSem-αs αs (Functor.F₀ π₁Env ρ) (SetSem ⊢F))
                              (extendSetSem-αs αs (Functor.F₀ π₁Env ρ) (SetSem ⊢G)))
  (η2 : NaturalTransformation (extendSetSem-αs αs (Functor.F₀ π₂Env ρ) (SetSem ⊢F))
                              (extendSetSem-αs αs (Functor.F₀ π₂Env ρ) (SetSem ⊢G)))
    : Set where

  field
    preserves :
```

```
∀ (Rs : Vec RelObj k)
→ preservesRelObj-prop-eq
    (Functor.F₀ (RelSem ⊢F) (Functor.F₀ (extendRelEnv-αs αs ρ) Rs))
    (Functor.F₀ (RelSem ⊢G) (Functor.F₀ (extendRelEnv-αs αs ρ) Rs))
    (nat-help1 αs ρ ⊢F Rs) (nat-help2 αs ρ ⊢F Rs)
    (nat-help1 αs ρ ⊢G Rs) (nat-help2 αs ρ ⊢G Rs)
    (NaturalTransformation.η η1 (vecfst Rs))
    (NaturalTransformation.η η2 (vecsnd Rs))
```

This record type consists of a single field preserves that expresses the condition for two natural transformations to be related given in the Nat case of Definition 7.4.1. This condition says that, for every vector of relations Rs, the components of η_1 and η_2 at vecfst Rs and vecsnd Rs must preserve related elements from the relational interpretation of ⊢F in an environment extended by Rs to the relational interpretation of ⊢G in an environment extended by Rs. Unfortunately, the components of the natural transformations here do not have the right types to comprise a morphism between these relations. But we can prove that the types of these components are propositionally equal to the desired types, so we can express this relation preservation property in terms of the function:

```
preservesRelObj-prop-eq : ∀ {A B A' B'} (R R' : RelObj)
  → A ≡ fst R → B ≡ snd R
  → A' ≡ fst R' → B' ≡ snd R'
  → (f : A → A') (g : B → B')
  → Set
preservesRelObj-prop-eq R R' ≡.refl ≡.refl ≡.refl ≡.refl f g = preservesRelObj R R' f g
```

The function preservesRelObj-prop-eq expresses that two functions preserve related elements between R and R' when the types of the functions are propositionally equal to the corresponding types of the relations. It is defined by pattern matching on the equality proofs, which causes the types of the relations and functions to match, and then calling preservesRelObj to express the usual relation-preserving condition. The proofs of equality for the instance of preservesRelObj-prop-eq above are given by the functions:

```
nat-help1 : ∀ {k : ℕ} {Γ : TCCtx} {Φ} {F : TypeExpr} (αs : Vec (FVar 0) k) (ρ : RelEnv)
  → (⊢F : Γ ≀ Φ ⊢ F)
  → (Rs : Vec RelObj k)
  → Functor.F₀ (extendSetSem-αs αs (Functor.F₀ π₁Env ρ) (SetSem ⊢F)) (vecfst Rs)
  ≡ RelObj.fst (Functor.F₀ (RelSem ⊢F) (Functor.F₀ (extendRelEnv-αs αs ρ) Rs))

nat-help2 : ∀ {k : ℕ} {Γ : TCCtx} {Φ} {F : TypeExpr} (αs : Vec (FVar 0) k) (ρ : RelEnv)
  → (⊢F : Γ ≀ Φ ⊢ F)
  → (Rs : Vec RelObj k)
  → Functor.F₀ (extendSetSem-αs αs (Functor.F₀ π₂Env ρ) (SetSem ⊢F)) (vecsnd Rs)
  ≡ RelObj.snd (Functor.F₀ (RelSem ⊢F) (Functor.F₀ (extendRelEnv-αs αs ρ) Rs))
```

In the syntax of [5], the types of these functions correspond to

$$[\![\Gamma;\overline{\alpha}\vdash F]\!]^{\mathsf{Set}}\pi_1\rho[\overline{\alpha:=\pi_1 R}] = \pi_1([\![\Gamma;\overline{\alpha}\vdash F]\!]^{\mathsf{Rel}}\rho[\overline{\alpha:=R}])$$

and

$$[\![\Gamma;\overline{\alpha}\vdash F]\!]^{\mathsf{Set}}\pi_2\rho[\overline{\alpha:=\pi_2 R}] = \pi_2([\![\Gamma;\overline{\alpha}\vdash F]\!]^{\mathsf{Rel}}\rho[\overline{\alpha:=R}])$$

The functions nat-help1 and nat-help2 are defined in terms of the proof that the relation semantics of types is over the set semantics of types.

The relational interpretation for a μ-type is given by:

```
RelSem (μ-I ⊢F Ks ⊢Ks) = MuRelSem ⊢F (RelSemVec ⊢Ks)
```

It is defined in terms of the function MuRelSem:

```
MuRelSem : ∀ {k : ℕ} {Γ : TCCtx} {H : TypeExpr}
                 → {φ : FVar k} {αs : Vec (FVar 0) k}
                 → Γ ≀ (∅ ,++ αs) ,, φ ⊢ H
                 → Functor RelEnvCat (Rels^ k) → Functor RelEnvCat Rels
MuRelSem {k} ⊢H SemKs =
  let fixRT : Functor RelEnvCat (RTCat k)
      fixRT = (fixHRT ∘F TEnvRT ⊢H)
```

```
    fixRels : Functor RelEnvCat [Rels^ k ,Rels]
    fixRels = embedRT ∘F fixRT
  in eval ∘F (fixRels ※ SemKs)
```

This definition has the same form as the definition of MuSem introduced in Section 4.4. Instead of taking a fixpoint of a higher-order functor of sets, we take the fixpoint of a higher-order relation transformer given by:

```
TEnvRT : ∀ {k : ℕ} {Γ : TCCtx} {H : TypeExpr}
           → {φ : FVar k} {αs : Vec (FVar 0) k}
           → Γ ≀ (∅ ,++ αs) ,, φ ⊢ H
           → Functor (RelEnvCat) (HRTCat k)
```

TEnvRT is defined in terms of environment extension and is analogous to the higher-order functor TSet introduced in Section 4.4. It corresponds to the higher-order relation transformer $T_{H,\rho}$ from Definition 7.4.1. In order to use eval to define a functor from RelEnvCat to Rels, we also must use the functor

