KU LEUVEN

FACULTY OF SCIENCE

Reasoning about Effect Parametricity Using Dependent Types

Joris CEULEMANS

Supervisor: Prof. dr. D. Devriese
KU Leuven (guest professor) and
Vrije Universiteit Brussel

Mentor: A. Nuyts
KU Leuven

Thesis presented in
fulfillment of the requirements
for the degree of Master of Science
in Mathematics

Academic year 2018-2019

Preface

Of course, one cannot write a master's thesis all by oneself without any help. As it would be a bit strange to express my gratitude in English towards people who share my mother tongue, the following paragraphs will be in Dutch.

Ten eerste wil ik mijn promotor Dominique Devriese en begeleider Andreas Nuyts enorm bedanken om mij de kans te geven me te verdiepen in dit boeiende onderwerp, om zeer uitgebreid te antwoorden op al mijn vragen, om allerlei administratieve zaken te regelen en om verschillende kladversies van dit werk na te lezen en becommentariëren. Ook bedankt om mij de kans te geven naar de Midlands Graduate School in Birmingham te gaan, wat een zeer leerrijke ervaring was.

Verder wil ik mijn lezers, Frank Piessens en Wim Veys, bedanken voor het lezen van dit werk, het bijwonen van de presentaties en het in vraag stellen waarom de return en bind operaties in een premonad parametrisch zijn (en of daar parametriciteitsresultaten uit volgen), wat uiteindelijk geleid heeft tot sectie 3.3.

Ten slotte gaat mijn oprechte dank uit naar mijn ouders en mijn zus, die het thuis altijd weer aangenaam maakten om aan deze thesis te werken door te zorgen voor ontspanning tussendoor en achteraf.

Bedankt iedereen!
Joris

Abstract

Before discussing the general aim of this thesis and giving some preview of the results, we first explain some words in the title more carefully.

Types are a tool present in a lot of programming languages and they prevent programmers from making some kinds of errors. For instance a type checker will complain when a function that expects an argument of type `int` is invoked with a value of type `string`. Of course type systems vary in strength and some languages provide types that can depend on other types. Java for example provides a generic type `ArrayList<T>` of lists with elements of type `T`. The advantage of such a type is that information on the type of elements in a list is stored in the type of that list, but at the same type list operations such as `isEmpty()` can be implemented once and work for all lists, irrespective of the type `T`. Moreover, the implementation of an operation such as `isEmpty()` does not inspect the type `T` and hence applies the same algorithm for every list. In this thesis, we will call functions such as `isEmpty()` that work for any type `T` polymorphic and if they additionally apply the same algorithm for any type, they are called **parametric**.

When we know of a certain polymorphic function that it is parametric, that information strongly restricts its possible behaviour and we can wonder whether there are some results that can be deduced from it. This is exactly what Philip Wadler did for System F, a theoretical programming language with support for parametric polymorphism [Wad89]. He developed a procedure to derive from a given type of parametric polymorphic functions a theorem that is satisfied by all functions of that type. This is a powerful feature for reasoning about programs.

The purpose of any practical computer program is to cause some **effect** (also called a side effect) in the world outside of that program. It could for instance write some output to the screen, read a file or send information over a network. Functions that do not cause any side effect are called pure and they can only take some arguments as input and produce some value as output. In this thesis we are mainly interested in purely functional programming languages, such as Haskell and Agda, in which every regular function is required to be pure. To write practically useful programs that cause side effects in such a language, we can use monads, a concept that originated in category theory. Different monads will give rise to different kinds of side effects.

It is then interesting to consider effect polymorphic functions, which are functions that are polymorphic not in the sense that they work for any type but in the sense that they work for any monad. In other words, these are functions that take some effectful computations as argument but for which the kind of side effect in those computations does not matter. An example of such an effect polymorphic function is the Haskell function `sequence` that takes a list of effectful computations and performs them sequentially. Just as for ordinary polymorphic functions, an effect polymorphic function is called parametric

if its implementation does not inspect the monad it is instantiated with and hence applies the same algorithm for any monad. **Effect parametricity** is then a procedure similar to Wadler's that allows us to deduce a theorem from a type of effect polymorphic functions that is satisfied by all functions of that type.

We already gave an example of a type that depends on another type. In type systems with **dependent types**, it is also allowed for a type to depend on a value of another type. We could for instance consider the type `Vec<T,n>` of lists of length `n` with elements of type `T`. Here `n` is a value of another type `int`. The introduction of dependent types makes our type system so expressive that we can formalize mathematics in it, using a principle called the Curry-Howard correspondence. Under this correspondence, mathematical statements correspond to types and proofs of these statements correspond to values of their corresponding types. In this way, it is possible to write programs in a dependently-typed language and prove properties of these programs in the same language.

In [NVD17], Nuyts, Vezzosi and Devriese describe a dependent type system ParamDTT which has support for parametricity. This means that they provide tools to prove parametricity theorems such as Wadler's internally in the system. Additionally, there is an extension of the dependently-typed programming language Agda that implements ParamDTT and that is called Agda parametric.

The general objective of this thesis is to study monads and in particular effect parametricity in the type system ParamDTT. We want to investigate how monads and (parametric) effect polymorphic functions can be defined and whether it is possible to prove theorems that follow from effect parametricity internally in the system. Additionally, we intend to take effect parametricity as a specific use-case to evaluate the type system ParamDTT and its implementation Agda Parametric as a tool for practical computer-verified reasoning about parametricity.

In chapter 1 and the first part of chapter 2, we present background material on type systems and parametricity. We also give an introduction to the type system ParamDTT. The last section of chapter 2 contains some examples of parametricity results involving functors and how they can be proved in ParamDTT.

Subsequently, chapter 3 provides both an introduction to the theory of monads and an implementation of this theory in the type system ParamDTT. We will see two different (but almost equivalent) formulations of a monad and we investigate whether there are some interesting results that follow from parametricity of the monad operations.

In chapter 4 we then explore how to implement effect polymorphic functions in the type system ParamDTT and how to prove effect parametricity results. We describe a more or less general procedure in section 4.2 and apply it to concrete examples of parametricity results in the subsequent sections.

Finally, in chapter 5 we present some related work and chapter 6 contains the conclusion and some directions for further research.

Our own contributions in this thesis consist of the formulation of Wadler's discussion on parametricity (see [Wad89]) in a concrete term model instead of an abstract term model in section 2.1, the formalization of the parametricity results about functors in section 2.3, the formalization of monads and the corresponding theory in chapter 3 and the formalization of the effect parametricity results in chapter 4. The results in section 3.3 about the independence of the monad laws were probably known already (in the absence of parametricity), but we could not find any reference stating or proving those results.

List of Symbols

$\mathcal{T}$	Set of terms in a formal system.
$\mathcal{V}$	Set of variables in a formal system.
$\lambda x \,.\, t$	Lambda abstraction.
$t[s/x]$	Substitution in of a variable x in a term t by a term s.
$\rightsquigarrow$	Evaluation relation in a formal system.
$\rightsquigarrow^*$	Multi-step evaluation relation.
$\mathbb{T}$	Set of types in a formal system.
ι	Base type of the simply typed lambda calculus.
$\rightarrow$	Function type constructor.
$\Gamma \vdash t : T$	Typing relation in simply typed lambda calculus and System F, well-typedness judgement in MLTT and ParamDTT.
$\Lambda X \,.\, t$	Type abstraction in System F.
$t\,[T]$	Type application in System F.
$\forall X \,.\, T$	Universal type in System F.
MLTT	Martin-Löf type theory.
$=$	Definitional equality.
$\mathcal{U}_\ell$	Universe type of level ℓ.
$\Pi(x : A).B$	Dependent product type.
$\Sigma(x : A).B$	Dependent sum type.
$\top$	Unit type.
tt	Value of type $\top$.
$\bot$	Empty type.
$\mathbb{N}$	Type of natural numbers.
$a \equiv_A b$	Identity type between $a, b : A$.
CBool	Church encoding of the type of booleans.
ctrue	Church encoding of the boolean value true.
cfalse	Church encoding of the boolean value false.
$\mathbb{T}_k$	Set of System F types with at most k free type variables.
$\mathcal{T}_k$	Set of System F terms with at most k free ordinary variables.
D_T	Set of all well-typed System F terms of type T that are in normal form.
$\llbracket T \rrbracket$	Relational interpretation of the type T.
$\sharp$	Parametric modality.
$\P$	Pointwise modality.
$\Pi^\mu(x : A).B$	Dependent product type with modality μ.
$\Sigma^\mu(x : A).B$	Dependent sum type with modality μ.
$\forall(x : A).B$	Parametric dependent product type.

$\mathbb{I}$	Interval (pseudo) type in ParamDTT.
$/ \cdot \backslash$	Graph relation former of ParamDTT.
$i \doteq j$	(Face) predicate on values $i, j : \mathbb{I}$.
obj_F	Action of a functor F on types.
hom_F	Action of a functor F on functions.
$\mathrm{Maybe}\, A$	Type consisting of $\mathsf{just}\, a$ for all values $a : A$ + one value $\mathsf{nothing}$.
return	Return operation of a premonad.
$\gg=$	Bind operation of a premonad.
$\mathsf{id\text{-}pm}$	Identity premonad.
$\mathsf{writer\text{-}pm}_{R,r}$¶	Writer premonad for a magma $(R, \cdot)$ and a value $r : R$.
$\mathcal{C}_M$	Kleisli category for a monad M.
$\mathrm{List}\, A$	Type of lists with elements of type A.
$[\,]$	The empty list.
$::$	Cons, operation for prepending a value to a list.

Contents

Some Preliminary Remarks

The results of this thesis have been formalized in the proof assistant Agda parametric, which is an extension of the dependently-typed programming language Agda implementing the type system ParamDTT. All code can be found in the GitHub repository accompanying the thesis.[1] More information on how to install Agda parametric can also be found there. The code is an integral part of this thesis, and we will sometimes refer to it in the text when writing out all details on paper would become very cumbersome.

The following sections and chapters contain results that are formalized in Agda parametric.

- Section 2.3 about functors and corresponding parametricity results. The code for this section can be found in the file `Functors.agda`.
- Chapter 3 about the theory of monads. The code for this chapter is contained in the folder `Monads`.
- Chapter 4 about effect parametricity. For this chapter, the code is in the folder `EffectParametricity`. Every section of this chapter has a corresponding file in this folder containing formalizations of material of that section.

The reason why we do not show any Agda code in this text, is that in our opinion code is easier to understand when one can navigate inside it. Moreover, this gives us the opportunity to be somewhat sloppy in our notation and for instance write F for not only a functor, but also its action on types and its action on functions (see section 2.3 for the definition of a functor), which is better for human understanding but is not allowed in a proof assistant.

An extended abstract on the results of this thesis has been submitted to the TyDe 2019 workshop on Type-Driven Development[2] and is under review at the time of writing.

[1]See `https://github.com/JorisCeulemans/effect-param-agda`.

[2]See `https://icfp19.sigplan.org/home/tyde-2019`.

Chapter 1

Type Systems

This chapter provides background information on the formal systems that will be used in the subsequent chapters. We start in section 1.1 by introducing the untyped lambda calculus, which is a very simple formal system and theoretical functional programming language. Despite the title of this chapter it does not contain any types, but it will serve as the basis of the typed systems that will follow. Adding types to the untyped lambda calculus in the simplest possible way gives rise to the simply typed lambda calculus, which is the subject of section 1.2. Section 1.3 then introduces System F, which is an extension of the simply typed lambda calculus that allows us to define polymorphic functions. Finally, we will consider dependent types, another extension of the simply typed lambda calculus (and in some sense also of System F), in section 1.4.

1.1 Untyped Lambda Calculus

Alonzo Church introduced the (untyped) lambda calculus in 1936 to provide a negative solution to the so-called Entscheidungsproblem or decision problem in mathematical logic [Chu36]. That problem was formulated by David Hilbert and Wilhelm Ackerman in 1928 and it was concerned with the existence of a procedure that would take a set of axioms and a logical formula as input and would then provide an answer to the question whether the formula is provable from the axioms. Initially, a major difficulty in solving the Entscheidungsproblem was the absence of a precise notion of algorithm or computable procedure, and that was exactly what Church intended to provide with his lambda calculus. Using this definition of an algorithm, he proved that a procedure as required by the decision problem cannot exist.

The lambda calculus is, however, not only interesting to logicians and mathematicians. As it was developed to provide a precise definition of an algorithm, it is not surprising that the lambda calculus is nowadays also a subject of study in theoretical computer science. There, it serves as a theoretical programming language that is simple enough so that theorems about it can be rigorously proved. At the same time the lambda calculus (or in most cases some extension of it) serves as the basis for many functional programming languages.

We include in this thesis an introductory section on the lambda calculus because it is the basis of the other systems, such as System F and dependent type theory, that we will need in the next chapters. This section is based on [Pie02, chapter 5].

1.1.1 Syntax

The central notion of the lambda calculus is that of a function. Contrary to the standard mathematical definition of a function as a set of pairs representing its graph, we will think of a function as a procedure that takes some input and consequently produces some output (more or less like a function in a programming language). A function that produces output e for the input x will be denoted by $\lambda x \,.\, e$, where e is an expression that probably, but not necessarily, contains x. In fact, the only operations that are available in the untyped lambda calculus are function definition and function application. Data like booleans, natural numbers or lists can then be encoded in terms of these functions, see for instance example 1.12.

More formally, we will define the set $\mathcal{T}_\lambda$ of all valid programs of the untyped lambda calculus. Such a program is called a *lambda term* or simply a *term*. We assume that we have a countably infinite set $\mathcal{V}$ of variables at our disposal, and that $x, y, z, x_i, y_i, z_i \in \mathcal{V}$ for all $i \in \mathbb{N}$.

Definition 1.1. The set $\mathcal{T}_\lambda$ of terms of the untyped lambda calculus is defined as the smallest set that satisfies the following conditions.

- If $x \in \mathcal{V}$ then $x \in \mathcal{T}_\lambda$.
- If $x \in \mathcal{V}$ and $t \in \mathcal{T}_\lambda$ then $\lambda x \,.\, t \in \mathcal{T}_\lambda$.
- If $t_1, t_2 \in \mathcal{T}_\lambda$ then $t_1\, t_2 \in \mathcal{T}_\lambda$.

From now on, we will replace inductive definitions as this one by saying that the symbol x ranges over variables and the symbol t ranges over lambda terms and that the set of lambda terms is given by the following grammar.

$$\begin{array}{lll} t & ::= & x \\ & \mid & \lambda x \,.\, t \\ & \mid & t\, t \end{array}$$

A term of the form $\lambda x \,.\, t$ is called a *lambda abstraction* and a term of the form $t_1\, t_2$ for terms t_1 and t_2 is called an *application* (of t_1 to t_2).

There can be some ambiguity in the interpretation of a lambda term, so parentheses should be added whenever necessary. To avoid writing too many parentheses, we will adopt two conventions, which are also common in the literature.

- Application associates to the left. This means that an expression of the form $t_1\, t_2 \ldots t_n$ will be interpreted as $((t_1\, t_2) \ldots)\, t_n$.
- A lambda abstraction stretches as far to the right as possible. The term $\lambda x \,.\, xy$ will for instance be interpreted as $\lambda x \,.\, (xy)$ and not as $(\lambda x \,.\, x)\, y$.

Example 1.2. The following lambda terms give some idea about what kind of functions can be described in the untyped lambda calculus. The additional explanation is only based on the intuition that a lambda abstraction denotes a function. These statements will be precise when we have introduced semantics for the untyped lambda calculus in section 1.1.2.

- The term $\mathsf{id} = \lambda x \,.\, x$ represents a function that takes an argument x and returns x, so it is the identity function.
- The term $\mathsf{pr}_1 = \lambda x \,.\, \lambda y \,.\, x$ takes an argument x and maps it to the function that takes another argument and returns x. As a consequence, if we apply pr_1 to two arguments, it will return the first one. Hence we can see pr_1 as a (projection) function that takes two arguments and produces a result. The procedure of representing a function that takes multiple arguments as a higher order function, is called currying after the American logician Haskell Curry.
- The lambda abstraction $\mathsf{comp} = \lambda g \,.\, \lambda f \,.\, \lambda x \,.\, g\,(f\,x)$ represents a composition operator taking as its arguments two functions g and f and returning the composition $g \circ f$ [1].

We can now define bound and free variables of a lambda term.

Definition 1.3. Let $x \in \mathcal{V}$ be a variable. We define by induction on a lambda term $t \in \mathcal{T}_\lambda$ the *free* and *bound occurrences* of x in t.

- If $t = y$ is a variable and $y \neq x$, then x (obviously) does not occur in t.
- If $t = x$, then the occurrence of x in t is free.
- If $t = \lambda y \,.\, t'$ and $y \neq x$, then every free (bound) occurrence of x in t' is a free (bound) occurrence of x in t.
- If $t = \lambda x \,.\, t'$, then every occurrence of x in t' is a bound occurrence of x in t. The free occurrences of x in t' are said to be bound by this abstraction.
- If $t = t_1\, t_2$, then the free (bound) occurrences of x in t are the free (bound) occurrences of x in t_1 and t_2.

A *free (bound) variable* of a lambda term t is a variable with at least one free (bound) occurrence in t. A term with no free variables is called *closed*.

Example 1.4. In the term $x\,(\lambda y \,.\, y\, w\, (\lambda z \,.\, z))$ the occurrences of the variables x and w are free and those of y and z are bound. As this term contains free variables, it is not closed. All terms from example 1.2 are closed.

1.1.2 Semantics

Now that we know the syntactically valid programs of the untyped lambda calculus, we must specify how these programs are evaluated. As the lambda calculus only provides mechanisms for function definition and function application, the only interesting situation in the evaluation of a lambda term is the presence of an application where the left hand side is a lambda abstraction. Such a term of the form $(\lambda x \,.\, t)\, s$ is called a *β-redex* or simply a *redex* (from reducible expression). Following our intuition that $\lambda x \,.\, t$ is a function that maps x to t, the term $(\lambda x \,.\, t)\, s$ should be evaluated to the term t in which every

[1] Note that we are using here the mathematical notation $g \circ f$, but we emphasize that f and g are lambda terms and not mathematical functions.

free occurrence of x is substituted by s. However, giving a precise definition of such a substitution that corresponds to our intuition is not completely trivial. Hence we present in this section a precise definition of substitution and subsequently we introduce an evaluation relation for the untyped lambda calculus.

Consider the lambda terms $\lambda x \,.\, x$ and $\lambda y \,.\, y$. They are not syntactically equal, but they both represent the identity function. It is in general clear that, when t' is the term obtained from t by replacing all free occurrences of the variable x by y, then $\lambda x \,.\, t$ and $\lambda y \,.\, t'$ represent the same function.[2] We say that these lambda terms are *α-equivalent*. Terms that are equal up to some α-equivalent subterms are also defined to be α-equivalent. As an example $\lambda x \,.\, \lambda y \,.\, y$ and $\lambda x \,.\, \lambda z \,.\, z$ are α-equivalent. From now on we will consider lambda terms up to α-equivalence and hence we should verify that all definitions and results are invariant under α-equivalence.[3] However, this verification will usually be straightforward and therefore omitted, as is common in the literature.

Definition 1.5. Let $x \in \mathcal{V}$ be a variable and $s \in \mathcal{T}_\lambda$ a term. Then we define by induction on a term $t \in \mathcal{T}_\lambda$ the *substitution* of x by s in t, written as $t[s/x]$.

- If $t = y$ is a variable and $y \neq x$, then $t[s/x] = y$.
- If $t = x$, then $t[s/x] = s$.
- If $t = \lambda y \,.\, t'$ with $y \neq x$ and y not a free variable of s, then $t[s/x] = \lambda y \,.\, t'[s/x]$.
- If $t = t_1\, t_2$, then $t[s/x] = t_1[s/x]\, t_2[s/x]$.

The above definition appears to be incomplete because it does not cover the cases $t = \lambda x \,.\, t'$ or $t = \lambda y \,.\, t'$ where y is a free variable of s. However, using an α-equivalent lambda abstraction, we can convert these cases into the lambda abstraction case that is covered in the definition. In order for this to work, it is important that the set of variables $\mathcal{V}$ is infinite so that we can always find a variable not appearing in t or s.

The reason why we use the above definition and not the naive idea of replacing all free occurrences of x in t by s, is to avoid the capture of free variables in s. Suppose for instance that $t = \lambda x \,.\, y$ and that $s = x$. Then t represents some function mapping an argument x to a free variable y. If we calculated the substitution of y by s in t naively by replacing all free occurrences of y in t by s, we would get the identity function $\lambda x \,.\, x$, which is clearly not the intended behaviour of substitution. Using the above definition, however, we first convert t to an α-equivalent term $\lambda z \,.\, y$ and then get $t[s/x] = \lambda z \,.\, x$. The definition of substitution as in Definition 1.5 is therefore called capture-avoiding.

Using our definition of substitution, we can define an *evaluation relation* (sometimes also called a reduction relation) for the untyped lambda calculus. This will be a binary relation on the set of lambda terms $\mathcal{T}_\lambda$, denoted by the symbol $\rightsquigarrow$ and expressing the semantics of the untyped lambda calculus. For any two terms $t, s \in \mathcal{T}_\lambda$ we intuitively say that $t \rightsquigarrow s$ if and only if a reduction step or computation step can be performed on the

[2]Actually this is not true. Consider for instance the case in which $t = \lambda y \,.\, y\, x$ so that $t' = \lambda y \,.\, y\, y$. Then $\lambda x \,.\, \lambda y \,.\, y\, x$ and $\lambda y \,.\, \lambda y \,.\, y\, y$ do not at all represent the same function. However, it will be intuitively clear when two terms are α-equivalent.

[3]This is strictly speaking not true for the definition of free and bound variables. However, using the intuition that variable names are unimportant, it can be made correct by appropriately changing those variable names.

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_1}\ \textsc{E-App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{v_1\, t_2 \rightsquigarrow v_1\, t_2'}\ \textsc{E-App2} \qquad (\lambda x\,.\,t)\, v \rightsquigarrow t[v/x] \quad (\textsc{E-AppAbs})$$

Figure 1.1: Evaluation rules for the untyped lambda calculus.

term t, yielding the term s. Formally, the evaluation relation is given by the inference rules of Figure 1.1. In order to interpret these inference rules, we need to introduce a concept called value.

When specifying the semantics of a formal system, we must choose which terms will be considered as possible final results of computations. Those terms will in general be called the *values* of that formal system. In the specific case of the untyped lambda calculus, the set of values is defined to be the subset of $\mathcal{T}_\lambda$ consisting of all lambda abstractions. This means that we will consider a lambda abstraction as the valid final result of the evaluation of a lambda term and it implies that lambda abstractions cannot be reduced. We will use the symbol v to range over values.

The first inference rule now states that if a term t_1 can be reduced to a term t_1', then the application $t_1\, t_2$ can be reduced to the application $t_1'\, t_2$. Similarly, the second inference rule states that an application whose left subterm is a value (i.e. a lambda abstraction) and whose right subterm is reducible, can be reduced by reducing the right subterm. The last rule, having no premises, is strictly speaking not an inference rule, but an axiom. It states that an application of a lambda abstraction to a value can be reduced to the body of the lambda abstraction, substituting the value for the variable bound by the lambda abstraction. This kind of reduction of a redex is called *β-reduction*.

The statement that the evaluation relation $\rightsquigarrow$ is determined by the inference rules in Figure 1.1 means that we can conclude $t \rightsquigarrow s$ if and only if it is an instance of the axiom E-AppAbs, or it is the conclusion of an instance of one of the inference rules E-App1 or E-App2 whose premise was already established. More concretely the statement $t \rightsquigarrow s$ is derivable if and only if it is the conclusion of a finite derivation tree built up from the inference rules.

Example 1.6. We will prove the evaluation statement $\big((\lambda x\,.\,x)\,\big((\lambda y\,.\,y)\,z\big)\big)(\lambda w\,.\,w) \rightsquigarrow \big((\lambda x\,.\,x)\,z\big)(\lambda w\,.\,w)$ by constructing a derivation tree.

$$\dfrac{\dfrac{\dfrac{}{(\lambda y\,.\,y)\,z \rightsquigarrow z}\ \textsc{E-AppAbs}}{(\lambda x\,.\,x)\,\big((\lambda y\,.\,y)\,z\big) \rightsquigarrow (\lambda x\,.\,x)\,z}\ \textsc{E-App2}}{\big((\lambda x\,.\,x)\,\big((\lambda y\,.\,y)\,z\big)\big)(\lambda w\,.\,w) \rightsquigarrow \big((\lambda x\,.\,x)\,z\big)(\lambda w\,.\,w)}\ \textsc{E-App1}$$

In the first and second derivation steps (involving the inference rules E-AppAbs and E-App2) we used the fact that $\lambda w\,.\,w$ and $\lambda x\,.\,x$ are a lambda abstractions and hence a values.

Proposition 1.7. *The evaluation relation $\rightsquigarrow$ is deterministic, meaning that for any terms $t, t', t'' \in \mathcal{T}_\lambda$ if $t \rightsquigarrow t'$ and $t \rightsquigarrow t''$, then $t' = t''$.*

Proof. The proof uses induction on a derivation of $t \rightsquigarrow t'$. This means that for every inference rule of the evaluation relation we prove that if the statement of the theorem holds for all of its premises, then it also holds for its conclusion. As $t \rightsquigarrow t'$ holds, it must be in the conclusion of a derivation tree of this fact, and hence this proves the theorem. See [Pie02] for more details. □

Definition 1.8. A lambda term $t \in \mathcal{T}_\lambda$ is said to be in *normal form* if it cannot be reduced, or in other words if there is no $t' \in \mathcal{T}_\lambda$ such that $t \rightsquigarrow t'$.

From the inference rules in Figure 1.1 one can easily see that all lambda abstractions are in normal form. More generally, the fact that that values are normal forms will be true in all systems that follow, which is not surprising as values are valid results of a chain of reductions.

Definition 1.9. One defines the multi-step evaluation relation $\rightsquigarrow^*$ as the reflexive, transitive closure of $\rightsquigarrow$. This intuitively means that $t \rightsquigarrow^* s$ if and only if t can be reduced to s in zero or more steps.

A term $t \in \mathcal{T}_\lambda$ is said to have a normal form if there exists a term $t' \in \mathcal{T}_\lambda$ that is in normal form and such that $t \rightsquigarrow^* t'$.

Proposition 1.10. *Every term $t \in \mathcal{T}_\lambda$ has at most one normal form, so if $t \rightsquigarrow^* t'$ and $t \rightsquigarrow^* t''$ and t' and t'' are in normal form, then $t' = t''$.*

Proof. This follows straightforwardly from Proposition 1.7. □

It is however not the case that every term has a normal form. The lambda term $(\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x)$ for instance will reduce to itself and hence there is no finite sequence of evaluation steps leading to a normal form.

Remark 1.11. The semantics that we introduced in this section using an evaluation relation is one of many alternatives. First, the evaluation relation $\rightsquigarrow$ represents one computation step. This is kind of semantics is called small-step style, as opposed to big-step semantics with an evaluation relation describing the reduction of a term to its final result. An evaluation relation in the big-step style is a bit like our relation $\rightsquigarrow^*$ when the right-hand side is a normal form, but in our case the relation $\rightsquigarrow^*$ is not the primitive relation defining the semantics but is rather defined in terms of $\rightsquigarrow$.

Furthermore, the inference rules in Figure 1.1 determine a particular evaluation order in which the redexes in a lambda term should be reduced, namely first reduce the left-hand side of an application until it is a value, then reduce the right-hand side until it is a value and then reduce the obtained application using E-AppAbs. This order is called the call-by-value evaluation order. Other evaluation strategies exist such as call-by-reference, normal order and full β-reduction. The last two strategies also allow reduction within a lambda abstraction so that the definition of values should be adjusted (only those lambda abstractions with an irreducible body will then be values). Moreover those strategies will have different normal forms than the call-by-value order presented here. See [Bar85] and [Pie02, chapter 5] for more details.

Example 1.12. Although the untyped lambda calculus is very simple as a programming language, providing only function definition and function application as primitive constructs, data such as booleans and numbers can be encoded as lambda terms. This is called the *Church encoding* of these data. In this example, we will consider the Church encoding of booleans.

The essential feature of the booleans is that there are two values, true and false. Hence the encoding of the booleans must consist of two closed lambda terms, one encoding true and one encoding false. The traditional Church encoding of the booleans is

$$\begin{aligned}\mathsf{ctrue} &:= \lambda x \,.\, \lambda y \,.\, x\\ \mathsf{cfalse} &:= \lambda x \,.\, \lambda y \,.\, y.\end{aligned}$$

These two terms are particularly useful for encoding booleans because the booleans are mostly used for choosing between two alternatives and that is exactly what ctrue and cfalse allow us to do. Using this Church encoding, one can also start defining logical operations such as

$$\begin{aligned}\mathsf{and} &:= \lambda b \,.\, \lambda c \,.\, b\, c\, \mathsf{cfalse}\\ \mathsf{not} &:= \lambda b \,.\, b\, \mathsf{cfalse}\, \mathsf{ctrue}.\end{aligned}$$

Working out definitions then gives us that for instance $\mathsf{and}\,\mathsf{ctrue}\,\mathsf{cfalse} \rightsquigarrow^* \mathsf{cfalse}$ and that $\mathsf{not}\,\mathsf{cfalse} \rightsquigarrow^* \mathsf{ctrue}$.

1.2 Simply Typed Lambda Calculus

A practical functional programming language based on the lambda calculus usually has primitives for booleans, natural numbers, lists, etc. instead of relying on their Church encodings (among others for efficiency reasons). Consider for the moment the untyped lambda calculus extended with primitive support for natural numbers. It then becomes possible to write nonsensical programs such as the application of a number to some other term. Such a term that is not a value, but cannot be reduced either, is called *stuck* and it represents an erroneous program. A lot of programming languages have a type system that prevents us from evaluating nonsensical programs like the one above. In this section, we will add a type system to the untyped lambda calculus, giving rise to the simply typed lambda calculus (sometimes abbreviated to STLC). This section is based on [Pie02, chapter 9].

1.2.1 The Typing Relation

The idea of a type system is to assign types to certain terms indicating what kind of information those terms represent. In a language containing primitives for the natural numbers, there could be for instance a type of natural numbers and every natural number would have that type. However, we would not be able to assign a type to an application in which the left-hand side is a natural number because for the typing of an application the left-hand side will be required to have a function type. If we then restrict ourselves to the execution or evaluation of programs that can be assigned a type, the nonsensical program from the introduction to this section will be rejected by the type system.

When providing typing for the lambda calculus, we should first determine what are the possible types that lambda terms can have. As the only values of the lambda calculus are lambda abstractions representing functions, these types will be function types indicating that a term is a function. Moreover, it will turn out to be useful (and even necessary) to include in these function types the type of the argument that is expected and the type of result that is returned. More concretely, for any types A and B we will have a type $A \to B$ of functions that take an argument of type A and return a result of type B. Furthermore, we will also introduce a base type ι. We will not assign this type to any closed term, but as the construction of a function type $A \to B$ already requires two existing types A and B, we need some type to start from.

Formally, the symbol T will range over types and the set $\mathbb{T}_\to$ of types of the simply typed lambda calculus is given by the following grammar.

$$\begin{array}{lll} T & ::= & \iota \\ & | & T \to T \end{array}$$

Again parentheses should be added whenever necessary to resolve ambiguity, but we adopt the convention that $\to$ associates to the right. This means for instance that $A \to B \to C$ will be interpreted as $A \to (B \to C)$.

It is clear that the identity function $\lambda x\,.\,x$ is not an erroneous term and should therefore not be rejected by the type system. As a consequence, we should be able to assign a type to it. Furthermore, it is also clear that the identity function is a function taking an argument of a certain type A and returning a result of the same type A. The problem is to determine what the type A should be because the identity function can take an argument of any type. In fact, in the simply typed lambda calculus there will be infinitely many identity functions, one for every type $A \in \mathbb{T}_\to$. In order to tell what type of argument a function expects, we alter the syntax of lambda abstractions to include the argument type. The identity function for the type A will for example be written as $\lambda(x : A)\,.\,x$. The set $\mathcal{T}_\to$ of terms of the simply typed lambda calculus is then given by the following grammar.

$$\begin{array}{lll} t & ::= & x \\ & | & \lambda(x : T)\,.\,t \\ & | & t\,t \end{array}$$

All definitions from section 1.1 such as free and bound variables, α-equivalence, substitution, evaluation, etc. carry over to the simply typed lambda calculus.

Consider now two types $A, B \in \mathbb{T}_\to$ and their respective identity functions $\lambda(x : A)\,.\,x$ and $\lambda(x : B)\,.\,x$. In the body of the first abstraction, the variable x represents a term of type A whereas in the body of the second abstraction it represents a term of type B. From this we can see that the type of a variable depends on the context it lives in and therefore, we will introduce the concept of a *typing context*. A typing context is a sequence of distinct variables with for each variable the type it is assumed to have in that context. For instance, in the context $x : \iota, y : \iota \to \iota$ the variable x is assumed to be of type ι and the variable y to be of type $\iota \to \iota$. Formally the symbol Γ will range over typing contexts and the set of all typing contexts is given by the following grammar.

$$\begin{array}{lll} \Gamma & ::= & \emptyset \\ & | & \Gamma, x : T \end{array}$$

Here the symbol $\emptyset$ denotes the empty context containing no assumptions about variables. Furthermore, we require all variables in a typing context to be distinct (which is strictly speaking not enforced by the grammar). We write $(x : T) \in \Gamma$ if and only if the assumption $x : T$ is contained in the typing context Γ.

Now we can define the *typing relation* for the simply typed lambda calculus. It is a ternary relation written $\cdot \vdash \cdot : \cdot$ between the set of typing contexts, the set $\mathcal{T}_\to$ of terms

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad\qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A)\,.\,t : A \to B}\ \textsc{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\,t_2 : B}\ \textsc{T-App}$$

Figure 1.2: Typing rules for the simply typed lambda calculus.

and the set $\mathbb{T}_{\to}$ of types of the simply typed lambda calculus. Intuitively we say that $\Gamma \vdash t : T$ if and only if in the context Γ (i.e. under the assumptions of the context Γ) the term t has type T. Formally, the typing relation is given by the inference rules in Figure 1.2. In a typing statement involving the empty context, we will write $\vdash t : T$ rather than $\emptyset \vdash t : T$.

The typing rule T-Var tells us that if the assumption $x : T$ is contained in the context Γ, then the variable x has type T in that same context. The rule T-Abs states that a lambda abstraction $\lambda(x : A)\,.\,t$ has a function type $A \to B$ if, assuming that the variable x has type A, the term t has type B. Finally the inference rule T-App says that an application of a term with function type $A \to B$ to a term with type A has type B.

Just like with the evaluation relation in section 1.1.2, the fact that the typing relation is determined by the inference rules in Figure 1.2 means that $\Gamma \vdash t : T$ if and only if this is the conclusion of a derivation tree that can be built using the typing rules. Such a derivation tree is also called a *typing derivation* for the typing statement.

Definition 1.13. A term $t \in \mathcal{T}_{\to}$ is said to be *typable* or *well typed* if there exists a context Γ and a type $T \in \mathbb{T}_{\to}$ such that $\Gamma \vdash t : T$.

Example 1.14. We will show that $\vdash \big(\lambda(x : \iota \to \iota)\,.\,x\big)\,(\lambda(y : \iota)\,.\,y) : \iota \to \iota$ by constructing a typing derivation.

$$\frac{\dfrac{\dfrac{(x : \iota \to \iota) \in x : \iota \to \iota}{x : \iota \to \iota \vdash x : \iota \to \iota}\ \textsc{T-Var}}{\vdash \lambda(x : \iota \to \iota)\,.\,x : (\iota \to \iota) \to \iota \to \iota}\ \textsc{T-Abs} \qquad \dfrac{\dfrac{(y : \iota) \in y : \iota}{y : \iota \vdash y : \iota}\ \textsc{T-Var}}{\vdash \lambda(y : \iota)\,.\,y : \iota \to \iota}\ \textsc{T-Abs}}{\vdash \big(\lambda(x : \iota \to \iota)\,.\,x\big)\,(\lambda(y : \iota)\,.\,y) : \iota \to \iota}\ \textsc{T-App}$$

Example 1.15. We continue example 1.12 about the Church encoding of the booleans. There the intuition was that the terms ctrue and cfalse let us easily choose between two alternatives. However, in the simply typed lambda calculus a lambda abstraction must be annotated with the type of argument it expects. Hence, just like with the identity function above, we now have for any type $A \in \mathbb{T}_{\to}$ the Church booleans

$$\mathsf{ctrue}_A := \lambda(x : A)\,.\,\lambda(y : A)\,.\,x$$
$$\mathsf{cfalse}_A := \lambda(x : A)\,.\,\lambda(y : A)\,.\,y$$

and these closed terms are well typed since $\vdash \mathsf{ctrue}_A, \mathsf{cfalse}_A : A \to A \to A$. This situation is however rather unsatisfactory, as our basic intuition of booleans tells us there are only two boolean values and not infinitely many. Moreover the term ctrue_T behaves exactly the same for every type T (picking the first of two arguments), the type annotation is only present for type checking. We will see in section 1.3 how to resolve these problems in System F.

1.2.2 Properties of the Simply Typed Lambda Calculus

We start this section with a simple proposition about uniqueness of types.

Proposition 1.16. *If for any typing context Γ, term $t \in \mathcal{T}_\to$ and types $T, T' \in \mathbb{T}_\to$ we have that $\Gamma \vdash t : T$ and $\Gamma \vdash t : T'$, then $T = T'$*

Proof. Very simple induction on a typing derivation of $\Gamma \vdash t : T$, using the convention that all variables in a typing context should be distinct. □

In the introduction to this section we mentioned that one of the reasons to introduce a type system is to prevent us from writing erroneous programs. Such an erroneous program is more more concretely defined as a term that evaluates to a stuck term (which is a term in normal form that is not a value). It is now possible to prove that any well-typed closed term of the simply typed lambda calculus will not evaluate to a stuck term. This property is known as the type safety of the simply typed lambda calculus and it is a precise formulation of the fact that a type system prevents us from writing erroneous programs.

Theorem 1.17 (Type safety for STLC)**.** *The following two statements hold.*

1. *Progress: If a closed term $t \in \mathcal{T}_\to$ is well typed, then it is either a value, or there exists a term $t' \in \mathcal{T}_\to$ such that $t \rightsquigarrow t'$.*

2. *Preservation: If $\Gamma \vdash t : T$ and $t \rightsquigarrow t'$ then $\Gamma \vdash t' : T$.*

This theorem indeed tells us that the evaluation of a well-typed closed term will never reach a stuck term. More concretely, by the progress statement a well-typed closed term t is either a value (and then the result follows immediately), or it can be reduced. The preservation statement tells us that in the latter case, the term to which t is evaluated is also well-typed and hence we can apply the same argument again. Note that Theorem 1.17 does not tell us that every well-typed closed term can be reduced in a finite number of steps to a normal form as it does not exclude the possibility of infinite evaluation sequences.

Proof of Theorem 1.17.
1. The proof of the progress statement is by induction on a typing derivation for the term t. See [Pie02, p. 105] for more details.

2. This proof can be carried out by induction on a derivation of $t \rightsquigarrow t'$ (or alternatively on a typing derivation of $\Gamma \vdash t : T$). It makes use of the following substitution lemma. See again [Pie02, p. 107] for more details.

□

Lemma 1.18 (Substitution lemma for STLC)**.** *If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$ then $\Gamma \vdash t[s/x] : T$.*

Proof. Again by induction on a typing derivation of $\Gamma, x : S \vdash t : T$. See [Pie02, pp. 106-107] for more details. □

In the discussion after Proposition 1.10 we saw an example of a term in the untyped lambda calculus that does not have a normal form. However, in the simply typed lambda calculus, we can prove that every well-typed term can be reduced in a finite number of steps to a normal form. One says that the simply typed lambda calculus is normalizing.

Theorem 1.19 (Normalization of STLC)**.** *If $t \in \mathcal{T}_{\to}$ is a well-typed closed term, then there is a term t' in normal form such that $t \leadsto^* t'$.*

Proof. Proving this theorem requires more than just induction on a typing derivation, but involves a technique called logical relations. The complete proof is the subject of [Pie02, chapter 12]. □

Together with the translation of Proposition 1.10 to the simply typed lambda calculus, this theorem tells us that every well-typed closed term has a unique normal form and from Theorem 1.17 it follows that this normal form is a value.

1.3 System F

In the previous section we saw that the simply typed lambda calculus has infinitely many identity functions, namely $\lambda(x : A) \,.\, x$ for every type $A \in \mathbb{T}_{\to}$. The behaviour of all these functions with respect to evaluation is however identical and the type annotation is only used for type checking. The idea of this section is to introduce polymorphic functions that work for any type. In this way we can introduce a single polymorphic identity function that can be instantiated with a type A and will then reduce to the identity function for the type A. The formal system that lets us define such polymorphic functions is called System F or the polymorphic lambda calculus.[4]

System F is an extension of the simply typed lambda calculus, developed independently by Jean-Yves Girard [Gir72] and John Reynolds [Rey83]. This section is based on [Pie02, chapter 23].

1.3.1 Syntax and Semantics

The main new feature of System F is the possibility to abstract a type out of a term. In other words, we can define a function that takes a type as argument and then returns a term. This new kind of abstraction is called a type abstraction and it is written as $\Lambda X \,.\, t$, where X is a type variable and t is a term. We can then define the polymorphic identity function as $\Lambda X \,.\, \lambda(x : X) \,.\, x$, so it is a function taking a type X as argument and returning the identity function for that type X. System F also allows us to apply a polymorphic function to a concrete type T, which we write as $t\,[T]$ and which is called a type application.

As System F is a typed language, we should assign types to well-behaved polymorphic functions such as the polymorphic identity function. For this purpose, System F provides us with a new type constructor $\forall$. The polymorphic identity function will for instance have type $\forall X \,.\, X \to X$ because for any type X it yields a function from X to X.

Having discussed the intuition about System F, we can now consider its syntax and semantics more formally. In System F we distinguish between type variables like the variable X in the previous paragraphs and ordinary variables that serve the same purpose as the variables we used in the previous sections. We assume that we have countably infinite sets $\mathcal{V}_T$ of type variables and $\mathcal{V}_O$ of ordinary variables. The symbol X will range over type variables and x over ordinary variables. The set $\mathbb{T}_{\mathrm{F}}$ of System F types is then defined by the following grammar.

