A Type Theory with Reflection

by

Stefania Dumbrava

Thesis for the Conferral of a Master of Science in Smart Systems

---

Dr. Florian Rabe
Prof. Dr. Michael Kohlhase
(Jacobs University)

Date of Submission:

Contents

Abstract

Reflection is a mechanism central to human intelligence in the large and, in particular, to *reasoning* and to its expression through natural and *formal languages*. Its fundamental role is that it provides the means to talk about concepts and their properties, by enabling the unification of different levels of abstraction.

This thesis emphasizes the importance of reflection, as shown by its multifaceted use in many areas related to mathematics and computer science and presents theoretical and practical aspects related to its introduction as a primitive in a type theoretical framework.

Our proposed solution consists of extending a basic type theory with primitives for the reflection of terms and morphisms. This provides us with inductive and (weakly) co-inductive types, which can be seen as theory models and morphisms, respectively. We elaborate a type theory with reflected terms and outline a corresponding methodology for developing similar reflection extensions for morphisms and signatures.

Also, we present work on enriching the modular, foundation-independent MMT framework with reflection capabilities. To this end, we implement the type theoretical extensions for the reflection of terms and for the reflection of morphisms as separate, orthogonal, plug-in features, which we closely document.

Acknowledgements

The author is very grateful to Dr. Florian Rabe and to Prof. Dr. Michael Kohlhase, for their continuous collaboration, support and guidance, as well as to all members of the KWARC group, for their encouragement, feedback and numerous inspiring discussions.

The original idea for the topic together with the general framework on top of which the corresponding implementation was carried out were provided by Dr. Florian Rabe.

Chapter 1

Introduction

Epistemological inquiry into the nature of mathematics runs like a red thread through the development of modern logic, as it was marked by three main philosophical stances: **platonism** (**logicism**), **formalism** and **intuitionism**.

The quest for an appropriate metalanguage that could serve as a foundation for mathematics traces its roots back to Leibniz's dream in the 17th century. Leibniz envisioned a symbolic language general enough to represent human thought ("characteristica universalis"), yet precise enough to enable computation in a framework mirroring human reasoning ("calculus ratiocinator"), in which proofs could be *formalized* using a fixed set of inference rules and *verified* algorithmically. The project's scope was, nonetheless, too ambitious given the existing limitations in the field of logic, a discipline which had been and would continue to remain stagnated in Aristotelian tradition until Frege's "Begriffsschrift" [**?**].

Driven to realize Leibniz's idea, Frege founded the **logicist** movement through his seminal work on establishing logic as a foundation for mathematics. A precursor of and vastly intertwined with **platonism** during the period, the philosophical standpoint stemming from the two conjectured that mathematical objects and truths exist independently of our universe. In his pursuit to reduce mathematical (specifically, arithmetical) axioms to logical truths, Frege revolutionized the field, by introducing the language of predicate logic. However, the axiomatic system in his formalization of naive set theory was shown to be *inconsistent* by Russell, due to a naive formulation of the *comprehension axiom*, according to which, given any well-defined mathematical property, one can construct the set of the elements that satisfy it. Russell discovered that one could thus define a paradoxical construction of "a set containing all sets that are not members of themselves".

Ramsey distinguished between this kind of "logical" paradoxes and "semantical" ones, known from antiquity to lead to sophisms, such as the Liar's Paradox ("This sentence is false") [**?**]. While the latter class of paradoxes anticipates Gödel's result and poses the difficult problem that natural language is semantically closed, vague and ambiguous, the former can be circumvented by introducing a layering of the universe of discourse. Indeed, Russell solved the issue of "vicious circularity" or *impredicativity* by proposing a hierarchy of types, in which the objects of each type are built exclusively from objects of the lower level types. Relying on this construction, Russell and Whitehead continued the efforts towards a complete formalization of mathematics in their "Principia Mathematica" [**?**],

but their theory was more complex and far less intuitive than those based on set theory. A simplified version of this system was given decades later by Church's simply typed λ-calculus [**?**].

Following Frege and Russell, another approach to solving foundational problems, **formalism**, was proposed by Hilbert [**?**]. According to this philosophical perspective, mathematical objects and truths do not exist, but rather formal systems and derivations in them. Hilbert's program set to establish mathematics on a formal foundation, through a complete set of axioms together with a proof of its *consistency* using arithmetic. This endeavor was, however, demonstrated to be unrealizable by Gödel, whose Second Incompleteness Theorem showed that any consistent deduction system strong enough to prove Peano's postulates of elementary arithmetic cannot prove its own consistency. Gödel's famous proof of the result [**?**] is one of the earliest examples of working with self-reference (see Section 2.1).

A similar idea to that of Russell's ramified type theory is that of the von Neumann class of hereditary sets [**?**]. In the von Neumann, Bernays, Gödel formalization of set theory (NBG), loops are avoided by considering two sorts: sets and *proper classes*, the latter of which has sets as elements, but cannot be contained in other sets or classes. This theory motivated the development of another formal account of set theory (ZF), by Zermelo, Skolem and Fraenkel, which refined the comprehension axiom, allowing sets to be defined by comprehension only by collecting elements of a pre-existing set that satisfies a well-formed property. ZF, optionally extended with the axiom of choice (ZFC), is considered the standard axiomatic theory and one of the most common foundations for mathematics.

The main opposing view to **formalism** and **platonism** was introduced by Brouwer [**?**] and argued towards a ***constructivist*** approach, in which mathematical objects and their properties are creations of the human mind. According to Brouwer, intuitionistic truth means *provability*, although, in an informal sense, relying on *intuition*. In this setting, several constructive set theories and type theories have been proposed as foundations for mathematics.

Arguably among the most prominent axiomatizations realizing ***constructive set theories*** are Myhill's Intuitionistic Zermelo Fraenkel (IZF) [**?**], essentially ZFC without the axiom of choice and the law of the excluded middle, and Aczel's Constructive Zermelo Fraenkel (CZF) [**?**], giving a predicative account of IZF. On the type theory side, a leading account of constructivism is given by Martin-Löf's Intuitionistic Type Theory (ITT) [**?**], which is at the base of proof-systems such as LCF (see Section 2.2), HOL (see Section 2.2) and of the logical framework LF (see Section 2.2). Other important ***constructive type theories***, in the sense of Martin-Löf, are Coquand, Huet and Paulin's Calculus of Inductive Constructions (CIC) [**?**], implemented in the Coq prover (see Section 2.2), and Computational Type Theory (CTT) [**?**], implemented in the Nuprl system (see Section 2.2).

All of the above type theories are generalizations of simply typed lambda calculus, such that a type is inhabited, only if there is a valid mathematical construction for that type. The main type constructors encountered in type theory give rise to finite types, equality types, Π-types or dependent product types (see Section 2.1), Σ-types or dependent sum types, inductive types (see Section 2.1) and universes.

From what has been presented so far, one can see that the foundational accounts of

mathematics can be divided into those based on **set-theory** and those based on **type-theory**. Nonetheless, it is also worth mentioning that recent advances in **category theory** led to the formulation of an alternative basis for mathematics. The central idea here is that of a "topos" [**?**], which can define its own local mathematical framework, essentially a generalization of set-theoretical models.

The problem of formalizing mathematics remained only of theoretical interest until the invention of modern computers. While this development enabled the mechanization of mathematics, adding more ease and reliability to mathematical practice, it also reaffirmed the importance of finding suitable foundations on which large-scale formalization projects can be conducted. Moreover, despite the deliberate efforts to escape circularity we described above, such a separation of the levels of abstraction may not be always desirable in practice. In fact, as we will outline in Section 2.1, there are numerous applications to being capable of manipulating meta-level entities inside the object-level. The question thus becomes how to conflate these levels of generality, ultimately into a single level, while avoiding contradictions. In order to obtain such a self-referential language some form of *reflection* is necessary [**?**].

As we have briefly sketched, the journey to realize Leibniz's dream spanned decades and unraveled the central foundational problem of mathematics, which has been at the heart of research in the fields of mathematics, philosophy, logic and, with the advent of the computer age, automated theorem proving. Moreover, this quest has been notably intertwined with exploring the idea of *reflection* and the challenges it brings forth.

Although *reflection* is a fundamental notion, epitomizing the very essence of human cognition that Artificial Intelligence envisions to emulate, its treatment in the large or even within the same branch, for example in automated theorem proving, is disconcertingly heterogeneous, as we will see in Section 2.2. Hence, there is a real need to establish the conceptual foundations of *reflection* in a uniform manner, powerful enough to unify its different instances.

An ideal setting for this endeavor would be a logic-neutral language, capable of representing logical and mathematical theories, meta-theories and their interrelations. Such a language is represented by the "Module system for the development of Mathematical Theories" (MMT) [**?**], in which *we aim to integrate a type theory with reflection, study its properties and formalize examples from category theory, mathematics and logic.*

This thesis is organized as follows. In Section 2, we introduce reflection, inductive and co-inductive types (see Section 2.1) and then give a brief overview of related work in systems such as LCF, Nuprl, HOL, Coq, Chiron and logical frameworks (LF and MMT)(see Section 2.2).

Next, in Section 3, we present the *basic type theory* we will enrich with reflective features, outlining its syntax and proof theory, together with its semantics and the corresponding soundness check. Section 4 advances the *term reflection* addition to the minimal type theory from the previous section. Along the same lines as in Section 3, we introduce the needed reflection primitives at the syntax and inference level (Section 4.1 and Section 4.2), as well as their interpretation at the semantics level (Section 4.3) along with the a soundness proof (Section 4.4).