```
embedRT : ∀ {k} → Functor (RTCat k) [Rels^ k ,Rels]
```

that embeds a relation transformer into the functor category [Rels^ k ,Rels]. The action of embedRT on objects takes a relation transformer F and returns the k-ary functor of relations F* associated with F. The action of embedRT on morphisms takes a morphism of relation transformers d and returns the natural transformation d* associated with d. The fixpoint construction for relation transformers

```
fixHRT : ∀ {k} → Functor (HRTCat k) (RTCat k)
```

is defined in the next section. Its action on objects takes a higher-order relation transformer and returns its fixed point, which is a relation transformer. The action on morphisms takes a morphism of higher-order relation transformers from H1 to H2 and returns a morphism of relation transformers from the fixpoint of H1 to the fixpoint of H2.

7.4.2 Fixpoints of Relation Transformers in Agda

In this section we present our Agda formalization of fixpoints of higher-order relation transformers. The goal of this development is to define the function

```
fixHRT : ∀ {k} → Functor (HRTCat k) (RTCat k)
```

that is used in the relational interpretation of μ-types. In contrast with the set semantics of μ-types, we do not simply take the fixpoint of a higher-order functor of relations. Instead, we take the fixpoint of a higher-order relation transformer, which plays the role of an endofunctor on the category of relation transformers, but also has some extra structure. The extra structure is defined by two higher-order functors of sets. If we have a higher-order relation transformer RH whose set components are H1, H2 : [[[Sets^ k ,Sets] , [Sets^ k ,Sets]]], then the fixpoint of RH will be a relation transformer that is over the fixpoints of H1 and H2.

We import the Agda module that implements fixpoints of set functors by the following statement:

```
import HFixSet as SetFix
```

This allows us to refer to the fixpoints of set functors throughout this section. The TERMINATING and NO_POSITIVITY_CHECK flags needed for the constructs below are explained after we define each of these constructs.

The action on objects of fixHRT is defined by fixHRT_0:

```
module object-fixpoint-rel {k : ℕ} where
  HFunc = Functor [Sets^ k ,Sets] [Sets^ k ,Sets]


  {-# TERMINATING #-}
  fixHRT₀ : ∀ (H1 H2 : HFunc) (RH : HRTObj* H1 H2) → RTObj k
  fixHRT₀ H1 H2 RH = RT[ (SetFix.fixH₀ H1) , (SetFix.fixH₀ H2) , (fixHRT₀* H1 H2 RH) ]
```

The function fixHRT_0 takes two higher-order set functors H1 and H2 and a higher-order relation transformer RH that is over H1 and H2 and returns a relation transformer. The first two components of the fixpoint relation transformer are defined to be the fixpoints of the higher-order set functors H1 and H2. The third component is defined by:

```
{-# TERMINATING #-}
fixHRT₀* : ∀ (H1 H2 : HFunc) (RH : HRTObj* H1 H2) → RTObj* (SetFix.fixH₀ H1) (SetFix.fixH₀ H2)
    fixHRT₀* H1 H2 RH =
      record { F*Rel = λ Rs x y → HRTFixRel H1 H2 RH Rs x y
             ; F*Rel-preserves = fix-preserves }
```

The function fixHRT$_0$* returns the data of the record type RTObj* (SetFix.fixH$_0$ H1) (SetFix.fixH$_0$ H2). The action on relations F*Rel is defined by the data type:

```
{-# NO_POSITIVITY_CHECK #-}
data HRTFixRel (H1 H2 : HFunc) (RH : HRTObj* H1 H2)
  : ∀ (Rs : Vec RelObj k)
    → REL (SetFix.HFixObj H1 (vecfst Rs)) (SetFix.HFixObj H2 (vecsnd Rs))
```

HRTFixRel is parameterized by H1, H2, and RH. Given a vector of relations Rs, the type

HRTFixRel H1 H2 RH Rs

gives a relation between the fixpoint of H1 applied to vecfst Rs and the fixpoint of H2 applied to vecsnd Rs. Just like its analogue HFixObj for sets, HRTFixRel has a single constructor defined by:

```
data HRTFixRel H1 H2 RH where
  rhin : ∀ {Rs : Vec RelObj k}
        → {x : Functor.F₀ (Functor.F₀ H1 (SetFix.fixH₀ H1)) (vecfst Rs)}
        → {y : Functor.F₀ (Functor.F₀ H2 (SetFix.fixH₀ H2)) (vecsnd Rs)}
        → RTObj.F*Rel
            ((HEndo-obj HRT[ H1 , H2 , RH ]) (fixHRT₀ H1 H2 RH))
            Rs x y
        → HRTFixRel H1 H2 RH Rs (SetFix.hin {k} {H1} {vecfst Rs} x)
                                 (SetFix.hin {k} {H2} {vecsnd Rs} y)
```

The constructor rhin takes a vector of relations Rs, an element x of

Functor.F$_0$ (Functor.F$_0$ H1 (SetFix.fixH$_0$ H1)) (vecfst Rs) an element y of

$$\mathsf{Functor.F_0\,(Functor.F_0\,H2\,(SetFix.fixH_0\,H2))\,(vecsnd\,Rs)}$$

and a proof that x and y are related. In particular, x and y must be related by

$$\mathsf{RTObj.F{*}Rel\ ((HEndo\text{-}obj\,(HRT[\,H1,\ H2,\ RH\,]))\,(fixHRT_0\,H1\ H2\ RH))\,Rs}$$

Let us break this type down into its components. The relation transformer ($\mathsf{fixHRT_0\,H1\ H2\ RH}$) is the fixpoint of RH, and

$$\mathsf{(HEndo\text{-}obj\,(HRT[\,H1,\ H2,\ RH\,]))} : \mathsf{RTObj\,k} \to \mathsf{RTObj\,k}$$

is the action on objects of the endofunctor on $\mathsf{RTCat\ k}$ associated with the higher-order relation transformer $\mathsf{HRT[\,H1,\ H2,\ RH\,]}$. Thus

$$\mathsf{(HEndo\text{-}obj\,(HRT[\,H1,\ H2,\ RH\,]))\,(fixHRT_0\,H1\ H2\ RH)} : \mathsf{RTObj\,k}$$

is the application of RH to its own fixpoint. The relation transformer above has an action on vectors of relations:

$$\mathsf{RTObj.F{*}Rel\ ((HEndo\text{-}obj\,(HRT[\,H1,\ H2,\ RH\,]))\,(fixHRT_0\,H1\ H2\ RH))}$$

and applying this action on relations to Rs gives a relation

$$\mathsf{RTObj.F{*}Rel\ ((HEndo\text{-}obj\,(HRT[\,H1,\ H2,\ RH\,]))\,(fixHRT_0\,H1\ H2\ RH))\,Rs}$$

with domain

$$\mathsf{Functor.F_0\,(Functor.F_0\,H1\,(SetFix.fixH_0\,H1))\,(vecfst\,Rs)}$$

and codomain

$$\mathsf{Functor.F_0\,(Functor.F_0\,H2\,(SetFix.fixH_0\,H2))\,(vecsnd\,Rs)}$$

If x and y are related by the above relation, then $\mathsf{SetFix.hin\,x}$ and $\mathsf{SetFix.hin\,y}$ are related by

$$\mathsf{HRTFixRel\ H1\ H2\ RH\ Rs}$$

There is a lot going on with this data type, but it is easier to understand if we compare it to its analogue HFixObj for sets. Recall the type of the constructor for HFixObj:

```
data HFixObj H where
  hin : ∀ {As : Vec Set k}
       → Functor.F₀ (Functor.F₀ H (fixH₀ H)) As
       → HFixObj H As
```

In non-Agda syntax, the type of the constructor hin can be read as: for every vector of sets $\overline{A}$, if $x \in (H(\mu H))\overline{A}$, then $hin\, x \in (\mu H)\overline{A}$. The constructor for HRTFixRel can be read similarly as: for every vector of relations $\overline{R : \mathsf{Rel}(A, B)}$, $x \in (H1(\mu H1))\overline{A}$, $y \in (H2(\mu H2))\overline{B}$, if $(x, y) \in (RH(\mu RH))\overline{R}$, then $(hin\, x, hin\, y) \in (\mu RH)\overline{R}$.

To finish the definition of fixHRT₀*, we still need to define fix-preserves for the F*Rel-preserves field of fixHRT₀*. This function is defined in the following where clause of fixHRT₀*:

```
where fix-preserves : ∀ {Rs Ss : Vec RelObj k} (ms : Rels^ k [ Rs , Ss ])
                      → preservesRel (HRTFixRel H1 H2 RH Rs) (HRTFixRel H1 H2 RH Ss)
                                     (Functor.₁ (SetFix.fixH₀ H1) (vecmfst ms))
                                     (Functor.₁ (SetFix.fixH₀ H2) (vecmsnd ms))
   fix-preserves ms (rhin p) = rhin (H*RTRel-preserves RH (fixHRT₀ H1 H2 RH) ms p)
```

The function fix-preserves takes a morphism ms from Rs to Ss in Rels^ k and produces a proof that the actions of SetFix.fixH₀ H1 and SetFix.fixH₀ H2 on morphisms preserve related elements from

HRTFixRel H1 H2 RH Rs

to

HRTFixRel H1 H2 RH Ss

It is defined by pattern matching on a proof that two elements are related in HRTFixRel H1 H2 RH Rs, for which the only possible constructor is rhin. The returned proof is also necessarily defined by the rhin constructor. The type of p and the goal type for the argument of rhin on the right-hand side are given in comments below:

```
where fix-preserves : ∀ {Rs Ss : Vec RelObj k} (ms : Rels^ k [ Rs , Ss ])
                     → preservesRel (HRTFixRel H1 H2 RH Rs) (HRTFixRel H1 H2 RH Ss)
                                    (Functor.₁ (SetFix.fixH₀ H1) (vecmfst ms))
                                    (Functor.₁ (SetFix.fixH₀ H2) (vecmsnd ms))
      fix-preserves ms (rhin p) = rhin {!!}
      -- Goal:  H*RTRel RH (fixHRT₀ H1 H2 RH) Ss
      --         (Functor.F₁ (Functor.F₀ H1 (SetFix.fixH₀ H1)) (vecmfst ms) x)
      --         ((Functor.F₁ (Functor.F₀ H2 (SetFix.fixH₀ H2)) (vecmsnd ms) y)
      --
      -- p :  H*RTRel RH (fixHRT₀ H1 H2 RH) Rs x y
```

We can prove the above goal by applying the H*RTRel-preserves field of the higher-order relation transformer RH to the relation transformer fixHRT$_0$ H1 H2 RH, the morphism ms, and p. The fact that fix-preserves relies on the definition of fixHRT$_0$, which is defined in terms of fix-preserves, explains the need for the TERMINATING flags above.

The action of fixHRT on morphisms is defined by:

```
{-# TERMINATING #-}
fixHRT₁ : ∀ (H1 H2 : HFunc) (RH : HRTObj* H1 H2)
         → (K1 K2 : HFunc) (RK : HRTObj* K1 K2)
         → HRTMorph HRT[ H1 , H2 , RH ] HRT[ K1 , K2 , RK ]
         → RTMorph (fixHRT₀ H1 H2 RH) (fixHRT₀ K1 K2 RK)
fixHRT₁ H1 H2 RH K1 K2 RK σ@(HRTM[ σ1 , σ2 , σ-preserves ]) =
  RTM[ SetFix.fixH₁ H1 K1 σ1
     , SetFix.fixH₁ H2 K2 σ2
     , (λ {Rs} → fixHRT₁-preserves H1 H2 RH K1 K2 RK σ Rs ) ]
```

The function fixHRT$_1$ takes a morphism of higher-order relation transformers from HRT[H1, H2, , RH] to HRT[K1, K2, , RK] and returns a morphism of relation transformers between their fixpoints. The d1 and d2 fields of the returned morphism are defined by applying the action of

```
fixH : Functor [[ [Sets^ k ,Sets] , [Sets^ k ,Sets] ]]  [Sets^ k ,Sets]
```

on morphisms componentwise to σ_1 and σ_2, giving two natural transformations in [Sets^ k ,Sets]. The proof that these natural transformations preserve related elements is given by:

```
fixHRT₁-preserves : ∀ (H1 H2 : HFunc) (RH : HRTObj* H1 H2)
                    → (K1 K2 : HFunc) (RK : HRTObj* K1 K2)
                    → (σ : HRTMorph HRT[ H1 , H2 , RH ] HRT[ K1 , K2 , RK ])
                    → (Rs : Vec RelObj k)
                    → RTMorph-preserves (fixHRT₀ H1 H2 RH) (fixHRT₀ K1 K2 RK)
                                        (SetFix.fixH₁ H1 K1 (HRTMorph.σ1 σ))
                                        (SetFix.fixH₁ H2 K2 (HRTMorph.σ2 σ)) Rs
```

Here, RTMorph-preserves expands to the type of the d-preserves field for morphisms of relation transformers. We use this function to make the type of fixHRT$_1$-preserves more concise. It is defined by:

```
RTMorph-preserves : ∀ {k : ℕ} (H K : RTObj k)
  → (d1 : NaturalTransformation (RTObj.F1 H) (RTObj.F1 K))
  → (d2 : NaturalTransformation (RTObj.F2 H) (RTObj.F2 K))
  → (Rs : Vec RelObj k) → Set
RTMorph-preserves H K d1 d2 Rs =
  preservesRelObj (Functor.F₀ (RTObj.F* H) Rs)
                  (Functor.F₀ (RTObj.F* K) Rs)
                  (NaturalTransformation.η d1 (vecfst Rs))
                  (NaturalTransformation.η d2 (vecsnd Rs))
```

We elide the full definition of fixHRT$_1$-preserves because it is somewhat long and technical, but it is defined by pattern matching:

```
fixHRT₁-preserves H1 H2 RH K1 K2 RK σ Rs (rhin p) = rhin {! !}
```

Here p, has the type H*RTRel RH (fixHRT$_0$ H1 H2 RH) Rs x y.

With these definitions in hand, we can complete the definition of fixHRT:

```
fixHRT : ∀ {k} → Functor (HRTCat k) (RTCat k)
fixHRT =
  record
    { F₀ = λ { HRT[ H1 , H2 , RH ] → fixHRT₀ H1 H2 RH }
    ; F₁ = λ { {HRT[ H1 , H2 , RH ]} {HRT[ K1 , K2 , RK ]} σ → fixHRT₁ H1 H2 RH K1 K2 RK σ }
    ; identity = λ { {HRT[ H1 , H2 , _ ]} → SetFix.fixH₁-identity H1 , SetFix.fixH₁-identity H2 }
    ; homomorphism = λ { {HRT[ H1 , H2 , _ ]} {HRT[ K1 , K2 , _ ]} {HRT[ J1 , J2 , _ ]} {f} {g}
                       → SetFix.fixH₁-homomorphism H1 K1 J1 (HRTMorph.σ1 f) (HRTMorph.σ1 g)
                       , SetFix.fixH₁-homomorphism H2 K2 J2 (HRTMorph.σ2 f) (HRTMorph.σ2 g) }
    ; F-resp-≈ = λ { {HRT[ H1 , H2 , _ ]} {HRT[ K1 , K2 , _ ]} {f} {g} (f1≈g1 , f2≈g2)
                  → (SetFix.fixH₁-F-resp H1 K1 (HRTMorph.σ1 f) (HRTMorph.σ1 g) f1≈g1)
                  , (SetFix.fixH₁-F-resp H2 K2 (HRTMorph.σ2 f) (HRTMorph.σ2 g) f2≈g2) }
    }
```

The identity, homomorphism, and F-resp-≈ fields of fixHRT are defined componentwise by the corresponding fields of its analogue fixH for sets.

Chapter 8

Parametricity

In this section we discuss parametricity for the calculus $\mathcal{N}$ and partially formalize some of the theorems required to prove our semantics is parametric.

Parametricity is a property expressing that a polymorphic function of type $\forall \alpha \rightarrow \mathsf{F}\,\alpha$ should behave the same way no matter the type we choose for α. For example, the polymorphic identity function in Agda

```
id : ∀ {A : Set} → A → A
id x = x
```

has the same behavior regardless of the type of its input. A more complex example is the function `map`

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f nil = nil
map f (cons x xs) = cons (f x) (map f xs)
```

that applies a function to every element in a list. The behavior of map is the same no matter how we instantiate the types A and B when applying `map` to a list. Parametric polymorphism is a very useful property of programming languages. For example, it can be used to define a program that safely uses a type independently of the type's implementation details, so the implementation can change without breaking the program.

Parametricity is formalized in terms of relations. It was first introduced by Reynolds for the polymorphic lambda calculus [9]. Types are given both a set semantics and a relation semantics, and for a semantics to be parametric, the interpretation of each term must send related inputs to related outputs. That is, the interpretation of a term of type $f : A \to B$ must send a pair of inputs related by the relational interpretation of the type A to a pair of outputs related by the relational interpretation of the type B. Parametricity has many interesting consequences, including so called "theorems for free" [10], which give a way to deduce properties of polymorphic programs solely from their types.

8.1 Parametricity for the Calculus $\mathcal{N}$

The key theorems used to prove a semantics is parametric are the Identity Extension Lemma and the Abstraction Theorem. The Identity Extension Lemma expresses that the relational intepretation of a type in an equality environment is equal to equality relation on the set interpretation of that type. The Identity Extension Lemma follows from the fact that if we interpret all type variables as equality relations, then these equality relations are preserved by the relational interpretations of types. The Identity Extension Lemma for the calculus $\mathcal{N}$ is given formally by:

Theorem 8.1.1. *If $\rho : \mathsf{SetEnv}$ and $\Gamma; \Phi \vdash F$ then $[\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Rel}} \mathsf{Eq}_\rho = \mathsf{Eq}_{[\![\Gamma;\Phi\vdash F]\!]^{\mathsf{Set}}\rho}$.*

The Abstraction Theorem expresses that for every term t, the set interpretations of t in related environments are related in the relational interpretation of t's type. The Abstraction Theorem for the calculus $\mathcal{N}$ follows from the fact that terms are interpreted as natural transformations and from the extra relation-preserving condition given in the set interpretation of Nat types. The Abstraction Theorem for $\mathcal{N}$ is given formally by:

Theorem 8.1.2. *If $\Gamma; \Phi \,|\, \Delta \vdash t : F$ and $\rho \in \mathsf{RelEnv}$, and if $(a, b) \in [\![\Gamma; \Phi \vdash \Delta]\!]^{\mathsf{Rel}}\rho$, then*
$([\![\Gamma; \Phi \,|\, \Delta \vdash t : F]\!]^{\mathsf{Set}}(\pi_1\rho)\, a\, , [\![\Gamma; \Phi \,|\, \Delta \vdash t : F]\!]^{\mathsf{Set}}(\pi_2\rho)\, b) \in [\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Rel}}\rho$

Although it is not always explicitly stated as a requirement for parametricity, we must also prove that the relation semantics of types is over the set semantics of types. That is, for every $\Gamma; \Phi \vdash F$ and relation environment ρ, we must have

$$\pi_1([\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Rel}}\rho) = [\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}\pi_1\rho$$

and

$$\pi_2([\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Rel}}\rho) = [\![\Gamma; \Phi \vdash F]\!]^{\mathsf{Set}}\pi_2\rho$$

In the next section we formalize these notions by giving their statements in Agda. Since we have not finished defining the term semantics, we do not prove the Abstraction Theorem. We partially prove the Identity Extension Lemma, but the case for μ-types in particular is highly nontrivial, and we have not yet determined how to adapt the corresponding proof in [5] for the Agda formalization.

8.2 Parametricity for the Calculus $\mathcal{N}$ in Agda

The first step to formalizing parametricity for the calculus $\mathcal{N}$ in Agda is to add the relation-preserving condition to the set interpretation of Nat types. To do this we add a new field to the record type defining the set interpretation of Nat types:

```
record NatTypeSem {k : ℕ} {Γ : TCCtx} {F G : TypeExpr}
  (αs : Vec (FVar 0) k) (ρ : SetEnv)
  (⊢F : Γ ≀ (∅ ,++ αs) ⊢ F) (⊢G : Γ ≀ (∅ ,++ αs) ⊢ G) : Set where
  constructor NatT[_,_]

  field
    nat : NaturalTransformation (extendSetSem-αs αs ρ (SetSem ⊢F))
                                (extendSetSem-αs αs ρ (SetSem ⊢G))

    preserves-relations :
      ∀ (Rs : Vec RelObj k)
      → preservesRelObj-prop-eq
          (Functor.F₀ (RelSem ⊢F) (Functor.F₀ (extendRelEnv-αs αs (Functor.F₀ EqEnv ρ)) Rs))
          (Functor.F₀ (RelSem ⊢G) (Functor.F₀ (extendRelEnv-αs αs (Functor.F₀ EqEnv ρ)) Rs))
          (nat-help1 αs (Functor.F₀ EqEnv ρ) ⊢F Rs)
          (nat-help2 αs (Functor.F₀ EqEnv ρ) ⊢F Rs)
          (nat-help1 αs (Functor.F₀ EqEnv ρ) ⊢G Rs)
          (nat-help2 αs (Functor.F₀ EqEnv ρ) ⊢G Rs)
          (NaturalTransformation.η nat (vecfst Rs))
          (NaturalTransformation.η nat (vecsnd Rs))
```

This preserves-relations field corresponds to the condition imposed on natural transformations in the set interpretation of Nat types in Definition 4.3.1. As we discussed in Section 7.4.1 when defining the relational interpretation of Nat types, the components of the natural transformation nat here do not have the right types to comprise a morphism between these relations. That is, the types of the components are not definitionally equal to the types of the domains and codomains of the relations. However, we can prove they are propositionally equal using the proofs given by the functions nat-help-1 and nat-help-2 introduced in Section 7.4.1. Using these proofs of equality, we can give the type of this field in terms of preservesRelObj-prop-eq. To prove the Abstraction Theorem, it is necessary to show that the set interpretations of terms of Nat type (e.g., map) satisfy this relation-preserving condition, but we leave this for future work since we have not yet formalized the semantics of these terms.

The next step in formalizing parametricity for the calculus $\mathcal{N}$ in Agda is to prove that the relational interpretations of types are over their set interpretations. This is proved by two separate functions:

```
SetSem-over-1 : ∀ {Γ} {Φ} {F} → (⊢F : Γ ≀ Φ ⊢ F) → ∀ (ρ : RelEnv)
                → Functor.F₀ (SetSem ⊢F) (Functor.F₀ π₁Env ρ)
                ≡ fst (Functor.F₀ (RelSem ⊢F) ρ)
SetSem-over-2 : ∀ {Γ} {Φ} {F} → (⊢F : Γ ≀ Φ ⊢ F) → ∀ (ρ : RelEnv)
                → Functor.F₀ (SetSem ⊢F) (Functor.F₀ π₂Env ρ)
                ≡ snd (Functor.F₀ (RelSem ⊢F) ρ)
```

We also have analogous functions for the interpretations of vectors of types:

```
SetSemVec-over-1 : ∀ {k} {Γ} {Φ} {Fs : Vec TypeExpr k} → (⊢Fs : foreach (Γ ≀ Φ ⊢_) Fs)
                   → ∀ (ρ : RelEnv)
                   → Functor.F₀ (SetSemVec ⊢Fs) (Functor.F₀ π₁Env ρ)
                   ≡ vecfst (Functor.F₀ (RelSemVec ⊢Fs) ρ)
SetSemVec-over-2 : ∀ {k} {Γ} {Φ} {Fs : Vec TypeExpr k} → (⊢Fs : foreach (Γ ≀ Φ ⊢_) Fs)
                   → ∀ (ρ : RelEnv)
                   → Functor.F₀ (SetSemVec ⊢Fs) (Functor.F₀ π₂Env ρ)
                   ≡ vecsnd (Functor.F₀ (RelSemVec ⊢Fs) ρ)
```