[4]According to Girard [Gir86, p. 160] the letter F for the name System F was chosen by chance.

$$\begin{array}{lll} T & ::= & X \\ & \mid & T \to T \\ & \mid & \forall X \,.\, T \end{array}$$

Note that we do not include a base type any more because we can now start building types using type variables.

Subsequently, the set $\mathcal{T}_{\mathrm{F}}$ of the terms in System F is defined by the following grammar.

$$\begin{array}{lll} t & ::= & x \\ & \mid & \lambda(x : T) \,.\, t \\ & \mid & t\,t \\ & \mid & \Lambda X \,.\, t \\ & \mid & t\,[T] \end{array}$$

A term of the form $\Lambda X \,.\, t$ is called a *type abstraction* and a term of the form $t\,[T]$ is called a *type application* (of t on T).

Similar to the definition of free and bound ordinary variables of a term, we can also define free and bound type variables of a term where a type variable gets bound by a type abstraction. For instance the term $\big(\Lambda X \,.\, \lambda(x : X \to Y) \,.\, x\big)\,[Z]$ has X as a bound type variable and Y and Z as free type variables. Furthermore, we can now also define the notion of free and bound type variables of a type where a type variable gets bound by the $\forall$ type constructor. As an example the type $\forall X \,.\, X \to Y \to X$ has X as a bound type variable and Y as a free type variable.

The notion of α-equivalence for terms must also be extended to type variables. It is clear that when t' is the term obtained by replacing in t all free occurrences of X by Y, then the terms $\Lambda X \,.\, t$ and $\Lambda Y \,.\, t'$ represent the same polymorphic functions. These terms will therefore also be called α-equivalent. In the same way, we can also talk about α-equivalent types, like $\forall X \,.\, X \to X$ and $\forall Y \,.\, Y \to Y$. Just like in the previous sections, we consider terms and types up to α-equivalence.

Another concept from section 1.1 that can be extended to type variables is that of substitution. In System F we can also substitute in a type T a type variable X by a type S, written as $T[S/X]$, where we pay attention to avoid the capture of free type variables in S by a $\forall$ constructor in T (by taking another type that is α-equivalent to T). Similarly, we can define the capture-avoiding substitution of a type variable X by a type S in a term t, written as $t[S/X]$.

The semantics of System F is now given by the evaluation relation $\rightsquigarrow$ that is determined by the inference rules in Figure 1.3. The rules E-App1, E-App2 and E-AppAbs are identical to the evaluation rules of the simply typed lambda calculus. Note however that the values of System F are defined to be the lambda abstractions and the type abstractions. In addition to the known evaluation rules, we now also have the rule E-Tapp stating that a type application of a term to a type can be reduced if that term can be reduced, and E-TappTabs which is the counterpart of E-AppAbs for type abstractions and type applications.

To introduce the typing relation for System F, we will modify the definition of typing contexts a little bit. Such a typing context is still a sequence of typing assumptions on ordinary variables, but we will now also keep track of the type variables that appear free in the types of the ordinary variables and in the type of the typing statement under consideration. Hence a typing context will now have the form $\Theta; \Gamma$ where Θ is a sequence

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_1}\ \text{E-App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{v_1\, t_2 \rightsquigarrow v_1\, t_2'}\ \text{E-App2} \qquad (\lambda(x:T)\,.\,t)\, v \rightsquigarrow t[v/x] \quad (\text{E-AppAbs})$$

$$\frac{t \rightsquigarrow t'}{t\,[T] \rightsquigarrow t'\,[T]}\ \text{E-Tapp} \qquad (\Lambda X\,.\,t)\,[T] \rightsquigarrow t[T/X] \quad (\text{E-TappTabs})$$

Figure 1.3: Evaluation rules for System F.

$$\frac{(x:T) \in \Gamma}{\Theta;\Gamma \vdash x : T}\ \text{T-Var} \qquad \frac{\Theta;\Gamma, x : A \vdash t : B}{\Theta;\Gamma \vdash \lambda(x:A)\,.\,t : A \to B}\ \text{T-Abs}$$

$$\frac{\Theta;\Gamma \vdash t_1 : A \to B \qquad \Theta;\Gamma \vdash t_2 : A}{\Theta;\Gamma \vdash t_1\, t_2 : B}\ \text{T-App}$$

$$\frac{\Theta, X;\Gamma \vdash t : T}{\Theta;\Gamma \vdash \Lambda X\,.\,t : \forall X\,.\,T}\ \text{T-Tabs} \qquad \frac{\Theta;\Gamma \vdash t : \forall X\,.\,T}{\Theta;\Gamma \vdash t\,[A] : T[A/X]}\ \text{T-Tapp}$$

Figure 1.4: Typing rules for System F.

of type variables and where Γ is a sequence of typing assumptions on ordinary variables, just as in section 1.2. The typing relation is then given by the inference rules in Figure 1.4. We will write $\vdash t : T$ rather than $\emptyset;\emptyset \vdash t : T$.

Example 1.20. This example is a continuation of examples 1.12 and 1.15 about the Church encoding of the booleans. We saw in example 1.15 that in the simply typed lambda calculus we need separate terms encoding true and false for every type $A \in \mathbb{T}_{\to}$. Moreover, the semantic behaviour of these terms does not depend on the type A. In System F we can abstract the type out of these terms to get the Church Booleans

$$\mathsf{ctrue} := \Lambda X\,.\,\lambda(x:X)\,.\,\lambda(y:X)\,.\,x$$
$$\mathsf{cfalse} := \Lambda X\,.\,\lambda(x:X)\,.\,\lambda(y:X)\,.\,y.$$

Hence we see that System F is well suited to the Church encoding of data. Both terms ctrue and cfalse are also of the same type, so we can define in System F the type of Church booleans

$$\mathsf{CBool} := \forall X\,.\,X \to X \to X.$$

The question is now of course whether this type CBool really only contains the two terms ctrue and cfalse as we would expect of a type encoding the type of booleans. This question will be answered in chapter 2.

1.3.2 Properties of System F

Just like for the simply typed lambda calculus, one can prove type safety and normalization for System F.

Theorem 1.21 (Type Safety for System F)**.** *The following two statements hold.*

1. *Progress: If a closed term $t \in \mathcal{T}_{\mathrm{F}}$ is well typed (i.e. $\vdash t : T$ for some type T), then t is either a value, or there exists a term $t' \in \mathcal{T}_{\mathrm{F}}$ such that $t \rightsquigarrow t'$.*

2. *Preservation: If* $\Theta; \Gamma \vdash t : T$ *and* $t \rightsquigarrow t'$ *then* $\Theta; \Gamma \vdash t' : T$.

Proof. Just like in Theorem 1.17 by induction on a typing derivation or derivation of an evaluation statement. □

Theorem 1.22 (Normalization of System F)**.** *For any well-typed closed term* $t \in \mathcal{T}_{\mathrm{F}}$ *there is a term* $t' \in \mathcal{T}_{\mathrm{F}}$ *that is in normal form and such that* $t \rightsquigarrow^* t'$.

Contrary to the normalization of the simply typed lambda calculus, the normalization of System F is very difficult to prove. It is one of the results of Girard's doctoral thesis [Gir72].

It is easy to show that the evaluation relation of System F is deterministic (just like Proposition 1.7 for the untyped lambda calculus). In combination with Theorem 1.22, this implies that any well-typed closed term t in System F has a unique normal form which we will denote by $\mathrm{nform}(t)$. By the preservation property of System F this normal form has the same type as t and as it is well-typed and in normal form, it must be a value by the progress property.

1.4 Dependent Type Theory

So far we have considered two type constructors $\to$ and $\forall$. The former takes two types A and B and produces the type $A \to B$ and the latter takes a type variable X and a type T to produce the type $\forall X \,.\, T$. As we can see, the function types and polymorphic types we can construct in this way depend on other types, for instance $A \to B$ depends on the types A and B. Dependent type theory will allow us to construct types that depend not only on other types, but also on terms (of some other type).

One of the reasons why considering dependent types is useful, is that a dependent type can contain more information than an ordinary simple type. For example, in a language with support for natural numbers (providing a type $\mathbb{N}$), one could define a type $\mathrm{Vec}_n\, A$ of lists of length n with elements of type A. Here n is a value of type $\mathbb{N}$, in other words it is a natural number, and hence $\mathrm{Vec}_n\, A$ is a dependent type that depends on this value n. The type $\mathrm{Vec}_n\, A$ does not only indicate that its values are lists, but it also gives information about the length of such a list. This can for instance be useful when implementing a function that would not be well defined for the empty list (such as the function head mapping a list to its first element).

Another reason why dependent types are important involves the Curry-Howard correspondence, which will be the subject of section 1.4.3. The essence is that using a dependently typed programming language, one cannot only write programs but also prove properties about these programs within the same language.

There are many type systems that provide dependent types. In this section we present a well-known system called *Martin-Löf type theory* (sometimes abbreviated as MLTT), after the Swedish logician Per Martin-Löf.[5] In section 2.2, we will discuss another system with dependent types called ParamDTT. This section is based on [DP16] and [Hof97, sections 1-2].

[5]In fact Martin-Löf also designed multiple dependent type systems and what we will present here is a system that is nowadays called MLTT and which resembles the system in [Mar82].

1.4.1 Judgements

Martin-Löf type theory provides inference rules that allow us to derive *judgements*. A judgement is a statement that has a very specific form and in this section we will introduce the different kinds of judgements present in MLTT. These judgements are the only meaningful statements that can be expressed in the formal system of MLTT, but in section 1.4.3 we will see that this gives us enough expressiveness to do predicate logic. Just as in the previous sections, we will say that a judgement can be derived if and only if it is the conclusion of a derivation tree that is built using the inference rules. Again we will assume throughout this section that we have infinitely many variables at our disposal (we will not need type variables like those in System F anymore, so there will be only ordinary variables).

The Different Kinds of Judgements

The first kind of judgements we will consider are judgements stating that a certain object Γ is a well-formed context. Such a judgement will be written as

$$\vdash \Gamma \text{ context}.$$

Note that the notion of a well-formed context will be defined using inference rules, rather than using a certain grammar like in our description of the simply typed lambda calculus or System F. This is due to the fact that with the introduction of dependent types, the well-formednes of a type can depend on the well-typedness of a term and the well-typedness of a term depends in its turn on the context. As we will only consider typing contexts that contain typing assumptions on variables involving well-formed types, there will be an interplay between the different kinds of judgements and hence the notions of well-formed context, well-formed type and well-typed term must be introduced simultaneously (by means of the inference rules that we will discuss later).

Another kind of judgement, at which we already hinted in the previous paragraph, is a judgement stating that in a context Γ an object T is a well-formed type, written as

$$\Gamma \vdash T \text{ type}.$$

Just as for well-formed contexts, it is this judgement that will say that T is a (well-formed) type that can be used in our type system and we will not define a set of types using a grammar, as we did in sections 1.2 and 1.3.

Furthermore, there are judgements stating that in a context Γ a certain object t is a well-formed term of type T, written as

$$\Gamma \vdash t : T.$$

This notation is reminiscent of the typing relation introduced in section 1.2.1, but we emphasize that in this presentation of MLTT there is no relation expressing typing on a certain predefined set of terms. There is even no set of terms defined by a grammar as was the case in sections 1.1 to 1.3. A judgement of this form is the only way to express that t is a term that can be used in our type system. Note that in this way, we cannot introduce a term in MLTT without mentioning its type.

Remark 1.23. When we write a judgement like $\Gamma \vdash T$ type, we implicitly assume that Γ is a well-formed context, so we assume that $\vdash \Gamma$ context can be derived. Similarly, when we write $\Gamma \vdash t : T$, we implicitly assume that we can derive $\vdash \Gamma$ context and $\Gamma \vdash T$ type.

Apart from the three judgement forms for introducing contexts, types and terms discussed above, there will also be judgements expressing *definitional equality* or *judgemental equality* between those objects. For terms, there is the judgement

$$\Gamma \vdash t = s : T$$

expressing that t and s are definitionally equal terms of type T in the context Γ. For instance, if we would add natural numbers to our type system and define addition for these numbers, we would be able to derive $\Gamma \vdash 2 + 2 = 4 : \mathbb{N}$. Again, in this kind of judgements we implicitly assume that t and s are well-formed terms of type T in the context Γ.

Since types can depend on terms, we then also need judgemental equality for types. Indeed, if we again consider the addition of natural numbers, it is clear that the types $\mathrm{Vec}_{2+2}\, A$ and $\mathrm{Vec}_4\, A$ should be equal. Hence we have judgements of the form

$$\Gamma \vdash T = S \text{ type}$$

expressing that T and S are definitionally equal types in the context Γ. Of course, here we again implicitly assume that T and S are well-formed types in the context Γ.

Finally, we can also consider definitional equality for contexts. Hence there will be a judgement stating that Γ and Δ are definitionally equal contexts, written as

$$\vdash \Gamma = \Delta \text{ context.}$$

Here, we also assume that Γ and Δ are well-formed contexts. In fact, a judgement expressing definitional equality of contexts is not strictly necessary as it can be defined in terms of the other judgements [Hof97, section 2] but we include it for completeness.

Structural Inference Rules

In section 1.4.2 we will introduce a variety of different types and terms that are present in Martin-Löf type theory by giving several inference rules. Before we start doing that, we will discuss some structural inference rules that have no connection with a specific type, but that take care of some general aspects like how to construct well-formed contexts.

First, we provide inference rules stating that definitional equality (for contexts, types and terms) is an equivalence relation.

$$\frac{\vdash \Gamma \text{ context}}{\vdash \Gamma = \Gamma \text{ context}}\,\text{CTX-=-REF} \qquad \frac{\vdash \Gamma = \Delta \text{ context}}{\vdash \Delta = \Gamma \text{ context}}\,\text{CTX-=-SYM}$$

$$\frac{\vdash \Gamma_1 = \Gamma_2 \text{ context} \qquad \vdash \Gamma_2 = \Gamma_3 \text{ context}}{\vdash \Gamma_1 = \Gamma_3 \text{ context}}\,\text{CTX-=-TRA}$$

$$\frac{\Gamma \vdash T \text{ type}}{\Gamma \vdash T = T \text{ type}}\,\text{TYPE-=-REF} \qquad \frac{\Gamma \vdash T = S \text{ type}}{\Gamma \vdash S = T \text{ type}}\,\text{TYPE-=-SYM}$$

$$\frac{\Gamma \vdash T_1 = T_2 \text{ type} \qquad \Gamma \vdash T_2 = T_3 \text{ type}}{\Gamma \vdash T_1 = T_3 \text{ type}}\,\textsc{type-=-tra}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t = t : T}\,\textsc{term-=-ref} \qquad \frac{\Gamma \vdash t = s : T}{\Gamma \vdash s = t : T}\,\textsc{term-=-sym}$$

$$\frac{\Gamma \vdash t_1 = t_2 : T \qquad \Gamma \vdash t_2 = t_3 : T}{\Gamma \vdash t_1 = t_3 : T}\,\textsc{term-=-tra}$$

We will also need inference rules allowing us to construct well-formed contexts. Just like in the simply typed lambda calculus, a context will be a sequence $x_1 : A_1, \ldots, x_n : A_n$ of typing assumptions on variables. However, we will now also require that in this sequence every type A_i is well-formed in the context $x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$. The following inference rules make this intuition precise.

$$\frac{}{\vdash \emptyset \text{ context}}\,\textsc{ctx-empty} \qquad \frac{\vdash \Gamma \text{ context} \qquad \Gamma \vdash T \text{ type}}{\vdash \Gamma, x : T \text{ context}}\,\textsc{ctx-ext}$$

The rule CTX-EMPTY tells us that the empty context is a well-formed context. A judgement of the form $\emptyset \vdash \mathcal{J}$ involving the empty context will also be written as $\vdash \mathcal{J}$.[6] Furthermore, the rule CTX-EXT states that we can extend a well-formed context Γ by adding the assumption that a variable x has a certain type T that is well formed in the context Γ. In this rule we assume that x is a variable that does not occur in Γ so that all variables in a context are distinct. We can now see that if Γ and Δ are two well-formed contexts with the property that no variable of Γ occurs in Δ and vice versa, then the combination of all assumptions in Γ and Δ is also a well-formed context, which we will denote by Γ, Δ. To conclude our discussion about contexts, we introduce a rule from which we can deduce that two contexts that assign to each variable equal types, are equal themselves.

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash T = S \text{ type}}{\vdash \Gamma, x : T = \Delta, x : S \text{ context}}\,\textsc{ctx-=-ext}$$

Again we assume in this inference rule that x does not occur in Γ or Δ.

The following rule tells us how the assumptions in a context can be used. More concretely, it states that in a context in which a variable x is assumed to have type T this variable x is a well-formed term of type T.

$$\frac{\vdash \Gamma, x : T, \Delta \text{ context}}{\Gamma, x : T, \Delta \vdash x : T}\,\textsc{var}$$

Furthermore, we introduce inference rules expressing the intuition that the judgements of the different kinds we have seen, respect definitional equality. The inference rule TERM-CONV for instance states that if a term t has type T (in a certain context) and if the types T and S are definitionally equal, then t also has type S (in a context that is definitionally equal to the first context).

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash T \text{ type}}{\Delta \vdash T \text{ type}}\,\textsc{type-conv}$$

[6]Note that this is only the case for judgements about types or terms, so we will not write $\vdash$ context to express that $\emptyset$ is a well-formed context.

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash T = S \text{ type}}{\Delta \vdash T = S \text{ type}}\ \textsc{type-=-conv}$$

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash T = S \text{ type} \qquad \Gamma \vdash t : T}{\Delta \vdash t : S}\ \textsc{term-conv}$$

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash T = S \text{ type} \qquad \Gamma \vdash t = s : T}{\Delta \vdash t = s : S}\ \textsc{term-=-conv}$$

Finally, we also add the following two general inference rule schemes where $\mathcal{J}$ is of the form T type, $T = S$ type, $t : T$ or $t = s : T$ (so we do not consider judgements of the form $\vdash \Gamma$ context or $\vdash \Gamma = \Delta$ context).

$$\frac{\Gamma, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash T \text{ type}}{\Gamma, x : T, \Delta \vdash \mathcal{J}}\ \textsc{weaken} \qquad \frac{\Gamma, x : T, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash t : T}{\Gamma, \Delta[t/x] \vdash \mathcal{J}[t/x]}\ \textsc{subst}$$

The rule weaken tells us that any judgement $\mathcal{J}$ remains valid if we extend the context. Here we assume that x does not appear in Γ or Δ (otherwise $\Gamma, x : T, \Delta$ would not be a well-formed context). The inference rule subst says that if we have a judgement $\mathcal{J}$ depending on a variable x of type T and if we have a term t of type T, then we can substitute t for x in this judgement $\mathcal{J}$. Note that like in sections 1.1 to 1.3 the substitution is here assumed to be capture-avoiding so that free variables in t do not become bound by some constructs in $\mathcal{J}$.

Remark 1.24. In fact, we do not need to explicitly mention the last two inference rules weaken and subst. It follows from the other rules discussed in this section that if the hypotheses of one of these two rules can be derived, then also the conclusion can be derived. Inference rules with this property are called admissible.

1.4.2 Typical Constructions in Martin-Löf Type Theory

In this section we will introduce various types and type constructors of Martin-Löf type theory together with their terms and some rules concerning judgemental equality. For the creation of each new type, we will provide the following inference rules.

- A *formation rule* that indicates how the new type we are introducing must be formed and on what data it depends.
- An *introduction rule*, giving us a way to construct terms of the newly created type.
- *Congruence rules* ensuring that the types and terms constructed using the formation and introduction rules respect definitional equality.
- An *elimination rule* that allows us to use a term of the newly constructed type in other terms or types. This is sometimes also called an *induction principle* because for the type of natural numbers, it will correspond to mathematical induction (or recursive definition).
- A *computation rule* (sometimes also called β-rule) telling how the elimination interacts with the terms constructed using the introduction rule.

This type creation scheme is not strict and some of the above rules may be missing or there may be more than one rule of some sort in the introduction of certain types. For some types there will be an additional *uniqueness rule* (or η-rule) expressing that a term of that type is completely determined by its behaviour with respect to the elimination rule.

Universe Types

Universe types are special types and their creation will not follow the scheme outlined in the introduction to this section. Nevertheless, they are an essential part of Martin-Löf type theory. The idea is to make the distinction between types and values less strict by considering types as values of some other type, called a *universe type* and denoted by $\mathcal{U}$. Such a universe type is therefore a type of types. It is then natural to wonder of what type this universe $\mathcal{U}$ should be. An early version of Martin-Löf's type theory contained an axiom stating that $\mathcal{U}$ is a value of itself (so it is of type $\mathcal{U}$), but Jean-Yves Girard showed that this leads to a contradiction in the sense that he could provide a term of the empty type described below (see [Gir72], and also [Hur95] for a simplified version of the paradox). Hence in the version of MLTT we are discussing here, there is a hierarchy of universes $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$ and every universe has the subsequent universe as its type. Note that the integers $0, 1, 2, \ldots$ that appear as the index in a universe type are metatheoretic and hence they will be different from the natural numbers we will introduce in one of the following paragraphs.

To make the above discussion more precise, we will have the following inference rules.

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \mathcal{U}_\ell : \mathcal{U}_{\ell+1}}\ \textsc{univ-intro} \qquad \frac{\Gamma \vdash T : \mathcal{U}_k \qquad k \leq \ell}{\Gamma \vdash T : \mathcal{U}_\ell}\ \textsc{univ-cumul}$$

$$\frac{\Gamma \vdash T : \mathcal{U}_\ell}{\Gamma \vdash T \text{ type}}\ \textsc{univ-type} \qquad \frac{\Gamma \vdash T = S : \mathcal{U}_\ell}{\Gamma \vdash T = S \text{ type}}\ \textsc{univ-type-=}$$

The rule UNIV-INTRO holds for all natural numbers ℓ, so we have that $\mathcal{U}_0$ has type $\mathcal{U}_1$, which in its turn has type $\mathcal{U}_2$, etc. Likewise the inference rule UNIV-CUMUL holds for all numbers k and ℓ, however we have here the hypothesis $k \leq \ell$, which is strictly speaking not a judgement as defined in section 1.4.1. To be extremely precise, we could modify this rule so that it has only one hypothesis (requiring that T has type $\mathcal{U}_\ell$ in a context Γ) and state as a side condition that this rule only holds for all natural numbers k and ℓ with $k \leq \ell$. This rule expresses the fact that the universes are cumulative, so every value of for instance $\mathcal{U}_0$, will also have the types $\mathcal{U}_1, \mathcal{U}_2, \ldots$, and hence we clearly see that typing is not unique. Types in the universe $\mathcal{U}_0$ are sometimes called *small types*.

Dependent Products

We will now start to apply the type creation scheme discussed in this section's introduction and we begin with *dependent product types*, also called dependent function types. These types are a generalization of the function types from the simply typed lambda calculus or System F. The main difference is that the output type of a dependent function (i.e. a value of a dependent product type) may depend on the actual argument that is passed to this function. Suppose that we have a type B depending on a variable x of type A. Then the type $\Pi(x : A).B$ is the dependent product type of functions that take an argument x of type A and produce a result of type B (depending on the argument x).

The formation rule for dependent products tells us that $\Pi(x : A).B$ is a well-formed type living in the same universe as the types A and B.

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell \qquad \Gamma, x : A \vdash B : \mathcal{U}_\ell}{\Gamma \vdash \Pi(x : A).B : \mathcal{U}_\ell}\ \Pi\text{-}\textsc{form}$$

Note that the variable x gets bound by the type constructor Π, so every occurrence of x in $\Pi(x : A).B$ will be bound. We also extend α-equivalence, so the types $\Pi(x : A).B$ and $\Pi(y : A).B[y/x]$ will be considered definitionally equal. In the following paragraphs, we will see more examples of structures that bind variables and the notion of α-equivalence will always be extended appropriately without explicitly mentioning it.

Just as with function types, we can construct terms of a dependent product type by giving a lambda abstraction.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A)\,.\,t : \Pi(x : A).B}\ \Pi\text{-}\textsc{intro}$$

Example 1.25. Although we have not yet introduced natural numbers and we will not formally introduce the type constructor Vec described in the introduction on page 14, we can use these constructs intuitively to consider an example of a dependent function. If we denote by $\mathsf{zerolist}_n$ the list of length n containing only zeroes, then the function $\lambda(n : \mathbb{N})\,.\,\mathsf{zerolist}_n$ is a dependent function of type $\Pi(n : \mathbb{N}).\,\mathrm{Vec}_n\,\mathbb{N}$.

There are two congruence rules stating that the type constructor Π and the term constructor λ respect definitional equalities.

$$\frac{\Gamma \vdash A = A' : \mathcal{U}_\ell \qquad \Gamma, x : A \vdash B = B' : \mathcal{U}_\ell}{\Gamma \vdash \Pi(x : A).B = \Pi(x : A').B' : \mathcal{U}_\ell}\ \Pi\text{-}\textsc{cong-type}$$

$$\frac{\Gamma \vdash A = A' : \mathcal{U}_\ell \qquad \Gamma, x : A \vdash t = t' : B}{\Gamma \vdash \lambda(x : A)\,.\,t = \lambda(x : A')\,.\,t' : \Pi(x : A).B}\ \Pi\text{-}\textsc{cong-term}$$

From now on we will not explicitly mention these congruence rules anymore, but we assume that all constructors respect definitional equality.

The elimination rule for dependent product types must tell us how we can use terms of such a type, or in other words how we can use a dependent function. This will be the familiar application of a function to some other term.

$$\frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B[a/x]}\ \Pi\text{-}\textsc{elim}$$

We indeed see that the output type of a dependent function depends on its argument, because if we apply f to a the result is of type B where we substitute the variable x by the supplied argument a. Note that in the rule Π-elim we do not assume that f is a lambda abstraction, it can be any term of the dependent product type involved in the inference rule (for instance a variable that is assumed to have this type). The computation rule for dependent products will then tell us what happens if we have an application in which the dependent function is a lambda abstraction.

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A) \,.\, t)\, a = t[a/x] : B[a/x]}\ \Pi\text{-}\textsc{comp}$$

This corresponds to the β-reduction in section 1.1.2 and explains why the computation rule is sometimes also called a β-rule.

For dependent products we will also have a uniqueness principle or η-rule stating that every term of a dependent product type is definitionally equal to a lambda abstraction. This principle of η-expansion can also be introduced in the untyped or simply typed lambda calculus, giving rise to another kind of semantics than the one we presented in this chapter (see for instance [Bar85]).

$$\frac{\Gamma \vdash f : \Pi(x : A).B}{\Gamma \vdash f = \lambda(x : A) \,.\, f\, x : \Pi(x : A).B}\ \Pi\text{-}\textsc{uniq}$$

A side condition for this inference rule is that the variable x must not appear free in the term f.

Remark 1.26. If the type B does not depend on the variable x of type A, we abbreviate the dependent product type $\Pi(x : A).B$ as $A \to B$. Indeed, in this case the elements of this type are ordinary non-dependent functions.

Dependent Sums

Dependent sums are types of pairs whose first component determines the type of the second component. More concretely, if B is a type depending on a variable x of type A, then $\Sigma(x : A).B$ is the type of pairs with a first component x of type A and a second component of type B. The formation rule is similar to that of dependent products.

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell \qquad \Gamma, x : A \vdash B : \mathcal{U}_\ell}{\Gamma \vdash \Sigma(x : A).B : \mathcal{U}_\ell}\ \Sigma\text{-}\textsc{form}$$

Again the variable x gets bound by the type constructor Σ.

The introduction rule tells us that we can construct pairs as values of a dependent sum type.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma(x : A).B}\ \Sigma\text{-}\textsc{intro}$$

Example 1.27. In the same style as example 1.25 we can see that the pair $(3, [0, 0, 0])$ is a value of type $\Sigma(n : \mathbb{N}).\, \mathrm{Vec}_n\, \mathbb{N}$.

The elimination rule for dependent sums looks a bit more complicated.

$$\frac{\Gamma, z : \Sigma(x : A).B \vdash C \text{ type} \qquad \Gamma, x : A, y : B \vdash c : C[(x, y)/z] \qquad \Gamma \vdash p : \Sigma(x : A).B}{\Gamma \vdash \mathsf{ind}_\Sigma(z.C, x.y.c, p) : C[p/z]}\ \Sigma\text{-}\textsc{elim}$$

This rule allows us to construct, given a type C depending on a variable z of type $\Sigma(x : A).B$ and given a value p of type $\Sigma(x : A).B$, a value of type $C[p/z]$. More concretely, the rule says that for this purpose, it is enough to specify a value c of type $C[(x,y)/z]$ where $x : A$ and $y : B$ are variables. In other words the elimination principle more or less tells us that in order to use a dependent pair p, we may assume that it has the form (x,y) for some x and y. The computation rule for dependent sums will then say what happens if p is indeed of this form.

$$\frac{\begin{array}{ll}\Gamma, z : \Sigma(x : a).B \vdash C \text{ type} & \Gamma, x : A, y : B \vdash c : C[(x,y)/z] \\ \Gamma \vdash a : A & \Gamma \vdash b : B[a/x]\end{array}}{\Gamma \vdash \mathsf{ind}_\Sigma(z.C, x.y.c, (a,b)) = c[a/x, b/y] : C[(a,b)/z]}\ \Sigma\text{-COMP}$$

Note that in $\mathsf{ind}_\Sigma(z.C, x.y.c, p)$ the variables z, x and y become bound, which is denoted by writing for instance $z.C$ instead of just C.

For any dependent sum type $\Sigma(x : A).B$ we can now define the projection functions $\mathsf{fst} : (\Sigma(x : A).B) \to A$ and $\mathsf{snd} : \Pi(p : \Sigma(x : A).B).B[\mathsf{fst}\, p/x]$ as

$$\begin{aligned}\mathsf{fst} &:= \lambda(p : \Sigma(x : A).B)\,.\,\mathsf{ind}_\Sigma(z.A, x.y.x, p)\\ \mathsf{snd} &:= \lambda(p : \Sigma(x : A).B)\,.\,\mathsf{ind}_\Sigma(z.B[\mathsf{fst}\, z/x], x.y.y, p).\end{aligned}$$

To conclude, there will also be an η-rule stating that every term of a dependent sum type is definitionally equal to a pair.

$$\frac{\Gamma \vdash p : \Sigma(x : A).B}{\Gamma \vdash p = (\mathsf{fst}\, p, \mathsf{snd}\, p) : \Sigma(x : A).B}\ \Sigma\text{-UNIQ}$$

Remark 1.28. If the type B does not depend on the variable $x : A$, then the dependent sum type $\Sigma(x : A).B$ will be abbreviated as $A \times B$. Furthermore, we could also have chosen to formulate the elimination rule for dependent sums in terms of the first and second projection functions fst and snd and subsequently define the induction principle ind_Σ in terms of these projections.

The Unit Type

The *unit type* is a very simple non-dependent type, denoted by $\top$. It contains only one value that will be written as tt. The formation and introduction rules are very simple.

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \top : \mathcal{U}_0}\ \top\text{-FORM} \qquad \frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \mathsf{tt} : \top}\ \top\text{-INTRO}$$

The elimination rule for the unit type is similar to the induction principle for dependent sums, making precise the intuition that tt is the only interesting value of type $\top$.

$$\frac{\Gamma, x : \top \vdash A \text{ type} \qquad \Gamma \vdash a : A[\mathsf{tt}/x] \qquad \Gamma \vdash t : \top}{\Gamma \vdash \mathsf{ind}_\top(x.A, a, t) : A[t/x]}\ \top\text{-ELIM}$$

Again the variable x is bound by $\mathsf{ind}_\top$. The computation rule will tell us what happens when $\mathsf{ind}_\top$ is used with the value tt instead of a general term $t : \top$.

$$\frac{\Gamma, x : \top \vdash A \text{ type} \qquad \Gamma \vdash a : A[\mathsf{tt}/x]}{\Gamma \vdash \mathsf{ind}_\top(x.A, a, \mathsf{tt}) = a : A[\mathsf{tt}/x]}\ \top\text{-COMP}$$

We do not have an η-rule for the unit type.

The Empty Type

The *empty type* is, as its name suggests, a type with no elements. Hence there will be no introduction or computation rules. The elimination rule for the empty type allows us to construct an element of any type in the hypothetical case that we are able to provide a value of the empty type (which is in fact possible if the context contains a variable that is assumed to have the empty type).

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \bot : \mathcal{U}_0}\ \bot\text{-FORM} \qquad \frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash t : \bot}{\Gamma \vdash \mathsf{ind}_\bot(t) : A}\ \bot\text{-ELIM}$$

Natural Numbers

The natural numbers can also be formally introduced in MLTT by following the type creation scheme outlined on page 18. There will be two introduction rules, one defining the element 0 and one providing the successor function $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. For the sake of readability, we will adopt the usual notation for natural numbers as $0, 1, 2, \ldots$ instead of $0, \mathsf{succ}\, 0, \mathsf{succ}\,(\mathsf{succ}\, 0), \ldots$

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash \mathbb{N} : \mathcal{U}_0}\ \mathbb{N}\text{-FORM}$$

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash 0 : \mathbb{N}}\ \mathbb{N}\text{-INTRO-0} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\, n : \mathbb{N}}\ \mathbb{N}\text{-INTRO-SUCC}$$

The elimination principle states that in order to construct a term of some type A depending on a natural number n, we must provide an element for the case $n = 0$ and provide a procedure to advance from the case n to $\mathsf{succ}\, n$. Therefore, this rule formalizes the principle of recursive definition (which under the Curry-Howard correspondence will correspond to proofs by induction). Using these rules, we can define the usual operations on natural numbers like addition and multiplication.

$$\frac{\begin{array}{ll}\Gamma, n : \mathbb{N} \vdash A \text{ type} & \Gamma \vdash z : A[0/n] \\ \Gamma \vdash m : \mathbb{N} & \Gamma, k : \mathbb{N}, a : A[k/n] \vdash s : A[\mathsf{succ}\, k/n]\end{array}}{\Gamma \vdash \mathsf{ind}_\mathbb{N}(n.A, z, k.a.s, m) : A[m/n]}\ \mathbb{N}\text{-ELIM}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash A \text{ type} \qquad \Gamma \vdash z : A[0/n] \qquad \Gamma, k : \mathbb{N}, a : A[k/n] \vdash s : A[\mathsf{succ}\, k/n]}{\Gamma \vdash \mathsf{ind}_\mathbb{N}(n.A, z, k.a.s, 0) = z : A[0/n]}\ \mathbb{N}\text{-COMP-0}$$

$$\frac{\begin{array}{ll}\Gamma, n : \mathbb{N} \vdash A \text{ type} & \Gamma \vdash z : A[0/n] \\ \Gamma \vdash m : \mathbb{N} & \Gamma, k : \mathbb{N}, a : A[k/n] \vdash s : A[\mathsf{succ}\, k/n]\end{array}}{\Gamma \vdash \mathsf{ind}_\mathbb{N}(n.A, z, k.a.s, \mathsf{succ}\, m) = s[m/k, \mathsf{ind}_\mathbb{N}(n.A, z, k.a.s, m)/a] : A[\mathsf{succ}\, m/n]}\ \mathbb{N}\text{-COMP-SUCC}$$

Identity Types

Besides definitional equality, MLTT provides another way to express that two things are equal, called *propositional equality*. Unlike definitional equality, which can be derived using special kinds of judgements, propositional equality is expressed using types, the so-called *identity types*. These identity types are an essential part of MLTT and they will play an important role in the Curry-Howard correspondence. For any type A and any two terms $a, b : A$ there will be an identity type $a \equiv_A b$ living in the same universe as A.

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell \qquad \Gamma \vdash a, b : A}{\Gamma \vdash a \equiv_A b : \mathcal{U}_\ell}\ \textsc{id-form}$$

The identity type $a \equiv_A b$ can be thought of as the type of proofs that a and b are equal. Hence we will only provide a way to construct an element of the type $a \equiv_A b$ if a and b are the same. More concretely, the introduction rule for identity types gives us for any $a : A$ a reflexivity proof that a is equal to itself.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}\, a : a \equiv_A a}\ \textsc{id-intro}$$

From the congruence rule for identity types, which we have not explicitly written out, it follows that if $a, b : A$ are definitionally equal, then so are the types $a \equiv_A a$ and $a \equiv_A b$. Using the constructor refl and the rule TERM-CONV, we can then construct an element of the type $a \equiv_A b$, so in some sense definitional equality implies propositional equality.[7]

The elimination rule for identity types is also called the *J-rule*.

$$\frac{\begin{array}{ll}\Gamma \vdash a, b : A & \Gamma, y : A, w : a \equiv_A y \vdash C \text{ type} \\ \Gamma \vdash e : a \equiv_A b & \Gamma \vdash c : C[a/y, \mathsf{refl}\, a/w]\end{array}}{\Gamma \vdash J(a, b, y.w.C, e, c) : C[b/y, e/w]}\ \textsc{id-elim}$$

It tells us that if we have values $a, b : A$ and a proof $e : a \equiv_A b$ that a and b are equal and if we have a type C depending on variables $y : A$ and $w : a =_A y$, then in order to construct a value of type $C[b/y, e/w]$ it suffices to give a value c of type $C[a/y, \mathsf{refl}\, a/w]$. In other words, if we want to construct a value of a type depending on y and w and we know there is a value $e : a \equiv_A b$, then we must only consider the case where a and b are the same and where w is the reflexivity proof that a is equal to itself. The computation rule for identity types tells us what happens if a and b are indeed the same and w is indeed the reflexivity proof.

$$\frac{\Gamma \vdash a : A \qquad \Gamma, y : A, w : a \equiv_A y \vdash C \text{ type} \qquad \Gamma \vdash c : C[a/y, \mathsf{refl}\, a/w]}{\Gamma \vdash J(a, a, y.w.C, \mathsf{refl}\, a, c) = c : C[a/y, \mathsf{refl}\, a/w]}\ \textsc{id-comp}$$

Other Types

There are many more types that can be defined using the type creation scheme from page 18. For instance for any types A and B we can consider the coproduct type $A + B$ that behaves as a kind of disjoint union of A and B. Its values are of the form $\mathsf{inl}\, a$ for some $a : A$ or $\mathsf{inr}\, b$ for some $b : B$. As another example, one can also define a type constructor List such that the values of List A are the lists whose elements are of type A. In the same way, the type $\mathrm{Vec}_n\, A$ as described on page 14 could be defined for any $n : \mathbb{N}$ and any type A.

[7]The statement that definitional equality implies propositional equality is meant to be informal and we emphasize that definitional and propositional equality are expressed using different kinds of judgements.

1.4.3 The Curry-Howard Correspondence

There is a connection between type theory and logic called the Curry-Howard correspondence. The general philosophy is that a proposition or mathematical statement can be regarded as the type of its proofs. So under the Curry-Howard correspondence, propositions will correspond to types and a proof of a proposition will correspond to a value of the type corresponding to that proposition. The interpretation of the identity type $a \equiv_A b$ as the type of proofs that a and b are equal fits exactly into this picture. In the following paragraphs we will see that not only identity types but also most of the other type constructors described in the previous section have such a logical interpretation.

Let us first consider logical conjunctions. Suppose that we have propositions P and Q corresponding to types A and B respectively. Then proving the conjunction $P \wedge Q$ amounts to giving both a proof of P and a proof of Q. As A is the type of all proofs of P and B the type of proofs of Q, giving a proof of $P \wedge Q$ is therefore equivalent to giving a value of A and a value of B, or in other words giving a value of $A \times B$. Hence we see that the proposition $P \wedge Q$ corresponds to the type $A \times B$.

In the same way, logical implication corresponds to non-dependent function types. Indeed, proving $P \Rightarrow Q$ is the same as giving a procedure for transforming a proof of P into a proof of Q. Translated into type theory, we get a procedure that transforms values of type A into values of type B, which is exactly what a function of type $A \to B$ does.

Similarly, logical disjunction $P \vee Q$ corresponds to the coproduct type $A + B$.

Finding a type-theoretic equivalent for negation is less straightforward. Proving the negation of a proposition P amounts to showing that there is no proof of P. This corresponds to proving that the type A is empty. We can do this by giving a function of type $A \to \bot$. Indeed, if there were a value of type A, we could then pass it as an argument to this function to obtain a value of the empty type $\bot$, which is a contradiction.

Suppose now that we have a type A and let B be a type depending on a variable $x : A$. Using the interpretation of propositions as types, B corresponds to a proposition depending on a value of type A, so it is a predicate on the type A. Now universal quantification over A, saying that all values of type A have property B, corresponds to the dependent product type $\Pi(x : A).B$. Indeed, a function of this type maps every element $a : A$ to a value of type $B[a/x]$, which can be seen as a proof that a has property B. Clearly, a function that maps any $a : A$ to a proof that a has property B is itself a proof that every a has property B.

In the same way, existential quantification, saying that there exists a value of type A having property B, corresponds to the dependent sum type $\Sigma(x : A).B$. Indeed, a value of this sum type is a pair (a, b) where $a : A$ and $b : B[a/x]$, so it is a pair of a value a of type A together with a proof that this a has property B.

See Table 1.1 for a summary of the Curry-Howard correspondence.

Example 1.29. Translating the proposition that for any natural number n the sum $n+n$ is even gives us the type

$$\Pi(n : \mathbb{N}).\Sigma(m : \mathbb{N}).n + n \equiv_{\mathbb{N}} 2 * m$$

and proving the proposition is equivalent to giving a value of the above type (which the inference rules for natural numbers allow us to do).

Proposition	Type
true	$\top$
false	$\bot$
equality $a = b$	identity $a \equiv_A b$
conjunction $P \wedge Q$	product $A \times B$
disjunction $P \vee Q$	sum $A + B$
implication $P \Rightarrow Q$	function $A \to B$
negation $\neg P$	function $A \to \bot$
universal quantification $\forall x \in A : Q$	dependent product $\Pi(x : A).B$
existential quantification $\exists x \in A : Q$	dependent sum $\Sigma(x : A).B$

Table 1.1: Summary of the Curry-Howard correspondence.