We give an outlook towards a *general theory of reflection* in Section 5. Specifically, we sketch and illustrate the syntax and inference rules of *morphism reflection* in Section 5.1. We briefly discuss an approach towards elaborating *signature reflection* in Section

5.2, outlining the emerging symmetries between the presented reflection extension that characterize a *signature-based reflection theory*. Section 6 offers a description of the implementational aspects of constructing MMT-plugins for term reflection (Section 6.1) and morphism reflection (Section 6.2), respectively. We present our conclusions in Section 7.

Chapter 2

Related Work

2.1 State of the Art

Inherently, the notion of reflection permits languages to attain self-referentiality. A common example of this is given by natural language, where one can differentiate the *denotation* of a word from its *connotation*, by using inverted comas or quoting, i.e *What is the meaning of "meaning"?*.

Reflection is prevalent not only in natural language, but also in the theory of formal systems, where it has diverse applications. In Section 2.1, we give a general overview of how reflection occurs and is handled in computer science and mathematics. As will be illustrated in Section 4 and Section 5.1, we single out two forms of reflection, namely that of the terms and of the morphisms out of a language. In the former case we obtain inductive types and, in the latter, weakly coinductive or dependent record types. These are introduced in Section 2.1 and Section 2.1.

Reflection

From a computational perspective, reflection is the ability of a system to manipulate as data a representation of its internal state. Reflective mechanisms enable programs to perform "reflective computation", aimed at solving problems and at returning information about the internal problem domain. Consequently, programs can observe and reason about their state (introspection) and, possibly, modify their execution at runtime as a result (intercession). The practical importance of reflection arises from its various applications in software engineering, where it is used in debugging and performance testing, and in artificial intelligence, where it serves the purpose of enabling the self-optimization, adaptation and control of learning systems [**?**].

In the study of **programming languages**, the use of "computational reflection" originated in Brian Smith's work on developing Lisp [**?**, **?**] as a framework for *extensible programming*, which allows the user to write programs that modify the language itself. He distinguished between *structural reflection*, implemented in 2-Lisp and dealing with the representation of the static structure of a system, and *procedural (behavioral) reflection*, implemented in 3-Lisp and dealing with the system's ongoing activity. The reflective framework of 3-Lisp relies on a "reflective tower" mechanism, i.e a stack of circular meta-

interpreters, each interpreted by the one above it, with the user's program running at ground level. Intuitively, through "reflection" and the inverse operation of "reification", higher-class representation constructs (data) can be thought of as turning into lower-class implementation constructs (programs) and vice-versa. More specifically, data structures of the n^{th} level program can be "reflected" as values of the $(n+1)^{\text{th}}$ level interpreter, allowing the latter to observe its implementation. Conversely, through "reification", data structures of the $(n+1)^{\text{th}}$ level interpreter are made available to the n^{th} level program it is running, allowing the latter to observe its execution.

In **mathematics**, reflection arises naturally, since usually no distinction is made between *syntax* and *semantics*, i.e syntactic expressions are identified with their meaning. Indeed, when iteratively developing theories, working mathematicians define certain mathematical structures, "seal" them, and use instantiations of those structures (**models**) as proper objects, in order to prove theorems about their properties (**model functors**) (see Section 5.1 for examples). However, these theorems often cannot be stated within the framework of the structures themselves and are, essentially, *meta-theorems*.

Two interesting instances of this observation were made by Aiello and Weyrauch in [**?**]. Firstly, they pointed out that the notion of a "finite group" cannot be formalized in the first order language of group theory and needs another meta-level of abstraction, for instance set theory, in order to be axiomatized. Secondly, the "duality principle" in projective geometry provides a nice setting in which one could use "reflection" or "reification" to establish results in a dual theory, by analogy. According to the aforementioned principle, given a projective space of dimension N, any theorem that is valid in a subspace of dimension R is also valid in the complementary subspace of dimension $N-R-1$. To give an example, Pascal's Theorem stating that "for any hexagon inscribable in a conic section, the intersection points of the pairs of opposite sides are collinear" can be connected to Brianchon's Theorem stating that "for any hexagon circumscribed about a conic section, the polygon diagonals are concurrent", by regarding points as the "reflection" of lines and the property of "collinearity" as the "reflection" of concurrency. However, there are also cases where reflection is explicitly used. For example, when performing symbolic computations such as differentiation, integration, linear algebra, matricial and polynomial operations, as well as simplifications (rewriting) of algebraic expressions, one is not interested in the value of the elements, but in their syntactic form. Consequently, in these cases, one actually uses the mathematical elements as data objects, essentially working with their "quotation" (reflection).

In **mathematical logic**, one can regard logical theories as object-level constructs, defined within a meta-language. Through reflection, the meta-level syntax and semantics may be "quoted" (reflected) inside the logic, such that formulas and proofs at the object-level may refer to corresponding entities at the meta-level. As mentioned in the introduction (see Section 1), a famous example of reflection is that used by Gödel to establish the incompleteness of Hilbert's axiomatic system based on Peano Arithmetic (PA). As seen in the preliminaries, this is related to the Liar's Paradox, if one is to replace "this sentence" with a statement G in a theory T and "false" with "not provable, to obtain "G is not provable in T". The key idea behind the proof is to encode metamathematical statements, denoted by formulas and proofs in PA, as natural numbers in the arithmetic object language and to express the provability predicate as a logical formula. Thus, by taking a provable formula Φ in PA, its reflection $\ulcorner\Phi\urcorner$ in $\mathbb{N}$ and the provability predicate

Provable($\lceil\Phi\rceil$), definable as $\exists x.\text{Proof}(x, \lceil\Phi\rceil)$, one can state the following sentence which is true, but undecidable in PA: $PA \vdash \Phi \Leftrightarrow \neg\text{Provable}(\Phi)$. This reflection mechanism is also called *implicit reflection*, since provability is represented implicitly by an existential quantifier, which does not give any specification of the proof.

Gödel's result impacted not only the foundations of mathematics, but also those of **formal verification**. Indeed, if one were to strengthen an axiomatic system with its consistency assertion, via implicit reflection, one would obtain a new axiomatic system, with a different provability predicate and consistency assertion. The operation can be infinitely iterated resulting in a "reflective tower" [**?**] of theories, which cannot be formally verified. Nevertheless, this problem can be circumvented, by following a ***constructivist*** paradigm and defining a form of *explicit reflection*, based on a provability predicate employing a family of explicit proof terms [**?**, **?**]. This form of reflection is implemented in Nuprl (see Section 2.2).

In **automated theorem proving**, one also distinguishes between the *object language* and the *meta-language* [**?**]. In this setting, the object language, be it a logic, a formal language, a deductive system or their interpretation, provides the "ground level" on which proofs are produced. The meta-language, used to talk about the object language, contains the internal representation of the latter. This *large-scale reflection* enables the implementation of decision procedures, together with its correctness properties, directly as functions in the logic of the prover.

Indeed, considering a formula F and its reification into an abstract representation $\lfloor F \rfloor$, a proof of F can be obtained by reflection, through executing the procedure on $\lfloor F \rfloor$ and applying the soundness lemma, as depicted in Figure 2.1. In effect, this allows to transform theorem *proving* and *explicit* proof steps in the object-level theory into *evaluation*/computation and *implicit* proof steps in the meta-level theory.

Given that the meta-language has fewer primitives than the object language, it may be easier/more economical to prove meta-theorems certifying the *provability* of certain results at the meta-level, than to give their proofs explicitly at the object level. Also, through reflection, one can verify the correctness of derived object-level results and can reliably add more efficient, untrusted reasoning principles, thus ensuring the *extensibility* and *scalability* of formalization projects carried out in the system.

Another form of reflection, namely *small-scale reflection*, is used in the SSReflect module for the Coq proof assistant (see Section 2.2). This technique facilitates the local interchangeability of logical and symbolical representations inside a proof, by applying the `reflect` predicate. This allows the user to replace deductive steps by computation steps, taking advantage of the fact that the `bool` inductive type can be used as a reification of the `Prop` type.

Inductive Types

Inductive types are central to Martin-Löf's ***constructive*** type theory. In **set theory**, they correspond to the smallest set consistent with certain formation rules. Similarly, in **type theory**, inductive types can be seen as the closure over the set of the type's basic constructors. They were introduced in λ-calculus by Hagino [**?**] and in dependent type theory by Coquand [**?**].

An inductive type is obtained constructively, through an **inductive** definition, fol-

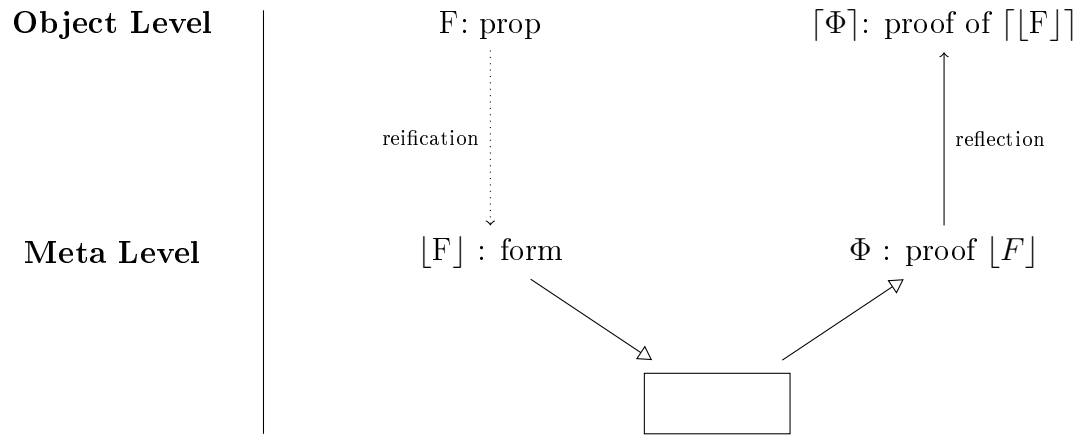


Figure 2.1: Reflection Scheme

lowing a **"bottom-up"** approach (hence the intuition behind our notation $\downarrow S$ in Section 5.2). Essentially, in the beginning of its creation, the only elements in the set of a given inductive type are its "founders" (basic constructors/base cases) and one can then incrementally populate the type, by using its constructors to combine these elements into new ones. These constructors are given by the *introduction forms/rules* and provide a canonical way of forming elements of that type through a finite number of applications. Conversely, through *elimination forms/rules*, objects of the inductive type can be used to define objects of other types.