Below we only present the proof for SetSem-over-1 because the proof for SetSem-over-2 has exactly the same structure. The proof for SetSemVec-over-1 (resp., SetSemVec-over-2) is given by pattern matching on the input vector and applying SetSem-over-1 (resp., SetSem-over-2) to each type in the vector. The proof for SetSem-over-1 is given by:

```
SetSem-over-1 𝟘-I ρ = ≡.refl
SetSem-over-1 𝟙-I ρ = ≡.refl
SetSem-over-1 (AppT-I {φ = φ} Γ∋φ Fs ⊢Fs) ρ =
  ≡.cong (Functor.F₀ (RTObj.F1 (RelEnv.tc ρ φ))) (SetSemVec-over-1 ⊢Fs ρ)
SetSem-over-1 (AppF-I {φ = φ} Φ∋φ Fs ⊢Fs) ρ =
  ≡.cong (Functor.F₀ (RTObj.F1 (RelEnv.fv ρ φ))) (SetSemVec-over-1 ⊢Fs ρ)
SetSem-over-1 (+-I ⊢F ⊢G) ρ = ≡.cong₂ _⊎_ (SetSem-over-1 ⊢F ρ) (SetSem-over-1 ⊢G ρ)
SetSem-over-1 (×-I ⊢F ⊢G) ρ = ≡.cong₂ _×'_ (SetSem-over-1 ⊢F ρ) (SetSem-over-1 ⊢G ρ)
SetSem-over-1 (Nat-I ⊢F ⊢G) ρ = ≡.refl
SetSem-over-1 (μ-I ⊢F Gs ⊢Gs) ρ rewrite (SetSemVec-over-1 ⊢Gs ρ) = ≡.refl
```

Because of the way we set up the relational semantics of types, the proof of SetSem-over-1 is relatively straightforward. The 𝟘-I and 𝟙-I cases are trivial because the first component of Rel⊥ (resp., Rel⊤) is ⊥ (resp., ⊤) by definition.

For the AppT-I case, we must prove:

```
-- Goal:  Functor.F₀ (RTObj.F1 (SetEnv.tc ρ φ))
--         (Functor.F₀ (SetSemVec ⊢Fs) (Functor.F₀ π₁Env ρ))
--        ≡
--        Functor.F₀ (RTObj.F1 (SetEnv.tc ρ φ))
--         (vecfst (Functor.F₀ (RelSemVec ⊢Fs) ρ))
```

That is, we must prove that the results of applying the same function to different arguments are equal when the arguments are equal. We prove this using ≡.cong and SetSemVec-over-1. The case for AppF-I has the same structure.

For the sum case, we must prove:

```
-- Goal:  (Functor.F₀ (SetSem ⊢F) (Functor.F₀ π₁Env ρ)
--        ⊎
--        Functor.F₀ (SetSem ⊢G) (Functor.F₀ π₁Env ρ)
--        ≡
--        (fst (Functor.F₀ (RelSem ⊢F) ρ)
--        ⊎
--        fst (Functor.F₀ (RelSem ⊢G) ρ))
```

That is, we must prove that the two sums are equal when their summands are equal. We prove this using $\equiv$.cong$_2$ and the proofs of SetSem-over-1 for the summands. The case for products has the same structure.

For the Nat-I case, the proof is trivial. The goal we must prove is:

```
-- Goal:  NatTypeSem αs (NatEnv (Functor.F₀ π₁Env ρ)) ⊢F ⊢G
--       ≡ fst (NatTypeRelObj αs (RelEnvironments.NatEnv ρ) ⊢F ⊢G)
```

This goal reduces to:

```
-- Goal:  NatTypeSem αs
--        SetEnv[ (λ φ → RTObj.F1 (SetEnv.tc ρ φ)) , (λ _ → ConstF ⊤) ] ⊢F ⊢G
--        ≡
--        NatTypeSem αs
--        SetEnv[ (λ φ → RTObj.F1 (SetEnv.tc ρ φ)) , (λ _ → ConstF ⊤) ] ⊢F ⊢G
```

The left-hand side and right-hand side reduce to the same type because NatEnv (Functor.F_0 π_1Env ρ) and Functor.F_0 π_1Env (RelEnvironments.NatEnv ρ)[1] are equivalent. Here RelEnvironments.NatEnv maps every functorial variable to the constant Rel$\top$ relation transformer, whose set functor components are given by ConstF $\top$.

The μ-I case has the same structure as the AppT-I and AppF-I cases. The goal we must prove is:

```
-- Goal:  Functor.F₀ (MuSem ⊢F (SetSemVec ⊢Gs)) (Functor.F₀ π₁Env ρ)
--        ≡ fst (Functor.F₀ (MuRelSem ⊢F (RelSemVec ⊢Gs)) ρ)
```

This goal reduces to:

```
-- Goal:  (HFixObj ...  )  (Functor.F₀ (SetSemVec ⊢Gs) (Functor.F₀ π₁Env ρ))
--        ≡ (HFixObj ...  )  (vecfst (Functor.F₀ (RelSemVec ⊢Gs) ρ))
```

Here, the two occurrences of HFixObj ... are equal because of the way we defined fixpoints of relation transformers to be over the fixpoints of their set components. We elide the argument to HFixObj because it is a somewhat long type. Again, we have the same function applied to two different arguments, and we know the arguments are equal. We could prove this using $\equiv$.cong as we did for AppT-I and AppF-I

[1] This occurence of π_1Env comes from the definition of NatTypeRelObj.

above, but instead we opt to use Agda's rewrite construct. The rewrite construct takes a proof of equality of the form lhs ≡ rhs and tells Agda to update any goals by replacing lhs with rhs in each goal. Using rewrite is more convenient than ≡.cong in this case because it means we can avoid writing out the type represented by HFixObj .... It also happens to type-check faster than a version using ≡.cong. After invoking rewrite, the goal becomes:

```
-- Goal:  (HFixObj ...  )  (Functor.F₀ (SetSemVec ⊢Gs) (Functor.F₀ π₁Env ρ))
--       ≡ (HFixObj ...  )  (Functor.F₀ (SetSemVec ⊢Gs) (Functor.F₀ π₁Env ρ))
```

Now the left-hand side and right-hand side of the equality we would like to prove are identical, so we can prove it using ≡.refl.

The proof for SetSem-over-2 is identical to the one we just presented for SetSem-over-1. The only difference is that the goal types involve snd and π_2Env rather than fst and π_1Env.