Example 1.30. The statement that equality for a certain type A is symmetric corresponds to the following type.

$$\Pi(a, b : A).(a \equiv_A b) \to (b \equiv_A a)$$

A function of this type will take values a and b of type A and a proof that a is equal to b and it transforms this into a proof that b is equal to a. The J-rule allows us to construct such a dependent function. Similarly, the proposition that equality is transitive translates to the type

$$\Pi(a, b, c : A).(a \equiv_A b) \to (b \equiv_A c) \to (b \equiv_A c)$$

and the J-rule also provides a way to construct a dependent function of this type.

Example 1.31. We can also use the J-rule to construct a term that proves that propositional equality is a congruence. More specifically, for any types A and B we can construct a function of type

$$\Pi(f : A \to B).\Pi(a, b : A).(a \equiv_A b) \to (f\,a \equiv_B f\,b).$$

Indeed, application of the J-rule to the term of type $a \equiv_A b$ tells us that it is sufficient to provide a term of type $f\,a \equiv_B f\,a$ and for that we can take $\mathsf{refl}\,(f\,a)$. Hence we can construct a function

$$\mathsf{cong}_{A,B} := \lambda(f : A \to B)\,.\,\lambda(a, b : A)\,.\,\lambda(e : a \equiv_A b)\,.\,J\big(a, b, y.w.(f\,a \equiv_B f\,y), e, \mathsf{refl}\,(f\,a)\big)$$

which has the type that was given above.

Remark 1.32. An important remark is that at the logic side of the Curry-Howard correspondence we do not have classical logic but constructive logic. This means that we cannot use the law of excluded middle stating that $P \vee \neg P$ holds for every proposition P. Indeed the law of excluded middle translates to the type

$$\mathsf{LEM}_\ell := \Pi(X : \mathcal{U}_\ell).X + (X \to \bot)$$

and it can be shown that we cannot construct a function of this type only using the inference rules introduced in this chapter. On the other hand, it can also be shown that we cannot construct a function of type $\mathsf{LEM}_\ell \to \bot$, so we cannot prove the negation of the law of excluded middle either. This means that the law of excluded middle is independent of the inference rules we provided so far, and we could add an axiom providing a value of the above type. However, we will not need such a value for what follows in this thesis.

Example 1.33. In mathematics, we say that two set-theoretic functions f and g from A to B are equal if $f(a) = g(a)$ for all elements $a \in A$. This principle is known as *function extensionality*. In MLTT we could wonder given two dependent functions $f, g : \Pi(x : A).B$ and a proof $e : \Pi(x : A).f\,x \equiv_B g\,x$ that f and g map all values of A to equal elements, whether we can construct a proof that f and g are equal, or in other words a term of type $f \equiv_{\Pi(x:A).B} g$. Just as for the law of excluded middle, this principle of function extensionality cannot be proved nor disproved and hence it is independent of all inference rules we have introduced so far. In this thesis we will assume that we have function extensionality and hence we include a new inference rule

$$\frac{\Gamma \vdash f, g : \Pi(x : A).B \qquad \Gamma \vdash e : \Pi(x : A).f\,x \equiv_B g\,x}{\Gamma \vdash \mathsf{funext}\,e : f \equiv_{\Pi(x:A).B} g}\,\text{FUNEXT}$$

providing us for any proof e that f and g map all values of type A to equal values of type B with an axiomatic element of type $f \equiv_{\Pi(x:A).B} g$. Note that this element does not evaluate to anything, so if we encounter an application of function extensionality during some computation, this computation will be blocked.

1.4.4 Polymorphism in Dependent Type Theory

Although we have only ordinary variables and no type variables in MLTT, we can construct polymorphic functions that behave in a similar way as the ones in System F. This crucially depends on the universe types introduced on page 19. Indeed, we can replace a type variable X of System F with an ordinary variable x that is assumed to have some universe type $\mathcal{U}_\ell$. This leads for instance to a polymorphic identity function

$$\mathsf{id}_\ell := \lambda(X : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,x$$

which has type $\Pi(X : \mathcal{U}_\ell).X \to X$ in the empty context. Hence we see that the $\forall$ type constructor gets replaced with the construct $\Pi(X : \mathcal{U}_\ell).\ldots$ In the same way we can consider the Church encoding of the booleans

$$\mathsf{ctrue}_\ell := \lambda(X : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,\lambda(y : X)\,.\,x$$
$$\mathsf{cfalse}_\ell := \lambda(X : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,\lambda(y : X)\,.\,y$$

which are both of type

$$\mathsf{CBool}_\ell := \Pi(X : \mathcal{U}_\ell).X \to X \to X.$$

However, there is a big difference between the polymorphic functions of System F and those of MLTT. In System F, a polymorphic function can be applied to any type, even to its own type. For instance the term

$$\big(\Lambda X\,.\,\lambda(x : X)\,.\,x\big)\,[\forall X\,.\,X \to X]$$

is well typed in System F. For this reason, System F is called impredicative. On the other hand, a polymorphic function in MLTT can never be applied to its own type. For instance, the function id_ℓ as defined above expects a type argument from the universe $\mathcal{U}_\ell$. However, the type of id_ℓ is $\Pi(X : \mathcal{U}_\ell).X \to X$, which is a type living in the universe $\mathcal{U}_{\ell+1}$. Martin-Löf type theory is said to be predicative.

There is another difference between the polymorphism of System F and that of MLTT, namely polymorphism in System F will automatically be parametric and this is not the case for polymorphism in MLTT. Parametricity and its consequences will be the subject of the next chapter.

Chapter 2

Parametricity

In this chapter we will study, as the title suggests, parametricity and its consequences. Intuitively, a polymorphic function is called parametric if its implementation does not inspect the type it is instantiated with and therefore applies the same algorithm for every type. In languages like System F or Haskell, every polymorphic function is automatically parametric because the type system does not allow the definition of non-parametric functions. The intuition about parametric polymorphism in System F was formalized by John Reynolds using relations, stating that a parametrically polymorphic function maps related arguments to related results [Rey83]. Using this relational interpretation Philip Wadler subsequently described a procedure to derive from a polymorphic System F type a theorem which is satisfied by all polymorphic functions of that type [Wad89]. This will be the subject of section 2.1.

Subsequently, in section 2.2 we will study ParamDTT which is a dependent type system with support for parametricity. This concretely means that there are parametric dependent product types whose values are functions that are enforced to be parametric. Moreover, as we have a dependent type system, we can use the Curry-Howard correspondence (see section 1.4.3) to prove theorems inside the system and ParamDTT provides tools to derive parametricity results à la Wadler internally.

Finally, we will see in section 2.3 some examples of parametricity results in ParamDTT involving Functors. Some of these results will also be useful in the next chapter.

2.1 Parametricity in System F

In example 1.20 we introduced the type CBool of Church encoded booleans in System F as $\forall X \,.\, X \to X \to X$ and we constructed to values ctrue and cfalse of this type. At the end of the example, we wondered whether ctrue and cfalse are the only values of type CBool. The following intuitive argument shows that indeed any closed term of type CBool behaves as either ctrue or cfalse. A closed term t of type CBool must be a type abstraction because its type starts with $\forall X$. Since the type then contains two function types, the body of the type abstraction must consist of a lambda abstraction in another lambda abstraction. In summary, t must be of the form $t = \Lambda X \,.\, \lambda(x : X) \,.\, \lambda(y : X) \,.\, u$ for some term u of type X. However, since this term u must be of type X for any type X, we cannot use any specific constant. Our only possibilities for u are the variables x or y of type X that are in the context at this point. Moreover, there is no possible way in System F for the function t to inspect the type X it is instantiated with. Hence t

cannot use information about X to choose between x or y as return value and therefore it either always returns x (behaving like ctrue) or always returns y (behaving like cfalse). This intuitive argument can be made precise using relational parametricity, which is the subject of this section.

2.1.1 Some Substitution Properties for System F

Before we formally develop the theory of parametricity in System F, we will work out some properties that will turn out to be useful. We could also have discussed these properties in section 1.3.2, but we choose to do it here because we will introduce some notation that is only used in sections 2.1.1 and 2.1.2. This is based on section 5 of [Wad89], but instead of working in an abstract frame model, we simplified the discussion by working in a term model. This has the advantage that we do not have to discuss the semantics of System F in frame models (see for instance [BMM90]) and moreover it will suffice for our description of parametricity in section 2.1.2. More concretely, Wadler's universe $\mathbf{U}$ of type values will be represented here by the set of closed System F types and the other notions are adjusted accordingly, although not always mentioned explicitly (for instance the interpretation of a type or a term in an environment will be replaced by multiple substitutions). Wadler's proposition about soundness of types corresponds to our Proposition 2.2.

We introduce the notation $\mathbb{T}_0 \subset \mathbb{T}_\mathrm{F}$ for the set of all closed types (i.e. types without free type variables) and $\mathcal{T}_0 \subset \mathcal{T}_\mathrm{F}$ for closed terms (i.e. terms without free ordinary variables). More generally, $\mathbb{T}_k \subset \mathbb{T}_\mathrm{F}$ will be the set of all types containing at most k free type variables and $\mathcal{T}_k$ the set of terms containing at most k free ordinary variables.[1] If a type $T \in \mathbb{T}_k$ has free (mutually distinct) type variables $X_1, \ldots, X_k$, we will sometimes denote it by $T(X_1, \ldots, X_k)$. Similarly, we will sometimes write $t(x_1, \ldots, x_k)$ for a term in which the (mutually distinct) ordinary variables $x_1, \ldots, x_k$ appear free.

When A and B are two sets, we denote by B^A the set of mathematical functions from A to B (so these functions are not lambda abstractions, but sets of pairs representing their graph). We can then define a map $\Xi \in \left({\mathbb{T}_0}^{\mathbb{T}_0}\right)^{\mathbb{T}_1}$ by specifying for every type $T \in \mathbb{T}_1$ a function $\Xi(T) : \mathbb{T}_0^{\mathbb{T}_0}$ as

$$\Xi(T)(S) = \begin{cases} T[S/X] & \text{if } T \text{ has exactly one free type variable } X \\ T & \text{if } T \in \mathbb{T}_0. \end{cases}$$

Similarly, we can define a map $\xi \in \left(\mathcal{T}_0^{\mathcal{T}_0}\right)^{\mathcal{T}_1}$ by specifying for every term $t \in \mathcal{T}_1$ a function $\xi(t) \in \mathcal{T}_0^{\mathcal{T}_0}$ where

$$\xi(t)(s) = \begin{cases} t[s/x] & \text{if } t \text{ has exactly one free ordinary variable } x \\ t & \text{if } t \in \mathcal{T}_0. \end{cases}$$

Given a closed type $T \in \mathbb{T}_0$, we can define the set D_T of all closed terms of type T that are in normal form, more specifically

$$D_T = \{t \in \mathcal{T}_0 \mid \vdash t : T \text{ and } t \text{ is in normal form}\}.$$

[1]One would expect at first sight that we should consider a set $\mathcal{T}_{k,l}$ of terms with k free ordinary variables and l free type variables. However, it turns out that the type variables in terms do not have to be mentioned in the results of the current section.

Suppose now that we have two types $T, S \in \mathbb{T}_0$ and a term $u \in D_{T \to S}$. Then the typing rules of System F allow us to conclude that for any $t \in D_T$ the term $u\,t$ has type S. By the preservation property (see Theorem 1.21) the normal form $\mathrm{nform}(u\,t)$ has type S and furthermore it is in normal form. Hence $\mathrm{nform}(u\,t) \in D_S$. This allows us to define a function $\varphi_{TS}(u) : D_T \to D_S$ (not confusing $\to$ with the constructor of function types) by letting $\varphi_{TS}(u)(t) = \mathrm{nform}(u\,t)$ for all $t \in D_T$. In this way, we constructed a mapping $\varphi_{TS} : D_{T \to S} \to (D_S)^{D_T}$.

Suppose now that we are given a type $\forall X\,.\,T \in \mathbb{T}_0$ and a term $t \in D_{\forall X\,.\,T}$. Then we can define the function $\Phi_{\forall X\,.\,T}(t) : \mathbb{T}_0 \to \mathcal{T}_0$ where $\Phi_{\forall X\,.\,T}(t)(S) = \mathrm{nform}(t\,[S])$ for all $S \in \mathbb{T}_0$. Notice that $\Phi_{\forall X\,.\,T}(t)(S) \in D_{T[S/X]} = D_{\Xi(T)(S)}$ for all $S \in \mathbb{T}_0$.

Let Θ be a sequence $X_1, \ldots, X_m$ of mutually distinct type variables and $\bar{A}$ a sequence $A_1, \ldots, A_m$ of closed types. Then for any type $T(\Theta) = T(X_1, \ldots, X_m) \in \mathbb{T}_\mathrm{F}$ we define the multiple substitution

$$T[\bar{A}/\Theta] = T[A_1/X_1] \ldots [A_m/X_m].$$

Notice that the order in which we substitute every type variable is not important since every type A_i is closed and the type variables are assumed to be mutually distinct. In the same way, we can also define $t[\bar{A}/\Theta]$ for a term t. If in addition Γ is a sequence of typing assumptions $x_1 : T_1, \ldots, x_n : T_n$, and $\bar{a}$ is a sequence $a_1, \ldots, a_n$ of closed typable terms (with closed types) in normal form, then we define for every term $t(x_1, \ldots, x_n)$ the multiple substitution

$$t[\bar{A}, \bar{a}/\Theta, \Gamma] = t[\bar{A}/\Theta][a_1/x_1] \ldots [a_n/x_n].$$

Again the order in which the variables are substituted is unimportant (among others because the types of the terms $a_1, \ldots, a_n$ are assumed to be closed). We say that $\bar{A}, \bar{a}$ respect $\Theta; \Gamma$ if $a_i \in D_{T_i[\bar{A}/\Theta]}$ for every i.

Lemma 2.1. *Suppose that $\Theta, \Omega; \Gamma \vdash t : T$ and that $\bar{A}$ is a sequence of closed types for Θ. Then we have that $\Omega; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$.*

Of course by $\Gamma[\bar{A}/\Theta]$ we mean the sequence of type assumptions on ordinary variables where $x_i : T_i$ is replaced with $x_i : T_i[\bar{A}/\Theta]$ for every i.

Proof. First notice that it makes sense to have Ω as a type variable context in the result because no type variables from Θ appear free due to the substitutions. The proof proceeds by induction on a derivation of $\Theta, \Omega; \Gamma \vdash t : T$. There are 5 possibilities for the last applied inference rule.

- T-Var: $t = x_i$ and $T = T_i$ and $x_i : T_i$ is contained in Γ.
  Then we have that $x_i : T_i[\bar{A}/\Theta]$ is contained in $\Gamma[\bar{A}/\Theta]$. Hence, since $t[\bar{A}/\Theta] = x_i[\bar{A}/\Theta] = x_i$, we get that $\Omega; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$.

- T-Abs: $t = \lambda(x : U)\,.\,v$ and $\Theta, \Omega; \Gamma, (x : U) \vdash v : V$ and $T = U \to V$.
  Using the induction hypothesis, we see that

  $$\Omega; \Gamma[\bar{A}/\Theta], (x : U[\bar{A}/\Theta]) \vdash v[\bar{A}/\Theta] : V[\bar{A}/\Theta].$$

  Since $t[\bar{A}/\Theta] = \lambda(x : U[\bar{A}/\Theta])\,.\,\big(v[\bar{A}/\Theta]\big)$ and $(U \to V)[\bar{A}/\Theta] = U[\bar{A}/\Theta] \to V[\bar{A}/\Theta]$, it follows from inference rule T-Abs that $\Omega; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$.

- T-App: $t = u\,v$ and $\Theta, \Omega; \Gamma \vdash u : V \to T$ and $\Theta, \Omega; \Gamma \vdash v : V$.
  Using the induction hypothesis, we get that

  $$\begin{aligned}\Omega; \Gamma[\bar{A}/\Theta] &\vdash u[\bar{A}/\Theta] : V[\bar{A}/\Theta] \to T[\bar{A}/\Theta] \\ \Omega; \Gamma[\bar{A}/\Theta] &\vdash v[\bar{A}/\Theta] : V[\bar{A}/\Theta].\end{aligned}$$

  As $t[\bar{A}/\Theta] = \big(u[\bar{A}/\Theta]\big)\,\big(v[\bar{A}/\Theta]\big)$, we get that $\Omega; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$.

- T-Tabs: $t = \Lambda X\,.\,u$ and $\Theta, \Omega, X; \Gamma \vdash u : U$ and $T = \forall X\,.\,U$.
  The induction hypothesis tells us that

  $$\Omega, X; \Gamma[\bar{A}/\Theta] \vdash u[\bar{A}/\Theta] : U[\bar{A}/\Theta].$$

  Because $t[\bar{A}/\Theta] = \Lambda X\,.\,\big(u[\bar{A}/\Theta]\big)$ and $T[\bar{A}/\Theta] = \forall X\,.\,\big(U[\bar{A}/\Theta]\big)$, we get that $\Omega; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$.

- T-Tapp: $t = u\,[V]$ and $\Theta, \Omega; \Gamma \vdash u : \forall X\,.\,U$ and $T = U[V/X]$.
  Now we can apply the induction hypothesis to see that

  $$\Omega; \Gamma[\bar{A}/\Theta] \vdash u[\bar{A}/\Theta] : \forall X\,.\,\big(U[\bar{A}/\Theta]\big)\,.$$

  From this it follows that $\Omega; \Gamma[\bar{A}/\Theta] \vdash \big(u[\bar{A}/\Theta]\big)\,[V[\bar{A}/\Theta]] : U[\bar{A}/\Theta][V[\bar{A}/\Theta]/X]$. Since $t[\bar{A}/\Theta] = \big(u[\bar{A}/\Theta]\big)\,[V[\bar{A}/\Theta]]$ and $U[\bar{A}/\Theta][V[\bar{A}/\Theta]/X] = T[\bar{A}/\Theta]$, we get that $\Omega; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$. $\square$

Proposition 2.2. *If $\Theta; \Gamma \vdash t : T$, then for all $\bar{A}, \bar{a}$ respecting $\Theta; \Gamma$, we have that* $\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]) \in D_{T[\bar{A}/\Theta]}$.

Proof. Since Θ contains all type variables occurring free in T and since all A_i are closed, $T[\bar{A}/\Theta]$ is a closed type so that the set $D_{T[\bar{A}/\Theta]}$ is defined. Using Lemma 2.1, we get that $\emptyset; \Gamma[\bar{A}/\Theta] \vdash t[\bar{A}/\Theta] : T[\bar{A}/\Theta]$. Furthermore, since $\bar{A}, \bar{a}$ respects $\Theta; \Gamma$, we have that $\vdash a_i : T_i[\bar{A}/\Theta]$. Hence we can use a substitution lemma stating that if $\Omega; \Delta, (y : U) \vdash s : S$ and if $\Omega; \Delta \vdash u : U$ then also $\Omega; \Delta \vdash s[u/y] : S$ (which can be proved by induction just like Lemma 2.1), to conclude that

$$\vdash t[\bar{A}/\Theta][a_1/x_1] \ldots [a_n/x_n] : T[\bar{A}/\Theta],$$

and hence $\vdash t[\bar{A}, \bar{a}/\Theta, \Gamma] : T[\bar{A}/\Theta]$. Since Γ contains all ordinary variables occurring free in t and since all a_i are closed, it is clear that $t[\bar{A}, \bar{a}/\Theta, \Gamma]$ is closed and therefore $\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]) \in D_{T[\bar{A}/\Theta]}$. $\square$

2.1.2 Relational Parametricity

We will now formally introduce the theory of parametricity in System F. The discussion is based on section 6 of [Wad89], but as discussed in section 2.1.1 in order to avoid the need for an introduction to semantics of System F using frame models, we decided to make everything more concrete by using a term model. Throughout this section, we work in System F and we use notation such as $\mathbb{T}_k$ and D_T introduced in section 2.1.1.

The intuitive idea of relational parametricity is as follows. To every closed type T we will associate a binary relation $[\![T]\!]$ on the set D_T. The parametricity theorem for

System F (Theorem 2.4) will then tell us that for every closed term $t \in D_T$ we have that $(t,t) \in [\![T]\!]$, that is, t is related to itself by the relation determined by its type. This information will allow us to prove interesting properties of closed terms with certain types. However, in order to define the relation $[\![T]\!]$ and to prove the parametricity theorem, we will work inductively and hence we need to consider relations $[\![T]\!]$ as well for types T that contain free type variables. For this purpose, the multiple substitution and the properties of section 2.1.1 will be useful.

Suppose that we have types $T, T', S, S' \in \mathbb{T}_0$ and relations $\mathcal{R} \subseteq D_T \times D_{T'}$ and $\mathcal{S} \subseteq D_S \times D_{S'}$. Then we can construct the relation $\mathcal{R} \to \mathcal{S} \subseteq D_{T\to S} \times D_{T'\to S'}$ defining $u \in D_{T\to S}$ and $u' \in D_{T'\to S'}$ to be related (so $(u,u') \in \mathcal{R} \to \mathcal{S}$) if and only if for all $(t,t') \in \mathcal{R}$ we have that $(\varphi_{TS}(u)(t), \varphi_{T'S'}(u')(t')) \in \mathcal{S}$.

For any two types $T, S \in \mathbb{T}_0$ we will denote by $\mathrm{Rel}(T,S)$ the set of relations between D_T and D_S, so we have that $\mathrm{Rel}(T,S) = \mathcal{P}(D_T \times D_S)$ where $\mathcal{P}(A)$ denotes the power set of a set A. Suppose now that we have closed types $\forall X\,.\,T, \forall X'\,.\,T' \in \mathbb{T}_0$ and for any types $S, S' \in \mathbb{T}_0$ a function $\mathcal{F}_{SS'} : \mathrm{Rel}(S,S') \to \mathrm{Rel}(\Xi(T)(S), \Xi(T')(S'))$. Then we define the relation $\forall\mathcal{X}.\mathcal{F}(\mathcal{X})$ between $D_{\forall X\,.\,T}$ and $D_{\forall X'\,.\,T'}$ saying that $(t,t') \in \forall\mathcal{X}.\mathcal{F}(\mathcal{X})$ if and only if for all types $S, S' \in \mathbb{T}_0$ and all relations $\mathcal{S} \in \mathrm{Rel}(S,S')$ we have that $(\Phi_{\forall X\,.\,T}(t)(S), \Phi_{\forall X'\,.\,T'}(t')(S')) \in \mathcal{F}_{SS'}(\mathcal{S})$.

Let Θ be a sequence of type variables $X_1, \ldots, X_m$. Then we define a relation environment for Θ to be a partial function $\bar{\mathcal{A}}$ from the set $\mathcal{V}_T$ of type variables to $\bigcup_{S,S'\in\mathbb{T}_0} \mathrm{Rel}(S,S')$ with domain Θ. If, moreover $\bar{A}$ and $\bar{A}'$ are type environments for Θ, we say that $\bar{\mathcal{A}}$ is a relation environment between $\bar{A}$ and $\bar{A}'$ if $\bar{\mathcal{A}}(X) \in \mathrm{Rel}(\bar{A}(X), \bar{A}'(X))$ for every type variable X in Θ. Given a type $T(X_1, \ldots, X_m)$ and a relation environment $\bar{\mathcal{A}}$ for $\Theta = X_1, \ldots, X_m$, we can define by induction on the structure of T the relation corresponding to T in the environment $\bar{\mathcal{A}}$, denoted by $[\![T]\!]\,\bar{\mathcal{A}}$. We say that

$$\begin{aligned}
[\![X]\!]\,\bar{\mathcal{A}} &= \bar{\mathcal{A}}(X)\\
[\![T \to S]\!]\,\bar{\mathcal{A}} &= [\![T]\!]\,\bar{\mathcal{A}} \to [\![S]\!]\,\bar{\mathcal{A}}\\
[\![\forall X\,.\,T]\!]\,\bar{\mathcal{A}} &= \forall\mathcal{X}.\,[\![T]\!]\,\bar{\mathcal{A}}[\mathcal{X}/X],
\end{aligned}$$

where $\bar{\mathcal{A}}[\mathcal{X}/X]$ is the relation environment that maps X to $\mathcal{X}$ and otherwise behaves as $\bar{\mathcal{A}}$.[2] We must be careful in the last part of this recursive definition (for the type $\forall X\,.\,T$) because in order to construct the relation $\forall\mathcal{X}.\ldots$, we need a function mapping relations to relations with certain properties. This will be made clear in the proof of the following proposition, which could also be interpreted as stating that the definition above is makes sense.

Proposition 2.3. *Let $T(\Theta)$ be a type and $\bar{A}, \bar{A}'$ type environments for Θ. Furthermore, let $\bar{\mathcal{A}}$ be a relation environment between $\bar{A}$ and $\bar{A}'$. Then we have that $[\![T]\!]\,\bar{\mathcal{A}} \in \mathrm{Rel}(T[\bar{A}/\Theta], T[\bar{A}'/\Theta])$.*

Proof. By induction on T. There are 3 possibilities.

- $T = X$ for some X in Θ. In this case, we know that $[\![T]\!]\,\bar{\mathcal{A}} = \bar{\mathcal{A}}(X)$, which is a relation between $\bar{A}(X) = T[\bar{A}/\Theta]$ and $\bar{A}'(X) = T[\bar{A}'/\Theta]$ by the assumption that $\bar{\mathcal{A}}$ is a relation environment between $\bar{A}$ and $\bar{A}'$.

[2]Note that in this definition of $\bar{\mathcal{A}}[\mathcal{X}/X]$ it is not important whether X is already in the domain of $\bar{\mathcal{A}}$. If it is, it remains in the domain and we only change the relation to which it is mapped to $\mathcal{X}$. If X is not in the domain of $\bar{\mathcal{A}}$, we simply add it to the domain.

- $T = U \to V$. Then, by the induction hypothesis we know that

  $$[\![U]\!]\,\bar{\mathcal{A}} \in \mathrm{Rel}(U[\bar{A}/\Theta], U[\bar{A}'/\Theta])$$

  and

  $$[\![V]\!]\,\bar{\mathcal{A}} \in \mathrm{Rel}(V[\bar{A}/\Theta], V[\bar{A}'/\Theta]).$$

  It follows that

  $$\begin{aligned}[\![T]\!]\,\bar{\mathcal{A}} = [\![U]\!]\,\bar{\mathcal{A}} \to [\![V]\!]\,\bar{\mathcal{A}} &\in \mathrm{Rel}(U[\bar{A}/\Theta] \to V[\bar{A}/\Theta], U[\bar{A}'/\Theta] \to V[\bar{A}'/\Theta]) \\ &= \mathrm{Rel}(T[\bar{A}/\Theta], T[\bar{A}'/\Theta]).\end{aligned}$$

- $T = \forall X \,.\, U$. As X is not free in T, we may assume that it does not occur in Θ. Now take any two types $S, S' \in \mathbb{T}_0$ and any relation $\mathcal{S} \in \mathrm{Rel}(S, S')$. Then we see that $\bar{\mathcal{A}}[\mathcal{S}/X]$ is a relation environment between $\bar{A}, S$ and $\bar{A}', S'$. Hence by the induction hypothesis $[\![U]\!]\,\bar{\mathcal{A}}[\mathcal{S}/X] \in \mathrm{Rel}(U[\bar{A}/\Theta][S/X], U[\bar{A}'/\Theta][S'/X])$. Since X does not occur in Θ, we see that

  $$[\![U]\!]\,\bar{\mathcal{A}}[\mathcal{S}/X] \in \mathrm{Rel}\big(\Xi(U[\bar{A}/\Theta])(S), \Xi(U[\bar{A}'/\Theta])(S')\big).$$

  Therefore, the function $\mathcal{F}_{SS'}$ defined by $\mathcal{F}_{SS'}(\mathcal{S}) = [\![U]\!]\,(\bar{\mathcal{A}}[\mathcal{S}/X])$ is a mapping from $\mathrm{Rel}(S, S')$ to $\mathrm{Rel}\big(\Xi(U[\bar{A}/\Theta])(S), \Xi(U[\bar{A}'/\Theta])(S')\big)$, as required by the definition of the $\forall$-operation on relations. Hence we see that $\forall \mathcal{X}.\ [\![U]\!]\,\bar{\mathcal{A}}[\mathcal{X}/X]$ is well-defined and that it is a relation between $\forall X \,.\, U[\bar{A}/\Theta]$ and $\forall X \,.\, U[\bar{A}'/\Theta]$, so that it is an element of $\mathrm{Rel}(T[\bar{A}/\Theta], T[\bar{A}'/\Theta])$. □

In particular, the previous proposition tells us that for a closed type $T \in \mathbb{T}_0$, we have that $[\![T]\!]$ (without relation environment because it is unnecessary) is a relation between D_T and D_T.

Suppose that we are given a context $\Theta; \Gamma$ and two environments $\bar{A}, \bar{a}$ and $\bar{A}', \bar{a}'$ that both respect $\Theta; \Gamma$ and a relation environment $\bar{\mathcal{A}}$ between $\bar{A}$ and $\bar{A}'$. Then we say that $\bar{\mathcal{A}}, \bar{A}, \bar{A}', \bar{a}, \bar{a}'$ respect $\Theta; \Gamma$ if $(a_i, a_i') \in [\![T_i]\!]\,\bar{\mathcal{A}}$ for all assumptions $x_i : T_i$ in Γ.

Theorem 2.4 (Parametricity for System F). *If $\Theta; \Gamma \vdash t : T$ then for all environments $\bar{\mathcal{A}}, \bar{A}, \bar{A}', \bar{a}, \bar{a}'$ that respect $\Theta; \Gamma$ we have that $\big(\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]), \mathrm{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])\big) \in [\![T]\!]\,\bar{\mathcal{A}}$.*

Proof. Notice first that the statement of the theorem makes sense because $[\![T]\!]\,\bar{\mathcal{A}} \in \mathrm{Rel}(T[\bar{A}/\Theta], T[\bar{A}'/\Theta])$ by Proposition 2.3 and also $\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]) \in D_{T[\bar{A}/\Theta]}$ and $\mathrm{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma]) \in D_{T[\bar{A}'/\Theta]}$ by Proposition 2.2.

Now we proceed by induction on a derivation of $\Theta; \Gamma \vdash t : T$. There are 5 possibilities for the inference rule last applied.

- T-Var: $t = x_i$ and $T = T_i$ and $x_i : T_i$ is contained in Γ.
  We see that $\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]) = a_i$ and $\mathrm{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma]) = a_i'$. Because $\bar{\mathcal{A}}, \bar{A}, \bar{A}', \bar{a}, \bar{a}'$ respects $\Theta; \Gamma$, we have that

  $$\big(\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]), \mathrm{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])\big) = (a_i, a_i') \in [\![T_i]\!]\,\bar{\mathcal{A}} = [\![T]\!]\,\bar{\mathcal{A}},$$

  which was to be proved.

- T-Abs: $t = \lambda(x : U)\,.\,v$ and $\Theta; \Gamma, (x : U) \vdash v : V$ and $T = U \to V$.
  Now we have that $[\![T]\!]\,\bar{\mathcal{A}} = [\![U]\!]\,\bar{\mathcal{A}} \to [\![V]\!]\,\bar{\mathcal{A}}$. Take arbitrary $(u, u') \in [\![U]\!]\,\bar{\mathcal{A}}$. Then we see that $\bar{\mathcal{A}}, \bar{A}, \bar{A}', (\bar{a}, u), (\bar{a}', u')$ respect $\Theta; \Gamma, (x : U)$. Hence the induction hypothesis tells us that

  $$\big(\text{nform}(v[\bar{A}, (\bar{a}, u)/\Theta, (\Gamma, x : U)]), \text{nform}(v[\bar{A}', (\bar{a}', u')/\Theta, (\Gamma, x : U)])\big) \in [\![V]\!]\,\bar{\mathcal{A}}.$$

  Furthermore, as we assume that x does not occur in Γ, we have that

  $$\text{nform}(v[\bar{A}, (\bar{a}, u)/\Theta, (\Gamma, x : U)]) = \text{nform}(v[\bar{A}, \bar{a}/\Theta, \Gamma][u/x]) = \varphi_{UV}(t[\bar{A}, \bar{a}/\Theta, \Gamma])(u)$$

  and similarly $\text{nform}(v[\bar{A}', (\bar{a}', u')/\Theta, (\Gamma, x : U)]) = \varphi_{UV}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])(u')$. Therefore, we see that

  $$\big(\varphi_{UV}(t[\bar{A}, \bar{a}/\Theta, \Gamma])(u), \varphi_{UV}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])(u')\big) \in [\![V]\!]\,\bar{\mathcal{A}}.$$

  By definition of $\to$ for relations, it follows that $\big(t[\bar{A}, \bar{a}/\Theta, \Gamma], t[\bar{A}', \bar{a}'/\Theta, \Gamma]\big) \in [\![T]\!]\,\bar{\mathcal{A}}$ from which we can conclude the result since $\text{nform}(t) = t$.

- T-App: $t = u\,v$ and $\Theta; \Gamma \vdash u : V \to T$ and $\Theta; \Gamma \vdash v : V$.
  Using the induction hypothesis, we see that

  $$\begin{aligned}\big(\text{nform}(u[\bar{A}, \bar{a}/\Theta, \Gamma]), \text{nform}(u[\bar{A}', \bar{a}'/\Theta, \Gamma])\big) &\in [\![V]\!]\,\bar{\mathcal{A}} \to [\![T]\!]\,\bar{\mathcal{A}} \\ \big(\text{nform}(v[\bar{A}, \bar{a}/\Theta, \Gamma]), \text{nform}(v[\bar{A}', \bar{a}'/\Theta, \Gamma])\big) &\in [\![V]\!]\,\bar{\mathcal{A}}.\end{aligned}$$

  Furthermore, we see that

  $$\begin{aligned}\varphi_{VT}(\text{nform}(u[\bar{A}, \bar{a}/\Theta, \Gamma]))(\text{nform}(v[\bar{A}, \bar{a}/\Theta, \Gamma])) &= \text{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]) \\ \varphi_{VT}(\text{nform}(u[\bar{A}', \bar{a}'/\Theta, \Gamma]))(\text{nform}(v[\bar{A}', \bar{a}'/\Theta, \Gamma])) &= \text{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma]).\end{aligned}$$

  From the definition of the constructor $\to$ for relations we can then conclude that $(\text{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]), \text{nform}(t[\bar{A}', \bar{a}/\Theta, \Gamma]')) \in [\![T]\!]\,\bar{\mathcal{A}}$.

- T-Tabs: $t = \Lambda X\,.\,u$ and $\Theta, X; \Gamma \vdash u : U$ and $T = \forall X\,.\,U$.
  We can see that $[\![T]\!]\,\bar{\mathcal{A}} = \forall \mathcal{X}.\,[\![U]\!]\,(\bar{\mathcal{A}}[\mathcal{X}/X])$. Now take arbitrary types $S, S' \in \mathbb{T}_0$ and an arbitrary relation $\mathcal{S} \in \text{Rel}(S, S')$. Then we see that the environments $(\bar{\mathcal{A}}[\mathcal{S}/X]), (\bar{A}, S), (\bar{A}', S'), \bar{a}, \bar{a}'$ respect $\Theta, X; \Gamma$. Therefore, the induction hypothesis gives us that

  $$\big(\text{nform}(u[(\bar{A}, S), \bar{a}/(\Theta, X), \Gamma]), \text{nform}(u[(\bar{A}', S'), \bar{a}'/(\Theta, X), \Gamma])\big) \in [\![U]\!]\,\bar{\mathcal{A}}[\mathcal{S}/X].$$

  As we assume that X does not occur in Θ, we also have that

  $$\text{nform}(u[(\bar{A}, S), \bar{a}/(\Theta, X), \Gamma]) = \text{nform}(u[\bar{A}, \bar{a}/\Theta, \Gamma][S/X]) = \Phi_{\forall X\,.\,U}(t[\bar{A}, \bar{a}/\Theta, \Gamma])(S)$$

  and similarly $\text{nform}(u[(\bar{A}', S'), \bar{a}'/(\Theta, X), \Gamma]) = \Phi_{\forall X\,.\,U}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])(S')$. Hence we see that $\big(\Phi_{\forall X\,.\,U}(t[\bar{A}, \bar{a}/\Theta, \Gamma])(S), \Phi_{\forall X\,.\,U}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])(S')\big) \in [\![U]\!]\,\bar{\mathcal{A}}[\mathcal{S}/X]$. By definition of $\forall$ for relations, we have that $\big(t[\bar{A}, \bar{a}/\Theta, \Gamma], t[\bar{A}', \bar{a}'/\Theta, \Gamma]\big) \in [\![T]\!]\,\bar{\mathcal{A}}$ and the result follows since $\text{nform}(t) = t$.

- T-Tapp $t = u\,[V]$ and $\Theta; \Gamma \vdash u : \forall X\,.\,U$ and $T = U[V/X]$.
  The induction hypothesis tells us that

  $$\big(\mathrm{nform}(u[\bar{A}, \bar{a}/\Theta, \Gamma]), \mathrm{nform}(u[\bar{A}', \bar{a}'/\Theta, \Gamma])\big) \in [\![\forall X\,.\,U]\!]\,\bar{\mathcal{A}}.$$

  Moreover, we know that $[\![\forall X\,.\,U]\!]\,\bar{\mathcal{A}} = \forall \mathcal{X}.\,[\![U]\!]\,(\bar{\mathcal{A}}[\mathcal{X}/X])$. Using the relation $[\![V]\!]\,\bar{\mathcal{A}} \in \mathrm{Rel}(V[\bar{A}/\Theta], V[\bar{A}'/\Theta])$, we can apply the definition of $\forall$ for relations to see that

  $$\big(\mathrm{nform}(u[\bar{A}, \bar{a}/\Theta, \Gamma])\,[V[\bar{A}/\Theta]], \mathrm{nform}(u[\bar{A}', \bar{a}'/\Theta, \Gamma])\,[V[\bar{A}'/\Theta]]\big) \in [\![U]\!]\,\bar{\mathcal{A}}[([\![V]\!]\,\bar{\mathcal{A}})/X].$$

  It can be easily shown that $[\![U]\!]\,\bar{\mathcal{A}}[([\![V]\!]\,\bar{\mathcal{A}})/X] = [\![U[V/X]]\!]\,\bar{\mathcal{A}} = [\![T]\!]\,\bar{\mathcal{A}}$ and that

  $$\mathrm{nform}(u[\bar{A}, \bar{a}/\Theta, \Gamma])\,[V[\bar{A}/\Theta]] = \mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma])$$

  and similarly we also have

  $$\mathrm{nform}(u[\bar{A}', \bar{a}'/\Theta, \Gamma])\,[V[\bar{A}'/\Theta]] = \mathrm{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma]).$$

  Hence we can conclude that $\big(\mathrm{nform}(t[\bar{A}, \bar{a}/\Theta, \Gamma]), \mathrm{nform}(t[\bar{A}', \bar{a}'/\Theta, \Gamma])\big) \in [\![T]\!]\,\bar{\mathcal{A}}$. □

In the special case where $T \in \mathbb{T}_0$ is a closed type and $t \in D_T$ is a closed term in normal form, the theorem states that $(t, t) \in [\![T]\!]$. So a closed term of a closed type is related to itself by the relation corresponding to the type.

Notice that all results proved in this section remain valid if we extend the language of System F with a concrete type A with values $D_A = \{a_1, \ldots, a_n\}$ and interpret $[\![A]\!]$ as the identity relation on D_A. In particular, we can consider a unit type $\top$ with $D_\top = \{\mathsf{tt}\}$ or a type of booleans Bool with $D_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

Example 2.5. We can now apply the parametricity result for System F to the Church Booleans from example 1.20 and the beginning of this section. Suppose that we have a closed term f of type CBool that is in normal form, so $f \in D_{\mathsf{CBool}}$. Then Theorem 2.4 tells us that $(f, f) \in [\![\mathsf{CBool}]\!]$. Furthermore, we have that $[\![\mathsf{CBool}]\!] = \forall \mathcal{X}.\mathcal{X} \to \mathcal{X} \to \mathcal{X}$. This means that for any types $S, S' \in \mathbb{T}_0$ and any relation $\mathcal{S} \in \mathrm{Rel}(S, S')$ we have that $(f\,[S], f\,[S']) \in \mathcal{S} \to \mathcal{S} \to \mathcal{S}$. It then follows that for every $s_1, s_2 \in S$ and $s_1', s_2' \in S$ if $(s_1, s_1') \in \mathcal{S}$ and $(s_2, s_2') \in \mathcal{S}$ then also $(f\,[S]\,s_1\,s_2, f\,[S']\,s_1'\,s_2') \in \mathcal{S}$.

Take now an arbitrary closed type $T \in \mathbb{T}_0$ and arbitrary $t_1, t_2 \in D_T$. If we assume that we have a type Bool with $D_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$, we can consider the case where $S = \mathsf{Bool}, S' = T, s_1 = \mathsf{true}, s_2 = \mathsf{false}, s_1' = t_1, s_2' = t_2$ and $\mathcal{S} = \{(\mathsf{true}, t_1), (\mathsf{false}, t_2)\}$. We then get that $(f\,[\mathsf{Bool}]\,\mathsf{true}\,\mathsf{false}, f\,[T]\,t_1\,t_2) \in \mathcal{S}$. As $f\,[\mathsf{Bool}]\,\mathsf{true}\,\mathsf{false} \in D_{\mathsf{Bool}}$, there are two possibilities. Either $f\,[\mathsf{Bool}]\,\mathsf{true}\,\mathsf{false} = \mathsf{true}$ and then $(\mathsf{true}, f\,[T]\,t_1\,t_2) \in \mathcal{S}$ so that $f\,[T]\,t_1\,t_2 = t_1$ for all T, t_1, t_2 or $f\,[\mathsf{Bool}]\,\mathsf{true}\,\mathsf{false} = \mathsf{false}$ and then $(\mathsf{false}, f\,[T]\,t_1\,t_2) \in \mathcal{S}$ so that $f\,[T]\,t_1\,t_2 = t_2$ for all T, t_1, t_2. This proves that a closed term of type CBool in normal form behaves exactly as ctrue or cfalse.