In **category-theoretical** terms, we can think of inductive types as freely generated by their introductory forms (surjective operators). Also, they form initial objects in the category of algebras for an endofunctor, whose elimination form can be expressed in terms of the universal morphism mapping out of them.

In order to reason about inductive types, we need a way to analyze their inhabitants. To this end, inductive types provide the principles for conducting proofs-by-induction (structural induction) and for defining functions from them, via case analysis over the constructors and (primitive) recursion.

Common examples of inductive types are the natural numbers (see Section 4), pairs, lists, trees, booleans, logical quantifiers and connectives.

Coinductive Types. Dependent Record Types

Coinductive types follow the **set-theory** intuition of defining the largest set consistent with a certain of formation rules. Dual to inductive types, which enable the construction of data structures and *use recursion*, coinductive come with basic operation for destructing values and are usually *defined through recursion*. Examples of coinductive types are streams and infinite lists.

Essentially, coinductive types can be seen as dependent record types. We define a record to be a tuple of labled fields $\langle L_1 = t_1, \cdots, L_n = t_n \rangle$, where L_i are string labels and t_i corresponding assignment, which can in turn be typed by T_i. A dependent record type is given by a record consisting of labeled types $\langle L_1 : T_1, \cdots, L_n : T_n \rangle$, where the types T_i of each label L_i depends on the types of the preceding ones. The operation of performing field selection is called projection and can essentially be seen as a destructor

for the type.

In type theory, dependently-typed records have been studied in the context of developing modular systems and of formalizing mathematical structures, as well as abstract datatypes with their specification.

2.2 State of the Craft

LCF

The LCF system [**?**] is a *logic-independent* interactive proof-checking system implemented in ML and based on the logic of computable functions. The semantics of the logic is given by partially ordered sets with respect to termination properties. The primary focus of the logic design was to allow reasoning about computable functions, in particular about recursion schemes. This was realized by treating recursive definitions as fixed-points of functions and by introducing the principle of *fixed-point induction*.

On the practical side, LCF is regarded as seminal to the technological development of theorem provers, given the functionality it introduced. One of its most important aspects is that it permits users to write their own proof search **tactics** using arbitrary computation, without affecting the soundness of the proof checker, which only depends on a relatively small, isolated kernel of code. To this effect, the system provides high *flexibility* and *extensibility*. Also, more control and proof *modularity* can be achieved via the **refinement** of proof obligations into subgoals.

As a result, almost all proof systems are influenced by LCF, with HOL (Section 2.2) being its direct descendent and with Nuprl (Section 2.2) providing a variation of the language.

Nuprl

Nuprl ("new pearl") [**?**] is a proof-development system for constructive mathematics, based on a variant of Martin-Löf's polymorphic, extensional type theory. Due to this extensionality, type checking is undecidable.

The flexibility of Nuprl's polymorphism and the fact that it does not impose computability of types enables the system to use a rich and open-ended type theory, which has a *predicative* cumulative hierarchy of type universes, dependent function types, subset types, quotient types, parameterized **inductive types** and *record types*. Following Curry Howard's propositions-as-types principle, propositions are interpreted as types inhabited by the computational content of their proofs. This content can then be extracted, thus realizing the *proofs-as-programs* paradigm.

Based on this, *computational reflection* (see Section 2.1) is achieved via an evaluation operator which can perform computation on terms forming either the hypothesis or conclusion of a sequent, replacing them with the result of their evaluation.

HOL

HOL [**?**] denotes a family of interactive theorem provers for higher-order logic, vastly influenced by LCF (see Section 2.2). To this effect, the systems are highly programmable

in ML and can be seen, in this sense, as programming languages in themselves. Also following the LCF-style methodology, HOL systems are implemented as libraries, in which, starting from a trusted kernel of proved theorem, new results can be established through syntactic manipulations, using functions from the library. These functions correspond to inference rules in higher-order logic.

Reflection is realized by implementing inference patterns using "proforma theorems". The intuition behind them is that, through proving general lemmas, one can encode general transformation schemas, such that proof steps can be justified via instantiation. Essentially, the mechanism enables one to reflect syntactic transformation into inference. Taking this further, one could also consider general verification procedures that, once established, can be relied upon, without the need of producing additional proofs.

Coq

Coq [**?**] is an interactive proof management system, whose underlying formal language is the Calculus of Inductive Constructions (CIC), the richest calculus in Barendregt's lambda cube [**?**], supporting polymorphism, type operators and dependent types. Essentially, the type system can be seen as containing only function types and inductive types (see Section 2.1). Unlike Nuprl, Coq's equality is *intensional*, meaning that type checking is decidable. The system in implemented in Objective Caml, a dialect of SML, and comes with an automatic extraction mechanism from Coq proof specifications that can be used to build certified and efficient functional programs. Two types of reflection are supported.

The general, *large-scale* one, enables the translation of propositions from the specification language (Gallina) into values of inductive types that represent syntax and that can be analyzed within the specification language itself, instead of within the Objective Caml meta-language. Through *small-scale* reflection, one can switch from definitional to propositional equality, hence inducing non-trivial computation during the proof checking of logical propositions.

Chiron

Chiron [**?**, **?**] is a general-purpose logic for mechanizing mathematics meant to enable the integration of computer theorem proving and computer algebra systems. The logic is based on a derivative of typed von-Neumann-Bernays-Gödel (NBG) set theory (see Section 1) and supports a mechanisms for dealing with *reflection* (see Section 2.1), which permits reasoning about syntax, and with *partiality*, by allowing undefined terms.

Central to Chiron is the notion of biform theories, which are both axiomatic and algorithmic theories that provide a uniform formalism for combining deduction and computation. On the axiomatic side, it consists of a set of *formulas* and, on the *algorithmic* side, it employs *transformer rules* and *meaning formulas*. While transformers essentially relate the input and output expressions syntactically, via their **quotation**, meaning formulas map between the values of these expressions, via their **evaluation**.

Logical Frameworks

Logical frameworks are a tool for specifying logical systems. They consist of a meta-language and a description of both the class of logics to be represented and of the mechanisms used to this extent. The motivation behind developing such logical frameworks can be identified as two-fold.

At a theoretical level, logical frameworks provide a formal basis for describing logical reasoning, which is an especially valuable insight, given the foundational crisis in mathematics (see Section 1). At a practical level, they constitute a suitable environment for developing reliable and powerful formal verification tools, since they allow for logic-independent proof development.

Logical frameworks can be **set-theoretical**, based on Tarski's view of consequence and model theoretic semantics, or **type-theoretical**, based on the Curry-Howard isomorphism and proof theoretic semantics. The former are exemplified by institutions [?, ?] and General Logics [?], while the latter by Automath [?], Isabelle [?] and LF [?]. An overview is given in [?].

The latter example, LF, is a corner of the lambda-cube [?] obtained by adding dependent types to the typed λ-calculus. The resulting $\lambda\Pi$-calculus consists of simply typed terms, types and kinded type families. Following the Curry-Howard isomorphism, LF represents all object language judgements as types and proofs as terms. Recently, a module system for the Twelf implementation of LF was developed [?]

Another logical framework is that given by the MMT language [?], which provides a foundation-independent scalable module system for the development of mathematical theories, based on the notions of **theories** and **theory morphisms**. Given its high degree of generality, the language supports not only the representation of mathematical and logical structures, but also of their meta-theoretical foundations and of the interrelations between them.

Chapter 3

Basic Type Theory

In building a type theory with reflection, we follow a constructivist approach. Namely, starting from an "empty" type theory, in the sense that it does not contain any type constructors, we iteratively add, as orthogonal features, the reflection of its meta-judgements.

An exhaustive theory of reflection in this setting would have six such reflected meta-judgements: three for signature-based reflection (the well-formedness of reflected signatures and, respectively, that of terms and morphisms with respect to a signature) and three for context-based reflection (the well-formedness of reflected contexts, that of terms with respect to a context and that of substitutions). The scope of this thesis is limited to elaborating the syntax and semantics of a type theory with signature-based reflection of terms.

This basic type theory that we consider in the initial phase is inspired by MMT, due to its generality and foundational unconstrainedness. The language distinguishes between two levels of expressivity: the object-level, inhabited by expressions, and the meta-level, consisting of meta-objects (signatures and contexts) and meta-morphisms between them (signature morphisms and "context morphisms" or substitutions). In Section 4, we will extend this basic type theory with type constructors for reflected terms.

3.1 Syntax

At the *object-level*, this basic type theory contains only simple expressions E, subsuming **constants** c, **variables** x and a distinguished **type** universe `type`. At the *meta-level*, it comprises of **signatures** Σ, **contexts** Γ, (signature) **morphisms** σ and, respectively, (context) **substitutions** γ.

Signatures Σ are taken to be fixed constructs with an absolute semantics, thus forming a closed world, whose inhabitants, typed constants, exist only by virtue of being declared. Due to the fact that, in practice, the size of these signatures is relatively large, they are provided with names S, for ease of multiple reference.