The statement of the Identity Extension Lemma is given in Agda as:

```
IEL : ∀ {k : ℕ} {Γ} {Φ} {F} (ρ : SetEnv)
    → (⊢F : Γ ≀ Φ ⊢ F)
    → Functor.F₀ (RelSem ⊢F) (Functor.F₀ EqEnv ρ)
    ≡ Functor.F₀ Eq (Functor.F₀ (SetSem ⊢F) ρ)
```

In order to prove these relations are equal, we must give proofs of equality for their domains, their codomains, and their underlying relations. We can give the proofs that their domains and their codomains are equal in terms of SetSem-over-1 and SetSem-over-2. The difficult part is in proving that the relations themselves are equal. In particular, we must prove that if two elements x and y are related by the relational interpretation of a type with respect to an equality environment, then x and y must be equal. That is, we would like to prove something along the lines of:

```
IEL-eq : ∀ {Γ} {Φ} {F} (ρ : SetEnv)
       → (⊢F : Γ ≀ Φ ⊢ F)
       → ∀ {x y}
       → x , y ∈ rel (Functor.F₀ (RelSem ⊢F) (Functor.F₀ EqEnv ρ))
       → x ≡ y
```

But the equality type in the last line above does not type-check, because the type of x

```
fst (Functor.F₀ (RelSem ⊢F) (Functor.F₀ EqEnv ρ))
```

and the type of y

```
snd (Functor.F₀ (RelSem ⊢F) (Functor.F₀ EqEnv ρ))
```

are not definitionally equal. To make this equality type-check, we can use the *heterogeneous* equality type. Heterogeneous equality is defined in Agda's standard library by:

```
data _≅_ {a b : Level} {A : Set a} (x : A) : {B : Set b} → B → Set a where
  refl : x ≅ x
```

Heterogeneous equality allows us to compare two elements of different types for equality, but like the type of propositional equality, it only has a constructor for x ≅ x. Using heterogeneous equality, we can express the type of IEL-eq as:

```
IEL-eq : ∀ {Γ} {Φ} {F} (ρ : SetEnv)
         → (⊢F : Γ ≀ Φ ⊢ F)
         → ∀ {x y}
         → x , y ∈ rel (Functor.F₀ (RelSem ⊢F) (Functor.F₀ EqEnv ρ))
         → x ≅ y
```

We can prove the cases of IEL-eq for 𝟘-I and 𝟙-I trivially because their interpretations always yield the same relation (or set, for the set semantics). We can prove the cases for sums and products by straightforward induction. The remaining cases are nontrivial and have not yet been proved in Agda.

To state the Abstraction Theorem in Agda, we must first define a relational interpretation of term contexts. We define this analogously to the set interpretation of term contexts as the product of the functors interpreting the individual types in the term context:

```
ContextInterpRel : ∀ {Γ : TCCtx} {Φ : FunCtx} → TermCtx Γ Φ
                   → Functor RelEnvCat Rels
ContextInterpRel Δ∅ = ConstF Rel⊤
ContextInterpRel (Δ ,- _ : ⊢F ⟨ _ ⟩) = ContextInterpRel Δ ×Rels RelSem ⊢F
```

Like the type of IEL-eq above, the type of the Abstraction Theorem also requires some reasoning about equality to make things type-check. We might try to state the Abstraction Theorem as:

```
AbstractionThm : ∀ {Γ : TCCtx} {Φ : FunCtx} {Δ : TermCtx Γ Φ}
                 → {F : TypeExpr} → {⊢F : Γ ≀ Φ ⊢ F}
                 → {t : TermExpr}
                 → (⊢t : Γ ≀ Φ ∣ Δ ⊢ t ∶ ⊢F)
                 → (ρ : RelEnv)
                 → ∀ {a : Functor.F₀ (ContextInterp Δ) (Functor.F₀ π₁Env ρ)}
                     {b : Functor.F₀ (ContextInterp Δ) (Functor.F₀ π₂Env ρ)}
                 → a , b ∈ rel (Functor.F₀ (ContextInterpRel Δ) ρ)
                 →   NaturalTransformation.η (TermSetSem ⊢t) (Functor.F₀ π₁Env ρ) a
                   , NaturalTransformation.η (TermSetSem ⊢t) (Functor.F₀ π₂Env ρ) b
                     ∈ rel (Functor.F₀ (RelSem ⊢F) ρ)
```

There are two issues with this type. First, for a and b to be given as arguments to the components of the natural transformations in this type, they must have the types they are given above. But the types of a and b are not definitionally equal to the domain and codomain of (Functor.F$_0$ (ContextInterpRel Δ) ρ), so the last argument before the return type does not type-check. Second, the types of the terms related in the return type are not definitionally equal to the domain and codomain of (Functor.F$_0$ (RelSem ⊢F) ρ). To resolve these issues, we introduce the function:

```
_,_∈[_][_]_ : ∀ {A B C D : Set} → C → D → (A ≡ C) → (B ≡ D) → REL A B → Set
x , y ∈[ ≡.refl ][ ≡.refl ] R = x , y ∈ R
```

This function expresses that two terms x and y are related by R when the type of x is propositionally equal to the domain of R and the type of y is propositionally equal to the codomain of R. Using this function, we can give a well-typed statement of the Abstraction Theorem as:

```
AbstractionThm : ∀ {Γ : TCCtx} {Φ : FunCtx} {Δ : TermCtx Γ Φ}
                 → {F : TypeExpr} → {⊢F : Γ ≀ Φ ⊢ F}
                 → {t : TermExpr}
                 → (⊢t : Γ ≀ Φ ∣ Δ ⊢ t ∶ ⊢F)
                 → (ρ : RelEnv)
→ ∀ {a : Functor.F₀ (ContextInterp Δ) (Functor.F₀ π₁Env ρ)}
    {b : Functor.F₀ (ContextInterp Δ) (Functor.F₀ π₂Env ρ)}

→ a , b ∈[ ContextInterp-over-1 Δ ρ ][ ContextInterp-over-2 Δ ρ ]
      rel (Functor.F₀ (ContextInterpRel Δ) ρ)

→   NaturalTransformation.η (TermSetSem ⊢t) (Functor.F₀ π₁Env ρ) a
  , NaturalTransformation.η (TermSetSem ⊢t) (Functor.F₀ π₂Env ρ) b
    ∈[ ≡.sym (SetSem-over-1 ⊢F ρ) ][ ≡.sym (SetSem-over-2 ⊢F ρ) ]
    rel (Functor.F₀ (RelSem ⊢F) ρ)
```