2.2 Parametricity and Dependent Types

In section 1.4.4, we saw that it is possible to define polymorphic functions in MLTT by considering an abstraction of the form $\lambda(X : \mathcal{U}_\ell)\,.\,\cdots$ for some universe $\mathcal{U}_\ell$. A natural

$\circ$	¶	id	$\sharp$
¶	¶	¶	$\sharp$
id	¶	id	$\sharp$
$\sharp$	¶	$\sharp$	$\sharp$

(a) Definition of $\circ$.

$\backslash$	¶	id	$\sharp$
¶	¶	$\sharp$	$\sharp$
id	¶	id	$\sharp$
$\sharp$	¶	id	id

(b) Definition of $\backslash$.

Table 2.1: Definition of the operations $\circ$ and $\backslash$ on modalities in ParamDTT.

question is then of course whether functions defined in this way are, just like in System F, enforced to be parametric. This is not true in general and it is possible to construct polymorphic functions in MLTT that are not parametric. A concrete example was given by Nuyts, Vezzosi and Devriese in [NVD17]. However, being able to derive theorems about functions only based on their types is a very powerful feature of a type system or programming language and hence the authors presented in the same article a way to restore parametricity. More concretely, they developed ParamDTT, a dependent type system based on MLTT in which the type of a function can contain a parametricity annotation and which provides tools to prove results à la Wadler internally.

The reason why parametricity breaks down in MLTT is the fact that dependent product types are used as a generalization of both function types and universal types of System F. The main difference in System F between a function $f : A \to B$ and a polymorphic function $g : \forall X . C$ is that f can use its argument of type A in its implementation whereas the type argument of g is only available during type checking. The fact that a function g cannot use its type argument in its implementation appears to be crucial for parametricity. By generalizing a universal type $\forall X . C$ to $\Pi(X : \mathcal{U}_\ell).C$ in dependent type theory, polymorphic functions can start using their type argument in their implementation, which is why we lose parametricity. In ParamDTT, this problem is resolved by distinguishing between parametric dependent product types $\Pi^\sharp(x : A).B$ whose functions intuitively cannot use the argument x in their implementation and continuous dependent product types $\Pi^{\mathsf{id}}(x : A).B$ whose functions can use argument x in their implementation.

In the rest of this section, we will more formally explore the parts of ParamDTT that will be useful for this thesis. Note that we will not explicitly mention every difference between MLTT and ParamDTT, but only those differences that will be important in subsequent chapters. This section is completely based on [NVD17], to which we also refer for more details.

2.2.1 Modalities

In ParamDTT, every dependency is equipped with a *modality* which can be *parametric* ($\sharp$), *continuous* (id) or *pointwise* (¶). The continuous modality will be considered default and will henceforth be omitted. We define an order on modalities by ¶ $\leq$ id $\leq$ $\sharp$. Furthermore, ParamDTT provides two operations $\circ$ (composition) and $\backslash$ (left division) on modalities, defined in Table 2.1.[3]

An important place where modalities will show up are typing contexts. More concretely, an assumption $x^\mu : T$ in a context Γ will from now on always have some modality

[3]The motivation for the particular order on modalities as well as the intuition behind the operations on modalities can be found in [NVD17]

μ. The operation $\backslash$ on modalities can be extended to modalities and contexts. More concretely, a context Γ that is left-divided by a modality μ will be denoted by $\mu \backslash \Gamma$ and for every assumption $x^\nu : T$ in Γ the context $\mu \backslash \Gamma$ will contain the assumption $x^{\mu \backslash \nu} : T$. The modality with which a variable is contained in a context Γ is significant when constructing terms in Γ because only a variable x that appears pointwise or continuous in Γ will type-check as a term of the type it is assumed to have. In other words, the rule VAR from MLTT gets replaced by the following inference rule.[4]

$$\frac{\vdash \Gamma, x^\mu : T, \Delta \text{ context} \qquad \mu \leq \mathsf{id}}{\Gamma, x^\mu : T, \Delta \vdash x : T}\ \textsc{var}$$

Another important inference rule that gets replaced in ParamDTT is the rule UNIV-TYPE that allows us to convert a value of a universe into a type.

$$\frac{\sharp \backslash \Gamma \vdash T : \mathcal{U}_\ell}{\Gamma \vdash T \text{ type}}\ \textsc{univ-type}$$

In this way, we see that variables that are parametric in the context Γ can be used in the construction of a type. Indeed, by the left-division of Γ by $\sharp$, every parametric modality gets replaced with a continuous modality and hence such a variable can be used as a term in this context.

ParamDTT now provides for every modality μ a dependent product type $\Pi^\mu(x : A).B$. We will omit the formation rule for such types. The introduction and elimination rules are as follows.

$$\frac{\Gamma, x^\mu : A \vdash t : B}{\Gamma \vdash \lambda(x^\mu : A)\,.\,t : \Pi^\mu(x : A).B}\ \Pi\textsc{-intro} \qquad \frac{\Gamma \vdash f : \Pi^\mu(x : A).B \qquad \mu \backslash \Gamma \vdash a : A}{\Gamma \vdash f\, a^\mu : B[a/x]}\ \Pi\textsc{-elim}$$

We now see that in the definition of a parametric function $\lambda(x^\sharp : A)\,.\,t$ the variable x is parametric in the context when t is type-checked, and hence by inference rule VAR it cannot be used in t except at a parametric position (for instance it can be used as an argument of another parametric function, in which case the context gets left-divided by $\sharp$ an x appears continuous). This makes precise the intuition that "a parametric function cannot use its argument in its implementation." Furthermore, the elimination rule tells us that an application $f\, a^\mu$ depends on the term a with modality μ. As already explained, this implies that a variable that is parametric in the context can still be used as an argument of a parametric function. From now on, we will write $\forall(x : A).B$ for the parametric dependent product type $\Pi^\sharp(x : A).B$. For any modality μ we will also write $\mu A \to B$ instead of $\Pi^\mu(x : A).B$ if the type B does not depend on the variable x.

Similar to dependent products with any modality, ParamDTT also provides dependent sums $\Sigma^\mu(x : A).B$ for any modality μ. These types contain dependent pairs that depend on their first component with modality μ. More concretely, the introduction rule for dependent sums is as follows.

$$\frac{\Gamma \vdash \Sigma^\mu(x : A).B \text{ type} \qquad \mu \backslash \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a^\mu, b) : \Sigma^\mu(x : A).B}\ \Sigma\textsc{-intro}$$

[4]Note that we also use the name VAR for this rule. Since from now on, we will always work in the system ParamDTT, this will not cause any confusion.

In the following chapters, we will quite often use nested dependent sums to represent types of tuples. When specifying such a tuple, we will not always annotate the components with their modalities because this could cause some confusion, especially when the components of the tuple are long expressions. Just as with dependent sums in MLTT, the elimination rule is more difficult and we will therefore omit it. We make however the remark that using the induction principle for a pointwise dependent sum $\Sigma^{\P}(x : A).B$ one can define a first projection function

$$\mathsf{fst}^{\P} : \sharp(\Sigma^{\P}(x : A).B) \to A$$

that is parametric. This will for instance be useful in chapter 4.

Example 2.6. As an example of the modalities in ParamDTT, we can define the polymorphic identity function for a universe $\mathcal{U}_\ell$ as

$$\lambda(X^{\sharp} : \mathcal{U}_\ell) \,.\, \lambda(x : X) \,.\, x : \forall(X : \mathcal{U}_\ell).X \to X.$$

Indeed, when type-checking the body $\lambda(x : X) \,.\, x$ of this function, the variable X is parametric in the context, but it only appears as a type in $\lambda(x : X) \,.\, x$ and this is by the inference rule UNIV-TYPE a parametric position.

As another example, one can define the type of Church encoded booleans in ParamDTT as

$$\mathsf{CBool}_\ell = \forall(X : \mathcal{U}_\ell).X \to X \to X$$

and once we have tools to prove parametricity results internally, we will be able to show a theorem like in example 2.5 for any term of this type.

2.2.2 Internal Parametricity Proofs

In order to prove parametricity results internally in the type system, ParamDTT provides a way to express relations in the type system. There is an interval pseudotype $\mathbb{I}$ with two axiomatic elements $0, 1 : \mathbb{I}$. A relation between two types $C, D : \mathcal{U}_\ell$ is then expressed using a continuous function $B : \mathbb{I} \to \mathcal{U}_\ell$ with $B\,0 = C$ and $B\,1 = D$, which is called a *bridge* between C and D. Subsequently, a proof that $c : C$ and $d : D$ are related, is expressed using a parametric function $p : \forall(i : \mathbb{I}).B\,i$ with $p\,0^{\sharp} = c$ and $p\,1^{\sharp} = d$, which is called a *path* over B between c and d.

The modality of a function can now also be described in terms of these bridges and paths. Continuous functions respect bridges and paths and parametric functions respect paths and strengthen bridges to paths (compare this to section 2.1 where we said that parametric polymorphic functions map related types to related results). Furthermore, pointwise functions respect paths but have no action on bridges.

There is also a *path degeneracy axiom* asserting that the endpoints of a homogeneous path are propositionally equal.[5] We say that a path p is homogeneous if it is a path over a constant bridge, or in other words if p has type $\sharp\mathbb{I} \to A$ for some type A. The use of this axiom will be the final step in most of the parametricity proofs in this thesis. It is however an axiom that has no computational content. In other words, computations in which the path degeneracy axiom is used will block (this situation is similar to that of

[5]Actually the path degeneracy axiom in ParamDTT states that any homogeneous path is (propositionally) constant, but from this it obviously follows that its endpoints are propositionally equal.

function extensionality in example 1.33). More formally, we have the following inference rule.

$$\frac{\Gamma \vdash A \text{ type} \qquad \sharp \setminus \Gamma \vdash p : \forall(i : \mathbb{I}).A}{\Gamma \vdash p\,0 \equiv_A p\,1}\ \textsc{path-to-eq}$$

Furthermore, ParamDTT provides tools to interpret a function $f : C \to D$ as a relation. More concretely, for such a function f there will be a bridge $/f\backslash : \mathbb{I} \to \mathcal{U}_\ell$ with $/f\backslash\,0 = C$ and $/f\backslash\,1 = D$. The relation represented by this bridge will be the graph of the function f. Furthermore there are functions $\mathsf{push}\,f : \forall(i : \mathbb{I}).C \to /f\backslash\,i$ and $\mathsf{pull}\,f : \forall(i : \mathbb{I})./f\backslash\,i \to D$ with $\mathsf{push}\,f\,0^\sharp = \mathsf{id}_C$, $\mathsf{push}\,f\,1^\sharp = f$, $\mathsf{pull}\,f\,0^\sharp = f$ and $\mathsf{pull}\,f\,1^\sharp = \mathsf{id}_D$. This situation is illustrated in the following diagram.

$$\begin{tikzcd} C \arrow[r, "\mathsf{push}\,f\,i^\sharp"] \arrow[rr, bend right, "f"'] & /f\backslash\,i \arrow[r, "\mathsf{pull}\,f\,i^\sharp"] & D \end{tikzcd}$$

The functions push and pull allow us to construct a path between any $c : C$ and its image $f\,c : D$. An important remark is that the graph relation $/f\backslash$ and the functions $\mathsf{push}\,f$ and $\mathsf{pull}\,f$ depend pointwise on f.

However, the operations $/\cdot\backslash$, push and pull are not primitive in ParamDTT, but they are implemented in terms some other primitive constructs. In fact there are two ways to implement them: using Weld types or using Glue types. In the rest of this section, we will present a simplified discussion of Weld and Glue types that only contains the material we will need in the other parts of the thesis or for the implementation of $/\cdot\backslash$, push and pull.

Suppose that we have a type $E : \mathcal{U}_\ell$ that possibly depends on a variable $i : \mathbb{I}$ and suppose furthermore that we have types A and B (not depending on i) and functions $f : E[0/i] \to A$ and $g : E[1/i] \to B$. Then a Weld type will allow us to extend the types A and B and the functions f and g for any $i : \mathbb{I}$. In other words, we have a type

$$\mathsf{Weld}\{E \to (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$$

that is definitionally equal to A if i is equal to 0 (or in other words if the predicate $i \doteq 0$ holds) and is definitionally equal to B if i is equal to 1. Moreover, there is a function

$$\mathsf{weld}(i \doteq 0\,?\,f \mid i \doteq 1\,?\,g) : E \to \mathsf{Weld}\{E \to (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$$

that is definitionally equal to f if i is equal to 0 and is definitionally equal to g if i is equal to 1. This situation is illustrated in Figure 2.1. We make the remark that both the type $\mathsf{Weld}\{E \to (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$ and the function $\mathsf{weld}(i \doteq 0\,?\,f \mid i \doteq 1\,?\,g)$ depend pointwise on the functions f and g.

Using Weld types, the graph relation former and push function for a given function $h : U \to V$ can be implemented as

$$/h\backslash = \lambda(i : \mathbb{I})\,.\,\mathsf{Weld}\{U \to (i \doteq 0\,?\,U, \mathsf{id}_U \mid i \doteq 1\,?\,V, h)\},$$

and

$$\mathsf{push}\,h = \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{weld}(i \doteq 0\,?\,\mathsf{id}_U \mid i \doteq 1\,?\,h).$$

In order to implement the pull function, we need an induction principle for Weld types. This will tell us that if we have some value of type $\mathsf{Weld}\{E \to (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$

$$\begin{tikzcd}
E \arrow[r, "f"] \arrow[d, dash] & A \arrow[d, dash, dashed] \\
E \arrow[r, dotted, "\mathsf{weld}(i \doteq 0\,?\,f \mid i \doteq 1\,?\,g)"] \arrow[d, dash] & \mathsf{Weld}\{E \to (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\} \arrow[d, dash, dashed] \\
E \arrow[r, "g"] & B
\end{tikzcd}$$

Figure 2.1: Illustration of Weld types. The middle row depicts the situation for general i and reduces to the top row if $i \doteq 0$ holds and to the bottom row if $i \doteq 1$ holds.

and we want to construct a value of type C, then it is sufficient to give a value $c : C$ depending on a variable $x : E$, a value $d_0 : C$ depending on a variable $y : A$ assuming that $i \doteq 0$ holds and a value $d_1 : C$ depending on a variable $z : B$ assuming that $i \doteq 1$ holds. Moreover, we need to have definitional equality of c and $d_0[f\,x/y]$ if $i \doteq 0$ holds and of c and $d_1[g\,x/z]$ if $i \doteq 1$ holds. If these conditions are satisfied, we can construct

$$\mathsf{ind}_{\mathsf{Weld}}\big(C,\ (i \doteq 0\,?\,y.d_0 \mid i \doteq 1\,?\,z.d_1),\ x.c,\ w\big) : C$$

for any $w : \mathsf{Weld}\{E \to (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$. This will be definitionally equal to $d_0[w/y]$ if $i \doteq 0$ holds and definitionally equal to $d_1[w/z]$ if $i \doteq 1$ holds. Furthermore, if $w = \mathsf{weld}(i \doteq 0\,?\,f \mid i \doteq 1\,?\,g)\,e$ for some $e : E$, then this is definitionally equal to $c[e/x]$. We can now define $\mathsf{pull}\,h$ for $h : U \to V$ as

$$\mathsf{pull}\,h\,i^{\sharp}\,w = \mathsf{ind}_{\mathsf{Weld}}\big(V,\ (i \doteq 0\,?\,y.(h\,y) \mid i \doteq 1\,?\,z.z)\ x.(h\,x), w\big) : V.$$

Glue types can be seen as the opposite construction of Weld types. Hence we now assume we have a type $E : \mathcal{U}_\ell$ that possibly depends on a variable $i : \mathbb{I}$ and types A and B (not depending on i) and functions $f : A \to E[0/i]$ and $g : B \to E[1/i]$. In this case, a Glue type will enable us to extend the types A and B and the functions f and g for any $i : \mathbb{I}$. Hence we have a type

$$\mathsf{Glue}\{E \leftarrow (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$$

that is definitionally equal to A if $i \doteq 0$ holds and definitionally equal to B if $i \doteq 1$ holds. Additionally, we have a function

$$\mathsf{unglue}(i \doteq 0\,?\,f \mid i \doteq 1\,?\,g) : \mathsf{Glue}\{E \leftarrow (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\} \to E$$

that is definitionally equal to f if $i \doteq 0$ holds and is definitionally equal to g if $i \doteq 1$ holds. This situation is illustrated in Figure 2.2. Just as for Weld types, the type $\mathsf{Glue}\{E \leftarrow (i \doteq 0\,?\,A, f \mid i \doteq 1\,?\,B, g)\}$ and the term $\mathsf{unglue}(i \doteq 0\,?\,f \mid i \doteq 1\,?\,g)$ depend pointwise on the functions f and g.

Using Glue types, the graph relation former and pull function for a given function $h : U \to V$ may be implemented as

$$/h\backslash = \lambda(i : \mathbb{I})\,.\,\mathsf{glue}\{V \leftarrow (i \doteq 0\,?\,U, h \mid i \doteq 1\,?\,V, \mathsf{id}_V)\},$$

$$\begin{tikzcd}
A \arrow[rr, "f"'] \arrow[d, dash, dashed] & & E \arrow[d, dash] \\
\mathsf{Glue}\{E \leftarrow (i \doteq 0 \mathbin{?} A, f \mid i \doteq 1 \mathbin{?} B, g)\} \arrow[rr, dotted, "\mathsf{unglue}(i \doteq 0 \mathbin{?} f \mid i \doteq 1 \mathbin{?} g)"'] \arrow[d, dash, dashed] & & E \arrow[d, dash] \\
B \arrow[rr, "g"'] & & E
\end{tikzcd}$$

Figure 2.2: Illustration of Glue types. The middle row depicts the situation for general i and reduces to the top row if $i \doteq 0$ holds and to the bottom row if $i \doteq 1$ holds.

and

$$\mathsf{pull}\, h = \lambda(i^\sharp : \mathbb{I}) \,.\, \mathsf{unglue}(i \doteq 0 \mathbin{?} h \mid i \doteq 1 \mathbin{?} \mathsf{id}_V).$$

To implement the push function, we need a constructor for Glue types. In order to construct some value of type $\mathsf{Glue}\{E \leftarrow (i \doteq 0 \mathbin{?} A, f \mid i \doteq 1 \mathbin{?} B, g)\}$ we need to give a value $e : E$, a value $a : A$ assuming that $i \doteq 0$ holds and a value $b : B$ assuming that $i \doteq 1$ holds. Moreover, $f\, a$ must be definitionally equal to e if $i \doteq 0$ holds and of $g\, b$ must be definitionally equal to e if $i \doteq 1$ holds. If these conditions are satisfied, we can construct

$$\mathsf{glue}\big(e \hookleftarrow\ (i \doteq 0 \mathbin{?} a \mid i \doteq 1 \mathbin{?} b)\big) : \mathsf{Glue}\{E \leftarrow (i \doteq 0 \mathbin{?} A, f \mid i \doteq 1 \mathbin{?} B, g)\}.$$

This will be definitionally equal to a if $i \doteq 0$ holds and definitionally equal to b if $i \doteq 1$ holds. Moreover, we have that

$$\mathsf{unglue}(i \doteq 0 \mathbin{?} f \mid i \doteq 1 \mathbin{?} g)\big(\mathsf{glue}(e \hookleftarrow\ (i \doteq 0 \mathbin{?} a \mid i \doteq 1 \mathbin{?} b))\big) = e : E.$$

We can now define $\mathsf{push}\, h$ for any $h : U \to V$ as

$$\mathsf{push}\, h\, i^\sharp\, u = \mathsf{glue}\big(h\, u,\ (i \doteq 0 \mathbin{?} u \mid i \doteq 1 \mathbin{?} h\, u)\big) : /h\backslash\, i.$$

2.3 Examples Involving Functors

In this section we will see several examples of parametricity results about functors, functor laws and natural transformations. All results are proved in ParamDTT and have been formalized in Agda parametric.[6] Some of these results will be useful in chapter 3 when we define a monad in ParamDTT.

2.3.1 Definition of a Functor in ParamDTT

In the functional programming language Haskell, a *functor* (or more precisely an instance of the type class `Functor`) is defined to be a type operator `f` together with an action

[6]See `https://github.com/JorisCeulemans/effect-param-agda/blob/master/Functors.agda`.

`fmap :: (a -> b) -> f a -> f b` on functions which is assumed to respect composition and preserve identity functions [Lip11, chapters 8, 11] (as Haskell does not have dependent types, these functor laws cannot be imposed in the language). This is a special case of the more general category theoretic definition of a functor in the sense that a Haskell functor is a category theoretic functor from the category of Haskell types, which is sometimes called Hask, to itself.[7] The category of Haskell types has as objects all types present in Haskell and as morphisms between types `a` and `b` the functions of type `a -> b`. Strictly speaking, the fact that a Haskell program may be undefined or may not terminate poses some problems for Hask to be a category, but these difficulties are usually ignored [Bau16].

The definition of a functor that we will present in this section resembles more the definition of a Haskell functor than that of a category theoretic functor. It will also be a type operator equipped with some action on functions. However, via the Curry-Howard correspondence the dependent types in our system will enable us to enforce the functor laws in the definition of a functor, something which is not possible in Haskell.

More concretely, the first thing we need in order to specify a functor F is a type operator obj_F. Since in dependent type theory types are values of a universe type, such a type operator is nothing more than a function going from some universe to some (possibly other) universe, so $\mathsf{obj}_F : \mathcal{U}_k \to \mathcal{U}_\ell$ for some universe levels k and ℓ. Second, we need a way to convert a function of type $A \to B$ into a function of type $\mathsf{obj}_F\, A \to \mathsf{obj}_F\, B$. Hence the functor F will also contain an operation

$$\mathsf{hom}_F : \forall(X, Y : \mathcal{U}_k).(X \to Y) \to \mathsf{obj}_F\, X \to \mathsf{obj}_F\, Y,$$

which is parametric in its first two arguments X and Y, expressing our intuition that the procedure to convert a function $f : A \to B$ into $\mathsf{hom}_F\, f : \mathsf{obj}_F\, A \to \mathsf{obj}_F\, B$ should not inspect the types A and B.

Finally, we need to express that obj_F and hom_F satisfy the functor laws, stating that hom_F maps the identity function $\mathsf{id}_X := \lambda(x : X)\,.\,x$ for any type X to the identity function $\mathsf{id}_{\mathsf{obj}_F X}$ for the type $\mathsf{obj}_F\, X$ and that hom_F respects composition of functions, so $\mathsf{hom}_F(g \circ f)$ equals $\mathsf{hom}_F\, g \circ \mathsf{hom}_F\, f$ for any two composable functions f and g. Using the Curry-Howard correspondence, we can translate these functor laws to the types

$$\forall(X : \mathcal{U}_k).\Pi(x : \mathsf{obj}_F\, X).\mathsf{hom}_F\, \mathsf{id}_X\, x \equiv_{\mathsf{obj}_F X} x \tag{2.1}$$

and

$$\begin{gathered}\forall(X, Y, Z : \mathcal{U}_k).\Pi(f : X \to Y).\Pi(g : Y \to Z).\Pi(x : \mathsf{obj}_F\, X). \\ \mathsf{hom}_F(g \circ f)\, x \equiv_{\mathsf{obj}_F Z} (\mathsf{hom}_F\, g \circ \mathsf{hom}_F\, f)\, x.\end{gathered} \tag{2.2}$$

Note that in for instance (2.1) we should strictly speaking write $\mathsf{hom}_F\, X^\sharp\, X^\sharp\, \mathsf{id}_X$ instead of $\mathsf{hom}_F\, \mathsf{id}_X$, but we leave these two arguments $X^\sharp$ implicit for the sake of readability (they can be deduced from the type of id_X). Also note that both types in (2.1) and (2.2) are dependent products that are parametric in their first argument(s). This is possible because hom_F is parametric in its first two arguments and it expresses our intuition that in some sense the proofs that F satisfies the functor laws should be the same for all types (we should not inspect the type when constructing these proofs). This choice for parametric types is also practically justified by implementing some well-known functors

[7]Such a functor from a category $\mathcal{C}$ to itself is called an endofunctor.

such as the list functor, the covariant hom functor and the maybe functor, showing that we are not too restrictive when requiring parametric proofs of the functor laws.[8]

Taking into account the discussion in the last paragraphs, a first attempt to formally define a functor F could be as a 4-tuple consisting of a type operator obj_F, an action on functions hom_F and a value of each of the two types in (2.1) and (2.2) proving the two functor laws. However, we will see in section 2.3.2 that in the definition we only need to require that F preserves identity functions and using parametricity we will then construct a proof that F respects composition.

In summary, we define for all universes $\mathcal{U}_k$ and $\mathcal{U}_\ell$ the type of functors

$$\begin{aligned}\mathsf{Functor}_{k,\ell} := {} & \Sigma(\mathsf{obj} : \mathcal{U}_k \to \mathcal{U}_\ell).\\ & \Sigma^{¶}(\mathsf{hom} : \forall(X, Y : \mathcal{U}_k).(X \to Y) \to \mathsf{obj}\,X \to \mathsf{obj}\,Y).\\ & \Sigma^{¶}(\mathsf{funct\text{-}id} : \forall(X : \mathcal{U}_k).\Pi(x : \mathsf{obj}\,X).\mathsf{hom}\,\mathsf{id}_X\,x \equiv_{\mathsf{obj}\,X} x).\top.\end{aligned}$$

In other words, a functor is a 4-tuple consisting of a type operator, an action on functions, a proof that this action preserves identity functions and an element of the unit type $\top$. This last element is a trivial piece of information (there is only one value of type $\top$, namely tt) that is included because we want a functor to depend pointwise on its action on functions and on its proof of the functor law involving identity functions.[9] This can be done using pointwise sigma types, but only for the first component of a pair and hence we need to add the trivial element of type $\top$ to the definition of a functor.

From now on, if we have a functor $F : \mathsf{Functor}_{k,\ell}$ we will by abuse of notation write F not only for the entire functor, but also for its type former and for its action on functions. If we wanted to be extremely precise, we should write obj_F for the type operator of F and hom_F for its action on functions but this would severely reduce readability in formulae. We will denote by $\mathsf{funct\text{-}id}\,F$ the proof that F preserves identity functions.

Example 2.7. If our type system has for any type A the type $\mathrm{List}\,A$ of lists with elements of type A, then we can consider the list functor mapping a type A to $\mathrm{List}\,A$ and a function $f : A \to B$ to the function of type $\mathrm{List}\,A \to \mathrm{List}\,B$ that applies f to every element in a list of type $\mathrm{List}\,A$.

As another example, we can consider for any type X the covariant hom functor that maps a type Y to the function type $X \to Y$. A function $f : Y \to Z$ is then converted into the function of type $(X \to Y) \to (X \to Z)$ that maps $g : X \to Y$ to $f \circ g$.

Remark 2.8. Our definition of a functor corresponds to what is called in category theory a covariant functor. It is also possible to define and study in ParamDTT so-called contravariant functors, but we will not cover those in this thesis.

2.3.2 Functors and Composition

In this section we will prove two results about functors and the composition of functions. We start with a justification that the functor law involving composition is not included in the definition of a functor.

[8] See the module Examples in `https://github.com/JorisCeulemans/effect-param-agda/blob/master/Functors.agda`

[9] The reason why we want this dependency to be pointwise, is that in this case a parametric function taking a functor as argument can still use the action and the proof in its implementation.

Proposition 2.9. *Suppose that* Γ *is a context containing the assumptions*

- $F^\sharp : \mathsf{Functor}_{k,\ell}$
- $A^\sharp, B^\sharp, C^\sharp : \mathcal{U}_k$
- $f : A \to B$
- $g^\P : B \to C$.

Then we can construct in this context Γ *a term*

$$\mathsf{funct\text{-}comp} : \Pi(a : F\,A).F\,g\,\big(F\,f\,a\big) \equiv_{FC} F\,(g \circ f)\,a.$$

Proof. We will use parametricity of obj_F. The general idea is to construct a (heterogeneous) path between f and $g \circ f$ and subsequently apply the functor F to everything in order to get a (heterogeneous) path between $F\,f$ and $F\,(g \circ f)$. Then we will use the function $\mathsf{pull}\,g$ to construct a homogeneous path between $(F\,g) \circ (F\,f)$ and $(F\,\mathsf{id}_C) \circ F(g \circ f)$. Finally we will use the fact that F preserves identity functions which is part of the definition of $\mathsf{Functor}_{k,\ell}$.

More concretely, using the function g we first construct a bridge

$$/g\backslash : \mathbb{I} \to \mathcal{U}_k$$

from B to C. Here we need that g is pointwise in the context Γ because we want to apply $/\cdot\backslash$ to it. Then we apply the function $\mathsf{push}\,g$ to obtain for every $b : B$ a path

$$\mathsf{g\text{-}path}\,b \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{push}\,g\,i^\sharp\,b \quad : \quad \forall(i : \mathbb{I})./g\backslash\,i$$

over $/g\backslash$ between b and $g\,b$. This can subsequently be used to construct a path

$$\mathsf{func\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\lambda(a : A)\,.\,\mathsf{g\text{-}path}\,(f\,a)\,i^\sharp \quad : \quad \forall(i : \mathbb{I}).A \to /g\backslash\,i$$

between f and $g \circ f$. This situation is illustrated in figure 2.3, where the dashed line represents the bridge $/g\backslash$. One could think of this diagram as if $/g\backslash\,i$ can move up and down along the dashed line and the middle arrow (with label $\mathsf{func\text{-}path}\,i^\sharp$) follows accordingly. When i is 0, then we are at the top and $/g\backslash\,i$ is definitionally equal to B and $\mathsf{func\text{-}path}\,i^\sharp$ is definitionally equal to f. Likewise, if i is 1 then we are at the bottom of the diagram (definitionally).

We now have for any $i : \mathbb{I}$ a function $\mathsf{func\text{-}path}\,i^\sharp$ from A to $/g\backslash\,i$. We can then apply the functor F to this to get a function $F\,(\mathsf{func\text{-}path}\,i^\sharp)$ from $F\,A$ to $F\,(/g\backslash\,i)$. This gives us a path

$$\mathsf{F\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{func\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).F\,A \to F\,(/g\backslash\,i)$$

between $F\,f$ and $F\,(g \circ f)$. This situation is illustrated in the left part of figure 2.4. Additionally, we can apply for any $i : \mathbb{I}$ the functor F to the function $\mathsf{pull}\,g\,i^\sharp$ to get a function from $F\,(/g\backslash\,i)$ to $F\,C$. This then yields a path

$$\mathsf{Fpullg\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{pull}\,g\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).F\,(/g\backslash\,i) \to F\,C$$

$$\begin{tikzcd}
 & B \arrow[d, dash, dashed] \\
A \arrow[ur, "f"] \arrow[r, "\mathsf{func\text{-}path}\, i^\sharp"] \arrow[dr, "g\circ f"'] & /g\backslash\, i \arrow[d, dash, dashed] \\
 & C
\end{tikzcd}$$

Figure 2.3: Construction of a path between f and $g \circ f$.

$$\begin{tikzcd}
 & F\,B \arrow[d, dash, dashed] \arrow[dr, "F\,g"] & \\
F\,A \arrow[ur, "F\,f"] \arrow[r, "\mathsf{F\text{-}path}\, i^\sharp"] \arrow[dr, "F\,(g\circ f)"'] & F(/g\backslash\, i) \arrow[d, dash, dashed] \arrow[r, "\mathsf{Fpullg\text{-}path}\, i^\sharp"] & F\,C \\
 & F\,C \arrow[ur, "F\,\mathsf{id}_C"'] &
\end{tikzcd}$$

Figure 2.4: Construction of a path between $F\,g \circ F\,f$ and $F\,\mathsf{id}_C \circ F\,g \circ f$.

between $F\,g$ and $F\,\mathsf{id}_C$. This is illustrated on the right part of figure 2.4. As can be seen on that diagram, for any $i : \mathbb{I}$ we can compose the functions $\mathsf{F\text{-}path}\, i^\sharp$ and $\mathsf{Fpullg\text{-}path}\, i^\sharp$ to get a function of type $F\,A \to F\,C$. Applying this to an arbitrary $a : F\,A$ gives the path

$$\mathsf{final\text{-}path}\, a \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{Fpullg\text{-}path}\, i^\sharp\big(\mathsf{F\text{-}path}\, i^\sharp\, a\big) \quad : \quad \forall(i : \mathbb{I}).F\,C$$

between $F\,g\,(F\,f\,a)$ and $F\,\mathsf{id}_C\,(F\,(g \circ f)\,a)$. As we can see, this is a homogeneous path so that the path degeneracy axiom provides a term

$$\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}\, a) : F\,g\,(F\,f\,a) \equiv_{FC} F\,\mathsf{id}_C\,(F\,(g \circ f)\,a).$$

Using this proof term together with the term

$$\mathsf{funct\text{-}id}_F\,(F\,(g \circ f)\,a) : F\,\mathsf{id}_C(F\,(g \circ f)\,a) \equiv_{FC} F\,(g \circ f)\,a$$

obtained from the definition of a functor gives us by transitivity

$$\mathsf{funct\text{-}comp}\, a : F\,g\,\big(F\,f\,a\big) \equiv_{FC} F\,(g \circ f)\,a. \qquad \square$$

Remark 2.10. We formulate some remarks regarding Proposition 2.9.

1. Using function extensionality (see example 1.33), it is possible to derive from the term $\mathsf{funct\text{-}comp}$ a term of type $(F\,g) \circ (F\,f) \equiv_{FA \to FC} F\,(g \circ f)$. However, it is also possible to derive such a term directly by modifying the argument above a little bit and by applying function extensionality to the term $\mathsf{funct\text{-}id}_F$. This is also included in the mechanization of the previous proof in Agda and we refer to that code for more details.

2. Proposition 2.9 also holds in the case where F is continuous or pointwise in the context Γ. The proof is in those cases exactly the same. This can be verified by adjusting the modality of F in the mechanization of the proof in Agda.

3. The fact that good behaviour of a functor F with respect to composition follows from parametricity and the functor law involving identity functions, was already known.[10] However, to our knowledge it has not yet been proved in a dependent type system with support for parametricity.

4. A very similar argument can be applied to obtain a term of the same type as in Proposition 2.9 but with f appearing pointwise and g continuous in the context Γ. In this case the function f is used to create a bridge $/f\backslash$ from A to B and the rest of the proof needs to be adjusted accordingly.

By Proposition 2.9 we can omit the functor law involving composition in the definition of a functor. We could of course also wonder whether the functor law involving identity functions can be omitted and proved directly from the types of obj_F and hom_F by parametricity. This is however not the case, as can be seen by considering for instance the type operator

$$\Theta := \lambda(X : \mathcal{U}_0) \,.\, \mathbb{N} : \mathcal{U}_0 \to \mathcal{U}_0$$

mapping any type in $\mathcal{U}_0$ to the type of natural numbers. For the morphism part, we will map any function $f : X \to Y$ to the constant function of type $\mathbb{N} \to \mathbb{N}$ mapping every natural number to 0, so we have

$$\begin{aligned}\theta \quad := \quad & \lambda(X^\sharp, Y^\sharp : \mathcal{U}_0) \,.\, \lambda(f : X \to Y) \,.\, \lambda(n : \mathbb{N}) \,.\, 0 : \\ & \qquad\qquad \forall(X, Y : \mathcal{U}_0).(X \to Y) \to \mathbb{N} \to \mathbb{N}.\end{aligned}$$

Clearly the constant function $\lambda(n : \mathbb{N}) \,.\, 0$ is not the identity function for $\mathbb{N}$ and hence the functor law involving identity functions does not hold automatically by parametricity. Moreover, it is easy to check that $\theta\,(g \circ f)$ and $\theta\, g \circ \theta f$ map every natural number to 0 so that θ respects composition of functions. Hence it is not possible to define a functor alternatively by only requiring the functor law involving composition (and then proving identity preservation).

The following result is in some sense a generalization of Proposition 2.9.

Proposition 2.11. *Suppose that Γ is a context containing the following assumptions*

- $F^\sharp : \mathsf{Functor}_{k,\ell}$
- $A^\sharp, B^\sharp, C^\sharp, D^\sharp : \mathcal{U}_k$
- $f_1 : A \to B$
- $f_2 : C \to D$
- $g^\P : A \to C$
- $h^\P : B \to D$

[10] See for instance `https://github.com/quchen/articles/blob/master/second_functor_law.md`.

and assume that we can derive the judgement

$$\Gamma, a : A \vdash h\,(f_1\,a) = f_2\,(g\,a) : D.$$

Then we can construct in this context Γ *a term*

$$\mathsf{square\text{-}commute} : \Pi(a : F\,A).F\,h\,(F\,f_1\,a) \equiv_{FD} F\,f_2\,(F\,g\,a).$$

In other words, the previous proposition tells us that if the left diagram below commutes definitionally, then we can construct a proof that the right diagram commutes.

$$\begin{tikzcd} A \arrow[r, "f_1"] \arrow[d, "g"] & B \arrow[d, "h"] \\ C \arrow[r, "f_2"] & D \end{tikzcd} \qquad \begin{tikzcd} F\,A \arrow[r, "F\,f_1"] \arrow[d, "F\,g"] & F\,B \arrow[d, "F\,h"] \\ F\,C \arrow[r, "F\,f_2"] & F\,D \end{tikzcd}$$

Proof. Again we use parametricity of obj_F. The idea is now to construct a path from f_1 to f_2 and again apply the functor F to get a path from $F\,f_1$ to $F\,f_2$. Pre- and postcomposing this path with the right paths will then yield the result by applying the path degeneracy axiom. Throughout this proof we assume that the graph relation former $/\cdot\backslash$ is implemented using the Weld type former.

More concretely, we construct bridges $/g\backslash : \mathbb{I} \to \mathcal{U}_k$ between A and C and $/h\backslash : \mathbb{I} \to \mathcal{U}_k$ between B and D. This explains why g and h appear pointwise in the context Γ. Using the function $\mathsf{push}\,h$ we get for any $b : B$ a path

$$\mathsf{h\text{-}path}\,b \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{push}\,h\,i^\sharp\,b \quad : \quad \forall(i : \mathbb{I})./h\backslash\,i$$

between b and $h\,b$. Subsequently, we want to define a path between f_1 and f_2 over the bridge $\lambda(i : \mathbb{I})\,.\,/g\backslash\,i \to /h\backslash\,i$. For this purpose, we must specify for any $i : \mathbb{I}$ a function of type $/g\backslash\,i \to /h\backslash\,i$ and we will do this using the induction principle for Weld types, recalling that $/g\backslash$ was implemented using the Weld type former as

$$/g\backslash\,i = \mathsf{Weld}\{A \to (i \doteq 0\,?\,A, \mathsf{id}_A \mid i \doteq 1\,?\,C, g)\}.$$

Hence, in order to use the induction principle $\mathsf{ind}_{\mathsf{Weld}}$ to construct an element of type $/h\backslash\,i$, we need to specify an element of type $/h\backslash\,i$ given $a : A$ and to specify an element of type $/h\backslash\,i$ assuming that the predicate $(i \doteq 0) \vee (i \doteq 1)$ holds. The former can be given using the path $\mathsf{h\text{-}path}$ we just constructed, more concretely if $a : A$ then we have that $\mathsf{h\text{-}path}\,(f_1\,a)\,i^\sharp : /h\backslash\,i$. The latter element is given by $\big(i \doteq 0\,?\,a.(f_1\,a) \mid i \doteq 1\,?\,c.(f_2\,c)\big)$. Finally, we also need that if $(i \doteq 0)$ then $\mathsf{h\text{-}path}\,(f_1\,a)\,0^\sharp$ is definitionally equal to $f_1\,a$, which is the case because $\mathsf{push}\,h\,0^\sharp = \mathsf{id}_B$, and that if $(i \doteq 1)$ then $\mathsf{h\text{-}path}\,(f_1\,a)\,1^\sharp$ is definitionally equal to $f_2\,(g\,a)$ which is the case because the left diagram above commutes definitionally. We can then use the induction principle for Weld types to get a path

$$\begin{aligned} \mathsf{func\text{-}path} := {} & \lambda(i^\sharp : \mathbb{I})\,.\,\lambda(x : /g\backslash\,i)\,. \\ & \quad \mathsf{ind}_{\mathsf{Weld}}(/h\backslash\,i, (i \doteq 0\,?\,a.(f_1\,a) \mid i \doteq 1\,?\,c.(f_2\,c)), a.(\mathsf{h\text{-}path}\,(f_1\,a)\,i^\sharp), x) \end{aligned}$$

between f_1 and f_2. This situation is illustrated in figure 2.5.

$$\begin{tikzcd}
A \arrow[r, "f_1"] \arrow[d, dashed, no head] & B \arrow[d, dashed, no head] \\
/g\backslash\, i \arrow[r, "\mathsf{func\text{-}path}\, i"] \arrow[d, dashed, no head] & /h\backslash\, i \arrow[d, dashed, no head] \\
C \arrow[r, "f_2"] & D
\end{tikzcd}$$

Figure 2.5: Construction of a path between f_1 and f_2.

Now we apply for any $i : \mathbb{I}$ the functor F to the function $\mathsf{func\text{-}path}\, i^\sharp$ to get a path

$$\mathsf{F\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\, F\,(\mathsf{func\text{-}path}\, i^\sharp) \quad : \quad \forall (i : \mathbb{I}).F\,(/g\backslash\, i) \to F\,(/h\backslash\, i)$$

between $F\,f_1$ and $F\,f_2$. This is illustrated in the middle part of figure 2.6. Subsequently, we use the functions $\mathsf{push}\, g$ and $\mathsf{pull}\, h$ to construct paths

$$\mathsf{Fpushg\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\, F\,(\mathsf{push}\, g\, i^\sharp) \quad : \quad \forall (i : \mathbb{I}).F\,A \to F(/g\backslash\, i)$$

between $F\,\mathsf{id}_A$ and $F\,g$, and

$$\mathsf{Fpullh\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\, F\,(\mathsf{pull}\, h\, i^\sharp) \quad : \quad \forall (i : \mathbb{I}).F\,(/h\backslash\, i) \to F\,D$$

between $F\,h$ and $F\,\mathsf{id}_D$.