Contexts Γ are defined with respect to a signature and hence have a relative semantics, forming an open world, inhabited by typed variables.

Fixing two signatures Σ and Σ', (signature) morphisms, $\sigma : \Sigma \to \Sigma'$ that map between them, consist of type-preserving assignments of Σ-constants $c : A$ to Σ'-terms $\sigma(c) : \overline{\sigma}(A)$, where $\overline{\sigma}$ is the homomorphic extension of σ to $\Sigma-$terms.

Similarly to morphisms, assuming a signature Σ, (context) substitutions, $\gamma : \Gamma' \to \Gamma''$, map variables x, from a Σ-context Γ', to terms E, in Σ-context Γ''.

Next, we define the application of morphisms and substitutions to well-typed expressions E, contexts Γ' and substitutions γ.

Definition 1. Morphism Application. Considering a well-formed (signature) morphism $\Gamma \vdash \sigma : \Sigma \to \Sigma'$, the application of σ is given by the homomorphic extension $\overline{\sigma}(-)$:

$$\overline{\sigma}(E) ::= \begin{cases} x & \text{if } \Gamma \vdash E = x \\ t & \text{if } \Gamma \vdash E = c \text{ and } c \mapsto t \text{ in } \sigma \\ \texttt{type} & \text{if } \Gamma \vdash E = \texttt{type} \end{cases}$$

In the case of expressions, morphism application is straightforward, since it preserves variables and `type`, replacing only constants with terms assigned to them by the morphism.

$$\overline{\sigma}(\Gamma') ::= \begin{cases} \cdot & \text{if } \Gamma \vdash \Gamma' = \cdot \\ \sigma(\Gamma''), x : \sigma(A) & \text{if } \Gamma \vdash \Gamma' = \Gamma'', x : A \end{cases}$$

For non-trivial contexts, the function is applied recursively to the types of the variables, leaving the latter unchanged.

$$\overline{\sigma}(\gamma) ::= \begin{cases} \cdot & \text{if } \Gamma \vdash \gamma = \cdot \to \cdot \\ \sigma(\gamma'), x/\sigma(t) & \text{if } \Gamma \vdash \gamma = \gamma', x/t : \Gamma' \to \Gamma \end{cases}$$

Morphism application to non-trivial substitutions is carried out analogously.

Definition 2. Substitution Application. Considering a well-formed (context) substitution $\Gamma \vdash_\Sigma \gamma : \Gamma' \to \Gamma''$, where $\Gamma' = x_1 : A_1, ..., x_n : A_n$ and $\gamma = x_1/t_1, ..., x_n/t_n$, the application of γ is given by the homomorphic extension $\overline{\gamma}(-)$:

$$\overline{\gamma}(E) ::= \begin{cases} t_i & \text{if } \Gamma \vdash E = x_i \\ c & \text{if } \Gamma \vdash E = c \\ \texttt{type} & \text{if } \Gamma \vdash E = \texttt{type} \end{cases}$$

Substitution application to expressions preserves constants and `type`, replacing variables with the terms assigned to them by the substitution.

$$\overline{\gamma}(\Gamma') ::= \begin{cases} \cdot & \text{if } \Gamma \vdash \Gamma' = \cdot \\ \gamma(\Gamma''), x : \gamma(A) & \text{if } \Gamma \vdash \Gamma' = \Gamma'', x : A \end{cases}$$

$$\overline{\gamma}(\gamma') ::= \begin{cases} \cdot & \text{if } \Gamma \vdash_\Sigma \gamma' = \cdot \to \cdot \\ \gamma(\gamma''), \gamma(t) & \text{if } \Gamma \vdash_\Sigma \gamma' = \gamma'', t : \Gamma'' \to \Gamma' \end{cases}$$

We present an overview of the basic syntax in Table 3.1.

3.2 Proof Theory

To delimitate the meaningful syntactic constructs of our language, we give corresponding well-formedness judgments for its primitives. These rules are defined inductively, as follows.

	Grammar	Typing Judgment	Equality Judgment
Signatures (Σ)	$\Sigma \quad ::= \quad \cdot \mid \Sigma, c : E$	$\vdash \Sigma \ \ \texttt{Sig}$	$\vdash \Sigma = \Sigma'$
Morphisms (σ)	$\sigma \quad ::= \quad \cdot \mid \sigma, c/E$	$\Gamma \vdash \sigma : \Sigma \to \Sigma'$	$\Gamma \vdash \sigma = \sigma' : \Sigma \to \Sigma'$
Contexts (Γ)	$\Gamma \quad ::= \quad \cdot \mid \Gamma, x : E$	$\Gamma \vdash_\Sigma \Gamma' \ \texttt{Ctx}$	$\Gamma \vdash_\Sigma \Gamma' = \Gamma''$
Substitutions (γ)	$\gamma \quad ::= \quad \cdot \mid \gamma,\ x/E$	$\Gamma \vdash_\Sigma \gamma : \Gamma' \to \Gamma''$	$\Gamma \vdash_\Sigma \gamma = \gamma' : \Gamma' \to \Gamma''$
Expressions (E)	$E \quad ::= \quad c \mid x \mid \texttt{type}$	$\vdash_\Sigma E : E'$	$\vdash_\Sigma E = E'$

Table 3.1: Grammar, Typing and Equality Judgments for Basic Type Theory.

For signatures, the base case *sigempty* consists of the well-formed empty signature $\vdash \cdot \ \texttt{Sig}$. According to *sig*, once we have a well-formed signature Σ, we can extend it to a well-formed signature $\Sigma, c : A$, by adding the corresponding declaration for a fresh constant c, whose type is well-formed with respect to Σ.

Next, given a well-formed signature Σ, the empty morphism, mapping from the empty signature to Σ, is well-formed (*morempty*). A morphism $\sigma : \Sigma' \to \Sigma$ can be iteratively extended by adding the assignment $c \mapsto t$ of a constant $c : A$ declared in Σ' to a term t in Σ. The resulting morphism $\sigma, c \mapsto t$ maps from the extended signature $\Sigma', c : A$ to Σ. This extended morphism is well-formed (*mor*), if the initial morphism σ is well-formed and if the term t is well-formed over Σ and has the type $\sigma(A)$, obtained by translating the type of c along σ.

In the trivial case (*ctxempty*), the empty context, $\vdash_\Sigma \cdot$ over a well-formed signature Σ, is well-formed. A context Γ can be extended with a variable declaration $x : A$. According to *ctx*, the enlarged context $\Gamma, x : A$ is well-formed, if the initial context is well-formed and if the type of the variable is well-formed over Σ, in context Γ.

The well-formedness rules for substitutions, *subempty* and *sub*, are analogous to those for morphism. Lastly, constant and variable declarations over a signature Σ, in context Γ, are well-formed, if Γ is well-formed over Σ.

Equality is defined component-wise and the corresponding rules are straightforward. For morphisms, in the base case, the empty morphisms mapping to the same well-formed signature Σ are trivially equal. In the step case, by extending equal morphisms $\sigma_1 = \sigma_2$ with assignments $c \mapsto t_1$ and, respectively, $c \mapsto t_2$, equality is preserved, if the terms are equal, i.e $t_1 = t_2$. The case for substitution equality is analogous.

As stated in the rules *crefl*, *vrefl* and *trefl*, equality of atomic expressions in *reflexive*. Also, it is *symmetric* and *transitive*, thus forming an *equivalence relation* over the set of all terms definable by the grammar.

A summary of the typing and equality rules for basic type theory is given in Table 3.2 and Table 3.3, respectively.

$$\frac{}{\vdash \cdot \ \texttt{Sig}}\ sigempty \qquad \frac{\vdash \Sigma\ \texttt{Sig} \quad c\ \texttt{not in}\ \Sigma \quad \cdot \vdash_\Sigma A : \texttt{type}}{\vdash \Sigma, c : A\ \texttt{Sig}}\ sig$$

$$\frac{\vdash \Sigma\ \texttt{Sig}}{\vdash \cdot : \cdot \to \Sigma}\ morempty \qquad \frac{\vdash \sigma : \Sigma' \to \Sigma \quad \vdash_\Sigma t : \sigma(A)}{\vdash \sigma, c \mapsto t : \Sigma', c : A \to \Sigma}\ mor$$

$$\frac{\vdash \Sigma\ \texttt{Sig}}{\vdash_\Sigma \cdot\ \texttt{Ctx}}\ ctxempty \qquad \frac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad \Gamma \vdash_\Sigma A : \texttt{type}}{\vdash_\Sigma \Gamma, x : A\ \texttt{Ctx}}\ ctx$$

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\vdash_\Sigma \cdot : \cdot \to \Gamma}\ subempty \qquad \frac{\vdash_\Sigma \gamma : \Sigma' \to \Sigma \quad \Gamma' \vdash_\Sigma A : \texttt{type} \quad \Gamma \vdash_\Sigma t : \gamma(A)}{\vdash_\Sigma \gamma, t : \Gamma', x : A \to \Gamma}\ sub$$

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad c : E\ \texttt{in}\ \Sigma}{\Gamma \vdash_\Sigma c : E}\ con \qquad \frac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad x : E\ \texttt{in}\ \Gamma}{\Gamma \vdash_\Sigma x : E}\ var$$

Table 3.2: Inference System for Basic Type Theory

$$\frac{\vdash \Sigma\ \texttt{Sig}}{\vdash \cdot = \cdot : \cdot \to \Sigma}\ emptymoreq \qquad \frac{\vdash \sigma_1 = \sigma_2 : \Sigma' \to \Sigma \quad \vdash_\Sigma t_1 = t_2}{\vdash \sigma_1, c \mapsto t_1 = \sigma_2, c \mapsto t_2 : \Sigma', c : A \to \Sigma}\ moreq$$