The proofs of equality for the return type are given in terms of SetSem-over-1 and SetSem-over-2. We must also use ≡.sym there to swap the order of the arguments to the equality type. The proofs of equality for the last argument of AbstractionThm give that the the domain of

```
Functor.F₀ (ContextInterpRel Δ) ρ
```

is equal to the type of a and that its codomain is equal to the type of b. These proofs are given by:

```
ContextInterp-over-1 : ∀ {Γ : TCCtx} {Φ : FunCtx} (Δ : TermCtx Γ Φ)
                      → (ρ : RelEnv)
                      → fst (Functor.F₀ (ContextInterpRel Δ) ρ)
                      ≡ Functor.F₀ (ContextInterp Δ) (Functor.F₀ π₁Env ρ)
ContextInterp-over-2 : ∀ {Γ : TCCtx} {Φ : FunCtx} (Δ : TermCtx Γ Φ)
                      → (ρ : RelEnv)
                      → snd (Functor.F₀ (ContextInterpRel Δ) ρ)
                      ≡ Functor.F₀ (ContextInterp Δ) (Functor.F₀ π₂Env ρ)
```

The definition of ContextInterp-over-1 (resp., ContextInterp-over-2) is given by pattern matching on its input term context and using SetSem-over-1 (resp., SetSem-over-2) on each type in that term context.

Chapter 9

Conclusion

In this thesis, we formalized the syntax of the calculus $\mathcal{N}$ and partially formalized its semantics. Our formalization closely corresponds to the development in [5], but it deviates from the definitions given there in a few places. For example, our definition of the semantics of types corresponds exactly to the semantics of types presented in [5] except in the intepretation of μ-types. There, the fixpoint of a functor is defined in terms of a categorical construct called a colimit. To express fixpoints of functors in our formalization, we use an explicit construction based on an inductive data type in Agda. Another place our formalization deviates from [5] is in the interpretation of Nat types. In Agda, we have an extra step to explicitly "forget" the functorial variables in an environment (by sending them all to a constant functor on the singleton set) before using the environment to interpret Nat types. This extra step makes it easier to show that the interpretation of Nat types is a functor.

We have seen that Agda is very useful for doing bookkeeping tasks such as keeping track of proof goals and verifying proofs for correctness. But, by design, Agda is very pedantic, and many issues that can be elided on paper require tedious extra steps in Agda because every detail must be explicitly formalized. This extreme attention to detail can be both a useful tool and a drawback. For example, when we formalized the syntax of types, we identified (with the help of Agda) a subtle error in the syntax of μ-types that actually resulted in a change in [5]. On the other hand, there are several places throughout the formalization where we have proofs of propositional equality for two types, but we have to go through extra proof steps because the types are not definitionally equal. These extra steps are not required outside of Agda, because if we prove an equality of two objects on paper then we can just use the equal objects interchangeably.

9.1 Future Work

There are several possible directions for extending and improving the formalization presented in this thesis.

The most obvious direction for future work is to refactor the syntax of types to address the issues described at the end of Section 6.1.1. This would allow us to finish defining the set semantics of terms. In particular, we need to be able to express the fact that type contexts have at most one occurrence of each variable. This should allow us to prove that (syntactic) substitution of multiple variables interacts correctly with (semantic) environment extension of multiple variables. Another solution to this substitution issue could be to keep track of the variables that are actually *used* in a type, since variables that appear in a type's context are not necessarily used in that type. For example, the type $\mathbb{1}$ can be formed in a context with any variables, but these variables have no effect on the interpretation of the type $\mathbb{1}$. If we keep track of the variables actually used in each type, it might be easier to show that substitution and environment extension have the desired interaction.

With a complete definition of the term semantics, we could attempt to verify that the set interpretation of a term in $\mathcal{N}$ is equivalent (extensionally equal) to the corresponding Agda term. For example, we could attempt to verify that the interpretation of the map term for lists is equivalent to the map function for the list type in Agda.

Another direction for future work is to finish proving the Identity Extension Lemma and to prove the Abstraction Theorem. Also, if we prove both of these then we should be able to derive some free theorems for the types of $\mathcal{N}$.

To make the proofs of the Identity Extension Lemma and Abstraction Theorem more straightforward, we could potentially refactor the relation semantics of types so that it is over the set semantics of types by definition. If we did this, then we could state the Identity Extension Lemma without having to resort to heterogeneous equality. It would also eliminate the need for the propositional equality proofs in the statement of the Abstraction Theorem and in the set and relational semantics of Nat types.

If we were to finish defining the set semantics of terms, the Identity Extension Lemma, and the Abstraction Theorem, there are several possible directions for extending the calculus and model. For example, we could add more expressive recursion combinators to the term syntax, since the fold construct only captures a specific form of recursion, as discussed in [5]. Another possibility would be to generalize the semantics to interpret types in other categories, as many of the constructions in our formalization can be defined for categories with sufficiently Sets-like structure. We could also extend the type system of $\mathcal{N}$ to include richer classes of data types beyond nested types.

Bibliography

[1] Awodey, S. *Category Theory*, 2nd ed. Oxford University Press, Inc., USA, 2010.

[2] de Bruijn, N. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae 75*, 5 (1972), 381–392.

[3] Hu, J. Z. S., and Carette, J. Proof-relevant category theory in Agda. *CoRR abs/2005.07059* (2020).

[4] Jeffries, D. Agda code for this thesis. `https://github.com/jeffriesd/P4NT`.

[5] Johann, P., Ghiorzi, E., and Jeffries, D. Parametricity for primitive nested types. In *Foundations of Software Science and Computation Structures* (2021), pp. 324–343.

[6] Johann, P., and Polonsky, A. Higher-kinded data types: Syntax and semantics. In *Logic in Computer Science* (2019), pp. 1–13.

[7] Norell, U. The Agda user manual. `https://agda.readthedocs.io`.

[8] Norell, U. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (2008), p. 230–266.

[9] Reynolds, J. C. Types, abstraction, and parametric polymorphism. *Information Processing 83(1)* (1983), 513–523.

[10] Wadler, P. Theorems for free! In *Functional Programming Languages and Computer Architecture* (1989), pp. 347–359.

Vita

Daniel Jeffries was born in Stokesdale, North Carolina, to John and Valerie Jeffries. He graduated from Dalton L. McMichael High School in June 2015. In the fall of 2017, he enrolled at Appalachian State University to study Computer Science, and in December 2019 he was awarded the Bachelor of Science degree. In Spring of 2020, he accepted a research assistantship in Computer Science at Appalachian State University and began study toward a Master of Science degree.