We can then see on the diagram of figure 2.6 that for any $i : \mathbb{I}$ we can compose the functions $\mathsf{Fpushg\text{-}path}\, i^\sharp$, $\mathsf{F\text{-}path}\, i^\sharp$ and $\mathsf{Fpullh\text{-}path}\, i^\sharp$ to get for any $a : F\,A$ a path

$$\mathsf{final\text{-}path}\, a \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\, \mathsf{Fpullh\text{-}path}\, i^\sharp\, (\mathsf{F\text{-}path}\, i^\sharp\, (\mathsf{Fpushg\text{-}path}\, i^\sharp\, a) \quad : \quad \forall (i : \mathbb{I}).F\,D$$

between $F\,h\,(F\,f_1\,(F\,\mathsf{id}_A\,a))$ and $F\,\mathsf{id}_D\,(F\,f_2\,(F\,g\,a))$. Note that this path is homogeneous and hence the path degeneracy axiom provides us with a proof

$$\begin{aligned}&\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}\, a) : \\ &\qquad F\,h\,(F\,f_1\,(F\,\mathsf{id}_A\,a)) \equiv_{F\,D} F\,\mathsf{id}_D\,(F\,f_2\,(F\,g\,a)).\end{aligned}$$

Applying twice the proof $\mathsf{funct\text{-}id}_F$ that F preserves identity functions and using transitivity we finally get a term

$$\mathsf{square\text{-}commute}\, a : F\,h\,(F\,f_1\,a) \equiv_{FD} F\,f_2\,(F\,g\,a). \qquad \square$$

Remark 2.12. The remarks 1 to 3 from remark 2.10 remain valid here (of course appropriately modified to match the current proposition). We now also note that Proposition 2.11 a generalization of Proposition 2.9. It is indeed easy to see that if we assume in Proposition 2.11 that $A = C$ and that the function g is the identity function id_A, then the result of Proposition 2.9 follows.

$$\begin{tikzcd}
 & F\,A \arrow[r, "F\,f_1"] \arrow[d, dash, dashed] & F\,B \arrow[rd, "F\,h"] \arrow[d, dash, dashed] & \\
F\,A \arrow[ru, "F\,\mathsf{id}_A"] \arrow[r, "\mathsf{Fpushg\text{-}path}\,i^\sharp"] \arrow[rd, "F\,g"'] & F(/g\backslash\,i) \arrow[r, "\mathsf{F\text{-}path}\,i^\sharp"] \arrow[d, dash, dashed] & F(/h\backslash\,i) \arrow[r, "\mathsf{Fpullh\text{-}path}\,i^\sharp"] \arrow[d, dash, dashed] & F\,D \\
 & F\,C \arrow[r, "F\,f_2"] & F\,D \arrow[ru, "F\,\mathsf{id}_D"'] &
\end{tikzcd}$$

Figure 2.6: Construction of a path between $F\,h \circ F\,f_1 \circ F\,\mathsf{id}_A$ and $F\,\mathsf{id}_D \circ F\,f_2 \circ F\,g$.

2.3.3 Natural Transformations

In this section we show that if we have two functors $F, G : \mathsf{Functor}_{k,\ell}$ and a parametric polymorphic function

$$\rho : \forall(X : \mathcal{U}_k).F\,X \to G\,X,$$

then ρ is a *natural transformation* from F to G. This means that for any two types $A, B : \mathcal{U}_k$ and any function $f : A \to B$ the following diagram commutes.

$$\begin{tikzcd}
F\,A \arrow[r, "\rho\,A^\sharp"] \arrow[d, "F\,f"] & G\,A \arrow[d, "G\,f"] \\
F\,B \arrow[r, "\rho\,B^\sharp"] & G\,B
\end{tikzcd}$$

More concretely, we will prove the following Proposition.

Proposition 2.13. *Let Γ be a context containing the following assumptions*

- $F^\sharp, G^\sharp : \mathsf{Functor}_{k,\ell}$
- $\rho : \forall(X : \mathcal{U}_k).F\,X \to G\,X$
- $A^\sharp, B^\sharp : \mathcal{U}_k$
- $f^\P : A \to B.$

Then we can construct in this context Γ a term

$$\mathsf{naturality} : \Pi(a : F\,A).G\,f\,(\rho\,A^\sharp\,a) \equiv_{GB} \rho\,B^\sharp\,(F\,f\,a).$$

Proof. We will use the parametricity of ρ. The main idea is to construct a bridge from A to B and apply ρ to this bridge. As ρ is parametric in its first argument, this will give us a path.

We will need paths obtained by applying F to the function $\mathsf{push}\,f$ and G to the function $\mathsf{pull}\,f$. More concretely, we have the path

$$\mathsf{Fpushf\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{push}\,f\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).F\,A \to F(/f\backslash\,i)$$

$$\begin{tikzcd}
 & F\,A \arrow[r, "\rho\,A^\sharp"] \arrow[d, dash, dashed] & G\,A \arrow[dr, "G\,f"] \arrow[d, dash, dashed] & \\
F\,A \arrow[ur, "F\,\mathsf{id}_A"] \arrow[r, "F(\mathsf{push}\,f\,i^\sharp)"] \arrow[dr, "F\,f"'] & F(/f\backslash\,i) \arrow[r, "\rho\,(/f\backslash\,i)^\sharp"] \arrow[d, dash, dashed] & G(/f\backslash\,i) \arrow[r, "G(\mathsf{pull}\,f\,i^\sharp)"] \arrow[d, dash, dashed] & G\,B \\
 & F\,B \arrow[r, "\rho\,B^\sharp"] & G\,B \arrow[ur, "G\,\mathsf{id}_B"'] &
\end{tikzcd}$$

Figure 2.7: Constructing a path from $G\,f \circ \rho\,A^\sharp \circ F\,\mathsf{id}_A$ to $G\,\mathsf{id}_B \circ \rho\,B^\sharp \circ F\,f$.

from $F\,\mathsf{id}_A$ to $F\,f$ which is illustrated on the left of figure 2.7. Additionally, we consider the path

$$\mathsf{Gpullf\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,G\,(\mathsf{pull}\,f\,i^\sharp) \quad : \quad \forall(i:\mathbb{I}).G(/f\backslash\,i) \to G\,B$$

from $G\,f$ to $G\,\mathsf{id}_B$ which can be found on the right of figure 2.7. We can now compose the different paths appearing on that diagram to get for any $a : F\,A$ the path

$$\mathsf{final\text{-}path}\,a \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{Gpullf\text{-}path}\,i^\sharp\,(\rho\,(/f\backslash\,i)^\sharp\,(\mathsf{Fpushf\text{-}path}\,i^\sharp\,a)) : \\ \forall(i:\mathbb{I}).G\,B$$

from $G\,f\,(\rho\,A^\sharp\,(F\,\mathsf{id}_A\,a))$ to $G\,\mathsf{id}_B\,(\rho\,B^\sharp\,(F\,f\,a))$. Moreover, this path is homogeneous so that the path degeneracy axiom gives us a proof

$$\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}\,a) : G\,f\,(\rho\,A^\sharp\,(F\,\mathsf{id}_A\,a)) \equiv_{G\,B} G\,\mathsf{id}_B\,(\rho\,B^\sharp\,(F\,f\,a)).$$

Finally, using the fact that functors preserve identity functions and applying transitivity, we can construct

$$\mathsf{naturality}\,a : G\,f\,(\rho\,A^\sharp\,a) \equiv_{G\,B} \rho\,B^\sharp\,(F\,f\,a). \qquad \square$$

Remark 2.14. When appropriately adjusted, remarks 1 and 2 of remark 2.10 hold as well for this proposition. We also note that proposition 2.13 is a known result and it has already been proved in the type system ParamDTT by Nuyts, Vezzosi and Devriese [NVD17].[11] We include this result and proof in this thesis because we will need them in chapter 3.

[11] See also `https://github.com/Saizan/parametric-demo/blob/master/Naturality.agda`.

Chapter 3

Effects and Monads

Most computer programs interact with the outside world (i.e. the world outside the program, which could include other programs running on the same device) for instance by writing output to a screen, accepting input from a user, storing the value of some global variable or sending information over a network. These different kinds of interaction are called *side effects* or simply *effects*. In a purely functional programming language such as Haskell or Agda, an ordinary function cannot cause any side effect and hence it can only compute some output value for a given input.[1] Moreover, applying such a function at different instants of time to the same input will result in equal output values, as a pure function does not have access to some kind of state such as a global variable storing how many times the function has been applied. This is clearly a benefit for reasoning about programs.

However, in a programming language that does not allow any kind of side effects, one could not write many practically useful programs. This is why for instance in Haskell the concept of a monad was introduced. The idea is to define for a particular kind of side effects a new type `m a` for any type `a`, containing computations that yield a result of type `a` but that may also cause a side effect of this particular kind. Adding some requirements to the type former `m`, such as the possibility to sequentially compose different effectful computations, gives rise to the definition of a monad, which we will see in this chapter.

The concept of a monad originated in category theory and was first introduced in computer science by Eugenio Moggi for studying semantics of programming languages [Mog91]. Subsequently, Philip Wadler described their use for the treatment of side effects in purely functional programming languages [Wad92; Wad95].

The aim of this chapter is twofold. We will provide an introduction to the theory of monads, based on [Wad92; Lip11, chapter 12; Vou12; Mil18, chapters 20–22]. At the same time, we present this theory of monads in the type system ParamDTT and investigate the consequences of parametricity in this context (formalizations for this in Agda parametric are also available[2]).

[1]Strictly speaking, Haskell is almost purely functional because one can define ordinary functions of type `a -> b` that throw an exception using the value `undefined`.

[2]See `https://github.com/JorisCeulemans/effect-param-agda/tree/master/Monads`.

3.1 Introduction

Considered as a programming language, the type systems MLTT and ParamDTT are purely functional. Indeed, a function of type $A \to B$ can only compute an output value of type B given some argument of type A. Suppose now that we want to define a function pred of type $\mathbb{N} \to \mathbb{N}$ that maps a natural number to its predecessor. Clearly, we have to decide what to do with the number 0 as it has no predecessor contained in the type $\mathbb{N}$. We could choose to map 0 to itself but that would not be a proper definition of a predecessor function. In fact, we would like to throw some kind of error or exception when this function is applied to 0. That is however not possible in a purely functional language because the only thing a function of type $\mathbb{N} \to \mathbb{N}$ can do is produce output of type $\mathbb{N}$ given an argument of the same type $\mathbb{N}$.

A possible solution consists of the introduction of a new type $\mathrm{Maybe}\, A$ for any type A, containing besides all values of type A an element used for signaling that some error has occurred or some unexpected case has been reached. More concretely, we will extend our type system with the following inference rules.

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell}{\Gamma \vdash \mathrm{Maybe}\, A : \mathcal{U}_\ell}\ \textsc{Mb-form}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{just}\, a : \mathrm{Maybe}\, A}\ \textsc{Mb-intro-just} \qquad \frac{\Gamma \vdash \mathrm{Maybe}\, A \text{ type}}{\Gamma \vdash \mathsf{nothing} : \mathrm{Maybe}\, A}\ \textsc{Mb-intro-nothing}$$

$$\frac{\begin{array}{ll} \Gamma, x : \mathrm{Maybe}\, A \vdash B \text{ type} & \Gamma, y : A \vdash b_j : B[\mathsf{just}\, y/x] \\ \Gamma \vdash m : \mathrm{Maybe}\, A & \Gamma \vdash b_n : B[\mathsf{nothing}/x] \end{array}}{\Gamma \vdash \mathsf{ind}_{\mathrm{Maybe}}(x.B, y.b_j, b_n, m) : B[m/x]}\ \textsc{Mb-elim}$$

$$\frac{\begin{array}{ll} \Gamma, x : \mathrm{Maybe}\, A \vdash B \text{ type} & \Gamma, y : A \vdash b_j : B[\mathsf{just}\, y/x] \\ \Gamma \vdash a : A & \Gamma \vdash b_n : B[\mathsf{nothing}/x] \end{array}}{\Gamma \vdash \mathsf{ind}_{\mathrm{Maybe}}(x.B, y.b_j, b_n, \mathsf{just}\, a) = b_j[a/y] : B[\mathsf{just}\, a/x]}\ \textsc{Mb-comp-just}$$

$$\frac{\begin{array}{c} \Gamma, x : \mathrm{Maybe}\, A \vdash B \text{ type} \\ \Gamma, y : A \vdash b_j : B[\mathsf{just}\, y/x] \qquad \Gamma \vdash b_n : B[\mathsf{nothing}/x] \end{array}}{\Gamma \vdash \mathsf{ind}_{\mathrm{Maybe}}(x.B, y.b_j, b_n, \mathsf{nothing}) = b_n : B[\mathsf{nothing}/x]}\ \textsc{Mb-comp-nothing}$$

The introduction rules tell us that the type $\mathrm{Maybe}\, A$ has a specific value $\mathsf{nothing}$ (signaling an error) and values of the form $\mathsf{just}\, a$ for any value a of type A. Just like for most types in section 1.4.2, we have an induction principle stating that these values are essentially the only inhabitants of the type $\mathrm{Maybe}\, A$.

We can now consider functions from a type A to a type B that may cause an error as pure functions of type $A \to \mathrm{Maybe}\, B$ that return the value $\mathsf{nothing}$ in case an error has occurred. In other words, we can interpret $\mathrm{Maybe}\, B$ as the type of computations that result in some value of type B, but that may also produce an error. In this way the predecessor function pred may be defined as

$$\mathsf{pred} := \lambda(n : \mathbb{N})\, .\, \mathsf{ind}_{\mathbb{N}}(m.\, \mathrm{Maybe}\, \mathbb{N}, \mathsf{nothing}, k.a.\, \mathsf{just}\, k, n),$$

and it has as type $\mathbb{N} \to \mathrm{Maybe}\,\mathbb{N}$. We now have indeed that $\mathsf{pred}\,0$ is definitionally equal to $\mathsf{nothing}$ by the computation rules for the type $\mathbb{N}$.

In our interpretation of $\mathrm{Maybe}\,A$ as the type of computations that yield a result of type A but may also produce an error, we have for any value $a : A$ a pure computation $\mathsf{just}\,a$ that does not cause any effect but only yields the result a. This will be true in general, for any monad there will be a function mapping a value of a certain type X to pure a computation only yielding that value (and not causing any side effect).

We now know that functions from A to B that may produce an error are interpreted as pure functions of type $A \to \mathrm{Maybe}\,B$. However, at first sight this construction poses a problem for function composition. Indeed, suppose that we have functions f from A to B and g from B to C that may cause an error. It is clear that we would like to be able to compose f and g but in fact, taking into account that f and g may produce an error, we have functions

$$f : A \to \mathrm{Maybe}\,B \quad \text{and} \quad g : B \to \mathrm{Maybe}\,C,$$

and these cannot be composed because their types do not match. Intuitively however, it is obvious how the "composition" of f and g should behave: first apply f and then apply g and if an error occurs in the process, then propagate it. We can make this more precise using the induction principle for the Maybe type former, giving us the function

$$\lambda(x : A)\,.\,\mathsf{ind}_{\mathrm{Maybe}}\,(z.\,\mathrm{Maybe}\,C, y.(g\,y), \mathsf{nothing}, f\,x) \tag{3.1}$$

of type $A \to \mathrm{Maybe}\,C$ (which is indeed the type of functions from A to C that may cause an error). If we analyze this function more carefully, we see that we first applied f to the argument x, giving us some value of type $\mathrm{Maybe}\,B$, and then used the induction principle for Maybe to construct a value of type $\mathrm{Maybe}\,C$ out of this value of type $\mathrm{Maybe}\,B$ and the function g of type $B \to \mathrm{Maybe}\,C$. This procedure to convert a value of type $\mathrm{Maybe}\,X$ into a value of type $\mathrm{Maybe}\,Y$ using a function of type $X \to \mathrm{Maybe}\,Y$ is the key ingredient in the composition of functions that may cause an error and it is called the bind operation for Maybe. It will be denoted by the symbol $\gg\!=_{\mathrm{Maybe}}$ as an infix operation with a term of type $\mathrm{Maybe}\,X$ on the left and a function of type $X \to \mathrm{Maybe}\,Y$ on the right. More concretely, we have for $m : \mathrm{Maybe}\,X$ and $k : X \to \mathrm{Maybe}\,Y$ that

$$m \gg\!=_{\mathrm{Maybe}} k \quad := \quad \mathsf{ind}_{\mathrm{Maybe}}(z.\,\mathrm{Maybe}\,Y, x.(k\,x), \mathsf{nothing}, m).$$

Hence we can abbreviate the "composition" of f and g as given in (3.1) to

$$\lambda(x : A)\,.\,f\,x \gg\!=_{\mathrm{Maybe}} g.$$

Note that the bind operation $\gg\!=_{\mathrm{Maybe}}$ is in fact polymorphic[3] but we will omit the type arguments as they can be inferred from the types of the left and right arguments (just like we did with the hom operation for functors in section 2.3).

3.2 Definition of a Monad

In the previous section we saw that computations which may produce an error can be described using the type former Maybe. Of course there are many other kinds side effects

[3]Actually this operation is even polymorphic in the universe level as it works for types of any universe. This is however not expressible in MLTT or ParamDTT and hence we should strictly speaking annotate the symbol $\gg\!=_{\mathrm{Maybe}}$ with a universe level. We omit this for the sake of readability.

that we can imagine a programmer would want to include, such as writing output to the screen, accessing and updating some kind of state variable, etc. For every kind of side effects, there will be a different type former that can be used to model them. Additionally, any useful treatment of side effects should provide a way to compose effectful functions and hence we will require that such a type former comes with a bind operation as in the previous section. Equipping a type operator with the structure needed to model effects gives rise to the notion of a monad. In this section we will define monads and some other related concepts and we present these definitions in the type system ParamDTT.

3.2.1 A first formulation

As already mentioned, a *monad* M always consists of a type operator, which we also denote by M. Just like in section 2.3 such a type operator is a function with certain universe types as its domain and codomain. For a given type X, we will then interpret $M\,X$ as the type of computations that yield a result of type X but that may also cause a certain side effect (and the kind of side effect depends on the type operator M). In this context, it is not surprising that we require from a type operator in a monad that its domain and codomain are equal. Indeed, intuitively it is plausible that the type of computations which yield a result of type X must live in the same universe as X. We postpone the more technical motivation for this requirement to section 3.2.2. As a conclusion we have for a monad M a type operator $M : \mathcal{U}_\ell \to \mathcal{U}_\ell$.

Second, we need a way to produce for any type $A : \mathcal{U}_\ell$ and for any value $a : A$ a computation of type $M\,A$ that causes no side effect and only yields the result a. This will be achieved by a function called return which has type

$$\mathsf{return} : \forall(X : \mathcal{U}_\ell).X \to M\,X.$$

Note that this return function is parametric in its type argument, expressing the intuition that the procedure to embed X into $M\,X$ should not depend on the type X. This requirement is practically justified as we will see in example 3.2 that many well-known monads can be implemented in this way. Moreover, taking into account that polymorphism in Haskell is parametric, this type is the best approximation in ParamDTT for the type of the return method in the Haskell type class `Monad`.

Third, in order to compose effectful functions we need a *bind operation* $\gg\!=$ such as the one for Maybe in the previous section. We will use $\gg\!=$ as an infix operation and this function will have type

$$\_ \gg\!= \_ : \forall(X, Y : \mathcal{U}_\ell).M\,X \to (X \to M\,Y) \to M\,Y.$$

Again we have the requirement that $\gg\!=$ is parametric in its two type arguments which is the best approximation for the bind method in the Haskell type class `Monad` and which is practically justified by implementing some well-known monads in this way. Just like for Maybe, the composition of two effectful functions $f : A \to M\,B$ and $g : B \to M\,C$ is given by

$$\lambda(x : A) \,.\, f\,x \gg\!= g.$$

The return and bind functions give us enough structure on the type operator M to model side effects, but in order for this to be a monad, some extra conditions, called the monad laws, must be satisfied. However, in chapter 4 we will see that those structures that do not necessarily satisfy the monad laws are interesting and useful as well. Such structures will be called *premonads*.[4]

Definition 3.1. For any universe $\mathcal{U}_\ell$ we define the type

$$\begin{aligned}\mathsf{Premonad}_\ell := {} & \Sigma(M : \mathcal{U}_\ell \to \mathcal{U}_\ell).\\ & \Sigma^{\P}(\mathsf{return} : \forall(X : \mathcal{U}_\ell).X \to M\,X).\\ & \Sigma^{\P}(\_ \gg\!= \_ : \forall(X, Y : \mathcal{U}_\ell).M\,X \to (X \to M\,Y) \to M\,Y).\top\end{aligned}$$

and a value $M : \mathsf{Premonad}_\ell$ is called a premonad.

Notice that a premonad is a 4-tuple consisting of a type operator, return and bind functions and a value of type $\top$. Just as in the definition of a functor in section 2.3.1 this last component is an irrelevant piece of information that is included only so that a premonad can depend pointwise on its return and bind functions. The reason why this pointwise dependence is useful, will become clear in section 4.1.

From now on, if $M : \mathsf{Premonad}_\ell$ is a premonad, we will write return_M for its return function and $\gg\!=_M$ for its bind operation. Although both functions take one or two type arguments, we will usually omit these as they can be inferred from the other arguments. When these type arguments are not omitted, they can be distinguished from other arguments because they are annotated with a parametric modality (so $\mathsf{return}_M\,A^\sharp$ is a function of type $A \to M\,A$ and $\mathsf{return}_M\,a$ is a value of type $M\,A$ for any $a : A$). Sometimes, when the premonad is clear from the context, we will just write return and $\gg\!=$.

Example 3.2. We will now consider some examples of premonads, which are all formalized in Agda parametric[5].

1. The simplest premonad possible is the identity premonad $\mathsf{id\text{-}pm}_\ell : \mathsf{Premonad}_\ell$ (for any universe level ℓ). Its type operator is the identity function $\mathsf{id}_{\mathcal{U}_\ell} : \mathcal{U}_\ell \to \mathcal{U}_\ell$ and the return function is the polymorphic identity function

   $$\lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,x : \forall(X : \mathcal{U}_\ell).X \to X.$$

   The bind function has type $\forall(X, Y : \mathcal{U}_\ell).X \to (X \to Y) \to Y$ and maps $x : X$ and $k : X \to Y$ to $k\,x : Y$. In this way, the composition of functions $f : A \to \mathsf{id\text{-}pm}_\ell\,B$ and $g : B \to \mathsf{id\text{-}pm}_\ell\,C$ using bind corresponds to the usual composition of f and g (notice that $\mathsf{id\text{-}pm}_\ell\,B = B$ so that this makes sense). The identity premonad models no side effects and will be important in many proofs involving effect parametricity in chapter 4. From now on we will not explicitly write the level ℓ in $\mathsf{id\text{-}pm}_\ell$ anymore unless this is important in the reasoning.

2. From the previous section, we know that

   $$\big(\lambda(X : \mathcal{U}_\ell)\,.\,\mathrm{Maybe}\,X,\quad \lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,\mathsf{just}\,x,\quad \gg\!=_{\mathrm{Maybe}},\quad \mathsf{tt}\big)$$

   is a premonad, called the Maybe premonad. Strictly speaking we should have annotated the second and third component of this tuple with a pointwise modality, but we omitted this for the sake of readability.

[4]This terminology is not standard in the literature. Actually there is no generally accepted name for these structures, which is why we came up with the term premonad.

[5]For more details, see `https://github.com/JorisCeulemans/effect-param-agda/blob/master/Monads/Examples.agda`

3. For a type $S : \mathcal{U}_k$ we can define a premonad $\textsf{state-pm}_S : \textsf{Premonad}_\ell$ for any $\ell \geq k$. Its type operator maps a type $X : \mathcal{U}_\ell$ to the type $S \to (X \times S)$, which also lives in $\mathcal{U}_\ell$ because of cumulativity of the universes. The return function is given by

$$\textsf{return}_{\textsf{state-pm}_S} := \lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,\lambda(s : S)\,.\,(x, s)$$

and its bind function by

$$f \gg\!=_{\textsf{state-pm}_S} k := \lambda(s : S)\,.\,k\,(\textsf{fst}\,(f\,s))\,(\textsf{snd}\,(f\,s)).$$

We will later see that $\textsf{state-pm}_S$ is actually a monad and this monad is used to model computations that can read and update a state of type S. For instance if $S = \mathbb{N}$ the state could be the number of times a certain function has been called.

4. Finally, we consider the example of the writer premonad for a pointed magma. A *magma* is an algebraic structure consisting of a type (its carrier) and a binary operation on that carrier. More concretely, for any universe type $\mathcal{U}_\ell$ we define the type of magmas with carrier in $\mathcal{U}_\ell$ as

$$\begin{aligned}\textsf{Magma}_\ell := {} & \Sigma(X : \mathcal{U}_\ell). \\ & \Sigma^{\P}(\_ \cdot \_ : X \to X \to X).\top.\end{aligned}$$

We will denote by $(X, \cdot)$ the magma with carrier X and (infix) operation $\_ \cdot \_$.[6]

Suppose now that we are working in a context Γ and that $(R, \cdot) : \textsf{Magma}_k$ is a magma. Additionally, suppose that we can derive the judgement $\P \setminus \Gamma \vdash r : R$. Then we can construct (in the context Γ) for any $\ell \geq k$ the writer premonad $\textsf{writer-pm}_{R,r^{\P}} : \textsf{Premonad}_\ell$. Its type operator is given by

$$\textsf{writer-pm}_{R,r^{\P}} \quad := \quad \lambda(X : \mathcal{U}_\ell)\,.\,X \times R \quad : \quad \mathcal{U}_\ell \to \mathcal{U}_\ell$$

and the return function maps any $x : X$ to the tuple with r as its second component, so

$$\textsf{return}_{\textsf{writer-pm}_{R,r^{\P}}} \quad := \quad \lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,(x, r) \quad : \quad \forall(X : \mathcal{U}_\ell).X \to X \times R.$$

Finally, the bind operation takes $m : X \times R$ and $k : X \to Y \times R$ and produces a value of type $Y \times R$ by applying k to the first component of m, retaining the first component and multiplying the second component with that of m (using the magma operation). More concretely, we have

$$m \gg\!=_{\textsf{writer-pm}_{R,r^{\P}}} k := \big(\textsf{fst}\,(k\,(\textsf{fst}\,m)), \quad \textsf{snd}\,m \,\cdot\, \textsf{snd}\,(k\,(\textsf{fst}\,m))\big).$$

Notice that $\textsf{writer-pm}_{R,r^{\P}}$ depends pointwise on r because any premonad depends pointwise on its return function and in this case we needed r to implement that return function. More concretely, if we could only derive the judgement $\Gamma \vdash r : R$ (so with context Γ and not $\P \setminus \Gamma$), then r becomes available only for parametric use in the context $\P \setminus \Gamma$, which is the context in which the return function is type checked, and hence we would not be able to use r in the implementation of $\textsf{return}$.

[6]Notice that again a magma is actually a 3-tuple containing an irrelevant value of type $\top$, but we will not write this last trivial component.

As mentioned above, given a monad M we will interpret effectful functions from A to B as pure functions of type $A \to M\,B$ and the bind operation provides a way to compose such functions. This way of composing functions is possible even if M is merely a premonad and we could then try to build a category using this notion of composition. More concretely, suppose that $M : \mathsf{Premonad}_\ell$ is a premonad. Then the category $\mathcal{C}_M$ we want to construct has as objects the types in the universe $\mathcal{U}_\ell$ and the morphisms from a type $A : \mathcal{U}_\ell$ to $B : \mathcal{U}_\ell$ are all functions of type $A \to M\,B$. The identity morphism for a type A is given by $\mathsf{return}_M\,A^\sharp : A \to M\,A$ and the composition of $f : A \to M\,B$ and $g : B \to M\,C$ is as usual

$$\lambda(x : A)\,.\,f\,x \gg\!=_M g.$$

However, for $\mathcal{C}_M$ to be a category, composition needs to be associative and the identity morphisms must be left and right units for composition. Translating the law that the identity morphism is a left unit for composition gives us the requirement that

$$\mathsf{return}_M\,x \gg\!=_M k \;=\; k\,x \tag{3.2}$$

for any types $X, Y : \mathcal{U}_\ell$, any $x : X$ and any function $k : X \to M\,Y$. Similarly we can express the law that the identity morphism is a right unit for composition as

$$m \gg\!=_M (\mathsf{return}_M\,X^\sharp) \;=\; m \tag{3.3}$$

for all types $X : \mathcal{U}_\ell$ and all $m : M\,X$.[7] Finally the translation of the associativity of composition gives us that

$$(m \gg\!=_M k) \gg\!=_M q \;=\; m \gg\!=_M (\lambda(x : X)\,.\,k\,x \gg\!=_M q) \tag{3.4}$$

for all $X, Y, Z : \mathcal{U}_\ell$, and any $m : M\,X$ and $k : X \to M\,Y$ and $q : Y \to M\,Z$. Equations (3.2) to (3.4) are called the *monad laws* and every premonad satisfying these laws is called a *monad*. Using the Curry-Howard correspondence we can translate the monad laws to types and then define for any premonad M a type $\mathsf{IsMonad}\,M$ of proofs that M is a monad (i.e. proofs that M satisfies the three monad laws).

Definition 3.3. For every universe $\mathcal{U}_\ell$ we define a type family (i.e. a function with a universe as codomain) $\mathsf{IsMonad}_\ell : \mathsf{Premonad}_\ell \to \mathcal{U}_{\ell+1}$ by saying that

$$\begin{aligned}
\mathsf{IsMonad}_\ell\,M := {} & \Sigma^\P(\mathsf{return\text{-}law1} : \forall(X, Y : \mathcal{U}_\ell).\Pi(x : X).\Pi(k : X \to M\,Y).\\
& \qquad\qquad \mathsf{return}_M\,x \gg\!=_M k \equiv_{M\,Y} k\,x).\\
& \Sigma^\P(\mathsf{return\text{-}law2} : \forall(X : \mathcal{U}_\ell).\Pi(m : M\,X).m \gg\!=_M (\mathsf{return}_M\,X^\sharp) \equiv_{M\,X} m).\\
& \Sigma^\P(\mathsf{assoc\text{-}law} : \forall(X, Y, Z : \mathcal{U}_\ell).\Pi(m : M\,X).\\
& \qquad \Pi(k : X \to M\,Y).\Pi(q : Y \to M\,Z).\\
& \qquad (m \gg\!=_M k) \gg\!=_M q \equiv_{M\,Z} m \gg\!=_M (\lambda(x : X)\,.\,k\,x \gg\!=_M q)).\\
& \top.
\end{aligned}$$

A monad is then defined to be a premonad $M : \mathcal{U}_\ell$ together with a value of $\mathsf{IsMonad}_\ell\,M$.

[7] Strictly speaking the translation of this law should be $k\,z \gg\!=_M (\mathsf{return}_M\,X^\sharp) \;=\; k\,z$ for all $k : Z \to M\,X$ and any $z : Z$. However it is easy to see that this is equivalent to equation (3.3). A similar remark also holds for equation (3.4).

Remark 3.4. We formulate some remarks regarding the definition of a monad.

1. Strictly speaking, a monad is defined as a pair consisting of a premonad and a proof that this premonad satisfies the monad laws. In other words, it is a value of the type $\Sigma(M : \mathsf{Premonad}_\ell).\, \mathsf{IsMonad}_\ell\, M$. However, from now on we say by abuse of terminology that a monad is a premonad that satisfies the monad laws. Note that this could cause some problems because there could be different proofs that a premonad M satisfies the monad laws, so different values of $\mathsf{IsMonad}_\ell\, M$. These two different proofs strictly speaking give rise to different monads which is not taken into account when saying that a monad is a premonad satisfying the monad laws. However, when we consider a monad in this thesis it will be in general clear what its proof of the monad laws is. Note that this proof relevance is a general phenomenon in the formalization of mathematics using dependent type theory.

2. Just like in the definitions of a functor and of a premonad, the last component of type $\top$ in a value of type $\mathsf{IsMonad}_\ell\, M$ for a premonad M is an irrelevant piece of information.

3. Also notice that in the translation of the monad laws to types using the Curry-Howard correspondence, some modalities are introduced. The motivation for these parametric modalities is the same as in previous instances such as the types of return and $\gg\!=$.

4. From now on, if we have a premonad $M : \mathsf{Premonad}_\ell$ and a proof $\widetilde{M} : \mathsf{IsMonad}_\ell\, M$ that M is a monad, then we will write $\mathsf{return\text{-}law1}\, \widetilde{M}$ for its proof of the first return law, $\mathsf{return\text{-}law2}\, \widetilde{M}$ for its proof of the second return law and $\mathsf{assoc\text{-}law}\, \widetilde{M}$ for its proof of the associativity law.

5. As a final remark, the universe level ℓ in $\mathsf{IsMonad}_\ell\, M$ can always be deduced from the type of M and will henceforth be omitted.

From the discussion leading to the definition of a monad, we know that if the premonad $M : \mathsf{Premonad}_\ell$ is a monad, then $\mathcal{C}_M$ as constructed above really is a category. This is called the Kleisli category for the monad M after the Swiss mathematician Heinrich Kleisli [Kle65]. We can say that the morphisms in the Kleisli category $\mathcal{C}_M$ from a type X to a type Y are the effectful functions from X to Y (where the kind of effects depends on M).

Example 3.5. We now investigate which premonads of example 3.2 are monads. These examples are also worked out in Agda parametric (in the same file as that for example 3.2).

1. The identity premonad $\mathsf{id\text{-}pm}$ is a monad because composition is here just ordinary composition and the identity morphisms are the usual identity functions.

2. The Maybe premonad turns out to be a monad as well. The first return law follows immediately from the definition of $\gg\!=_{\text{Maybe}}$. The two other laws require some more work, but we refer to the Agda implementation for all details.

3. For any type S the state premonad $\mathsf{state\text{-}pm}_S$ is a monad. All monad laws follow trivially by working out definitions of return and $\gg\!=$.

4. It is not the case that $\textsf{writer-pm}_{R,r¶}$ is a monad for every magma $(R,\cdot)$ and every $r : R$. There will be an easy characterization for when this is the case but we postpone this to section 3.3.

A final notion that we introduce in this section and that will be important in some results of chapter 4 is that of a *monad morphism*. Despite its name a monad morphism can be defined between any two premonads, they do not necessarily have to satisfy the monad laws. More concretely, if $M_1, M_2 : \textsf{Premonad}_\ell$ are premonads, then a monad morphism from M_1 to M_2 is a polymorphic map h of type $\forall(X : \mathcal{U}_\ell).M_1\,X \to M_2\,X$ that is compatible with the return and bind functions of both premonads. Compatibility with the return functions means that

$$h\,(\textsf{return}_{M_1}\,x) = \textsf{return}_{M_2}\,x$$

for any type $X : \mathcal{U}_\ell$ and any $x : X$ and compatibility with the bind operations means that

$$h\,(m \gg\!=_{M_1} k) \;=\; (h\,m) \gg\!=_{M_2} (h \circ k)$$

for any types $X, Y : \mathcal{U}_\ell$ and any $m : M_1\,X$ and $k : X \to M_1\,Y$. Notice that h takes a type argument which is omitted for the sake of readability. Again we can translate the monad morphism laws to types using the Curry-Howard correspondence and subsequently we can define the type of monad morphisms from M_1 to M_2.

Definition 3.6. For any universe $\mathcal{U}_\ell$ we define the type family

$$\textsf{MonadMorphism} : \textsf{Premonad}_\ell \to \textsf{Premonad}_\ell \to \mathcal{U}_{\ell+1}$$

as follows

$$\begin{aligned}
\textsf{MonadMorphism}\,M_1\,M_2 := {} & \Sigma(h : \forall(X : \mathcal{U}_\ell).M_1\,X \to M_2\,X).\\
& \Sigma^{¶}(\textsf{morph-return} : \forall(X : \mathcal{U}_\ell).\Pi(x : X).\\
& \qquad\qquad h\,(\textsf{return}_{M_1}\,x) \equiv_{M_2\,X} \textsf{return}_{M_2}\,x).\\
& \Sigma^{¶}(\textsf{morph-bind} : \forall(X, Y : \mathcal{U}_\ell).\Pi(m : M_1\,X).\Pi(q : X \to M_1\,Y).\\
& \qquad\qquad h\,(m \gg\!=_{M_1} k) \equiv_{M_2\,Y} (h\,m) \gg\!=_{M_2} (h \circ k)).\\
& \top.
\end{aligned}$$

If $M_1, M_2 : \textsf{Premonad}_\ell$ are premonads and h is a value of type $\textsf{MonadMorphism}\,M_1\,M_2$ then h is called a monad morphism from M_1 to M_2.

Notice that again there is an irrelevant piece of information of type $\top$ contained in a monad morphism. Just like with the definition of a monad, we will say from now on by abuse of terminology that a monad morphism from $M_1 : \textsf{Premonad}_\ell$ to $M_2 : \textsf{Premonad}_\ell$ is a polymorphic function $h : \forall(X : \mathcal{U}_\ell).M_1\,X \to M_2\,X$ that satisfies the laws above.

Example 3.7. If $M : \textsf{Premonad}_\ell$ is a premonad, then we know that $\textsf{return}_M$ has type $\forall(X : \mathcal{U}_\ell).X \to M\,X$. In this example we will see that if M is a monad (witnessed by $\widetilde{M} : \textsf{IsMonad}\,M$) then this return function is a monad morphism from the identity monad id-pm to M. Indeed, using the definition of return for id-pm the monad morphism law involving return functions becomes trivial. Furthermore, taking into account the definition of $\gg\!=_{\mathsf{id\text{-}pm}}$, the morphism law involving bind functions requires a proof of type

$$\forall(X, Y : \mathcal{U}_\ell).\Pi(x : X).\Pi(k : X \to Y).\, \mathsf{return}_M\,(k\,x) \equiv_{M\,Y} (\mathsf{return}_M\,x) \gg\!=_M (\mathsf{return}_M \circ k).$$

This proof can be constructed using $\mathsf{return\text{-}law1}\,\widetilde{M}$ which gives for any $X, Y : \mathcal{U}_\ell$, $x : X$ and $k : X \to Y$ a value of type

$$(\mathsf{return}_M\,x) \gg\!=_M (\mathsf{return}_M \circ k) \equiv_{M\,Y} (\mathsf{return}_M \circ k)\,x.$$

Hence the result follows from symmetry of propositional equality (see example 1.30).

3.2.2 A different formulation

As mentioned in the introduction to this chapter, the concept of a monad originated in category theory. However, the usual category theoretic definition of a monad is quite different from the one given in Definitions 3.1 and 3.3. More concretely, a monad in category theory is defined as an endofunctor T on a category $\mathcal{C}$ together with natural transformations $\eta : \mathsf{id}_\mathcal{C} \dot{\to} T$ and $\mu : T^2 \dot{\to} T$ (where $\mathsf{id}_\mathcal{C}$ is the identity endofunctor on $\mathcal{C}$ and $T^2 = T \circ T$ is the composition of T with itself) such that the following diagrams commute.

$$\begin{tikzcd}
T^3 \arrow[r, Rightarrow, "\mu T"] \arrow[d, Rightarrow, "T\mu"] & T^2 \arrow[d, Rightarrow, "\mu"] \\
T^2 \arrow[r, Rightarrow, "\mu"] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, Rightarrow, "\eta T"] \arrow[dr, equal] & T^2 \arrow[d, Rightarrow, "\mu"] & T \arrow[l, Rightarrow, "T\eta"'] \arrow[dl, equal] \\
& T &
\end{tikzcd}$$

Here μT is the natural transformation from T^3 to T^2 such that $(\mu T)_X = \mu_{T\,X}$ for all objects X and likewise for ηT. Similarly, $T\mu$ is the natural transformation from T^3 to T^2 such that $(T\mu)_X = T(\mu_X)$ for all objects X and likewise for $T\eta$.

The category theoretic definition above is the inspiration for the following alternative definitions of a premonad and monad in ParamDTT.

Definition 3.8. For any universe $\mathcal{U}_\ell$ we define the type

$$\begin{aligned}
\mathsf{Premonad\text{-}rj}_\ell := {} & \Sigma(F : \mathsf{Functor}_{\ell,\ell}). \\
& \Sigma^{\P}(\eta : \forall(X : \mathcal{U}_\ell).X \to F\,X). \\
& \Sigma^{\P}(\mu : \forall(X : \mathcal{U}_\ell).F\,(F\,X) \to F\,X).\top.
\end{aligned}$$

If $M : \mathsf{Premonad\text{-}rj}_\ell$, then we write funct_M for its functor and η_M and μ_M for its two natural transformations. Most of the time M will be clear from the context and then we just write funct, η and μ. Just like with return and $\gg\!=$, we will often leave type arguments implicit as they can be inferred from other arguments.

Definition 3.9. The type family $\mathsf{IsMonad\text{-}rj}_\ell : \mathsf{Premonad\text{-}rj}_\ell \to \mathcal{U}_{\ell+1}$ is defined as

$$\begin{aligned}
\mathsf{IsMonad\text{-}rj}\,M := {} & \Sigma^{\P}(\mu\text{-}\mathsf{law} : \forall(X : \mathcal{U}_\ell).\Pi(x : \mathsf{funct}^3\,X).\mu\,(\mu\,x) \equiv_{\mathsf{funct}\,X} \mu\,(\mathsf{funct}\,\mu\,x)). \\
& \Sigma^{\P}(\eta\text{-}\mathsf{law1} : \forall(X : \mathcal{U}_\ell).\Pi(x : \mathsf{funct}\,X).\mu\,(\eta\,x) \equiv_{\mathsf{funct}\,X} x). \\
& \Sigma^{\P}(\eta\text{-}\mathsf{law2} : \forall(X : \mathcal{U}_\ell).\Pi(x : \mathsf{funct}\,X).\mu\,(\mathsf{funct}\,\eta\,x) \equiv_{\mathsf{funct}\,X} x).\top.
\end{aligned}$$

The types of the different non-trivial components of $\mathsf{IsMonad\text{-}rj}\, M$ are the translations (by the Curry-Howard correspondence) of the requirement that the diagrams above commute. This is not completely trivial to see at first sight because a lot of type arguments for η and μ are omitted.

Remark 3.10. We make two remarks concerning definitions 3.8 and 3.9.

1. In the category theoretic definition, η and μ are required to be natural transformations and hence one would expect to see naturality conditions for η and μ in the definition of Premonad-rj or IsMonad-rj as well. However, because η and μ are parametric in their type arguments, we can apply Proposition 2.13 to see that these two functions are automatically natural thanks to parametricity (of course for μ one should first show that a functor composed with itself gives again rise to a functor, but that is not too difficult).