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\vdash \cdot = \cdot : \cdot \to \Gamma}\ emptysubeq \qquad \frac{\vdash_\Sigma \gamma_1 = \gamma_2 : \Gamma' \to \Gamma \quad \Gamma \vdash_\Sigma t_1 = t_2}{\vdash_\Sigma \gamma_1, t_1 = \gamma_2, t_2 : \Gamma', x : A \to \Gamma}\ subeq$$

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma c = c}\ crefl \qquad \frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma x = x}\ vrefl \qquad \frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma \texttt{type} = \texttt{type}}\ trefl$$

$$\frac{\Gamma \vdash_\Sigma E = E'}{\Gamma \vdash_\Sigma E' = E}\ sym \qquad \frac{\Gamma \vdash_\Sigma E = E' \quad \Gamma \vdash_\Sigma E' = E''}{\Gamma \vdash_\Sigma E = E''}\ trans$$

$$\frac{\Gamma \vdash_\Sigma E_1 = E_1' \quad \Gamma \vdash_\Sigma E_2 = E_2' \quad \Gamma \vdash_\Sigma E_1 = E_2}{\Gamma \vdash_\Sigma E_1' : E_2'}\ congr$$

Table 3.3: Equality Rules for Basic Type Theory

3.3 Semantics

Starting from the type theory $\mathbb{T}$, whose syntax is outlined in 2.1, we set to define its set-theoretic semantics, with respect to a **model structure** M and to a **variable assignment function** α, and to prove its *soundness* (see Section4.4).

Definition 3. Model Structure. A **model structure** $M = \langle \mathtt{Set}, [\![-]\!] \rangle$ consists of a universe (domain of individuals) Set, which is the class of sets, and of an interpretation function $[\![-]\!] : \mathbb{T} \to \mathtt{Set}$, which is an assignment of semantic values to every syntactic entity in $\mathbb{T}$ (see Table 3.4).

Definition 4. Variable Assignment. Given a well-formed context Γ, a **variable assignment** function $\alpha : \Gamma \to \mathtt{Set}$ is defined extensionally as a set of tuples pairing the arguments and the values of the function, as follows:

$$\alpha = \begin{cases} \emptyset & \text{if } \Gamma = \cdot \\ \alpha_0 \cup \{(x, u)\}, \text{where } \alpha_0 = \text{assignment for } \Gamma_0 \text{ and } u \in [\![A]\!]^M & \text{if } \Gamma = \Gamma_0, x : A \end{cases}$$

Signatures are interpreted as sets and, hence, given a particular signature Σ, the set $[\![\Sigma]\!]$ of its models belongs to the category `Class`, whose objects are classes. Intuitively, a model of a signature consists of a set of tuples, where each tuple contains a declared constant together with an interpretation of its type. Looking at the production rules, given a signature Σ and a model $M \in [\![\Sigma]\!]$, the interpretation of the extended signature $\Sigma, c : A$ is formed by adding the element (c, u) to the set M, where u is the interpretation of A with respect to M. At the base case, the interpretation of the empty signature is the class containing just the empty set.

The semantics of signature morphisms is given by functors that are *contravariant* with respect to the interpretations of the domain and target signatures. A model of a signature morphism is a function that takes a variable assignment as input and returns a set of tuples, where each tuple contains a declared constant from the domain signature together with an interpretation of the expression assigned to it by the morphism. Specifically, given a morphism σ and a model $\alpha \mapsto [\![\sigma]\!](\alpha)$, the interpretation of the extended morphism $\sigma, c \mapsto E$ is formed by extending the function model with $(c, [\![\Gamma' \vdash E]\!])$ to the function $\alpha \mapsto ([\![\sigma]\!](\alpha) \cup \{(c, [\![\Gamma' \vdash E]\!])\})$.

The interpretation of contexts and substitutions is analogous to that of signatures and, respectively, of signature morphisms. The only difference is that, since contexts and substitutions are well-formed with respect to a given signature, their semantics is also depends on a fixed signature model.

Indeed, let us consider a well-formed context Γ with respect to a signature Σ. The interpretation of Γ, under a fixed Σ-model M, is a class specified by a set of variable assignment functions that contain ordered pairs of variables and members of their type's semantics. Starting from a context Γ and a Γ-model α, the interpretation of the extended context $\Gamma, x : A$ is constructed by adding the pair (x, u) to the set α, where u is a translation of the type A.

For substitutions, as in the case of morphisms, the interpretation function is a *contravariant* functor translating from the interpretation of the target signature to that of the domain signature. The interpretation of the extended substitution $\gamma, x/t$ is obtained,

as before, by adding to the set a tuple consisting of the added variable and the interpretation of its corresponding term, with respect to the signature model and variable assignment.

Semantically, for a fixed model and variable assignment, the well-typing of a term in context translates to the inhabitation of its interpreted type. In particular, following the production rules for expressions, we have that the interpretation of constants and variables is obtained through direct application of the model and, respectively, of the variable assignment. The type `type` is interpreted as `Set`.

Syntax	Semantics
$\vdash \Sigma$ `Sig`	$[\![\Sigma]\!] \in$ `Class`
$\vdash \sigma : \Sigma \to \Sigma'$	$[\![\sigma]\!] : [\![\Sigma']\!] \to [\![\Sigma]\!]$
$\Sigma ::= \cdot \mid \Sigma, c : A$	$[\![\cdot]\!] = \{\emptyset\} \mid [\![\Sigma, c : A]\!] = \{M \cup \{(c, u)\} \mid M \in [\![\Sigma]\!], u \in [\![\Gamma \vdash_\Sigma A]\!]^M\}$
$\sigma ::= \cdot \mid \sigma, c \mapsto \mathrm{E}$	$[\![\cdot]\!] : \emptyset \mapsto \emptyset \mid [\![\sigma, c \mapsto \mathrm{E}]\!] : \alpha \mapsto ([\![\sigma]\!](\alpha) \cup \{(c, [\![\Gamma \vdash_{\Sigma'} E]\!])\})$
For a fixed model $M \in [\![\Sigma]\!]$:	
$\vdash_\Sigma \Gamma$ `Ctx`	$[\![\Gamma]\!]^M \in$ `Set`
$\vdash_\Sigma \gamma : \Gamma \to \Gamma'$	$[\![\gamma]\!]^M : [\![\Gamma']\!]^M \to [\![\Gamma]\!]^M$
$\Gamma ::= \cdot \mid \Gamma, x : A$	$[\![\cdot]\!]^M = \{\emptyset\} \mid [\![\Gamma, x : A]\!]^M = \{\alpha \cup \{(x, u)\} \mid \alpha \in [\![\Gamma]\!]^M, u \in [\![\Gamma \vdash_\Sigma A]\!]^M\}$
$\gamma ::= \cdot \mid \gamma, x/t$	$[\![\cdot]\!]^M : \emptyset \mapsto \emptyset \mid [\![\gamma, x/t]\!]^M : \alpha \mapsto ([\![\gamma]\!]^M(\alpha) \cup \{(x, [\![\Gamma' \vdash_\Sigma t]\!]^{M,\alpha})\})$
For a fixed assignment $\alpha \in [\![\Gamma]\!]^M$:	
$\Gamma \vdash_\Sigma t : A$	$[\![\Gamma \vdash_\Sigma t]\!]^{M,\alpha} \in [\![\Gamma \vdash_\Sigma A]\!]^{M,\alpha}$
$E ::= c \mid x \mid$ `type`	$[\![\Gamma \vdash_\Sigma c]\!]^{M,\alpha} = M(c) \mid [\![\Gamma \vdash_\Sigma x]\!]^{M,\alpha} = \alpha(x) \mid [\![\Gamma \vdash_\Sigma$ `type`$]\!]^{M,\alpha} =$ `Set`

Table 3.4: Interpretation Function

A summary of the corresponding set-theoretic semantics is given in Table 4.4.

3.4 Soundness

Lemma 3.4.1. *Soundness Lemma: Let us assume a well-formed signature Σ, a fixed model $M \in [\![\Sigma]\!]$, a well-formed context Γ and a fixed variable assignment $\alpha \in [\![\Gamma]\!]^M$.*

It follows that, if $\Gamma \vdash_\Sigma t = t'$, then $[\![\Gamma \vdash t]\!]^{M,\alpha} = [\![\Gamma \vdash t']\!]^{M,\alpha}$.

Proof. We proceed by trivial induction on the derivation $\Gamma \vdash_\Sigma t = t'$ and we only have the trivial base cases to consider, as given in Table 3.2.

- case 1: $t = c$, $t' = c$. Then, $[\![\Gamma \vdash t]\!]^{M,\alpha} = M(c)$ and $[\![\Gamma \vdash t']\!]^{M,\alpha} = M(c)$, hence $[\![\Gamma \vdash t]\!]^{M,\alpha} = [\![\Gamma \vdash t']\!]^{M,\alpha}$.

- case 2: $t = x$, $t' = x$. Then, $[\![\Gamma \vdash t]\!]^{M,\alpha} = \alpha(x)$ and $[\![\Gamma \vdash t']\!]^{M,\alpha} = \alpha(x)$, hence $[\![\Gamma \vdash t]\!]^{M,\alpha} = [\![\Gamma \vdash t']\!]^{M,\alpha}$.
- case 3: $t = \texttt{type}$, $t' = \texttt{type}$. Then, $[\![\Gamma \vdash t]\!]^{M,\alpha} = \text{Set}$ and $[\![\Gamma \vdash t']\!]^{M,\alpha} = \text{Set}$, hence $[\![\Gamma \vdash t]\!]^{M,\alpha} = [\![\Gamma \vdash t']\!]^{M,\alpha}$.