2. The type operator of the functor in a value of type $\mathsf{Premonad\text{-}rj}_\ell$ must have the same domain and codomain $\mathcal{U}_\ell$. Indeed, otherwise we would not be able to compose this functor with itself and define μ. This reflects the fact that the functor in the category theoretic definition of a monad needs to be an endofunctor. Since we want this alternative definition of a monad to be equivalent to that from section 3.2.1 (see also the next paragraphs), we also required from the type operator in definition 3.1 to have equal domain and codomain.

Of course, a natural question that now emerges is whether this definition of a monad is equivalent to the definition we saw in section 3.2.1. This appears to be the case, so both definitions are different formulations of the same concept.[8] We call the formulation of the previous section the return-bind or simply bind formulation (for obvious reasons) and the one of this section the return-join or simply join formulation of a monad (because the operation μ is often called join, this also explains the suffix rj in for example Premonad-rj). However, it is not the case that both definitions of a premonad are equivalent. Every value of $\mathsf{Premonad\text{-}rj}_\ell$ can be turned into a value of type $\mathsf{Premonad}_\ell$ but the converse is not true. The reason for this is not fundamental but it is a consequence of our definition of a functor, more specifically of the requirement that a functor preserves identity functions. Both definitions would be equivalent if we had defined a notion of a prefunctor (with actions on types and functions not necessarily preserving identity functions) and had defined Premonad-rj using such a prefunctor.

We now give a more concrete idea of the equivalence of both definitions of a monad. All details are worked out in a formalization in Agda parametric.[9]

Given a premonad $M : \mathsf{Premonad}_\ell$ that is a monad (witnessed by $\widetilde{M} : \mathsf{IsMonad}\, M$), we can construct a functor whose type operator is just M. For that purpose, we need a procedure to convert a function $f : X \to Y$ into some function of type $M\, X \to M\, Y$. This can be done by first considering the composition $\mathsf{return}_M \circ f : X \to M\, Y$ and then applying the bind operation for M to obtain

$$\lambda(m : M\, X)\, .\, m \gg\!=_M \mathsf{return}_M \circ f \quad : \quad M\, X \to M\, Y.$$

[8]Although both definitions are equivalent, this cannot be proved completely in ParamDTT. The reason for this is not fundamental but exposes some shortcomings of the type system ParamDTT. We will come back to this later in this section and we will discuss in general the practical use of ParamDTT in chapter 6.

[9]See the following file in the GitHub repository `https://github.com/JorisCeulemans/effect-param-agda/blob/master/Monads/Return-Join-Isomorphism.agda`.

The fact that this procedure preserves identity functions corresponds exactly to the second monad law involving return_M and hence a proof is given by $\mathsf{return\text{-}law2}\,\widetilde{M}$. This implies that we need $\widetilde{M}$ in order to construct a functor out of the premonad M and it explains why we cannot build a value of type $\mathsf{Premonad\text{-}rj}_\ell$ out of a value of type $\mathsf{Premonad}_\ell$ alone. Subsequently, the polymorphic function η has exactly the same type as return_M and hence the only thing we still have to construct is μ. This can be done by noticing that $\gg\!=_M$ can be used to convert a function of type $A \to M\,B$ into a function of type $M\,A \to M\,B$ for all types $A, B : \mathcal{U}_\ell$. If we substitute A by $M\,X$ and B by X, then we can convert the identity function $\mathsf{id}_{M\,X} : M\,X \to M\,X$ into a function of type $M\,(M\,X) \to M\,X$ (for any $X : \mathcal{U}_\ell$) and hence we can define μ as

$$\mu\,X^\sharp := \lambda(m : M\,(M\,X))\,.\,m \gg\!=_M \mathsf{id}_{M\,X}.$$

At this point, we have constructed a value of type $\mathsf{Premonad\text{-}rj}_\ell$. A construction of a proof of the laws described in $\mathsf{IsMonad\text{-}rj}$ is more technically involved and can be found in the formalization in Agda parametric.

For the converse direction of the equivalence, suppose that we have $M : \mathsf{Premonad\text{-}rj}_\ell$ and that we want to construct a premonad of type $\mathsf{Premonad}_\ell$. The type operator is given by the type operator of funct_M and the return function is just η_M. The only non-trivial object to construct is the bind operation, so assume that we have $X, Y : \mathcal{U}_\ell$ and $k : X \to \mathsf{funct}_M\,Y$. We can then apply the functor funct_M to k to get a function of type $\mathsf{funct}_M\,X \to \mathsf{funct}_M\,(\mathsf{funct}_M\,Y)$. Subsequently, we compose this function with $\mu\,Y^\sharp : \mathsf{funct}_M\,(\mathsf{funct}_M\,Y) \to \mathsf{funct}_M\,Y$ to obtain a function of type $\mathsf{funct}_M\,X \to \mathsf{funct}_M\,Y$. In summary, the bind operation is given by

$$\lambda(X^\sharp, Y^\sharp : \mathcal{U}_\ell)\,.\,\lambda(m : \mathsf{funct}_M\,X)\,.\,\lambda(k : X \to \mathsf{funct}_M\,Y)\,.\,\mu\,(\mathsf{funct}_M\,k\,m).$$

This gives the procedure to convert a value M of type $\mathsf{Premonad\text{-}rj}_\ell$ into a value M' of type $\mathsf{Premonad}_\ell$. Then we would like to construct from any value of type $\mathsf{IsMonad\text{-}rj}\,M$ a proof that the corresponding premonad M' is a monad, so a value of type $\mathsf{IsMonad}\,M'$. This is however impossible in ParamDTT. The problem lies in the fact that in for instance the construction of a term of type

$$\forall(X, Y : \mathcal{U}_\ell).\Pi(x : X).\Pi(k : X \to M'\,Y).\,\mathsf{return}_{M'}\,x \gg\!=_{M'} k \equiv_{M'\,Y} k\,x$$

proving the first monad law for M' involving return, we need to apply naturality of η with respect to k. Hence we want to use Proposition 2.13, but the term $\mathsf{naturality}$ we constructed in that proposition depends pointwise on its function f and since k is only available for continuous use, we are not allowed to pass it as an argument to $\mathsf{naturality}$. A similar situation occurs when proving the monad law stating that $\gg\!=_{M'}$ is associative. A possible solution would be to make some monad laws depend pointwise on their function arguments instead of continuous (for instance in the type above we could replace $\Pi(k : X \to M'\,Y)$ with $\Pi^\P(k : X \to M'\,Y)$) so that these functions become available for pointwise use and can therefore be passed as an argument to $\mathsf{naturality}$. In fact, this was our first approach when working out monads and effect parametricity in ParamDTT and Agda parametric.[10] However, later we realized that this approach introduces new problems

[10]See `https://github.com/JorisCeulemans/effect-param-agda/releases/tag/isomorphism` for an earlier version of the code in which both formulations of a monad can still be proved to be equivalent.

when working with the glue constructor and $\mathsf{ind}_{\mathsf{Weld}}$ eliminator in the proofs of effect parametricity (see chapter 4).[11] Hence we decided to make the monad laws continuous in their function arguments and give up the equivalence of both definitions of a monad in ParamDTT. We want to emphasize that this is not a fundamental problem in both formulations of a monad, but rather a consequence of the fact that the graph relation former $/ \cdot \backslash$ in ParamDTT depends pointwise on its function argument and that therefore $\mathsf{naturality}$ in Proposition 2.13 also has this pointwise dependence.

3.3 Monads and Parametricity

We already pointed out in remark 3.10 that in the join formulation of a monad there are some laws that need not to be proved because of parametricity. More concretely, we do not have to show that a functor respects composition (see Proposition 2.9)[12] and that the polymorphic functions η and μ are natural transformations. These results could give us hope that also in the bind formulation of a monad some monad laws can be proved automatically using parametricity of return and $\gg\!=$ and that we can then simplify the definition of a monad just like we did for functors in section 2.3. That question will be the subject of this section.[13]

Unfortunately, we are not as lucky as in section 2.3 and the monad laws do not follow from parametricity. In fact, we will construct for each monad law a premonad that does not satisfy this law, but still satisfies the two other monad laws. As a consequence, this shows that no monad law follows from the other two by parametricity.

The examples we will construct all involve the writer premonad for a pointed magma from example 3.2. We will investigate for which magmas $(R, \cdot) : \mathsf{Magma}_\ell$ and which elements $r : R$ the writer premonad $\mathsf{writer\text{-}pm}_{R,r\P}$ is a monad. Recall that the return function for $\mathsf{writer\text{-}pm}_{R,r\P}$ is given by

$$\mathsf{return}\, x := (x, r)$$

for any $X : \mathcal{U}_\ell$ and $x : X$ and that the bind function was defined as

$$m \gg\!= k := \big(\mathsf{fst}\,(k\,(\mathsf{fst}\, m)), \quad \mathsf{snd}\, m \ \cdot\ \mathsf{snd}\,(k\,(\mathsf{fst}\, m))\big)$$

for any $X, Y : \mathcal{U}_\ell$ and $m : X \times R$ and $k : X \to Y \times R$. Expanding these definitions of return and bind in the first monad law involving return, we get the requirement that

$$\big(\mathsf{fst}\,(k\, x), r \cdot \mathsf{snd}\,(k\, x)\big) \ = \ k\, x \tag{3.5}$$

for all $X : \mathcal{U}_\ell$, $x : X$ and $k : X \to Y \times R$. This is clearly satisfied if r is a left unit for the magma operation $\_ \cdot \_$. Moreover, given any $s : R$ we can take for k the function $\lambda(x : \top) . (x, s)$ of type $\top \to \top \times R$ and hence we see that equation (3.5) is equivalent to r being a left unit for $\_ \cdot \_$.

[11]These problems did not directly show up in Agda parametric because it does not check modalities when applying rewrite rules, see also `https://github.com/agda/agda/issues/3839`.

[12]Since the definition of a monad in the join formulation involves a functor, this result indeed reduces the amount of work needed to construct a monad using this formulation.

[13]A formalization of the results in this section is available at `https://github.com/JorisCeulemans/effect-param-agda/blob/master/Monads/Examples.agda`, along with examples of premonads and monads.

$\cdot$	a	b
a	a	a
b	b	b

(a) Associative magma with right units but no left unit.

$\cdot$	a	b
a	a	b
b	a	b

(b) Associative magma with left units but no right unit.

$\cdot$	e	a	b
e	e	a	b
a	a	b	a
b	b	a	a

(c) Non-associative magma with unit e. We have for instance that $a \cdot (b \cdot b) = b \neq a = (a \cdot b) \cdot b$.

Table 3.1: Multiplication tables of the different magmas for the construction of premonads that do not satisfy exactly one monad law.

Using the definitions of return and bind for $\mathsf{writer\text{-}pm}_{R,r}$¶ in the second monad law involving return, we obtain the requirement that

$$\big(\mathsf{fst}\, m, \mathsf{snd}\, m \;\cdot\; r\big) \;=\; m \tag{3.6}$$

for any $X : \mathcal{U}_\ell$ and $m : X \times R$. Hence it is easy to see that this monad law for $\mathsf{writer\text{-}pm}_{R,r}$¶ is equivalent to r being a right unit for the magma operation $\_ \cdot \_$.

Finally, we expand the definitions of return and $\gg=$ in the monad law involving associativity of bind to get the requirement that

$$\begin{aligned}&\Big(\mathsf{fst}\,(q\,(\mathsf{fst}\,(k\,(\mathsf{fst}\, m)))),\quad \big(\mathsf{snd}\, m \,\cdot\, \mathsf{snd}\,(k\,(\mathsf{fst}\, m))\big) \,\cdot\, \mathsf{snd}\,(q\,(\mathsf{fst}\,(k\,(\mathsf{fst}\, m))))\Big)\\ &\qquad = \Big(\mathsf{fst}\,(q\,(\mathsf{fst}\,(k\,(\mathsf{fst}\, m)))),\quad \mathsf{snd}\, m \,\cdot\, \big(\mathsf{snd}\,(k\,(\mathsf{fst}\, m)) \,\cdot\, \mathsf{snd}\,(q\,(\mathsf{fst}\,(k\,(\mathsf{fst}\, m))))\big)\Big)\end{aligned} \tag{3.7}$$

for any $X, Y, Z : \mathcal{U}_\ell$, $m : X \times R$, $k : X \to Y \times R$ and $q : Y \to Z \times R$. It is clear that this requirement is satisfied if the magma operation $\_ \cdot \_$ is associative. Moreover, by taking for k and q functions with constant second components, one can see that equation (3.7) is equivalent to the associativity of $\_ \cdot \_$.

In summary, equations (3.5) to (3.7) tell us that $\mathsf{writer\text{-}pm}_{R,r}$¶ is a monad if and only if $(R, \cdot)$ is a monoid and r is its unit. Moreover, we can construct a premonad that does not satisfy the first return law but does satisfy the other monad laws by taking the writer premonad for the magma given by the multiplication table in Table 3.1a. We see that this magma is associative and has a right unit (both a and b are right units) but it has no left unit. Similarly a premonad that does not satisfy the second return law but does satisfy the other monad laws is given by the writer premonad for the magma given in Table 3.1b. This magma is associative and has a left unit (both a and b are left units), but no right unit. Finally, the magma defined in Table 3.1c has a (left and right) unit e but is not associative and hence its corresponding writer premonad (with the element e) satisfies both return laws but not the associativity law.

Chapter 4

Effect Parametricity and Dependent Types

In the previous chapter we saw that in a purely functional programming language such as Haskell, functions causing side effects can be treated using monads and that different monads give rise to different kinds of side effects. In this chapter, we will be interested in so-called *effect polymorphic* functions, which are functions that work for any of monad or in other words for any kind of side effect. Consider for example the following function from the Haskell Prelude (a standard library for Haskell).[1]

```
sequence :: Monad m => [m a] -> m [a]
```

For any monad `m` it takes a list of monadic computations (that yield a result of any type `a`) and performs them sequentially, collecting their results in a list. This function works for any monad and its implementation cannot inspect the monad it is instantiated with because the Haskell type system prohibits this. Therefore we can say that `sequence` is parametric in the monad `m`.

In [Voi09], Janis Voigtländer extended the procedure we saw in section 2.1 for deriving free theorems from types of polymorphic functions, to include effect polymorphic functions such as `sequence`.[2] In this way he was able to prove various useful properties of such functions metatheoretically. For instance, a parametricity argument shows that if all monadic computations in a list of type `[m a]` are pure, then so is the result of applying `sequence` to this list. This use of parametricity to prove theorems about effect polymorphic functions is called *effect parametricity*. In this chapter, we will describe how properties involving effect parametricity can be proved internally in the type system ParamDTT. More concretely, in section 4.1 we will see how effect polymorphism can be expressed in ParamDTT and in section 4.2 we will give a general idea of how to apply the machinery of this type system to prove properties using effect parametricity. In the subsequent sections we will make this general idea more concrete and prove Theorems

[1]See `http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html`.

[2]These effect polymorphic functions cannot be expressed in System F as they involve quantification over type operators (i.e. functions that map types to types). A well-known extension of System F that allows such quantification is called System Fω. The relational interpretation of System Fω types that is behind the procedure to derive free theorems is referred to in [Voi09] as being folklore, but has in the meantime been formalized in [Atk12].

1–5 from [Voi09] in ParamDTT (the other theorems in that article do not involve effect parametricity). All results in this chapter have been formalized in Agda parametric.[3]

4.1 Effect Polymorphism

Before we can describe how effect parametricity in ParamDTT works, we need to know how we can express effect polymorphic functions in this type system, which is the subject of this section. We start with a translation of the Haskell function `sequence` that we saw in the introduction to this chapter. For that purpose, we need to introduce lists in our type system (note that we used lists in some examples in the previous chapters but we never formally introduced them) and hence we add the following inference rules.

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell}{\Gamma \vdash \mathsf{List}\, A : \mathcal{U}_\ell}\ \textsc{List-form}$$

$$\frac{\Gamma \vdash \mathsf{List}\, A \text{ type}}{\Gamma \vdash [\,] : \mathsf{List}\, A}\ \textsc{List-intro-empty} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash l : \mathsf{List}\, A}{\Gamma \vdash a :: l : \mathsf{List}\, A}\ \textsc{List-intro-cons}$$

$$\frac{\begin{array}{ll}\Gamma, x : \mathsf{List}\, A \vdash B \text{ type} & \Gamma \vdash b_e : B[[\,]/x] \\ \Gamma \vdash l : \mathsf{List}\, A & \Gamma, y : A, z : \mathsf{List}\, A, w : B[z/x] \vdash b_c : B[y :: z/x]\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathrm{List}}(x.B, b_e, y.z.w.b_c, l) : B[l/x]}\ \textsc{List-elim}$$

$$\frac{\begin{array}{cc} & \Gamma, x : \mathsf{List}\, A \vdash B \text{ type} \\ \Gamma \vdash b_e : B[[\,]/x] & \Gamma, y : A, z : \mathsf{List}\, A, w : B[z/x] \vdash b_c : B[y :: z/x]\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathrm{List}}(x.B, b_e, y.z.w.b_c, [\,]) = b_e : B[[\,]/x]}\ \textsc{List-comp-empty}$$

$$\frac{\begin{array}{ll}\Gamma, x : \mathsf{List}\, A \vdash B \text{ type} & \\ \Gamma \vdash b_e : B[[\,]/x] & \Gamma, y : A, z : \mathsf{List}\, A, w : B[z/x] \vdash b_c : B[y :: z/x] \\ \Gamma \vdash a : A & \Gamma \vdash l : \mathsf{List}\, A\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathrm{List}}(x.B, b_e, y.z.w.b_c, a :: l) = b_c[a/y, l/z, \mathsf{ind}_{\mathrm{List}}(x.B, b_e, y.z.w.b_c, l)/w] : B[a :: l/x]}$$

(List-comp-cons)

The introduction rules allow us to construct values of type $\mathsf{List}\, A$ (for any type A) that are of the form $a_1 :: a_2 :: \ldots :: a_n :: [\,]$ with $a_1, a_2, \ldots, a_n$ values of type A, and of course these represent the lists with elements $a_1, a_2, \ldots, a_n$ (and consequently $[\,]$ denotes the empty list). The induction principle then intuitively tells us that these are essentially the only values of type $\mathsf{List}\, A$. More concretely, it allows us to construct a value of a type B that depends on a variable $x : \mathsf{List}\, A$ by giving both a value of B assuming that x is the empty list $[\,]$ and a procedure to convert $B[l/x]$ into $B[a :: l/x]$ for any list $l : \mathsf{List}\, A$ and any value $a : A$. Using this induction principle we can for instance define a length function $\mathsf{length} : \mathsf{List}\, A \to \mathbb{N}$ for any type A as

$$\mathsf{length} := \lambda(l : \mathsf{List}\, A)\,.\, \mathsf{ind}_{\mathrm{List}}(x.\mathbb{N}, 0, y.z.w.(\mathsf{succ}\, w), l).$$

[3]See the folder `https://github.com/JorisCeulemans/effect-param-agda/tree/master/EffectParametricity` in the GitHub repository.

Furthermore, we can implement a function map that takes for any types A and B a function $f : A \to B$ and a list $l : \mathrm{List}\, A$ and returns the list of type $\mathrm{List}\, B$ obtained by applying f to all elements of l. More concretely we have that

$$\mathsf{map}\, f\, l := \mathsf{ind}_{\mathrm{List}}(x.\, \mathrm{List}\, B, [\,], y.z.w.(f\, y :: w), l),$$

and this allows us to convert f into a function $\mathsf{map}\, f$ of type $\mathrm{List}\, A \to \mathrm{List}\, B$. In this way one can construct a functor as defined in section 2.3 with the List type former as its action on types.

Now we can proceed with the definition in ParamDTT of a function $\mathsf{sequence}$ that behaves like its Haskell namesake. For reasons that will be clear in section 4.2, we will want this function to work for any premonad instead of any monad.[4] Hence for a given universe $\mathcal{U}_\ell$ the function $\mathsf{sequence}$ should take a premonad $M : \mathsf{Premonad}_\ell$, a type $X : \mathcal{U}_\ell$ and a list l of type $\mathrm{List}\,(M\, X)$. It should then return a value of type $M\,(\mathrm{List}\, X)$ obtained by sequentially composing all computations in l. We will use the induction principle for lists to express this more formally. That induction principle tells us that we can first assume that l is the empty list, in which case the result should be a monadic computation that just yields the empty list as its result and this computation is given by $\mathsf{return}_M\, [\,]$. Then we can assume that l is of the form $m :: l'$ with $m : M\, X$ and $l' : \mathrm{List}\,(M\, X)$ and that we know the result of $\mathsf{sequence}$ applied to l'. In this case, we execute the effectful computation m and we compose this with $\mathsf{sequence}$ applied to l' using the bind operation for M. More concretely, we define $\mathsf{sequence}$ as

$$\begin{aligned}\mathsf{sequence} := {} & \lambda(M^\sharp : \mathsf{Premonad}_\ell)\,.\,\lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\lambda(l : \mathrm{List}\,(M\, X))\,.\\ & \mathsf{ind}_{\mathrm{List}}\big(z.M\,(\mathrm{List}\, X),\\ & \qquad \mathsf{return}_M\,[\,],\\ & \qquad m.l'.w.\; m \gg\!=_M \big(\lambda(x : X)\,.\, w \gg\!=_M (\lambda(y : \mathrm{List}\, X)\,.\; \mathsf{return}_M\,(x :: y))\big),\\ & \qquad l\big).\end{aligned}$$

Its type is $\forall(M : \mathsf{Premonad}_\ell).\forall(X : \mathcal{U}_\ell).\, \mathrm{List}\,(M\, X) \to M\,(\mathrm{List}\, X)$ and hence we clearly see that $\mathsf{sequence}$ is parametric in its premonad argument. In fact we should annotate $\mathsf{sequence}$ with the universe level ℓ but we will omit this for the sake of readability. Note that the expression defining the behaviour of $\mathsf{sequence}$ when l is non-empty could be written more elegantly using the do-notation present in Haskell (see for instance [Lip11, chapter 12]), but we do not have this in our type system nor in Agda parametric for the (pre)monads we defined in chapter 3.

Remark 4.1. Of course we should verify that the definition of $\mathsf{sequence}$ above is allowed by the type system. For instance, at first sight it is not clear why we can use the return and bind operations of the monad M in the implementation of that function even though M appears parametric in the context at those points. It is for exactly this reason that we wanted a premonad to depend pointwise on its return and bind functions in section 3.2.1.

[4]Actually, this also better reflects the behaviour of the Haskell function `sequence` because in Haskell an instance of the type class `Monad` is not guaranteed to satisfy the monad laws. Moreover a function that works for any premonad can be applied in more cases than a function that works for any monad because every monad is also a premonad. It means however that the implementation of this function cannot use any monad laws.

Indeed, for a pointwise dependent sum type $\Sigma^{¶}(x : A).B$ there is a first projection $\mathsf{fst}^{¶} : \sharp\big(\Sigma^{¶}(x : A).B\big) \to A$ that is parametric. Hence we can write functions of type

$$\forall(M : \mathsf{Premonad}_\ell).\forall(X : \mathcal{U}_\ell).X \to M\,X$$

and

$$\forall(M : \mathsf{Premonad}_\ell).\forall(X, Y : \mathcal{U}_\ell).M\,X \to (X \to M\,Y) \to M\,Y$$

that map a premonad to its return and bind operation respectively and that are both parametric in the premonad. As a consequence, we can say that in some sense the return and bind functions depend parametrically on their premonad. Therefore, the monad M always appears in parametric position in the implementation of $\mathsf{sequence}$ and hence this function is well defined. Note that one could argue that we should write $\mathsf{return}_{M^\sharp}$ and $\gg\!=_{M^\sharp}$ instead of return_M and $\gg\!=_M$ but this would reduce readability.

We now consider another example of an effect polymorphic function, this time involving booleans. Hence we assume that there is a type $\mathsf{Bool} : \mathcal{U}_0$ and values $\mathsf{true} : \mathsf{Bool}$ and $\mathsf{false} : \mathsf{Bool}$ in our type system. Furthermore, we will use an operation $\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_$ which takes a term of type Bool and two other terms of a (possibly other) equal type and which behaves as one would intuitively expect. Formally, we should add inference rules for all these constructs, but we believe that it is clear from the description above how they behave. Alternatively, we could have decided to define the type of booleans using the other types already available as $\top + \top$, but we will not do this.

The effect polymorphic function that we consider, let us call it f, will take for any premonad $M : \mathsf{Premonad}_0$ a list of effectful computations that yield a result of type Bool, so in other words a list of type $\mathrm{List}\,(M\,\mathsf{Bool})$. It will then perform these effectful computations sequentially until it reaches a computation which yields the result false. The computation that f returns will yield the result true if and only if all computations in the input list yield true. In this context, one could interpret the boolean result of a monadic computation as an indicator of the success of that computation. The function f will then try to execute all computations in a list and will report on whether they all succeeded or not. More concretely, f is defined as

$$\begin{aligned}
f := \lambda(M^\sharp : \mathsf{Premonad}_0)\,.\,&\lambda(l : \mathrm{List}\,(M\,\mathsf{Bool}))\,.\\
&\mathsf{ind}_{\mathrm{List}}\big(x.(M\,\mathsf{Bool}),\\
&\qquad \mathsf{return}_M\,\mathsf{true},\\
&\qquad m.l'.w.\ m \gg\!=_M \big(\lambda(b : \mathsf{Bool})\,.\,\mathsf{if}\ b\ \mathsf{then}\ w\ \mathsf{else}\ (\mathsf{return}_M\,\mathsf{false})\big),\\
&\qquad l\big)
\end{aligned} \tag{4.1}$$

and it has as type $\forall(M : \mathsf{Premonad}_0).\,\mathrm{List}\,(M\,\mathsf{Bool}) \to M\,\mathsf{Bool}$. Again remark 4.1 applies in this example.

In this section, we have seen two examples of effect polymorphic functions in the type system ParamDTT. Of course many more examples can be given, but these two make clear how effect polymorphic functions and their types look like in ParamDTT. Essentially, the types always contain a parametric quantification over a premonad. In the rest of this chapter, we will see how the structures available in ParamDTT enable us to exploit this parametric quantification over a premonad and prove interesting theorems about effect polymorphic functions.

4.2 Constructing a Bridge inside Premonad

In this chapter we focus on effect polymorphic functions f which have a type of the form $\forall(M : \mathsf{Premonad}_\ell).A$ where A is a type that depends on the variable M. Generally speaking, effect parametricity results relate the behaviour of f with respect to one premonad to its behaviour with respect to another premonad in the presence of a certain relation between these two premonads, such as a monad morphism.[5] More concretely, suppose that we have two premonads $\kappa_1, \kappa_2 : \mathsf{Premonad}_\ell$ and a monad morphism $h : \mathsf{MonadMorphism}\,\kappa_1\,\kappa_2$. The general idea of an effect parametricity proof in ParamDTT is then to use h to construct a bridge inside the type $\mathsf{Premonad}_\ell$ with endpoints κ_1 and κ_2. In other words, we want to construct a function $\mathsf{pm\text{-}bridge}$ of type $\mathbb{I} \to \mathsf{Premonad}_\ell$ such that $\mathsf{pm\text{-}bridge}\,0 = \kappa_1$ and $\mathsf{pm\text{-}bridge}\,1 = \kappa_2$. We can then apply f to this bridge and since f is parametric in its premonad argument, this bridge will be strengthened to a path from a term of type $A[\kappa_1/M]$ to a term of type $A[\kappa_2/M]$. Further manipulations of this path will eventually yield a homogeneous path and we can then apply the path degeneracy axiom to conclude the proof.

Of course this raises the question how to construct a bridge $\mathsf{pm\text{-}bridge}$ inside the type $\mathsf{Premonad}_\ell$ from κ_1 to κ_2 using the monad morphism h. The idea is to use the fact that a premonad is a 4-tuple and to construct bridges and paths between the corresponding components of κ_1 and κ_2. More concretely, we will first construct a bridge $\mathsf{type\text{-}op\text{-}bridge}$ in $\mathcal{U}_\ell \to \mathcal{U}_\ell$ between the type operators of κ_1 and κ_2. For this purpose, we must give for any $i : \mathbb{I}$ and any $X : \mathcal{U}_\ell$ a type in the universe $\mathcal{U}_\ell$ that is equal to $\kappa_1\,X$ if $i = 0$ and to $\kappa_2\,X$ if $i = 1$. This can be done by considering the function $h\,X^\sharp : \kappa_1\,X \to \kappa_2\,X$ and using the graph relation former $/\cdot\backslash$. Indeed, we have that $/h\,X^\sharp\backslash$ is a bridge in $\mathcal{U}_\ell$ from $\kappa_1\,X$ to $\kappa_2\,X$ and hence we define

$$\mathsf{type\text{-}op\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\lambda(X : \mathcal{U}_\ell)\,.\,/h\,X^\sharp\backslash\,i \quad : \quad \mathbb{I} \to \mathcal{U}_\ell \to \mathcal{U}_\ell. \tag{4.2}$$

We see that indeed $\mathsf{type\text{-}op\text{-}bridge}\,0$ is the type operator of κ_1 and $\mathsf{type\text{-}op\text{-}bridge}\,1$ is the type operator of κ_2.

Second, we need for any $i : \mathbb{I}$ a function of type $\forall(X : \mathcal{U}_\ell).X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,X$ that can be used as the return function of $\mathsf{pm\text{-}bridge}\,i$. The construction of this function is possible using the return function for κ_1 and the function $\mathsf{push}\,h$. More concretely, for any $X : \mathcal{U}_\ell$ we have $\mathsf{return}_{\kappa_1}\,X^\sharp : X \to \kappa_1\,X$ and $\mathsf{push}\,(h\,X^\sharp)\,i^\sharp : \kappa_1\,X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,X$ and hence we can consider their composition

$$\begin{aligned}\mathsf{return\text{-}path}\,i^\sharp \quad := \quad \lambda(X : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,\mathsf{push}\,(h\,X^\sharp)\,i^\sharp\,(\mathsf{return}_{\kappa_1}\,x) :& \\ \forall(X : \mathcal{U}_\ell).X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,X.&\end{aligned} \tag{4.3}$$

Notice that in the implementation of $\mathsf{return\text{-}path}$ the variable i is used only in a parametric position and therefore this function can indeed depend parametrically on i (as was implied by writing $\mathsf{return\text{-}path}\,i^\sharp$ instead of $\mathsf{return\text{-}path}\,i$). This means that the function $\mathsf{return\text{-}path}$ has the type

$$\forall(i : \mathbb{I}).\forall(X : \mathcal{U}_\ell).X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,X$$

and hence it really is a path (and not a bridge) from $\mathsf{return}_{\kappa_1}$ to some other function. Moreover, since $\mathsf{push}\,(h\,X^\sharp)\,1 = h\,X^\sharp$, this other function maps any $x : X$ to $h\,(\mathsf{return}_{\kappa_1}\,x)$

[5]Note that in this sentence the words "relate" and "relation" are informally used and do not refer to mathematical relations as in section 2.1, although the procedure described in [Voi09] works in this way.

$$\begin{tikzcd}
\mathsf{pm\text{-}bridge}\,0 \arrow[d, dash, dashed] & = & (& \kappa_1, \arrow[d, dash, dashed] & \mathsf{return}_{\kappa_1}, \arrow[d, dash] & \gg\!=_{\kappa_1}, \arrow[d, dash] & \mathsf{tt} &) \\
\mathsf{pm\text{-}bridge}\,i \arrow[d, dash, dashed] & = & (& \mathsf{type\text{-}op\text{-}bridge}\,i, \arrow[d, dash, dashed] & \mathsf{return\text{-}path}\,i^\sharp, \arrow[d, dash] & \mathsf{bind\text{-}path}\,i^\sharp, \arrow[d, dash] & \mathsf{tt} &) \\
\mathsf{pm\text{-}bridge}\,1 & = & (& \kappa_2, & \mathsf{return}_{\kappa_2}, & \gg\!=_{\kappa_2}, & \mathsf{tt} &)
\end{tikzcd}$$

Figure 4.1: Construction of the bridge $\mathsf{pm\text{-}bridge}$ inside the type $\mathsf{Premonad}_\ell$. The top row gives the endpoint for $i = 0$, the middle row is the situation for general i and the bottom row gives the endpoint for $i = 1$. Dashed lines represent bridges and solid lines represent paths.

which is equal to $\mathsf{return}_{\kappa_2}\, x$ because h is a monad morphism.[6] As a conclusion $\mathsf{return\text{-}path}$ is a path from $\mathsf{return}_{\kappa_1}$ to $\mathsf{return}_{\kappa_2}$.

Third, we need to consider the bind operation for $\mathsf{pm\text{-}bridge}\,i$ and hence we have to construct for any $i : \mathbb{I}$ a function of type

$$\forall (X, Y : \mathcal{U}_\ell).\mathsf{type\text{-}op\text{-}bridge}\,i\,X \to (X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,Y) \to \mathsf{type\text{-}op\text{-}bridge}\,i\,Y.$$

At first glance, it is not evident how to do this and it turns out we cannot construct such a function using only push and pull. We will work out the details for this construction in the next sections, gradually going from simpler to more difficult cases. For now, we assume that we have a path

$$\begin{aligned}\mathsf{bind\text{-}path} : \forall (i : \mathbb{I}).\forall (X, Y : \mathcal{U}_\ell).\mathsf{type\text{-}op\text{-}bridge}\,i\,X \to (X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,Y)\\ \to \mathsf{type\text{-}op\text{-}bridge}\,i\,Y\end{aligned}$$

from $\gg\!=_{\kappa_1}$ to $\gg\!=_{\kappa_2}$.

Now we are ready to define the bridge $\mathsf{pm\text{-}bridge}$ as

$$\begin{aligned}\mathsf{pm\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\big(\mathsf{type\text{-}op\text{-}bridge}\,i,\ \mathsf{return\text{-}path}\,i^\sharp,\ \mathsf{bind\text{-}path}\,i^\sharp,\ \mathsf{tt}\big) :\\ \mathbb{I} \to \mathsf{Premonad}_\ell.\end{aligned}$$

See also Figure 4.1 for an illustration of this situation.

Notice that it is important that $\mathsf{return\text{-}path}$ and $\mathsf{bind\text{-}path}$ are paths (and hence their argument i is in parametric position). Indeed, we know from definition 3.1 that a premonad depends pointwise on its return and bind operations. Hence the implementations of these operations are type checked in the current context left-divided by the modality ¶, which means that i is only available for parametric use at these points. We can also look

[6]This equality of $h\,(\mathsf{return}_{\kappa_1}\, x)$ and $\mathsf{return}_{\kappa_2}\, x$ is propositional and not definitional because of the definition of a monad morphism. We will ignore this important detail in this section for simplicity but we will come back to it in the following sections.

at this situation in the following more intuitive way. As a premonad depends pointwise on its return operation, a bridge between different return functions will be broken when going to the premonad level and hence we need a (stronger) path between the return functions in order to get some relational structure between the premonads. The same is true for the bind operations.

Another remark is that the bridge pm-bridge is actually not a bridge from κ_1 to κ_2 but from a premonad which has the same type operator, return and bind functions as κ_1 to a premonad which has the same type operator, return and bind functions as κ_2. Indeed, although intuitively the unit type $\top$ only has one inhabitant, it is not possible to derive that any two values of type $\top$ are definitionally equal (because we do not have an η-rule for the unit type). Hence, the last components of κ_1 and κ_2 are not necessarily definitionally equal to tt. However, this will not pose a problem because the value of type $\top$ in a premonad is irrelevant and will therefore not be used anywhere in the implementation of an effect polymorphic function or in a proof of an effect parametricity result. Hence we will be able to show that if an effect parametricity result holds for pm-bridge 0 and pm-bridge 1, then it also holds for κ_1 and κ_2. We refer to the following sections for more details.

At this point we can also say something more about why we want an effect polymorphic function to take a premonad as its argument instead of a monad. In the latter case, our effect polymorphic functions would have a type of the form $\forall(M : \mathsf{Premonad}_\ell).\forall(\widetilde{M} : \mathsf{IsMonad}\, M).A$ for some type A depending on M and $\widetilde{M}$. If we wanted to prove theorems using effect parametricity for such a function, we would need to construct not only a bridge $\mathsf{pm\text{-}bridge} : \mathbb{I} \to \mathsf{Premonad}_\ell$ from a premonad κ_1 to a premonad κ_2, but also a bridge of type $\forall(i : \mathbb{I}).\, \mathsf{IsMonad}\,(\mathsf{pm\text{-}bridge}\, i)$ from a given term of type $\mathsf{IsMonad}\, \kappa_1$ to a given term of type $\mathsf{IsMonad}\, \kappa_2$. In other words, we would have to build paths between the corresponding proofs of the monad laws for κ_1 and κ_2 but we did not succeed in doing this. However, as illustrated in section 4.1 the implementations of many effect polymorphic functions only require access to the return and bind operations of a premonad and not to proofs that they satisfy the monad laws.

4.3 Purity Preservation

In this section, we are going to use effect parametricity to show for effect polymorphic functions of a certain type that if all of their arguments are pure monadic computations (i.e. computations not causing any side effect), then so is their result. More concretely, we start with a function f of type

$$\forall(M : \mathsf{Premonad}_\ell).M\, A \to M\, A$$

for a given closed type A (so for instance $A = \mathsf{Bool}$ or $A = \mathbb{N}$) and we will prove that if κ is a monad and $m : \kappa\, A$ is pure, then so is $f\, \kappa^\sharp\, m$. In fact, this result even holds if κ is a premonad satisfying only the first monad law involving return. Of course, before stating and proving the theorem we should first clearly define what we mean by a pure monadic computation. We say for any premonad $\kappa : \mathsf{Premonad}_\ell$ that a computation $m : \kappa\, A$ is pure if and only if it can be written as $\mathsf{return}_\kappa\, a$ for some value $a : A$.

Theorem 4.2. *Suppose that Γ is a context containing the following assumptions*

- $A^\sharp : \mathcal{U}_\ell$
- $f : \forall(M : \mathsf{Premonad}_\ell).M\,A \to M\,A$
- $\kappa : \mathsf{Premonad}_\ell$
- $a : A$

and assume that we can derive the judgement

$$¶ \setminus \Gamma,\ X^\sharp : \mathcal{U}_\ell,\ Y^\sharp : \mathcal{U}_\ell,\ x : X,\ q : X \to \kappa\, Y \vdash \mathsf{return}_\kappa\, x \gg\!=_\kappa q \ = \ q\, x : \kappa\, Y. \tag{4.4}$$

In other words, we assume that the first monad law for κ involving return *holds definitionally. Then we can construct in this context Γ a term*

$$\mathsf{thm} : \mathsf{return}_\kappa\,(f\ \mathsf{id\text{-}pm}^\sharp\, a) \equiv_{\kappa A} f\, \kappa^\sharp\,(\mathsf{return}_\kappa\, a).$$

This theorem indeed tells us that f applied to a pure computation $\mathsf{return}_\kappa\, a$ can be written as return_κ applied to some other term and is therefore also pure itself. Recall that id-pm is the identity premonad as defined in example 3.2.

Proof. Throughout this proof we will assume that the graph relation former $/ \cdot \setminus$ is implemented using the Weld type former. We will follow the procedure outlined in section 4.2 to construct a bridge pm-bridge inside $\mathsf{Premonad}_\ell$ using the return operation of κ (recall that we saw in example 3.7 that return_κ is a monad morphism from id-pm to κ)[7]. More concretely, we have a bridge

$$\mathsf{type\text{-}op\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\lambda(X : \mathcal{U}_\ell)\,.\,/\,\mathsf{return}_\kappa\, X^\sharp \setminus i \quad : \quad \mathbb{I} \to \mathcal{U}_\ell \to \mathcal{U}_\ell$$

from $\mathsf{id}_{\mathcal{U}_\ell}$ to the type operator of κ and a path

$$\begin{aligned}\mathsf{return\text{-}path} \quad := \quad & \lambda(i^\sharp : \mathbb{I})\,.\,\lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\mathsf{push}\,(\mathsf{return}_\kappa\, X^\sharp)\, i^\sharp : \\ & \qquad \forall(i : \mathbb{I}).\forall(X : \mathcal{U}_\ell).X \to \mathsf{type\text{-}op\text{-}bridge}\, i\, X\end{aligned}$$

from $\mathsf{return}_{\mathsf{id\text{-}pm}}$ to return_κ. Next, we need to construct the bind operation for the premonad bridge, so we have to specify for any $i : \mathbb{I}$ a function of type

$$\forall(X, Y : \mathcal{U}_\ell).\mathsf{type\text{-}op\text{-}bridge}\, i\, X \to (X \to \mathsf{type\text{-}op\text{-}bridge}\, i\, Y) \to \mathsf{type\text{-}op\text{-}bridge}\, i\, Y.$$

In other words, given $i : \mathbb{I}$, $X, Y : \mathcal{U}_\ell$, $b : /\,\mathsf{return}_\kappa\, X^\sharp \setminus i$ and a function $q : X \to /\,\mathsf{return}_\kappa\, Y^\sharp \setminus i$ we must provide a value of type $/\,\mathsf{return}_\kappa\, Y^\sharp \setminus i$. We will do this by noticing that $/ \cdot \setminus$ is assumed to be implemented using the Weld type former and by applying the induction principle for Weld types to b. More concretely, we know that

$$/\,\mathsf{return}_\kappa\, X^\sharp \setminus i = \mathsf{Weld}\{X \to (i \doteq 0\,?\,X, \mathsf{id}_X \mid i \doteq 1\,?\,\kappa\, X, (\mathsf{return}_\kappa\, X^\sharp))\},$$

and hence in order to use $\mathsf{ind}_{\mathsf{Weld}}$ for b to give a value of type $/\,\mathsf{return}_\kappa\, Y^\sharp \setminus i$, we must specify and verify the following things (see also Figure 4.2).

[7]Actually, we assumed in example 3.7 that κ is a monad. However, if we analyze this example again, we see that we only used the first monad law involving return which we also assume in this theorem.