□

Lemma 3.4.2. *Substitution Lemma: Let us assume a well-formed signature Σ, a fixed model $M \in [\![\Sigma]\!]$, well-formed contexts Γ, Γ', a Σ-substitution $\gamma : \Gamma \to \Gamma'$ and a fixed variable assignment $\alpha \in [\![\Gamma]\!]^M$.*

It follows that, if $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$ and $\Gamma \vdash_\Sigma E : E'$, then $[\![\Gamma' \vdash \gamma(E)]\!]^{M,\alpha} = [\![\Gamma \vdash E]\!]^{M,[\![\gamma]\!]^M(\alpha)}$.

Proof. We proceed by induction on the derivation $\Gamma \vdash_\Sigma E : E'$.

- case 1: $E = c$ implies $[\![\Gamma \vdash E]\!]^{M,[\![\gamma]\!]^M(\alpha)} = [\![\Gamma \vdash c]\!]^{M,[\![\gamma]\!]^M(\alpha)} = M(c) = [\![\Gamma' \vdash c]\!]^{M,\alpha} = [\![\Gamma' \vdash \gamma(c)]\!]^{M,\alpha} = [\![\Gamma' \vdash \gamma(E)]\!]^{M,\alpha}$
- case 2: $E = x$ implies $[\![\Gamma \vdash E]\!]^{M,[\![\gamma]\!]^M(\alpha)} = [\![\Gamma \vdash x]\!]^{M,[\![\gamma]\!]^M(\alpha)} = [\![\gamma]\!]^M(\alpha)(x) = [\![\Gamma' \vdash t]\!]^{M,\alpha}$
- case 3: $E = \text{type}$ implies $[\![\Gamma \vdash E]\!]^{M,[\![\gamma]\!]^M(\alpha)} = [\![\Gamma \vdash \text{type}]\!]^{M,[\![\gamma]\!]^M(\alpha)} = \text{Set} = [\![\Gamma' \vdash \text{type}]\!]^{M,\alpha} = [\![\Gamma' \vdash \gamma(\text{type})]\!]^{M,\alpha} = [\![\Gamma' \vdash \gamma(x)]\!]^{M,\alpha}$

□

Chapter 4

Reflecting Terms

In this chapter we present the *type-theoretical foundations* of a minimal theory supporting term reflection (Section 4.1 and Section 4.2), together with its corresponding *semantics* (Section 4.3).

4.1 Syntax

In order to be able to talk about term reflection, it is necessary to expand the syntax of the basic language we start from (see Section 3.1) with the needed reflection primitives, as well as with auxiliary meta-entities.

At the object-level, additional expressions are obtained through the *type formation* of a reflected type $\uparrow_S A$ inhabited by all reflected (quoted) terms of type A over S, through the *quotation* $\lceil t \rceil_S$ of term t over a named signature S, through the *evaluation* $\lfloor q \rfloor_S$ of a reflected term q over a named signature S and through the *elimination* q^σ of a quoted term q via a morphism σ (see Table 5.4).

At the meta-level, the extended grammar includes a *theory graph* G, consisting of all theories definable in our language, and a *stack of frames* W, comprising of named signatures (theories) S and their contexts Γ. Such a *stack* is used to keep track of the theories that have been quoted.

The corresponding typing and equality judgements are also given in Table 5.4 and will be discussed at length in Section 4.2.

Next, we illustrate the introduced reflection constructs as part of a running example, given by an encoding for the theory of natural numbers.

Example 1. Natural Numbers

Let us begin from the meta-level signature `Nat`, which defines the *set of terms over the natural numbers*, by providing a generic type `nat` and the basic constructors `zero` and `succ`.

```
1   Nat = nat : type.
          zero : nat. succ : nat → nat.
```

	Grammar	Typing Judgment	Equality Judgment
Theory graphs (G)	$G \ ::= \ \cdot \mid G,\ \text{sig}\ S = \{\Sigma\}$	$\vdash \ G \ \texttt{TGph}$	$\vdash \ G = G'$
Frame stacks (W)	$W \ ::= \ \cdot \mid W,\ (S, \Gamma)$	$\vdash^G W \ \texttt{Stack}$	$\vdash^G W = W'$
Signatures (Σ)	$\Sigma \ ::= \ \cdot \mid \Sigma,\ c : E \mid S$	$\vdash^G \Sigma \ \texttt{Sig}$	$\vdash^G \Sigma = \Sigma'$
Morphisms (σ)	$\sigma \ ::= \ \cdot \mid \sigma,\ c/E$	$W \vdash^G \sigma : S$	$W \vdash^G \sigma = \sigma' : S$
Contexts (Γ)	$\Gamma \ ::= \ \cdot \mid \Gamma,\ x : E$	$W \vdash^G \Gamma \ \texttt{Ctx}$	$W \vdash^G \Gamma = \Gamma'$
Substitutions (γ)	$\gamma \ ::= \ \cdot \mid \gamma,\ x/E$	$W \vdash^G \gamma : \Gamma$	$W \vdash^G \gamma = \gamma' : \Gamma$
Expressions (E, A, t, q) Reflected terms (q)	$E \ ::= \ c \mid x \mid \texttt{type} \mid E \to E \mid E\ E \mid [E]\ E \mid \uparrow_S A \mid \lceil t \rceil_S \mid \lfloor q \rfloor_S \mid q^\sigma \mid$	$W \vdash^G E : E'$	$W \vdash^G E = E'$

Table 4.1: Grammar, Typing and Equality Judgments for a Type Theory with Term Reflection

This generic specification of the naturals is, however, not enough to form an inductive abstract datatype. Specifically, in order to obtain the *inductive type of natural numbers* $\uparrow_{\texttt{Nat}} \texttt{nat}$, one has to take the reflection of all the terms of type `nat` over the signature `Nat`. The constructors `zero` and `succ` are reflected into the terms $\lceil \texttt{zero} \rceil_{\texttt{Nat}}$ and $\lceil \texttt{succ} \rceil_{\texttt{Nat}}$ of a new signature `N`. These are abbreviated by $\mathbb{N}$ and, respectively, `0` and `s`.

```
N = {
      ℕ : type   = ↑Nat nat.
      0 : ℕ      = ⌈zero⌉Nat .
      s : ℕ → ℕ  = ⌈succ⌉Nat .
    }.
```

Furthermore, we illustrate in our next example how our approach gives us not only inductive types, but also what is generally referred to as *inductive families*. Indeed, let us look at the corresponding encoding of the datatype for vectors.

Example 2. Encoding Vectors.

First, let us note that the type of a vector having a certain fixed length can be fully specified with respect to the type of its entries. In our case, we take the latter to be a type which we denote as `a` and which is kinded by `type`, the base type.

Starting from the above encoding for the natural numbers, we have that a vector type `vec` is constructed by taking the type $\uparrow_{Nat} \texttt{nat}$ of all reflected naturals `nat` over the theory `Nat`.

Consequently, the type constructor $\emptyset$, for the type of all vectors of length zero, is formed by the reflected type of `zero` over `Nat`. Lastly, the `cons` type constructor is given by the function type formed from the dependent type of all vector types of length x and from that corresponding to the type `a` of the vector's entries. It returns the type of all vectors of length $x + 1$, value computable by applying the reflected succesor operation `s` to the evaluation $\lfloor x \rfloor_{Nat}$ of x with respect to the theory `Nat` and by reflecting the outcome back to `Nat`.

As a result, the same signature `Vector` gives multiple inductive types. Thus, we obtain an elegant solution for a common problem with inductive types, namely that one cannot take the fixpoint over an inductive family, since it is defined relative to an index.

```
Vector = {
         a : type.
         vec : ↑_Nat nat → type.
4        ∅ : vec ↑_Nat zero.
         cons : Πx : ↑_Nat nat. vec x → a → vec ⌈s⌊x⌋_Nat⌉_Nat.
         }.
```

Definition 5. Frame Well-Ordering.

Let us consider a theory graph G and a stack of frames containing (bottom-to-top) the frames $W_1, \ldots, W_n$, with corresponding theories $T_1, \ldots, T_n$.

We define a partial relation $<^G$ between two theories from G, as denoting the degree of their expressivity. Particularly, the "larger" the theory, the more expressive it is, in the sense that it can contain quoted elements from any of the "smaller" theories. In turn, a theory cannot contain any quoted inhabitants of the "larger" theory, but only evaluations.

We say that the frames $W_1, \ldots, W_n$ are *well-ordered* with respect to $<^G$, if the following holds:

$$W_n <^G \cdots <^G W_1 \tag{4.1}$$

Since terms can contain arbitrarily many quotations and evaluations, it becomes necessary to appropriately handle such nested operations, when dealing with morphism application or when defining the semantics (see Section 4.3).

This is realized through two similar families of meta-theoretical functions, $\{\sigma^n\}_{n\in\mathbb{N}^*}$ and $\{(M\alpha)^n\}_{n\in\mathbb{N}^*}$, that operate structurally on a given term t, preserving the subterms of t, if the number of quotations and evaluations is well-balanced. If, however, the top-level subterm of t is governed by one extra evaluation that returns it to the object level, the functions correspond to normal morphism application and, respectively, to choosing a representative from the semantics of the top-level term.

In the following, we will discuss the auxiliary morphism application functions $\{\sigma^n\}_{n\in\mathbb{N}^*}$. The auxiliary evaluation functions $\{(M\alpha)^n\}_{n\in\mathbb{N}^*}$ will be discussed in Section 4.3.