- First, we need a way to construct a value of type $/\,\mathsf{return}_\kappa\, Y^\sharp\backslash\, i$ given $x : X$. This is easy because we can just take $q\,x$.
- Next, we must give a value of the same type $/\,\mathsf{return}_\kappa\, Y^\sharp\backslash\, i$ under the assumption that the predicate $(i \doteq 0) \vee (i \doteq 1)$ holds. If $i \doteq 0$ holds, then we have that q has type $X \to Y$ and we must produce a value of type Y given $x : X$. Hence we just take $q\,x$. On the other hand, if $i \doteq 1$ holds, then q has type $X \to \kappa\, Y$ and we must give a value of type $\kappa\, Y$ given $b' : \kappa\, X$. This means that we can take the term $b' \gg\!=_\kappa q$.
- Finally, we have to verify that if $i \doteq 0$ holds, then $q\,x$ is definitionally equal to $q\,x$ and if $i \doteq 1$ holds, then $\mathsf{return}_\kappa\, x \gg\!=_\kappa q$ is definitionally equal to $q\,x$. The former equality is trivial and the latter is satisfied because we assumed in (4.4) that the first monad law for κ involving return holds definitionally.[8]

We can then construct a path

$$\begin{aligned}\mathsf{bind\text{-}path} := {} & \lambda(i^\sharp : \mathbb{I})\,.\,\lambda(X^\sharp, Y^\sharp : \mathcal{U}_\ell)\,.\,\lambda(b : \mathsf{type\text{-}op\text{-}bridge}\, i\, X)\,.\\ & \lambda(q : X \to \mathsf{type\text{-}op\text{-}bridge}\, i\, Y)\,.\\ & \mathsf{ind}_{\mathsf{Weld}}\big(\mathsf{type\text{-}op\text{-}bridge}\, i\, Y,\ \ (i \doteq 0\,?\,x.(q\,x) \mid i \doteq 1\,?\,b'.(b' \gg\!=_\kappa q)),\ \ x.(q\,x),\ \ b\big)\end{aligned}$$

from $\gg\!=_{\mathsf{id\text{-}pm}}$ to $\gg\!=_\kappa$.[9] In this way, we get a bridge

$$\mathsf{pm\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\big(\mathsf{type\text{-}op\text{-}bridge}\, i,\ \ \mathsf{return\text{-}path}\, i^\sharp,\ \ \mathsf{bind\text{-}path}\, i^\sharp,\ \mathsf{tt}\big)$$

from $\mathsf{id\text{-}pm}$ to κ just as described in section 4.2. Note that $\mathsf{pm\text{-}bridge}\, 1$ is not exactly κ but κ with the last (irrelevant) component replaced with tt. This problem will be solved at the end of the proof and it will not be important in the meantime because we will never use this trivial component of a premonad. However, when mentioning the endpoints of a path we will sometimes write κ instead of $\mathsf{pm\text{-}bridge}\, 1$ for clarity and readability.

Next, we use push for the function $\mathsf{return}_\kappa\, A^\sharp$ to get a path

$$\mathsf{returna\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{push}\,(\mathsf{return}_\kappa\, A^\sharp)\, i^\sharp\, a \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\, i\, A$$

from a to $\mathsf{return}_\kappa\, a$. Since $\mathsf{type\text{-}op\text{-}bridge}\, i$ is the type operator of the premonad $\mathsf{pm\text{-}bridge}\, i$, we can apply the function f to $\mathsf{pm\text{-}bridge}\, i$ and $\mathsf{returna\text{-}path}\, i^\sharp$ to obtain a path

$$\mathsf{f\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,f\,(\mathsf{pm\text{-}bridge}\, i)^\sharp\,(\mathsf{returna\text{-}path}\, i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\, i\, A$$

from $f\,\mathsf{id\text{-}pm}^\sharp\, a$ to $f\,\kappa^\sharp\,(\mathsf{return}_\kappa\, a)$. This situation is illustrated in Figure 4.3.

Finally, we can use pull applied to $\mathsf{return}_\kappa\, A^\sharp$ to get a path

$$\mathsf{final\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{pull}\,(\mathsf{return}_\kappa\, A^\sharp)\, i^\sharp\,(\mathsf{f\text{-}path}\, i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\kappa\, A$$

[8]We can now explain why the context Γ in assumption (4.4) is left-divided by the modality ¶. This is because the path we are constructing between $\gg\!=_{\mathsf{id\text{-}pm}}$ and $\gg\!=_\kappa$ will be used in pointwise position in the bridge $\mathsf{pm\text{-}bridge}$. Hence everything that contributes to the construction of that path will be type-checked in the context $¶ \setminus \Gamma$ and not in Γ.

[9]Indeed, $\mathsf{bind\text{-}path}$ is a path with $\gg\!=_{\mathsf{id\text{-}pm}}$ and $\gg\!=_\kappa$ as its endpoints because for $i = 0$ or $i = 1$ the result of $\mathsf{ind}_{\mathsf{Weld}}$ is definitionally equal to the values we provided in case the predicate $(i \doteq 0) \vee (i \doteq 1)$ holds and these are exactly applications of the bind operations for $\mathsf{id\text{-}pm}$ and κ.

$$\begin{tikzcd}
X \arrow[r, "\mathsf{id}_X"'] \arrow[rr, bend left, "q"] \arrow[d, dash] & X \arrow[r, "q"'] \arrow[d, dashed, dash] & Y \arrow[d, dashed, dash] \\
X \arrow[r, "\mathsf{push}\,(\mathsf{return}_\kappa\, X^\sharp)\, i^\sharp"'] \arrow[rr, bend left, "q"] \arrow[d, dash] & /\,\mathsf{return}_\kappa\, X^\sharp \backslash\, i \arrow[r, dotted] \arrow[d, dashed, dash] & /\,\mathsf{return}_\kappa\, Y^\sharp \backslash\, i \arrow[d, dashed, dash] \\
X \arrow[r, "\mathsf{return}_\kappa\, X^\sharp"'] \arrow[rr, bend left, "q"] & \kappa\, X \arrow[r, "\_ \gg\!=_\kappa q"'] & \kappa\, Y
\end{tikzcd}$$

Figure 4.2: Diagram similar to that in [NVD17, p. 10] illustrating a part of the construction of the path bind-path in the proof of Theorem 4.2. The middle row depicts the situation for general i and reduces to the top row when $i = 0$ and to the bottom row when $i = 1$. Dashed lines represent bridges that were already constructed and the dotted arrow is the function we are constructing using $\mathsf{ind}_{\mathsf{Weld}}$.

$$\begin{tikzcd}
f & \mathsf{id\text{-}pm} \arrow[d, dashed, dash] & a \arrow[d, dash] & & f\ \mathsf{id\text{-}pm}^\sharp\ a \arrow[d, dash] \\
f & \mathsf{pm\text{-}bridge}\, i \arrow[d, dashed, dash] & \mathsf{returna\text{-}path}\, i^\sharp \arrow[d, dash] & \rightsquigarrow & \mathsf{f\text{-}path}\, i^\sharp \arrow[d, dash] \\
f & \kappa & \mathsf{return}_\kappa\, a & & f\ \kappa^\sharp\ (\mathsf{return}_\kappa\, a)
\end{tikzcd}$$

Figure 4.3: Application of f to the bridge pm-bridge and the path returna-path gives rise to the path f-path on the right. Note that for $i = 1$ (bottom row) we write κ instead of pm-bridge 1 even though they are not exactly equal (definitionally).

from $\mathsf{return}_\kappa\,(f\,\mathsf{id\text{-}pm}^\sharp\,a)$ to $f\,\kappa^\sharp\,(\mathsf{return}_\kappa\,a)$. Since this path is homogeneous, we can use the path degeneracy axiom to get a term

$$\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}) : \mathsf{return}_\kappa\,(f\,\mathsf{id\text{-}pm}^\sharp\,a) \equiv_{\kappa A} f\,(\mathsf{pm\text{-}bridge}\,1)^\sharp\,(\mathsf{return}_\kappa\,a).$$

As a last step, we then need to use the fact that any two values of type $\top$ are propositionally equal and replace in this way the last component tt of $\mathsf{pm\text{-}bridge}\,1$ with the trivial component of κ. We refer to the Agda code for the details of this step. Eventually, we get a proof

$$\mathsf{thm} : \mathsf{return}_\kappa\,(f\,\mathsf{id\text{-}pm}^\sharp\,a) \equiv_{\kappa A} f\,\kappa^\sharp\,(\mathsf{return}_\kappa\,a). \qquad \square$$

The next theorem is a generalization of Theorem 4.2. We are now considering an effect polymorphic function f with type

$$f : \forall(M : \mathsf{Premonad}_\ell).F\,(M\,A) \to M\,A$$

for a given closed type A and a given functor F (as defined in section 2.3). An example of such a function is given in equation (4.1) for $A = \mathsf{Bool}$ and F equal to the list functor.

Theorem 4.3. *Let Γ be a context containing the following assumptions*

- $F^\sharp : \mathsf{Functor}_{\ell,\ell}$
- $A^\sharp : \mathcal{U}_\ell$
- $f : \forall(M : \mathsf{Premonad}_\ell).F\,(M\,A) \to M\,A$
- $\kappa : \mathsf{Premonad}_\ell$
- $a_F : F\,A$

and assume that the first monad law for κ involving return *holds definitionally (just as in Theorem 4.2). Then we can construct in this context Γ a term*

$$\mathsf{thm} : \mathsf{return}_\kappa\,(f\,\mathsf{id\text{-}pm}^\sharp\,a_F) \equiv_{\kappa A} f\,\kappa^\sharp\,(F\,(\mathsf{return}_\kappa\,A^\sharp)\,a_F).$$

If we take for F the list functor, this theorem intuitively tells us that if f is applied to a list of pure computations, then its result will also be pure. The specialization of Theorem 4.3 to the list functor is a translation of Theorem 1 from [Voi09] to ParamDTT. Note that Theorem 4.2 is indeed an instance of Theorem 4.3 by taking for F the identity functor.

Proof. Again we assume that the graph relation former is implemented using the Weld type former. The construction of the bridges type-op-bridge and pm-bridge is then exactly the same as in the proof of Theorem 4.2.

Using the action of F on functions and using push applied to $\mathsf{return}_\kappa\,A^\sharp$, we can construct a path

$$\mathsf{a_F\text{-}path} \;:=\; \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{push}\,(\mathsf{return}_\kappa\,A^\sharp))\,a_F \quad : \quad \forall(i : \mathbb{I}).F\,(\mathsf{type\text{-}op\text{-}bridge}\,i\,A)$$

from $F\,\mathsf{id}_A\,a_F$ to $F\,(\mathsf{return}_\kappa\,A^\sharp)\,a_F$. Then we apply the function f to the bridge pm-bridge and the path $\mathsf{a_F\text{-}path}$ to get a path

$$\mathsf{f\text{-}path} \;:=\; \lambda(i^\sharp : \mathbb{I})\,.\,f\,(\mathsf{pm\text{-}bridge}\,i)^\sharp\,(\mathsf{a_F\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\,i\,A$$

from $f\,\textsf{id-pm}^\sharp\,(F\,\textsf{id}_A\,a_F)$ to $f\,\kappa^\sharp\,(F\,(\textsf{return}_\kappa\,A^\sharp)\,a_F)$. Using $\textsf{pull}$ we can then build a path

$$\textsf{final-path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\textsf{pull}\,(\textsf{return}_\kappa\,A^\sharp)\,i^\sharp\,(\textsf{f-path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\kappa\,A$$

from $\textsf{return}_\kappa\,(f\,\textsf{id-pm}^\sharp\,(F\,\textsf{id}_A\,a_F))$ to $f\,\kappa^\sharp\,(F\,(\textsf{return}_\kappa\,A^\sharp)\,a_F)$. Since this final path is homogeneous, the path degeneracy axiom gives us a term

$$\begin{aligned}&\textsf{path-to-eq}\,(\textsf{final-path}) :\\ &\qquad\qquad \textsf{return}_\kappa\,(f\,\textsf{id-pm}^\sharp\,(F\,\textsf{id}_A\,a_F)) \equiv_{\kappa\,A} f\,(\textsf{pm-bridge}\,1)^\sharp\,(F\,(\textsf{return}_\kappa\,A^\sharp)\,a_F).\end{aligned}$$

Just like in the proof of Theorem 4.2 we then replace $\textsf{pm-bridge}\,1$ with κ using the fact that any two values of type $\top$ are propositionally equal. Applying the fact that a functor preserves identity functions finally gives us the result

$$\textsf{thm} : \textsf{return}_\kappa\,(f\,\textsf{id-pm}^\sharp\,a_F) \equiv_{\kappa\,A} f\,\kappa^\sharp\,(F\,(\textsf{return}_\kappa\,A^\sharp)\,a_F). \qquad \square$$

4.4 Value Extraction

The inspiration for the results in this section comes from Theorem 2 of [Voi09], so we consider the extraction of a result or a value out of a monadic computation. More concretely, we suppose that we have a premonad $\kappa : \textsf{Premonad}_\ell$ and a polymorphic function $p : \forall(X : \mathcal{U}_\ell).\kappa\,X \to X$ that maps a computation $m : \kappa\,X$ to its result of type X.[10] Of course, this is not possible for any (pre)monad[11] and even when it is possible, p will have to satisfy some requirements before we can interpret it as a projection of a computation to its result. Indeed, for any $x : X$ the computation $\textsf{return}_\kappa\,x$ yields the result x and hence we want that $p\,X^\sharp\,(\textsf{return}_\kappa\,x)$ equals x.

Just as in the previous section, we first consider effect polymorphic functions f of type

$$f : \forall(M : \textsf{Premonad}_\ell).M\,A \to M\,A$$

for a given closed type A and we wonder whether for any $m : \kappa\,A$ (for a premonad $\kappa : \textsf{Premonad}_\ell$) the result of $p\,(f\,\kappa^\sharp\,m)$ can be derived from only $p\,m$. In other words, we want to know if the result of the computation obtained by applying f to m is determined by the result of the computation m (and we do not have to take any side effects in m into account). This will not be the case for any κ and any p. As Voigtländer points out in [Voi09], we need to require that

$$p\,(m \gg\!\!=_\kappa q) = p\,(q\,(p\,m))$$

for any types $X, Y : \mathcal{U}_\ell$, any $m : \kappa\,X$ and $q : X \to \kappa\,Y$. Together with the requirement above involving $\textsf{return}$, this will ensure that p is a monad morphism from κ to $\textsf{id-pm}$.

[10] Just as with other polymorphic functions in this thesis, we will not explicitly write a type argument of p if this can be inferred from the context.

[11] For instance, if we had such a function p for the Maybe monad, we could apply it to the empty type $\bot$ and the value $\textsf{nothing}$: Maybe $\bot$ to get a value of the empty type. A premonad for which we can construct such a function p is the writer premonad for some magma $(R, \cdot)$ because we can then project a computation $m : X \times R$ to its first component of type X.

Theorem 4.4. *Suppose that Γ is a context with the following assumptions*

- $A^\sharp : \mathcal{U}_\ell$
- $f : \forall(M : \mathsf{Premonad}_\ell).M\,A \to M\,A$
- $\kappa : \mathsf{Premonad}_\ell$
- $p^{\P} : \forall(X : \mathcal{U}_\ell).\kappa\,X \to X$
- $a_\kappa : \kappa\,A$

and assume that we can derive the judgements

$$\P \setminus \Gamma,\ X^\sharp : \mathcal{U}_\ell,\ x : X \vdash p\,(\mathsf{return}_\kappa\,x) = x : X \tag{4.5}$$

and

$$\P \setminus \Gamma,\ X^\sharp : \mathcal{U}_\ell,\ Y^\sharp : \mathcal{U}_\ell,\ x_\kappa : \kappa\,X,\ q : X \to \kappa\,Y \vdash p\,(x_\kappa \gg\!=_\kappa q) = p\,(q\,(p\,x_\kappa)) : Y. \tag{4.6}$$

Then we can construct in Γ a term

$$\mathsf{thm} : p\,(f\,\kappa^\sharp\,a_\kappa) \equiv_A f\,\mathsf{id\text{-}pm}^\sharp\,(p\,a_\kappa).$$

Proof. Throughout this proof, we assume that the graph relation former $/\cdot\setminus$ is implemented using the Glue type former. We will again apply the procedure of section 4.2 to construct a bridge in the type $\mathsf{Premonad}_\ell$ from κ to $\mathsf{id\text{-}pm}$ using the monad morphism p. More concretely, we have a bridge

$$\mathsf{type\text{-}op\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\lambda(X : \mathcal{U}_\ell)\,.\,/p\,X^\sharp\setminus i \quad : \quad \mathbb{I} \to \mathcal{U}_\ell \to \mathcal{U}_\ell$$

from the type operator κ to the type operator $\mathsf{id}_{\mathcal{U}_\ell}$ of $\mathsf{id\text{-}pm}$. Note that it is important that p is parametric in its type argument, because $/\cdot\setminus$ depends pointwise on its function argument and hence X is only available for parametric use at the point it appears in the implementation of $\mathsf{type\text{-}op\text{-}bridge}$. Furthermore, we have a path

$$\begin{aligned}\mathsf{return\text{-}path} \quad := \quad & \lambda(i^\sharp : \mathbb{I})\,.\,\lambda(X^\sharp : \mathcal{U}_\ell)\,.\,\lambda(x : X)\,.\,\mathsf{push}\,(p\,X^\sharp)\,i^\sharp\,(\mathsf{return}_\kappa\,x) : \\ & \qquad\qquad \forall(i : \mathbb{I}).\forall(X : \mathcal{U}_\ell).X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,X\end{aligned}$$

from return_κ to some other function. This other function maps a type $X : \mathcal{U}_\ell$ and a value $x : X$ to $p\,(\mathsf{return}_\kappa\,x)$ and by assumption (4.5) of the theorem, this is definitionally equal to x. Therefore, the endpoint $\mathsf{return\text{-}path}\,1$ is definitionally equal to $\mathsf{return}_{\mathsf{id\text{-}pm}}$, or in other words $\mathsf{return\text{-}path}$ is a path from return_κ to $\mathsf{return}_{\mathsf{id\text{-}pm}}$.[12]

Then we need to construct a path from $\gg\!=_\kappa$ to $\gg\!=_{\mathsf{id\text{-}pm}}$, which is equivalent to specifying for any $i : \mathbb{I}$, $X, Y : \mathcal{U}_\ell$, $b : /(p\,X^\sharp)\setminus i$ and $q : X \to /(p\,Y^\sharp)\setminus i$ a value of type $/(p\,Y^\sharp)\setminus i$. We will do this using the fact that $/\cdot\setminus$ is implemented using Glue types and applying the constructor for such types. More concretely, we know that

$$/(p\,Y^\sharp)\setminus i = \mathsf{Glue}\{Y \leftarrow (i \doteq 0\,?\,\kappa\,Y, p\,Y^\sharp \mid i \doteq 1\,?\,Y, \mathsf{id}_Y)\}$$

and hence constructing an element of $/(p\,Y^\sharp)\setminus i$ using glue amounts to specifying and verifying the following things (see also Figure 4.4).

[12]Similar to footnote 8, the reason why assumption (4.5) contains $\P \setminus \Gamma$ instead of Γ is that it is used in to replace $\mathsf{return\text{-}path}\,1^\sharp$ in the bridge $\mathsf{pm\text{-}bridge}\,1$ (that we still have to construct) with $\mathsf{return}_{\mathsf{id\text{-}pm}}$. Since a premonad depends pointwise on its return function, everything involved with the return operation of $\mathsf{pm\text{-}bridge}$ is type-checked in the context $\P \setminus \Gamma$ instead of Γ. A similar remark applies to assumption (4.6).

- First, we need to give an element of type Y. For this purpose, we apply $\mathsf{pull}\,(p\,X^\sharp)\,i^\sharp$ to b to get a value of type X. Then we can apply the function q to this value and in this way we obtain a value of type $/(p\,Y^\sharp)\backslash\,i$.[13] Finally we can feed this into the function $\mathsf{pull}\,(p\,Y^\sharp)\,i^\sharp$ and obtain a value of type Y. In summary, the value of type Y we constructed is

$$\big((\mathsf{pull}\,(p\,Y^\sharp)\,i^\sharp) \circ q \circ (\mathsf{pull}\,(p\,X^\sharp)\,i^\sharp)\big)\;b.$$

  For easy reference in the rest of this proof, we denote by $\xi_{i^\sharp}$ the function

$$\xi_{i^\sharp} \quad := \quad (\mathsf{pull}\,(p\,Y^\sharp)\,i^\sharp) \circ q \circ (\mathsf{pull}\,(p\,X^\sharp)\,i^\sharp) \quad : \quad /(p\,X^\sharp)\backslash\,i \to Y.$$

- Second, we have to specify a value of type $\kappa\,Y$ assuming that $i \doteq 0$ holds and of type Y assuming that $i \doteq 1$ holds. In case $i \doteq 0$ holds, q has type $X \to \kappa\,Y$ and b has type $\kappa\,X$. Hence we can take $b \gg\!=_\kappa q$. On the other hand, if $i \doteq 1$ holds, then q has type $X \to Y$ and b has type X, so we can take the value $q\,b$.

- Finally, we must verify that if $i \doteq 0$ holds then $p\,(b \gg\!=_\kappa q)$ is definitionally equal to $\xi_{0^\sharp}\,b$ and if $i \doteq 1$ holds, then $q\,b$ is definitionally equal to $\xi_{1^\sharp}\,b$. In the former case, $i \doteq 0$ holds and hence $\xi_{0^\sharp} = (p\,Y^\sharp) \circ q \circ (p\,X^\sharp)$ which means that we must verify that $p\,(q\,(p\,b))$ is definitionally equal to $p\,(b \gg\!=_\kappa q)$ and this follows immediately from assumption (4.6) in the theorem. In the latter case, we have that $i \doteq 1$ holds and hence $\xi_{1^\sharp} = q$ from which we see that the equality we need to verify follows by reflexivity.

In summary, we can construct a path

$$\begin{aligned}\mathsf{bind\text{-}path} := {} & \lambda(i^\sharp : \mathbb{I})\,.\,\lambda(X^\sharp, Y^\sharp : \mathcal{U}_\ell)\,.\,\lambda(b : \mathsf{type\text{-}op\text{-}bridge}\,i\,X)\,.\\ & \lambda(q : X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,Y)\,.\\ & \mathsf{glue}\{\xi_{i^\sharp}\,b \hookleftarrow (i \doteq 0\,?\,b \gg\!=_\kappa q \mid i \doteq 1\,?\,q\,b)\}\end{aligned}$$

from $\gg\!=_\kappa$ to $\gg\!=_{\mathsf{id\text{-}pm}}$. Indeed, the endpoints of $\mathsf{bind\text{-}path}$ are $\gg\!=_\kappa$ and $\gg\!=_{\mathsf{id\text{-}pm}}$ because if $i = 0$, the result of glue is definitionally equal to the value we provided for the case where $i \doteq 0$ holds and in the implementation of $\mathsf{bind\text{-}path}$ this is an application of $\gg\!=_\kappa$. A similar argument applies to the case $i = 1$.

We can then construct a bridge

$$\mathsf{pm\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\big(\mathsf{type\text{-}op\text{-}bridge}\,i,\;\mathsf{return\text{-}path}\,i^\sharp,\;\mathsf{bind\text{-}path}\,i^\sharp,\;\mathsf{tt}\big)$$

from κ to $\mathsf{id\text{-}pm}$. Like in the proof of Theorem 4.2, the endpoint $\mathsf{pm\text{-}bridge}\,0$ is not definitionally equal to κ because its last component tt is not definitionally equal to the last component of κ. Again we will solve this problem at the end of the proof and in the meantime we will sometimes write κ instead of $\mathsf{pm\text{-}bridge}\,0$ when specifying endpoints of paths or bridges.

[13]The attentive reader might wonder why we make things so complicated and do not just take $q\,(\mathsf{pull}\,(p\,X^\sharp)\,i^\sharp\,b)$ as our final result. After all, we are using the glue constructor to create a value of type $/(p\,Y^\sharp)\backslash\,i$. The problem is that for $i = 0$ this will not necessarily be definitionally equal to $b \gg\!=_\kappa q$ and hence this would not result in a path with $\gg\!=_\kappa$ as one of its endpoint.

$$\begin{tikzcd}
\kappa\, X \arrow[r, "\_\gg\!=_\kappa q"'] \arrow[rr, bend left, "\xi_{0^\sharp}"] \arrow[d, dash, dashed] & \kappa\, Y \arrow[r, "p\, Y^\sharp"'] \arrow[d, dash, dashed] & Y \arrow[d, dash] \\
/p\, X^\sharp \backslash\, i \arrow[r, dotted] \arrow[rr, bend left, "\xi_{i^\sharp}"] \arrow[d, dash, dashed] & /p\, Y^\sharp \backslash\, i \arrow[r, "\mathsf{pull}\,(p\, Y^\sharp)\, i^\sharp"'] \arrow[d, dash, dashed] & Y \arrow[d, dash] \\
X \arrow[r, "q"'] \arrow[rr, bend left, "\xi_{1^\sharp}"] & Y \arrow[r, "\mathsf{id}_Y"'] & Y
\end{tikzcd}$$

Figure 4.4: Diagram illustrating a part of the construction of bind-path in the proof of Theorem 4.4. Just as in Figure 4.2, the middle row depicts the situation for general i and reduces to the top row if $i = 0$ and to the bottom row if $i = 1$. Dashed lines represent bridges that are already constructed. The dotted arrow represents the function we are constructing using glue. Note that in the construction of bind-path we apply this function to the value $b : /p\, X^\sharp \backslash\, i$.

We know for any $i : \mathbb{I}$ that $\mathsf{push}\,(p\, A^\sharp)\, i^\sharp\, a_\kappa$ has type $\mathsf{type\text{-}op\text{-}bridge}\, i\, A$ and hence we can apply f to $\mathsf{pm\text{-}bridge}\, i$ and this value to obtain a path

$$\mathsf{f\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\, .\, f\, (\mathsf{pm\text{-}bridge}\, i)^\sharp \left(\mathsf{push}\,(p\, A^\sharp)\, i^\sharp\, a_\kappa\right) \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\, i\, A$$

from $f\, \kappa^\sharp\, a_\kappa$ to $f\, \mathsf{id\text{-}pm}^\sharp\, (p\, a_\kappa)$. Next, we apply $\mathsf{pull}\,(p\, A^\sharp)$ to get a path

$$\mathsf{final\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\, .\, \mathsf{pull}\,(p\, A^\sharp)\, i^\sharp\, (\mathsf{f\text{-}path}\, i^\sharp) \quad : \quad \forall(i : \mathbb{I}).A$$

from $p\, (f\, \kappa^\sharp\, a_\kappa)$ to $f\, \mathsf{id\text{-}pm}^\sharp\, (p\, a_\kappa)$. As this final path is homogeneous, we can apply the path degeneracy axiom to get a term

$$\mathsf{path\text{-}to\text{-}eq}\, (\mathsf{final\text{-}path}) : p\, (f\, (\mathsf{pm\text{-}bridge}\, 0)^\sharp\, a_\kappa) \equiv_A f\, \mathsf{id\text{-}pm}^\sharp\, (p\, a_\kappa).$$

Finally, we use the fact that any two values of the unit type $\top$ are propositionally equal to replace the last component tt in $\mathsf{pm\text{-}bridge}\, 0$ with the trivial component of κ (see the Agda code for details of this step). In this way, we get a proof

$$\mathsf{thm} : p\, (f\, \kappa^\sharp\, a_\kappa) \equiv_A f\, \mathsf{id\text{-}pm}^\sharp\, (p\, a_\kappa). \qquad \square$$

Remark 4.5. In Theorem 4.4, we required that $p\, (\mathsf{return}_\kappa\, x)$ is definitionally equal to x for any $x : X$ because then we can conclude in its proof that $\mathsf{return\text{-}path}\, 1$ is really definitionally equal to $\mathsf{return}_{\mathsf{id\text{-}pm}}$. However, we could wonder if the procedure to replace the last component of $\mathsf{pm\text{-}bridge}\, 0$ with the last component of κ works for the return functions as well. In other words, would it also be sufficient to impose the requirement that $p\, (\mathsf{return}_\kappa\, x)$ and x are propositionally equal – witnessed by some term t – and could we then use this term t at the end of the proof to replace the return function of $\mathsf{pm\text{-}bridge}\, 1$ with $\mathsf{return}_{\mathsf{id\text{-}pm}}$? If we analyze this situation a bit further, we actually want to use the term t to show that $\mathsf{pm\text{-}bridge}\, 1$ and id-pm are propositionally equal, so to construct a term of type $\mathsf{pm\text{-}bridge}\,1 \equiv_{\mathsf{Premonad}_\ell} \mathsf{id\text{-}pm}$. Because all components of these premonads except their return functions are definitionally equal, a first idea to build such a term is to apply the congruence rule of example 1.31 (translated to ParamDTT) to the term t and to a function

$$\lambda(r : \forall(X : \mathcal{U}_\ell).X \to X)\,.\,(\mathsf{id}_{\mathcal{U}_\ell},\ r,\ \gg\!=_{\mathsf{id\text{-}pm}},\ \mathsf{tt}) : (\forall(X : \mathcal{U}_\ell).X \to X) \to \mathsf{Premonad}_\ell.$$

However, we know from Definition 3.1 that a premonad depends pointwise on its return function and hence the above function will not pass type checking (because r becomes available only for parametric use at the point it gets used).

A second idea would then be to make the above function pointwise in its argument r and apply some kind of congruence rule for pointwise functions. This congruence rule would take a function $g : \P A \to B$ and transform a proof of type $a \equiv_A b$ into $g\,a^\P \equiv_B g\,b^\P$. However, we cannot implement such a congruence rule for pointwise functions in ParamDTT in the same way as we did in example 1.31. The problem lies in the fact that the J-rule in ParamDTT (see [NVD17, p. 9]) requires us to give a goal type C that depends parametrically on a variable $y : A$ and with any modality on a variable $w : a \equiv_A y$. As we are constructing a type, the variable y becomes available for continuous use, but this does not let us use it as an argument of a pointwise function because it will then be only available for parametric use again. Hence we cannot take $g\,a^\P \equiv_B g\,y^\P$ for C because it does not type-check. This does not mean that it is impossible to prove that propositional equality is a congruence for pointwise functions in ParamDTT, but the standard and intuitive way to do it does not work, which is in our opinion rather remarkable.

In summary, the technique we apply for replacing the last component of a premonad does not work for the return functions because of the pointwise dependence of a premonad on its return function. As a consequence, we require that $p\,(\mathsf{return}_\kappa\,x)$ and x are definitionally equal in Theorem 4.4. We emphasize that we did not prove that this requirement for definitional equality cannot be weakened to propositional equality, but we did not succeed in doing so.

Just as in section 4.3, we now generalize Theorem 4.4 to effect polymorphic functions f of type

$$f : \forall(M : \mathsf{Premonad}_\ell).F\,(M\,A) \to M\,A$$

for a given closed type $A : \mathcal{U}_\ell$ and functor $F : \mathsf{Functor}_{\ell,\ell}$. The following theorem is a translation of Theorem 2 from [Voi09] to ParamDTT, generalized to hold for any functor instead of only the list functor.

Theorem 4.6. *Suppose that Γ is a context containing the following assumptions*

- $F^\sharp : \mathsf{Functor}_{\ell,\ell}$
- $A^\sharp : \mathcal{U}_\ell$
- $f : \forall(M : \mathsf{Premonad}_\ell).F\,(M\,A) \to M\,A$
- $\kappa : \mathsf{Premonad}_\ell$
- $p^\P : \forall(X : \mathcal{U}_\ell).\kappa\,X \to X$
- $a_{F\kappa} : F\,(\kappa\,A)$

$$\begin{tikzcd}
f & \kappa \arrow[d, no head, dashed] & F\,\mathsf{id}_{\kappa A}\,a_{F\kappa} \arrow[d, no head] & & f\,\kappa^\sharp\,(F\,\mathsf{id}_{\kappa A}\,a_{F\kappa}) \arrow[d, no head] \\
f & \mathsf{pm\text{-}bridge}\,i \arrow[d, no head, dashed] & \mathsf{a}_{\mathsf{F}\kappa}\text{-}\mathsf{path}\,i^\sharp \arrow[d, no head] & \rightsquigarrow & \mathsf{f\text{-}path}\,i^\sharp \arrow[d, no head] \\
f & \mathsf{id\text{-}pm} & F\,(p\,A^\sharp)\,a_{F\kappa} & & f\,\mathsf{id\text{-}pm}^\sharp\,(F\,(p\,A^\sharp)\,a_{F\kappa})
\end{tikzcd}$$

Figure 4.5: Application of f to the bridge $\mathsf{pm\text{-}bridge}$ and the path $\mathsf{a}_{\mathsf{F}\kappa}\text{-}\mathsf{path}$ gives rise to the path $\mathsf{f\text{-}path}$ on the right. This is a part of the proof of Theorem 4.6.

and assume that the judgements from (4.5) *and* (4.6) *can be derived. Then we can construct in this context* Γ *a term*

$$\mathsf{thm} : p\,(f\,\kappa^\sharp\,a_{F\kappa}) \equiv_A f\,\mathsf{id\text{-}pm}^\sharp\,(F\,(p\,A^\sharp)\,a_{F\kappa}).$$

Proof. We assume that the graph relation former $/\cdot\backslash$ is implemented using the Glue type former. The construction of a bridge $\mathsf{pm\text{-}bridge}$ from κ to $\mathsf{id\text{-}pm}$ is exactly the same as in the proof of Theorem 4.4.

Using the action of F on functions applied to $\mathsf{push}\,p$ we obtain a path

$$\mathsf{a}_{\mathsf{F}\kappa}\text{-}\mathsf{path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{push}\,(p\,A^\sharp)\,i^\sharp)\,a_{F\kappa} \quad : \quad \forall(i : \mathbb{I}).F\,(\mathsf{type\text{-}op\text{-}bridge}\,i\,A)$$

from $F\,\mathsf{id}_{\kappa A}\,a_{F\kappa}$ to $F\,(p\,A^\sharp)\,a_{F\kappa}$. Next, we can apply f to the bridge $\mathsf{pm\text{-}bridge}$ and the path $\mathsf{a}_{\mathsf{F}\kappa}\text{-}\mathsf{path}$ to get a path

$$\mathsf{f\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,f\,(\mathsf{pm\text{-}bridge}\,i)^\sharp\,(\mathsf{a}_{\mathsf{F}\kappa}\text{-}\mathsf{path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\,i\,A$$

from $f\,\kappa^\sharp\,(F\,\mathsf{id}_{\kappa A}\,a_{F\kappa})$ to $f\,\mathsf{id\text{-}pm}^\sharp\,(F\,(p\,A^\sharp)\,a_{F\kappa})$ (see also Figure 4.5). Then the use of $\mathsf{pull}\,p$ yields a path

$$\mathsf{final\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{pull}\,(p\,A^\sharp)\,i^\sharp\,(\mathsf{f\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).A$$

from $p\,(f\,\kappa^\sharp\,(F\,\mathsf{id}_{\kappa A}\,a_{F\kappa}))$ to $f\,\mathsf{id\text{-}pm}^\sharp\,(F\,(p\,A^\sharp)\,a_{F\kappa})$. Since this final path is homogeneous, we can apply the path degeneracy axiom and get

$$\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}) : p\,(f\,(\mathsf{pm\text{-}bridge}\,0)^\sharp\,(F\,\mathsf{id}_{\kappa A}\,a_{F\kappa})) \equiv_A f\,\mathsf{id\text{-}pm}^\sharp\,(F\,(p\,A^\sharp)\,a_{F\kappa}).$$

Applying the fact that a functor preserves identity functions and replacing the last component of $\mathsf{pm\text{-}bridge}\,0$ with that of κ gives us the final result

$$\mathsf{thm} : p\,(f\,\kappa^\sharp\,a_{F\kappa}) \equiv_A f\,\mathsf{id\text{-}pm}^\sharp\,(F\,(p\,A^\sharp)\,a_{F\kappa}). \qquad \square$$

4.5 Effect Parametricity and Monad Morphisms

In this section, we will consider the interaction of an effect polymorphic function with an arbitrary monad morphism. Inspiration for this material comes from sections 4.3 and 4.4 of [Voi09]. We also refer to that article for background on how the results proved in the current section (and more generally in this chapter) can be used for reasoning about the behaviour of effect polymorphic functions with respect to specific monads.[14] Just as in the previous sections, we start with an effect polymorphic function f of type

$$f : \forall(M : \mathsf{Premonad}_\ell).M\,A \to M\,A$$

for a given closed type A.

Theorem 4.7. *Let Γ be a context containing the following assumptions*

- $A^\sharp : \mathcal{U}_\ell$
- $f : \forall(M : \mathsf{Premonad}_\ell).M\,A \to M\,A$
- $\kappa_1, \kappa_2 : \mathsf{Premonad}_\ell$
- $h^\P : \mathsf{MonadMorphism}\,\kappa_1\,\kappa_2$
- $a_{\kappa_1} : \kappa_1\,A$

and assume that we can derive the judgements[15]

$$\P \setminus \Gamma,\ X^\sharp : \mathcal{U}_\ell,\ x : X \vdash h\,(\mathsf{return}_{\kappa_1}\,x) = \mathsf{return}_{\kappa_2}\,x : \kappa_2\,X \tag{4.7}$$

and

$$\begin{aligned}\P \setminus \Gamma,\ X^\sharp : \mathcal{U}_\ell,\ Y^\sharp : \mathcal{U}_\ell,\ x_{\kappa_1} : \kappa_1\,X,\ q : X \to \kappa_1\,Y \vdash&\\ h\,(x_{\kappa_1} \gg\!\!=_{\kappa_1} q) = (h\,x_{\kappa_1}) \gg\!\!=_{\kappa_2} (h \circ q) : \kappa_2\,Y.&\end{aligned} \tag{4.8}$$

In other words, we assume that the requirements for h to be a monad morphism hold definitionally. Then we can construct in the context Γ a term

$$\mathsf{thm} : h\,(f\,\kappa_1^\sharp\,a_{\kappa_1}) \equiv_{\kappa_2 A} f\,\kappa_2^\sharp\,(h\,a_{\kappa_1}).$$

This theorem states intuitively that a monad morphism h will in some sense commute with the effect polymorphic function f.

Proof. Throughout this proof, we assume that the graph relation former $/ \cdot \backslash$ is implemented using the Glue type former. We can now exactly follow the procedure outlined in section 4.2 to construct a bridge in $\mathsf{Premonad}_\ell$ from κ_1 to κ_2. More concretely, we have a bridge $\mathsf{type\text{-}op\text{-}bridge} : \mathbb{I} \to \mathcal{U}_\ell \to \mathcal{U}_\ell$ from the type operator of κ_1 to the type operator of κ_2 as defined in equation (4.2) and a path $\mathsf{return\text{-}path} : \forall(i : \mathbb{I}).\forall(X : \mathcal{U}_\ell).X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,X$ from $\mathsf{return}_{\kappa_1}$ to $\mathsf{return}_{\kappa_2}$ as defined in equation (4.3). Notice that below

[14]Note however that the reasoning in [Voi09] is metatheoretic and does not involve dependent types.

[15]The reason why in these judgements the context Γ is left-divided by the modality $\P$ is the same as in footnotes 8 and 12.

equation (4.3) we made the remark that $h\,(\mathsf{return}_{\kappa_1}\,x)$ needs to be definitionally equal to $\mathsf{return}_{\kappa_2}\,x$ for any $x : X$ in order to conclude that $\mathsf{return\text{-}path}\,1^\sharp$ is definitionally equal to $\mathsf{return}_{\kappa_2}$. This requirement is now fulfilled because of the assumption (4.7) in the theorem.

Next, we need to construct a path from $\gg\!=_{\kappa_1}$ to $\gg\!=_{\kappa_2}$. Hence we must specify for any $i : \mathbb{I}$, $X, Y : \mathcal{U}_\ell$, $b : /(h\,X^\sharp)\backslash\,i$ and $q : X \to /(h\,Y^\sharp)\backslash\,i$ a value of type $/(h\,Y^\sharp)\backslash\,i$. The procedure for this is very similar to that of the proof of Theorem 4.4. We now have that

$$/(h\,Y^\sharp)\backslash\,i = \mathsf{Glue}\{\kappa_2\,Y \leftarrow (i \doteq 0\,?\,\kappa_1\,Y, h\,Y^\sharp \mid i \doteq 1\,?\,\kappa_2\,Y, \mathsf{id}_{\kappa_2 Y})\}$$

and we will use the constructor glue to create a value of this type. For this purpose, we must specify and verify the following things (see also Figure 4.6).

- First, we must give a value of type $\kappa_2\,Y$. We notice that $\mathsf{pull}\,(h\,Y^\sharp)\,i^\sharp$ has type $/(h\,Y^\sharp)\backslash\,i \to \kappa_2 Y$ and hence $\big(\mathsf{pull}\,(h\,Y^\sharp)\,i^\sharp\big) \circ q$ will have type $X \to \kappa_2\,Y$. In order to construct a value of type $\kappa_2\,Y$ we could then apply the bind operation of κ_2 to this function and to a value of type $\kappa_2\,X$. Now we notice that $\mathsf{pull}\,(h\,X^\sharp)\,i^\sharp\,b$ has type $\kappa_2\,X$ and hence our value of type $\kappa_2\,Y$ will be given by

  $$\mathsf{pull}\,(h\,X^\sharp)\,i^\sharp\,b \gg\!=_{\kappa_2} \big(\mathsf{pull}\,(h\,Y^\sharp)\,i^\sharp\big) \circ q.$$

  For easy reference in the rest of this proof, we denote by $\xi_{i^\sharp}$ the function

  $$\xi_{i^\sharp} \quad := \quad \lambda(b' : /(h\,X^\sharp)\backslash\,i)\,.\,\mathsf{pull}\,(h\,X^\sharp)\,i^\sharp\,b' \gg\!=_{\kappa_2} \big(\mathsf{pull}\,(h\,Y^\sharp)\,i^\sharp\big) \circ q : \\ /(h\,X^\sharp)\backslash\,i \to \kappa_2\,Y.$$

- Next, we need to specify a value of type $\kappa_1\,Y$ assuming that $i \doteq 0$ holds and of type $\kappa_2\,Y$ assuming that $i \doteq 1$ holds. If $i \doteq 0$ holds, then q has type $X \to \kappa_1\,Y$ and b has type $\kappa_1\,X$, so that we can take $b \gg\!=_{\kappa_1} q$. On the other hand, if $i \doteq 1$ holds, then we know that q has type $X \to \kappa_2\,Y$ and b has type $\kappa_2\,X$ and hence we can take $b \gg\!=_{\kappa_2} q$.