Definition 6. Auxiliary Morphism Application.

Let us define the auxiliary morphism application to reflected terms $\sigma^n(t)$, which maps terms $W, (S, \cdot) \vdash^G t : A$ to closed terms $(S, \cdot) \vdash^G \sigma^n(t) : \sigma^n(A)$, for $n \in \mathbb{N}$.

$$
\begin{array}{rcl}
\sigma^{n+1}(c) & = & c \\
\sigma^{1}(c) & = & \sigma_c \\
\sigma^{n}(x) & = & x \\
\sigma^{n}(\texttt{type}) & = & \texttt{type} \\
\sigma^{n}(\uparrow_S A) & = & \uparrow_S \sigma^{n+1}(A) \\
\sigma^{n}(\lceil t \rceil_S) & = & \lceil \sigma^{n+1}(t) \rceil_S \\
\sigma^{n+1}(\lfloor t \rfloor_S) & = & \lfloor \sigma^{n}(t) \rfloor_S \\
\sigma^{1}(\lfloor t \rfloor_S) & = & t^{\sigma} \\
\sigma^{n}(t^{\tau}) & = & \sigma^{n}(t)^{\sigma^{n}(\tau)}
\end{array}
\tag{4.2}
$$

As we can see, the index n monitors the number of quoted theories on the stack of frames. Explicitly, when the auxiliary morphism σ^n is applied to the quotation $\lceil t \rceil_S$ of a term, the index count is incremented by a unit and, conversely, when it is applied to the evaluation $\lfloor t \rfloor_S$ of a term, its index count is decremented by a unit. Note that, if the index count is one, the auxiliary morphism σ^1 coincides with the meta-morphism σ and its application to an evaluated term returns the elimination t^{σ} of that term.

For index values greater than one, auxiliary morphism application preserves constants c, variables x, as well as the `type` base type. However, in the case of constants, if the index count is one, σ^1 and σ are the same, as seen previously. Hence, the application of σ^1 to a constant c coincides with the normal morphism application in Section 3. Finally, looking at the elimination t^{τ} of a term t by a morphism τ, the auxiliary morphism σ^n is applied compositionally to both the term and the morphism, for all values of n.

4.2 Proof Theory

We proceed to briefly discuss the well-formedness inference rules corresponding to the new constructs introduced in Section 4.1. The *typing rules* for term reflection are summarized in Table 4.2, while an overview of the *equality rules* is given in Table 4.3.

Typing Rules. To start with, let us give an account of the *typing rules*, starting from that for the *type formation* $\uparrow_{typ}$ of the reflected type corresponding to quoted terms with type A, under a named signature (theory) S . This $\uparrow_S A$ type is well-formed with respect to a stack W, if the type A is itself well-formed of type `type` with respect to the stack W onto which an additional frame has been pushed, containing the quoted theory S and its corresponding (empty) context. Additionally, we have to check for the theory W being "larger" than S, according to the $<^G$ relation. Intuitively, as seen in Section 4.1, we defined the relation $<^G$ as enforcing the fact that entities in any of the theories belonging to W can quote from the "smaller" theory S, but not vice-versa.

Next, we have the *introduction* rule $\uparrow_I$ for the type $\uparrow_S A$ of reflected terms over a named signature S. The rule states that the quotation $\lceil t \rceil_S$ of a term t over a named signature S is well-formed with respect to a stack W (has reflected type $\uparrow_S A$), if the

Type formation	$\dfrac{W, (S, \cdot) \vdash^G A : \texttt{type} \quad W >^G S}{W \vdash^G \uparrow_S A : \texttt{type}} \uparrow_{typ}$
Introduction (reflection)	$\dfrac{W, (S, \cdot) \vdash^G t : A \quad W >^G S}{W \vdash^G \lceil t \rceil_S : \uparrow_S A} \uparrow_I$
Evaluation	$\dfrac{W \vdash^G q : \uparrow_S A}{W, (S, \cdot) \vdash^G \lfloor q \rfloor_S : A} \uparrow_{ev}$
Elimination (induction)	$\dfrac{W \vdash^G q : \uparrow_S A \quad W \vdash \sigma : S}{W \vdash^G q^\sigma : \sigma(A)} \uparrow_E$

Table 4.2: Inference System for a Type Theory with Term Reflection

term t itself is well-formed with respect to the extended stack $W, (S, \cdot)$ and has (the well-formed) type A. Moreover, the same well-ordering check as in the type formation case has to be performed.

The *evaluation* rule $\uparrow_{ev}$ states that the evaluation of a quoted term q under a named signature S is well-formed with respect to the extended stack $W, (S, \cdot)$ and has type A, if q is well-formed under W and has the reflected type $\uparrow_S A$ over S. Note that, when type checking evaluations, the top-level frame recording the quoted theory and its context is popped from the stack. Thus, in the implementation, it is necessary to perform an additional check against the frame stack being empty (see Section 6).

The final rule $\uparrow_E$, namely that for morphism *elimination* is the most involved. In particular, the elimination of a quoted term q through a morphism σ is well-formed with respect to the stack W (and has type $\sigma(A)$), if two constraints are satisfied. Firstly, q has to be well-formed with respect to W and have the reflected type $\uparrow_S A$ of quoted terms of type A over S. Secondly, the morphism σ has to be well-formed with respect to W and translate terms from the theory S.

Equality Rules. Looking at Table 4.3, we first remark on the fact that the $=^t_I$ *introduction* and $=^t_E$ *elimination* rules for reflected terms are straight-forwardly stating that quotation and, respectively, the elimination via a morphism, preserve term equality.

According to the $=^t_C$ *computation* rule, the term $\lceil t \rceil^\sigma_S$, obtained by applying a morphism σ to a reflected term $\lceil t \rceil_S$, is equal to that obtained by applying the meta-level morphism σ to the term t, with respect to the frame W, given that, in turn, σ is well-formed with respect to W.

Next, the $=^t_{sound}$ *soundness* rule states that a well-formed term is preserved via quoting and then evaluating the result over the same theory S, i.e that quotation and evaluation are partially inverse operations.

Analogous to the soundness rule, the $=^t_{compl}$ *completeness* rule asserts that, by first evaluating a reflected term q under a theory S and then reflecting it over the same theory, the result q' is equal to q under a frame stack W, if equality holds between q and

Reflecting...	...terms
Introduction	$\dfrac{W,(S,\cdot) \vdash^{G_0} t = t'}{W \vdash^{G} \lceil t \rceil_S = \lceil t' \rceil_S} =^t_I$
Elimination	$\dfrac{W \vdash^{G} q = q'}{W \vdash^{G} q^\sigma = q'^\sigma} =^t_E$
Computation	$\dfrac{W \vdash^{G} \sigma : S}{W \vdash^{G} \lceil t \rceil^\sigma_S = \sigma(t)} =^t_C$
Soundness	$\dfrac{W,(S,\cdot) \vdash^{G_0} t = t'}{W,(S,\cdot) \vdash^{G_0} \lfloor \lceil t \rceil_S \rfloor_S = t'} =^t_{sound}$
Completeness	$\dfrac{W \vdash^{G} q = q'}{W \vdash^{G_0} \lceil \lfloor q \rfloor_S \rceil_S = q'} =^t_{compl}$
Extensionality	$\dfrac{W,(S,\cdot) \vdash^{G_0} \lfloor q \rfloor_S = \lfloor q' \rfloor_S}{W \vdash^{G} q = q'} =^t_{ext}$

Table 4.3: Equality Inference Rules a Type Theory with Term Reflection

q'. Together with the previous result, this introduces quotation and evaluation as inverse to each other.

Finally, conforming to the $=^t_{ext}$ *extensionality* reflection rule, equality between two well-formed quoted terms q and q', with respect to a frame W is preserved by evaluation under a theory S with respect to the extended frame $W,(S,\cdot)$.

Example 1. Natural Numbers (continued) One can now define basic operations for natural numbers, such as addition (`add`) and multiplication (`product`), by specifying morphisms σ and σ' from `Nat` to `N`. These morphisms make the case distinction on the structure of the first argument of the operations, returning a function $\mathbb{N} \to \mathbb{N}$, which is then applied to their second argument.

The assignment $\texttt{nat} \mapsto \mathbb{N} \to \mathbb{N}$ sets the type of m^σ, according to the rule $\uparrow_E$ (see Table 4.2). Consequently, $\sigma(\texttt{zero})$ must be of type $\sigma(\texttt{nat}) = \mathbb{N} \to \mathbb{N}$, and $\sigma(\texttt{succ})$ must have type $\sigma(\texttt{nat} \to \texttt{nat}) = (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$.

The operations are defined inductively on their first argument, by performing repeated successor applications, in the case of addition, and, correspondingly, repeated additions, in the case of multiplication.

```
N = {

    ℕ : type = ↑Nat nat.
    0 : ℕ    = ⌈zero⌉Nat .
    s : ℕ → ℕ = ⌈succ⌉Nat .

  add : ℕ → ℕ → ℕ
```

```
9                   = [m][n] m^σ n.
                      (where σ = { nat ↦ ℕ → ℕ.
                                   zero ↦ [n : ℕ] n.
                                   succ ↦ [f : ℕ → ℕ] ([n: ℕ] s(f(n))).
                                 })
14          product : ℕ → ℕ → ℕ
                    = [m][n] m^σ' n.
                      (where σ' = { nat ↦ ℕ → ℕ.
                                    zero ↦ [_] 0.
                                    succ ↦ [f : ℕ → ℕ] ([n: ℕ] (f(n) + n)).
19                                  })

            }.
```

Let us further illustrate the application of the above-defined inference rules, by presenting a proof for the type preservation of the *auxiliary morphism* application introduced in Section 4.1.