- Finally, we have to verify that if $i \doteq 0$ holds then $h\,(b \gg\!=_{\kappa_1} q)$ is definitionally equal to $\xi_{0^\sharp}\,b$ and if $i \doteq 1$ holds then $b \gg\!=_{\kappa_2} q$ is definitionally equal to $\xi_{1^\sharp}\,b$. In the former case $i \doteq 0$ holds and hence $\xi_{0^\sharp}\,b$ is equal to $h\,b \gg\!=_{\kappa_2} h \circ q$. Using assumption (4.8) in the theorem, this is indeed definitionally equal to $h\,(b \gg\!=_{\kappa_1} q)$. In the latter case we have that $i \doteq 1$ holds and therefore $\xi_{1^\sharp}\,b$ is equal to $b \gg\!=_{\kappa_2} q$ so that the equality we need to verify follows by reflexivity.

In summary, we construct a path

$$\begin{aligned}\mathsf{bind\text{-}path} := \lambda(i^\sharp : \mathbb{I})\,.\,&\lambda(X^\sharp, Y^\sharp : \mathcal{U}_\ell)\,.\,\lambda(b : \mathsf{type\text{-}op\text{-}bridge}\,i\,X)\,.\\ &\lambda(q : X \to \mathsf{type\text{-}op\text{-}bridge}\,i\,Y)\,.\\ &\mathsf{glue}\{\xi_{i^\sharp}\,b \hookleftarrow (i \doteq 0\,?\,b \gg\!=_{\kappa_1} q \mid i \doteq 1\,?\,b \gg\!=_{\kappa_2} q)\}\end{aligned}$$

from $\gg\!=_{\kappa_1}$ to $\gg\!=_{\kappa_2}$ and we use this to build a bridge

$$\mathsf{pm\text{-}bridge} \quad := \quad \lambda(i : \mathbb{I})\,.\,\big(\mathsf{type\text{-}op\text{-}bridge}\,i,\ \mathsf{return\text{-}path}\,i^\sharp,\ \mathsf{bind\text{-}path}\,i^\sharp,\ \mathsf{tt}\big)$$

from κ_1 to κ_2. Just as in the proofs of the previous theorems, the endpoints of $\mathsf{pm\text{-}bridge}$ are not exactly κ_1 and κ_2 because of their last components, but we solve this problem at

$$\begin{tikzcd}
\kappa_1\, X \arrow[r, "\_ \gg=_{\kappa_1} q"'] \arrow[rr, bend left, "\xi_{0^\sharp}"] \arrow[d, dash, dashed] & \kappa_1\, Y \arrow[r, "h\, Y^\sharp"'] \arrow[d, dash, dashed] & \kappa_2\, Y \arrow[d, dash] \\
/h\, X^\sharp\backslash\, i \arrow[r, dotted] \arrow[rr, bend left, "\xi_{i^\sharp}"] \arrow[d, dash, dashed] & /h\, Y^\sharp\backslash\, i \arrow[r, "\mathsf{pull}\,(h\, Y^\sharp)\, i^\sharp"'] \arrow[d, dash, dashed] & \kappa_2\, Y \arrow[d, dash] \\
\kappa_2\, X \arrow[r, "\_ \gg=_{\kappa_2} q"'] \arrow[rr, bend left, "\xi_{1^\sharp}"] & \kappa_2\, Y \arrow[r, "\mathsf{id}_{\kappa_2 Y}"'] & \kappa_2\, Y
\end{tikzcd}$$

Figure 4.6: Diagram illustrating a part of the construction of bind-path in the proof of Theorem 4.7. Just as in Figure 4.4, the middle row depicts the situation for general i and reduces to the top row if $i = 0$ and to the bottom row if $i = 1$. Dashed lines represent bridges that are already constructed. The dotted arrow represents the function we are constructing using glue. Note that in the construction of bind-path we apply this function to the value $b : /h\, X^\sharp\backslash\, i$.

the end of the proof and in the meantime we write κ_1 and κ_2 in the endpoints of bridges and paths.

Using push applied to h we can now build a path

$$\mathsf{a}_{\kappa_1}\text{-}\mathsf{path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{push}\,(h\,A^\sharp)\,i^\sharp\,a_{\kappa_1} \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\,i\,A$$

from a_{κ_1} to $h\,a_{\kappa_1}$. Then we apply the effect polymorphic function f to the bridge pm-bridge and this path to obtain a path

$$\mathsf{f\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,f\,(\mathsf{pm\text{-}bridge}\,i)^\sharp\,(\mathsf{a}_{\kappa_1}\text{-}\mathsf{path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\,i\,A$$

from $f\,\kappa_1^\sharp\,a_{\kappa_1}$ to $f\,\kappa_2^\sharp\,(h\,a_{\kappa_1})$. Finally, application of $\mathsf{pull}\,h$ yields the path

$$\mathsf{final\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{pull}\,(h\,A^\sharp)\,i^\sharp\,(\mathsf{f\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\kappa_2\,A$$

from $h\,(f\,\kappa_1^\sharp\,a_{\kappa_1})$ to $f\,\kappa_2^\sharp\,(h\,a_{\kappa_1})$. Since this final path is homogeneous, we apply the path degeneracy axiom and get a term

$$\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}) : h\,(f\,(\mathsf{pm\text{-}bridge}\,0)^\sharp\,a_{\kappa_1}) \equiv_{\kappa_2 A} f\,(\mathsf{pm\text{-}bridge}\,1)^\sharp\,(h\,a_{\kappa_1}).$$

Replacing the last components of $\mathsf{pm\text{-}bridge}\,0$ and $\mathsf{pm\text{-}bridge}\,1$ with those of κ_1 and κ_2 respectively, gives us the result

$$\mathsf{thm} : h\,(f\,\kappa_1^\sharp\,a_{\kappa_1}) \equiv_{\kappa_2 A} f\,\kappa_2^\sharp\,(h\,a_{\kappa_1}). \qquad \square$$

Remark 4.8. We can see that Theorem 4.2 is in fact an instance of Theorem 4.7 by taking for κ_1 the identity premonad id-pm, for κ_2 the premonad κ and for h the return operation return_κ (which is a monad morphism because of example 3.7). Similarly, also Theorem 4.4 is an instance of Theorem 4.7 by taking for κ_1 the premonad κ, for κ_2 the identity premonad id-pm and for h the function p (equipped with the necessary proofs that this is a monad morphism, which follows from the assumptions (4.5) and (4.6)).

We now again generalize the previous result to effect polymorphic functions f with type

$$f : \forall(M : \mathsf{Premonad}_\ell).F\,(M\,A) \to M\,A$$

for a given closed type A and a given functor F. The following theorem is a translation to ParamDTT of the core idea of Theorems 3 and 4 from [Voi09] (it is in fact equation (4) in the proof of Theorem 3 from that article), generalized to work for arbitrary functors instead of only the list functor.

Theorem 4.9. *Let Γ be a context containing the following assumptions*

- $F^\sharp : \mathsf{Functor}_{\ell,\ell}$
- $A^\sharp : \mathcal{U}_\ell$
- $f : \forall(M : \mathsf{Premonad}_\ell).F\,(M\,A) \to M\,A$
- $\kappa_1, \kappa_2 : \mathsf{Premonad}_\ell$
- $h^\P : \mathsf{MonadMorphism}\,\kappa_1\,\kappa_2$
- $a_{F\kappa_1} : F\,(\kappa_1\,A)$

and assume that the judgements from (4.7) *and* (4.8) *can be derived. Then we can construct in this context Γ a term*

$$\mathsf{thm} : h\,(f\,\kappa_1^\sharp\,a_{F\kappa_1}) \equiv_{\kappa_2 A} f\,\kappa_2^\sharp\,(F\,(h\,A^\sharp)\,a_{F\kappa_1}).$$

Proof. We assume that the graph relation former $/ \cdot \backslash$ is implemented using Glue types. The construction of a bridge $\mathsf{pm\text{-}bridge}$ in $\mathsf{Premonad}_\ell$ from κ_1 to κ_2 is then exactly the same as in the proof of Theorem 4.7.

Using $\mathsf{push}\,h$ and the action of F on functions, we can construct a path

$$\mathsf{a_{F\kappa_1}\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{push}\,(h\,A^\sharp)\,i^\sharp)\,a_{F\kappa_1} \quad : \quad \forall(i : \mathbb{I}).F\,(\mathsf{type\text{-}op\text{-}bridge}\,i\,A)$$

from $F\,\mathsf{id}_{\kappa_1 A}\,a_{F\kappa_1}$ to $F\,(h\,A^\sharp)\,a_{F\kappa_1}$. Next, we apply f to the bridge $\mathsf{pm\text{-}bridge}$ and the path we just constructed to obtain a path (see also Figure 4.7)

$$\mathsf{f\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,f\,(\mathsf{pm\text{-}bridge}\,i)^\sharp\,(\mathsf{a_{F\kappa_1}\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\,i\,A$$

from $f\,\kappa_1^\sharp\,(F\,\mathsf{id}_{\kappa_1 A}\,a_{F\kappa_1})$ to $f\,\kappa_2^\sharp\,(F\,(h\,A^\sharp)\,a_{F\kappa_1})$. Applying $\mathsf{pull}\,h$ then yields a path

$$\mathsf{final\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{pull}\,(h\,A^\sharp)\,i^\sharp\,(\mathsf{f\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\kappa_2\,A$$

from $h\,(f\,\kappa_1^\sharp\,(F\,\mathsf{id}_{\kappa_1 A}\,a_{F\kappa_1}))$ to $f\,\kappa_2^\sharp\,(F\,(h\,A^\sharp)\,a_{F\kappa_1})$. Finally, this path is homogeneous so that the path degeneracy axiom gives us a term

$$\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}) :$$
$$h\,(f\,(\mathsf{pm\text{-}bridge}\,0)^\sharp\,(F\,\mathsf{id}_{\kappa_1 A}\,a_{F\kappa_1})) \equiv_{\kappa_2 A} f\,(\mathsf{pm\text{-}bridge}\,1)^\sharp\,(F\,(h\,A^\sharp)\,a_{F\kappa_1}).$$

Replacing the last components of $\mathsf{pm\text{-}bridge}\,0$ and $\mathsf{pm\text{-}bridge}\,1$ with those of κ_1 and κ_2 respectively, and using the fact that a functor preserves identity functions yields the final result

$$\mathsf{thm} : h\,(f\,\kappa_1^\sharp\,a_{F\kappa_1}) \equiv_{\kappa_2 A} f\,\kappa_2^\sharp\,(F\,(h\,A^\sharp)\,a_{F\kappa_1}). \qquad \square$$

Remark 4.10. Similarly as in remark 4.8, we can see that Theorems 4.3 and 4.6 are instances of Theorem 4.9 by taking specific premonads for κ_1 and κ_2 and a specific monad morphism for h.

$$\begin{tikzcd}
f & \kappa_1 \arrow[d, dashed, dash] & F\,\mathsf{id}_{\kappa_1 A}\,a_{F\kappa_1} \arrow[d, dash] & & f\,\kappa_1^\sharp\,(F\,\mathsf{id}_{\kappa_1 A}\,a_{F\kappa_1}) \arrow[d, dash] \\
f & \mathsf{pm\text{-}bridge}\,i \arrow[d, dashed, dash] & \mathsf{a}_{\mathsf{F}\kappa_1}\text{-}\mathsf{path}\,i^\sharp \arrow[d, dash] & \rightsquigarrow & \mathsf{f\text{-}path}\,i^\sharp \arrow[d, dash] \\
f & \kappa_2 & F\,(h\,A^\sharp)\,a_{F\kappa_1} & & f\,\kappa_2^\sharp\,(F\,(h\,A^\sharp)\,a_{F\kappa_1})
\end{tikzcd}$$

Figure 4.7: Diagram similar to that in Figure 4.5 showing the construction of the path $\mathsf{f\text{-}path}$ in the proof of Theorem 4.9. Note that we write κ_1 and κ_2 as the endpoints of $\mathsf{pm\text{-}bridge}$ although this is not definitionally true.

4.6 A Parametricity Result for sequence

As a final result, we will consider a parametricity theorem for effect polymorphic functions that have a type similar to that of $\mathsf{sequence}$ (as defined in section 4.1). More specifically, we assume that we have a function f of type

$$f : \forall(M : \mathsf{Premonad}_\ell).\forall(X : \mathcal{U}_\ell).F\,(M\,X) \to M\,(F\,X)$$

for any given functor $F : \mathsf{Functor}_{\ell,\ell}$. The function $\mathsf{sequence}$ is an example if we take the list functor for F. Notice that the type of f contains two parametric quantifications: one involving a premonad and an ordinary one involving a type in the universe $\mathcal{U}_\ell$. As a consequence, the following theorem is a combination of two parametricity results: one that is similar to the results we have seen so far in this chapter and one that is more like the results from chapter 2. We could have proved both results separately and combine them afterwards, but we choose to illustrate how such a combined statement is proved in ParamDTT.

Recall that in section 3.2.2, we saw that if $\kappa : \mathsf{Premonad}_\ell$ is a monad, then we can construct a functor whose type operator is the same as that of κ and whose action on functions is given by postcomposing a function with return_κ and feeding this into $\gg\!=_\kappa$. Moreover, in the implementation of this action on functions, we never used the fact that κ is a monad and hence we can apply it for any premonad. From now on, we will write this action applied to a function $g : A \to B$ as $\kappa\,g : \kappa\,A \to \kappa\,B$. Notice that $\kappa\,\mathsf{id}_A$ is not necessarily $\mathsf{id}_{\kappa A}$ if κ is not a monad and hence we do not necessarily obtain a functor. If κ is a monad (witnessed by $\widetilde{\kappa} : \mathsf{IsMonad}\,\kappa$), a proof that this procedure preserves identity functions is given by $\mathsf{return\text{-}law2}\,\widetilde{\kappa}$ and hence in this case we do get a functor.

The following theorem is Theorem 5 from [Voi09], translated to ParamDTT and generalized to work for any functor instead of only the list functor.

Theorem 4.11. *Suppose that Γ is a context containing the following assumptions*

- $F^\sharp : \mathsf{Functor}_{\ell,\ell}$
- $f : \forall(M : \mathsf{Premonad}_\ell).\forall(X : \mathcal{U}_\ell).F\,(M\,X) \to M\,(F\,X)$

- $\kappa_1, \kappa_2 : \mathsf{Premonad}_\ell$
- $\widetilde{\kappa_1} : \mathsf{IsMonad}\,\kappa_1, \;\; \widetilde{\kappa_2} : \mathsf{IsMonad}\,\kappa_2$
- $h^\P : \mathsf{MonadMorphism}\,\kappa_1\,\kappa_2$
- $A^\sharp, B^\sharp : \mathcal{U}_\ell$
- $g^\P : A \to B$
- $a_{F\kappa_1} : F\,(\kappa_1\,A)$

and assume that we can derive the same judgements as in (4.7) *and* (4.8) *so that the laws required for h to be a monad morphism hold definitionally. Then we can construct in the context Γ a term*

$$\mathsf{thm} : \big(\kappa_2\,(F\,g)\big)\,\big(h\,(f\,\kappa_1^\sharp\,a_{F\kappa_1})\big) \equiv_{\kappa_2(FB)} f\,\kappa_2^\sharp\,\big(\big(F\,(\kappa_2\,g)\big)\,(F\,h\,a_{F\kappa_1})\big).$$

Proof. The construction of a bridge $\mathsf{type\text{-}op\text{-}bridge} : \mathbb{I} \to \mathcal{U}_\ell \to \mathcal{U}_\ell$ from the type former of κ_1 to the type former of κ_2 and of a bridge $\mathsf{pm\text{-}bridge} : \mathbb{I} \to \mathsf{Premonad}_\ell$ from κ_1 to κ_2 is exactly the same as in the proof of Theorem 4.7.

We can now construct a path

$$\mathsf{ha_{F\kappa_1}\text{-}path} \;:=\; \lambda(i^\sharp : \mathbb{I})\,.\,F\,(\mathsf{push}\,(h\,A^\sharp)\,i^\sharp)\,a_{F\kappa_1} \;:\; \forall(i : \mathbb{I}).F\,(\mathsf{type\text{-}op\text{-}bridge}\,i\,A)$$

from $F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1}$ to $F\,h\,a_{F\kappa_1}$. Next, we know for any $i : \mathbb{I}$ that $\mathsf{push}\,g\,i^\sharp$ is a function from A to $/g\backslash\,i$. Hence, we can apply the action of the premonad $\mathsf{pm\text{-}bridge}\,i$ on functions to $\mathsf{push}\,g\,i^\sharp$ and in this way we obtain a function $(\mathsf{pm\text{-}bridge}\,i)(\mathsf{push}\,g\,i^\sharp)$ from $\mathsf{type\text{-}op\text{-}bridge}\,i\,A$ to $\mathsf{type\text{-}op\text{-}bridge}\,i\,(/g\backslash\,i)$. Applying then the functor F gives us the path

$$\begin{aligned}\mathsf{gha_{F\kappa_1}\text{-}path} \;:=\; \lambda(i^\sharp : \mathbb{I})\,.\,F\,\big((\mathsf{pm\text{-}bridge}\,i)(\mathsf{push}\,g\,i^\sharp)\big)\,(\mathsf{ha_{F\kappa_1}\text{-}path}\,i^\sharp) :\\ \forall(i : \mathbb{I}).F\,(\mathsf{type\text{-}op\text{-}bridge}\,i\,(/g\backslash\,i))\end{aligned}$$

from $F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1})$ to $F\,(\kappa_2\,g)\,(F\,h\,a_{F\kappa_1})$. Notice that in the definition of the path $\mathsf{gha_{F\kappa_1}\text{-}path}$ the i in $\mathsf{pm\text{-}bridge}\,i$ is in parametric position although it is not annotated with $\sharp$ (and neither is any structure that contains it). This is because the implementation of the functor-like action of a premonad on functions only makes use of the premonad via its return and bind operations and from remark 4.1 we know that these return and bind functions depend parametrically on their premonad.

Then we can apply f to the bridge $\mathsf{pm\text{-}bridge}$ and the path we just constructed to get a path (see also Figure 4.8)

$$\begin{aligned}\mathsf{f\text{-}path} \;:=\; \lambda(i^\sharp : \mathbb{I})\,.\,f\,(\mathsf{pm\text{-}bridge}\,i)^\sharp\,(\mathsf{gha_{F\kappa_1}\text{-}path}\,i^\sharp) :\\ \forall(i : \mathbb{I}).\mathsf{type\text{-}op\text{-}bridge}\,i\,(F\,(/g\backslash\,i))\end{aligned}$$

from $f\,\kappa_1^\sharp\,\big(F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1})\big)$ to $f\,\kappa_2^\sharp\,\big(\big(F\,(\kappa_2\,g)\big)\,(F\,h\,a_{F\kappa_1})\big)$. The next step is to apply $\mathsf{pull}\,h$ to this, which yields a path

$$\begin{aligned}\mathsf{almost\text{-}final\text{-}path} \;:=\; \lambda(i^\sharp : \mathbb{I})\,.\,\mathsf{pull}\,\big(h\,(F\,(/g\backslash\,i))^\sharp\big)\,i^\sharp\,(\mathsf{f\text{-}path}\,i^\sharp) :\\ \forall(i : \mathbb{I}).\kappa_2\,(F\,(/g\backslash\,i))\end{aligned}$$

$$\begin{tikzcd}
f & \kappa_1 \arrow[d, dash, dashed] & F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1}) \arrow[d, dash] & & f\,\kappa_1^\sharp\,\big(F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1})\big) \arrow[d, dash] \\
f & \mathsf{pm\text{-}bridge}\,i \arrow[d, dash, dashed] & \mathsf{gha}_{\mathsf{F}\kappa_1}\mathsf{\text{-}path}\,i^\sharp \arrow[d, dash] & \rightsquigarrow & \mathsf{f\text{-}path}\,i^\sharp \arrow[d, dash] \\
f & \kappa_2 & F\,(\kappa_2\,g)\,(F\,h\,a_{F\kappa_1}) & & f\,\kappa_2^\sharp\,\big((F\,(\kappa_2\,g))\,(F\,h\,a_{F\kappa_1})\big)
\end{tikzcd}$$

Figure 4.8: Diagram similar to that in Figures 4.5 and 4.7 showing the construction of the path f-path in the proof of Theorem 4.11. Note that again we write κ_1 and κ_2 as the endpoints of pm-bridge although this is not definitionally true.

from $h\,\big(f\,\kappa_1^\sharp\,\big(F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1})\big)\big)$ to $f\,\kappa_2^\sharp\,\big(\big(F\,(\kappa_2\,g)\big)\,(F\,h\,a_{F\kappa_1})\big)$. The final path can be constructed by applying for any $i : \mathbb{I}$ the actions of the functor F and the premonad κ_2 to $\mathsf{pull}\,g\,i^\sharp$ which yields a function of type $\kappa_2\,(F\,(/g\backslash\,i)) \to \kappa_2\,(F\,B)$. As a consequence, we have the path

$$\mathsf{final\text{-}path} \quad := \quad \lambda(i^\sharp : \mathbb{I})\,.\,\kappa_2\,\big(F\,(\mathsf{pull}\,g\,i^\sharp)\big)\,(\mathsf{almost\text{-}final\text{-}path}\,i^\sharp) \quad : \quad \forall(i : \mathbb{I}).\kappa_2\,(F\,B)$$

from

$$\big(\kappa_2\,(F\,g)\big)\Big(h\,\big(f\,\kappa_1^\sharp\,\big(F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1})\big)\big)\Big)$$

to

$$\big(\kappa_2\,(F\,\mathsf{id}_B)\big)\Big(f\,\kappa_2^\sharp\,\big(\big(F\,(\kappa_2\,g)\big)\,(F\,h\,a_{F\kappa_1})\big)\Big).$$

Since this last path is homogeneous, the path degeneracy axiom provides us with a term

$$\begin{aligned}\mathsf{path\text{-}to\text{-}eq}\,(\mathsf{final\text{-}path}) : \big(\kappa_2\,(F\,g)\big)\Big(h\,\big(f\,(\mathsf{pm\text{-}bridge}\,0)^\sharp\,\big(F\,(\kappa_1\,\mathsf{id}_A)\,(F\,\mathsf{id}_{\kappa_1}\,a_{F\kappa_1})\big)\big)\Big) &\equiv_{\kappa_2(FB)} \\ \big(\kappa_2\,(F\,\mathsf{id}_B)\big)\Big(f\,(\mathsf{pm\text{-}bridge}\,1)^\sharp\,\big(\big(F\,(\kappa_2\,g)\big)\,(F\,h\,a_{F\kappa_1})\big)\Big)&.\end{aligned}$$

Finally, we need to replace the last components of $\mathsf{pm\text{-}bridge}\,0$ and $\mathsf{pm\text{-}bridge}\,1$ with those of κ_1 and κ_2 respectively, use the fact that F preserves identity functions and make use of $\mathsf{return\text{-}law2}\,\widetilde{\kappa_1}$ and $\mathsf{return\text{-}law2}\,\widetilde{\kappa_2}$ which prove that the actions of κ_1 and κ_2 on functions also preserve identity functions. This then gives us a term

$$\mathsf{thm} : \big(\kappa_2\,(F\,g)\big)\,\big(h\,(f\,\kappa_1^\sharp\,a_{F\kappa_1})\big) \equiv_{\kappa_2(FB)} f\,\kappa_2^\sharp\,\big(\big(F\,(\kappa_2\,g)\big)\,(F\,h\,a_{F\kappa_1})\big). \qquad \square$$

Remark 4.12. Using the last theorem with the list functor for F, the function sequence for f, the identity premonad id-pm for κ_1, the return operation $\mathsf{return}_{\kappa_2}$ for h and the identity function id_A for g (assuming that $A = B$), we obtain that sequence applied to a list of pure computations is itself pure. Of course, this is not a surprise given the implementation of sequence, but we see that by parametricity, any function of this type will have this property.

Chapter 5

Related Work

In this small chapter, we discuss some related work on parametricity, monads and effect parametricity.

Parametricity According to Reynolds [Rey83], the notions of parametric and ad hoc polymorphism were invented by Strachey in 1967 in a set of lecture notes [Str00]. However, the distinction between these kinds of polymorphism was only described intuitively (a polymorphic function being parametric if it behaves the same way for all types). In order to reason about parametric polymorphism, Reynolds formalized this notion using relations and he stated that parametric polymorphic functions map related arguments to related results [Rey83]. Subsequently, Wadler made use of Reynolds's relational interpretation to describe a procedure to derive from a System F type a theorem that is satisfied by all terms of that type [Wad89]. In his discussion, Wadler used a denotational semantics of System F given by frame models (described in for instance [BMM90]). A more operational point of view on parametricity can be found in [Sko15]. Our treatment of parametricity in section 2.1 has the same operational flavour of this last work, but we stay closer to Wadler's notation.

Parametricity and dependent types For parametricity in dependent type theory, we completely followed the article by Nuyts, Vezzosi and Devriese in which the type system ParamDTT is described [NVD17]. However, this is not the first time the relation between parametricity and dependent types is studied. For instance Bernardy et al. provide a way to translate types in a pure type system (a kind of dependent type system) to their corresponding logical relation and terms of a certain type to a proof that they are related to themselves by the relation corresponding to that type [BJP12]. Based on this work (among others), there are tools for the proof assistant Coq that automatically perform such a translation and enable us to use parametricity results in further reasoning.[1] The consistency of the type system ParamDTT and in particular the path degeneracy axiom has been proved by providing a presheaf model for the system [Nuy17], which was based on [BCM15].

Monads and effects Monads were first introduced in category theory under a variety of different names such as monad, triple, triad or standard construction. See [Vou12]

[1]See for instance `https://github.com/coq-community/paramcoq`.

for a further discussion of monads in category theory and an introduction of some more historical facts. Moggi was the first to use monads in computer science for the mathematical study of semantics of imperative programming languages [Mog91]. Subsequently, the use of monads for modeling side effects in purely functional languages was presented by Wadler [Wad92; Wad95].

Effect Parametricity Voigtländer extended the procedure in [Wad89] and studied parametricity properties for effect polymorphic functions [Voi09]. He provides many examples of the use of such theorems for concrete monads, but a formal development of the relational interpretation that is behind the procedure is not presented. This is for instance developed by Atkey [Atk12] for System Fω (an extension of System F that also supports type operators, operators on type operators, ...) and by Bernardy et al. [BJP12] for pure type systems. The validity of our effect parametricity proofs is justified by the consistency of ParamDTT and hence by the model provided in [Nuy17]. We based ourselves on Voigtländer's article for inspiration on effect parametricity results.

Chapter 6

Conclusion and Further Work

We have come to the end of this thesis. In this final chapter, we will reflect on the results we obtained and relate these to the general aim of the thesis as formulated in the beginning. Recall that our intention was to study how the dependent type system ParamDTT can be used to reason about monads in general and effect parametricity in particular. Furthermore, taking these subjects as specific use-cases, we wanted to evaluate ParmDTT (and its implementation Agda parametric) as a practical tool for reasoning about parametricity and identify possible difficulties that present themselves.

Our general conclusion is that ParamDTT is indeed suitable to reason about effect parametricity. This is demonstrated by the fact that all effect parametricity results from [Voi09] could be translated to and proved within the type system. Furthermore, in section 2.3 and chapter 3 we showed that reasoning about functors and monads in general is also possible in ParamDTT.

However, in the following paragraphs we describe some problems or difficulties that we were confronted with along the way and that somewhat reduce the applicability of our results and the elegance of our proofs.

Definitional equality for Glue and Weld types Proposition 2.11 and all theorems in chapter 4 are of the form "Take a context Γ with the assumptions ... and assume that we can derive the judgements ..., then the result ... holds." The fact that these propositions and theorems require that some judgements involving definitional equality can be derived, is partly caused by our use of the induction principle $\mathsf{ind}_{\mathsf{Weld}}$ for Weld types or the constructor glue for Glue types in their proofs.[1] The inference rules introducing both constructs indeed have a judgement requiring some definitional equality as one of their premises. Our proofs would be more elegant if these premises could be weakened to require only propositional equality. Indeed, in that case it would be for instance sufficient in Theorem 4.7 to require that h is a monad morphism from κ_1 to κ_2 (what we already do) and we would not have to impose that the laws for monad morphisms hold definitionally.

Moreover, the requirement for definitional equalities has an impact on the practical applicability of the formalizations in Agda parametric of the proofs presented in this thesis. More concretely, all formalizations of the theorems containing definitional equality premises are implemented in a module that postulates its arguments rather than taking

[1]This explains not all of the requirements for definitional equality. For instance in Theorem 4.4, the assumption (4.6) is needed because of the glue constructor, but assumption (4.5) is needed for another reason discussed below.

parameters. This is necessary because the definitional equality has to be implemented as a rewrite rule and Agda does not allow us to introduce in a module a rewrite rule that involves that module's parameters. However, this means that when we have a concrete effect polymorphic function, we cannot instantiate the module proving some effect parametricity result with this concrete function and use the result for further reasoning.

Pointwise dependence of parametricity results Most of the parametricity results in ParamDTT that we have seen in this thesis depend pointwise on one or more values or functions. In Theorem 4.4 for example, the term thm we constructed depends pointwise on the function p. This pointwise dependence is caused by the fact that in the proof of that theorem we use p in the graph relation former $/ \cdot \backslash$, which depends pointwise on its function and this is on its turn caused by the pointwise dependence of the constructors weld and glue on one of their arguments. In [NVD17], Nuyts, Vezzosi and Devriese already pointed out that this precludes iterated parametricity in ParamDTT. In our specific case, the pointwise dependence caused problems in chapter 3 when we wanted to prove the equivalence of the bind and join formulations of a monad. As a result, our two definitions of a monad (Definitions 3.3 and 3.9) cannot be proved equivalent in ParamDTT, but this would be possible if the term naturality in Proposition 2.13 depended continuously on the function f instead of pointwise.

Propositional equality and pointwise functions As already pointed out in remark 4.5, we cannot prove that propositional equality is a congruence for pointwise functions in the standard way using the J-rule and we did not find any alternative proof. This makes it for instance impossible to show for a premonad $\kappa : \mathsf{Premonad}_\ell$ and a polymorphic function $g : \forall(X : \mathcal{U}_\ell).X \to \kappa\, X$ which is propositionally equal to return_κ that κ is propositionally equal to the premonad obtained from κ by replacing its return function with g. As a consequence, we imposed in some results extra requirements for definitional equality (such as assumption (4.5) in Theorem 4.4). The loss of applicability this causes for the formalization of our results in Agda parametric has already been discussed in the paragraph about definitional equality for Glue and Weld types.

Moreover, it is probable that if we were able to prove that propositional equality is a congruence with respect to pointwise functions, then it would become possible to change the definition of a monad a little bit and prove the equivalence of both formulations of a monad while retaining all effect parametricity results. More concretely, we would in this case make the bind operation $m \gg\!\!=_\kappa q$ depend pointwise on the function q and additionally introduce some pointwise modalities in the monad laws, which would enable us to use the term naturality from Proposition 2.13 in the proof of the equivalence of both formulations. In a first attempt, we tried to work out this idea, but we found out that we would then need a proof that propositional equality is a congruence for pointwise functions in many places, which is why we left this approach.

Pointwise dependent sums In many definitions, such as Definitions 3.1, 3.3 and 3.6, we used pointwise dependent sums to introduce a pointwise dependence of a certain object on one of its components. If we want to do this for the last (relevant) component of a tuple, we need to introduce an irrelevant last component of the unit type $\top$. However, as a consequence we must show explicitly in the proof of many parametricity results that two premonads which are definitionally equal up to their last (irrelevant) component are propositionally equal, which is somewhat annoying. A possible solution would be to make the last trivial component of type $\top$ irrelevant, which means that it will not be inspected when checking whether to values (e.g. premonads) are equal.

The difficulties described above are not really an indication of a fundamental problem in our approach to prove the parametricity results, but rather of some possible improvements in the type system ParamDTT (and actually the last problem about pointwise dependent sums is rather an inconvenience caused by Agda parametric than by ParamDTT). In fact, Nuyts and Devriese already developed a successor for ParamDTT, called RelDTT, in which all of the problems above can be solved [ND18]. An interesting subject for further research would be to translate our proofs of parametricity results to RelDTT. The reason why we did not yet started doing this, is that there is currently no proof assistant available that implements this type system, but development in this direction is carried out by Andreas Nuyts.[2] We now sketch briefly how each difficulty above can be solved in RelDTT.

- The type system RelDTT supports extension types. These will eliminate the need to impose the definitional equality requirements in effect parametricity results that are now necessary because of the application of $\mathsf{ind}_{\mathsf{Weld}}$ or glue in proofs. More concretely, using extension types, it would be for instance sufficient in Theorem 4.4 that the equality in assumption (4.6) holds propositionally.

- In RelDTT, the weld and glue constructors depend continuously on the argument which is pointwise in ParamDTT. Hence most pointwise dependencies in parametricity results of this thesis could be converted into continuous dependencies.

- The inference rule introducing the J-rule for identity types in RelDTT has some different modalities in its premises than that of ParamDTT. This makes it possible to prove that propositional equality is a congruence for ad hoc functions (which is the name for pointwise functions in RelDTT).

- Finally, RelDTT has support for irrelevance. At first glance it would be interesting to use this for instance in the last (irrelevant) component of a premonad. However, if we tried to do this, we would be faced with the problem that dependent sums can again only be made irrelevant in their first component, which leads us to introducing another component of type $\top$ that is not irrelevant. Hence in a practical proof assistant based on ParamDTT or RelDTT, it would be convenient to have dependent sums that can also depend on their second component with some modality chosen by the user. Of course, one should then first investigate whether there is some interaction with the first component's modality and what would be the implications of such an interaction.

Another interesting subject for future research would be to consider in ParamDTT the Church encoding of a free monad for a functor. Given a functor F, we say that a monad M is the free monad for F if there is a natural transformation $\rho : \forall(X : \mathcal{U}_\ell).F\,X \to M\,X$ such that for any other monad M' and any natural transformation $\sigma : \forall(X : \mathcal{U}_\ell).F\,X \to M'\,X$

[2]See `https://github.com/anuyts/menkar`.

there is a unique monad morphism $h : \mathsf{MonadMorphism}\, M\, M'$ with the property that $\sigma = h \circ \rho$. Free monads can be used in the treatment of so-called algebraic effects. The idea for the Church encoding of a functor $F : \mathsf{Functor}_{\ell,\ell}$ would be to consider the type operator F^* defined as

$$F^*\, X := \forall(M : \mathsf{Premonad}_\ell).\forall(\widetilde{M} : \mathsf{IsMonad}\, M).\big(\forall(Y : \mathcal{U}_\ell).F\, Y \to M\, Y\big) \to M\, X$$

for any $X : \mathcal{U}_\ell$ and then verify that F^* can be equipped with the structure of a monad and that it is indeed the free monad for F. The inspiration for this construction comes from [JO15]. One important remark is that in order to show that F^* gives rise to the free monad, we will probably have to construct not only a bridge inside the type premonad, but also paths between different proofs of the monad laws, and we did not yet succeed in doing so as described at the end of section 4.2.

Bibliography

[Atk12] Robert Atkey. “Relational Parametricity for Higher Kinds”. In: *Computer Science Logic (CSL’12)*. Ed. by Patrick Cégielski and Arnaud Durand. Vol. 16. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012, pp. 46–61. ISBN: 978-3-939897-42-2. DOI: `10.4230/LIPIcs.CSL.2012.46`. URL: `http://drops.dagstuhl.de/opus/volltexte/2012/3663`.

[Bar85] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Rev. ed., 2nd print. Studies in logic and the foundations of mathematics 103. Amsterdam: Elsevier Science Publishers b.v., 1985. ISBN: 0-444-87508-5.

[Bau16] Andrej Bauer. *Hask is not a category*. Aug. 2016. URL: `http://math.andrej.com/2016/08/06/hask-is-not-a-category/` (visited on 05/27/2019).

[BCM15] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. “A Presheaf Model of Parametric Type Theory”. In: *Electronic Notes in Theoretical Computer Science* 319 (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)., pp. 67–82. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2015.12.006`. URL: `http://www.sciencedirect.com/science/article/pii/S1571066115000730`.

[BJP12] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. “Proofs for Free: Parametricity for Dependent Types”. In: *Journal of Functional Programming* 22.2 (Mar. 2012). ISSN: 1469-7653, 0956-7968. DOI: `10.1017/S0956796812000056`.

[BMM90] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. “The Semantics of Second-order Lambda Calculus”. In: *Information and Computation* 85.1 (1990), pp. 76–134. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(90)90044-I`. URL: `http://www.sciencedirect.com/science/article/pii/089054019090044I`.

[Chu36] Alonzo Church. “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363. ISSN: 0002-9327, 1080-6377. URL: `http://www.jstor.org/stable/2371045`.

[DP16] Peter Dybjer and Erik Palmgren. “Intuitionistic Type Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University, 2016. URL: `https://plato.stanford.edu/archives/win2016/entries/type-theory-intuitionistic/`.

[Gir72] Jean-Yves Girard. “Interprétation fonctionelle et élimination des coupures de l’arithmétique d’ordre supérieur”. PhD thesis. Université Paris VII, 1972.

[Gir86] Jean-Yves Girard. "The system F of variable types, fifteen years later". In: *Theoretical Computer Science* 45 (1986), pp. 159–192. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(86)90044-7`. URL: `http://www.sciencedirect.com/science/article/pii/0304397586900447`.

[Hof97] Martin Hofmann. "Syntax and Semantics of Dependent Types". In: *Semantics and Logics of Computation.* Ed. by Andrew M. Pitts and Peter Dybjer. Publications of the Newton Institute. Cambridge University Press, 1997, pp. 79–130. DOI: `10.1017/CBO9780511526619.004`.

[Hur95] Antonius J. C. Hurkens. "A simplification of Girard's paradox". In: *Typed Lambda Calculi and Applications.* Ed. by Mariangiola Dezani-Ciancaglini and Gordon Plotkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 266–278. ISBN: 978-3-540-49178-1.

[JO15] Mauro Jaskelioff and Russell O'Connor. "A Representation Theorem for Second-Order Functionals". In: *Journal of Functional Programming* 25 (2015). ISSN: 0956-7968, 1469-7653. DOI: `10.1017/S0956796815000088`.

[Kle65] Heinrich Kleisli. "Every Standard Construction Is Induced by a Pair of Adjoint Functors". In: *Proceedings of the American Mathematical Society* 16.3 (1965), pp. 544–546. ISSN: 0002-9939, 1088-6826. URL: `http://www.jstor.org/stable/2034693`.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* Apr. 2011. URL: `http://learnyouahaskell.com/chapters` (visited on 05/27/2019).

[Mar82] Per Martin-Löf. "Constructive Mathematics and Computer Programming". In: *Logic, Methodology and Philosophy of Science VI.* Ed. by Laurence Jonathan Cohen et al. Vol. 104. Studies in Logic and the Foundations of Mathematics. Elsevier, 1982, pp. 153–175. DOI: `https://doi.org/10.1016/S0049-237X(09)70189-2`. URL: `http://www.sciencedirect.com/science/article/pii/S0049237X09701892`.

[Mil18] Bartosz Milewski. *Category Theory for Programmers.* version 0.7.0. Apr. 2018. URL: `https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/`.

[Mog91] Eugenio Moggi. "Notions of Computation and Monads". In: *Information and computation* 93.1 (1991), pp. 55–92.

[Nuy17] Andreas Nuyts. *A Model of Parametric Dependent Type Theory in Bridge/Path Cubical Sets.* Tech. rep. 2017. URL: `https://arxiv.org/abs/1706.04383`.

[ND18] Andreas Nuyts and Dominique Devriese. "Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory". In: *Logic in Computer Science.* ACM, 2018. DOI: `10.1145/3209108.3209119`.

[NVD17] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. "Parametric Quantifiers for Dependent Type Theory". In: *Proceedings of the ACM on functional programming* 1.ICFP (Sept. 2017). DOI: `10.1145/3110276`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* Cambridge, Massachusetts: MIT Press, 2002. ISBN: 0-262-16209-1.

[Rey83] John C. Reynolds. “Types, Abstraction, and Parametric Polymorphism”. In: *Information Processing*. North Holland, 1983, pp. 513–523.

[Sko15] Lau Skorstengaard. *An Introduction to Logical Relations*. 2015. URL: `http://cs.au.dk/~lask/main.pdf` (visited on 06/05/2019).

[Str00] Christopher Strachey. “Fundamental Concepts in Programming Languages”. In: *Higher-Order and Symbolic Computation* 13.1 (Apr. 2000), pp. 11–49. ISSN: 1573-0557. DOI: `10.1023/A:1010000313106`. URL: `https://doi.org/10.1023/A:1010000313106`.

[Voi09] Janis Voigtländer. “Free Theorems Involving Type Constructor Classes: Functional Pearl”. In: *International Conference on Functional Programming*. ACM, 2009, pp. 173–184.

[Vou12] Anthony Voutas. *The basic theory of monads and their connection to universal algebra*. 2012. URL: `https://www.semanticscholar.org/paper/The-basic-theory-of-monads-and-their-connection-to-Voutas/98c0ca084f561160b6ddc11c6d64df18b6594b5b`.

[Wad89] Philip Wadler. “Theorems for Free!” In: *Functional Programming Languages and Computer Architecture*. ACM, 1989, pp. 347–359.

[Wad92] Philip Wadler. “The Essence of Functional Programming”. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’92. Albuquerque, New Mexico, USA: ACM, 1992, pp. 1–14. ISBN: 0-89791-453-8. DOI: `10.1145/143165.143169`. URL: `http://doi.acm.org/10.1145/143165.143169`.

[Wad95] Philip Wadler. “Monads for Functional Programming”. In: *Advanced Functional Programming*. Vol. 925. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 24–52.

Index

imec-DistriNet Research Group
Celestijnenlaan 200A
3001 Heverlee, BELGIUM
tel. + 32 16 32 76 40
fax + 32 16 32 79 96
www.kuleuven.be