Example 3. Auxiliary Morphism Type Preservation.
In order to prove the type preservation property of morphism application (see Definition 6), we first need to prove an auxiliary commutativity lemma, stated below and trivially verifiable by induction.

Lemma 4.2.1. ***Commutativity.***

$$\frac{W, S, R_1, \cdots, R_n, T \vdash^G A : \mathit{type} \quad W, S, R_1, \cdots, R_n \vdash^G \tau : T \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n \vdash^G \sigma^n(\tau(A)) = \sigma^n(\tau)(\sigma^{n+1}(A))} \tag{4.3}$$

According to the commutativity result, applying the auxiliary morphism σ^n to $\tau(A)$ is the same as first applying σ^n to τ and then composing the result with the application $\sigma^{n+1}(A)$ of the auxiliary morphism with incremented index σ^{n+1} to A.

Theorem 4.2.2. ***Type Preservation.***

Given the function σ^n defined above, the following invariant is preserved:

$$\frac{W, S, R_1, \cdots, R_n \vdash^G t : A \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n \vdash^G \sigma^n(t) : \sigma^n(A)} \tag{4.4}$$

Proof. The proof follows by structural induction on the term t.

- $t ::= c$

We know that:

$$W, S, R_1, \cdots, R_n \vdash^G c : A \tag{4.5}$$

$$W \vdash^G \sigma : S \tag{4.6}$$

Given that we have the following ordering of the theories in our frame stack:

$$R_n <^G \cdots <^G R_1 <^G S <^G W \tag{4.7}$$

we know that theory R_n can be quoted in any of the "larger" theories R_{n-1} to S, but cannot quote symbols from any of them. Since A inhabits theory R_n, it can only use its symbols and, also since it does not contain any evaluations, it is, therefore, left invariant with respect to the application of function σ^n.

Substituting $\sigma^n(A)$ and applying the corresponding definition for $\sigma^n(c)$ in 4.5:

$$W, S, R_1, \cdots, R_n \vdash^G \sigma^n(c) : \sigma^n(A) \tag{4.8}$$

Also, since no symbol from theory S is used, we can strengthen 4.9 to:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(c) : \sigma^n(A) \tag{4.9}$$

Hence, we can conclude:

$$\frac{W, S, R_1, \cdots, R_n \vdash^G c : A \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n \vdash^G \sigma^n(c) : \sigma^n(A)} \tag{4.10}$$

- $t ::= x$

Analogous to the previous case.

- $t ::= \texttt{type}$

Analogous to the previous case.

- $t ::= \uparrow_T A$

We know:

$$W, S, R_1, \cdots, R_n \vdash^G \uparrow_T A : \texttt{type} \tag{4.11}$$

$$W \vdash^G \sigma : S \tag{4.12}$$

Applying the $\uparrow_{typ}$ **Type Formation Rule** to 4.11:

$$W, S, R_1, \cdots, R_n, T \vdash^G A : \texttt{type} \tag{4.13}$$

The Induction Hypothesis on 4.55 and 4.12 yields:

$$W, R_1, \cdots, R_n, T \vdash^G \sigma^{n+1}(A) : \sigma^{n+1}(\texttt{type}) \tag{4.14}$$

which, by substituting the definition for $\sigma^{n+1}(\texttt{type})$, becomes:

$$W, R_1, \cdots, R_n, T \vdash^G \sigma^{n+1}(A) : \texttt{type} \tag{4.15}$$

The $\uparrow_{typ}$ **Type Formation Rule** leads to:

$$W, R_1, \cdots, R_n \vdash^G \uparrow_T \sigma^{n+1}(A) : \texttt{type} \tag{4.16}$$

Applying the definition of $\uparrow_T \sigma^{n+1}(A)$ and that of $\sigma^n(\texttt{type})$:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(\uparrow_T A) : \sigma^n(\texttt{type}) \tag{4.17}$$

Hence:

$$\frac{W, S, R_1, \cdots, R_n \vdash^G \uparrow_T A : \texttt{type} \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n \vdash^G \sigma^n(\uparrow_T A) : \sigma^n(\texttt{type})} \tag{4.18}$$

- $t ::= \lceil t \rceil_T$

We know:

$$W, S, R_1, \cdots, R_n \vdash^G \lceil t \rceil_T : \uparrow_T A \tag{4.19}$$

$$W \vdash^G \sigma : S \tag{4.20}$$

Applying the $\uparrow_I$ Introduction Rule to 4.19:

$$W, S, R_1, \cdots, R_n, T \vdash^G t : A \tag{4.21}$$

From 4.21 and 4.20, we obtain, via the Induction Hypothesis:

$$\frac{W, S, R_1, \cdots, R_n, T \vdash^G t : A \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n, T \vdash^G \sigma^{n+1}(t) : \sigma^{n+1}(A)} \tag{4.22}$$

Using the $\uparrow_I$ Introduction Rule again on the last derivation:

$$\frac{W, R_1, \cdots, R_n, T \vdash^G \sigma^{n+1}(t) : \sigma^{n+1}(A)}{W, R_1, \cdots, R_n \vdash^G \lceil \sigma^{n+1}(t) \rceil_T : \uparrow_T \sigma^{n+1}(A)} \tag{4.23}$$

Substitutions according to the corresponding definitions in 4.2 lead to:

$$\frac{W, R_1, \cdots, R_n \vdash^G \lceil \sigma^{n+1}(t) \rceil_T : \uparrow_T \sigma^{n+1}(A)}{W, R_1, \cdots, R_n \vdash^G \sigma^n(\lceil t \rceil_T) : \sigma^n(\uparrow_T A)} \tag{4.24}$$

Hence, we proved:

$$\frac{W, S, R_1, \cdots, R_n \vdash^G \lceil t \rceil_T : \uparrow_T A \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n \vdash^G \sigma^n(\lceil t \rceil_T) : \sigma^n(\uparrow_T A)} \tag{4.25}$$

- $t ::= \lfloor t \rfloor_T$

We know:

$$W, S, R_1, \cdots, R_n, T \vdash^G \lfloor t \rfloor_T : A \tag{4.26}$$

$$W \vdash^G \sigma : S \tag{4.27}$$

Applying the $\uparrow_{ev}$ **Evaluation Rule** to 4.106:

$$W, S, R_1, \cdots, R_n \vdash^G t : \uparrow_T A \tag{4.28}$$

The Induction Hypothesis on 4.28 and 4.27 yields:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(t) : \sigma^n(\uparrow_T A) \tag{4.29}$$

Substituting the definition for $\sigma^n(\uparrow_T A)$:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(t) : \uparrow_T \sigma^{n+1}(A) \tag{4.30}$$

The $\uparrow_{ev}$ **Evaluation Rule** on 4.30 leads to:

$$W, R_1, \cdots, R_n, T \vdash^G \lfloor \sigma^n(t) \rfloor_T : \sigma^{n+1}(A) \tag{4.31}$$

Substituting the definition for $\lfloor \sigma^n(t) \rfloor_T$:

$$W, R_1, \cdots, R_n, T \vdash^G \sigma^{n+1}(\lfloor t \rfloor_T) : \sigma^{n+1} \tag{4.32}$$

Hence:

$$\frac{W, S, R_1, \cdots, R_n, T \vdash^G \lfloor t \rfloor_T : A \quad W \vdash^G \sigma : S}{W, R_1, \cdots, R_n, T \vdash^G \sigma^{n+1}(\lfloor t \rfloor_T) : \sigma^{n+1}(A)} \tag{4.33}$$

- $t ::= t^\tau$

We know:

$$W, S, R_1, \cdots, R_n \vdash^G t^\tau : \tau(A) \tag{4.34}$$

$$W \vdash^G \sigma : S \tag{4.35}$$

Applying the $\uparrow_E$ **Elimination Rule** to 4.34, we have:

$$W, S, R_1, \cdots, R_n \vdash^G t : \uparrow_T A \tag{4.36}$$

and

$$W, S, R_1, \cdots, R_n \vdash^G \tau : T \tag{4.37}$$

Using the Induction Hypothesis on 4.36 and 4.37:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(t) : \sigma^n(\uparrow_T A) \tag{4.38}$$

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(\tau) : \sigma^n(T) \tag{4.39}$$

Substituting the corresponding definition for $\sigma^n(\uparrow_T A)$ in 4.38:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(t) : \uparrow_T \sigma^{n+1}(A) \tag{4.40}$$

Also, since T is a theory name, we can replace $\sigma^n(T)$ with T in 4.39 and obtain:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(\tau) : T \tag{4.41}$$

From the $\uparrow_E$ **Elimination Rule** applied to 4.40 and 4.41:

$$\frac{W, R_1, \cdots, R_n \vdash^G \sigma^n(t) : \uparrow_T \sigma^{n+1}(A) \quad W, R_1, \cdots, R_n \vdash^G \sigma^n(\tau) : T}{W, R_1, \cdots, R_n \vdash^G \sigma^n(t)^{\sigma^n(\tau)} : \sigma^n(\tau)(\sigma^{n+1}(A))} \tag{4.42}$$

Substituting the definition for $\sigma^n(t)^{\sigma^n(\tau)}$ and using the **Commutativity Lemma** 4.3.1:

$$W, R_1, \cdots, R_n \vdash^G \sigma^n(t^\tau) : \sigma^n(\tau(A)) \tag{4.43}$$

which is what had to be proven in this case. □

4.3 Semantics

Having presented the necessary term reflection primitives in Section 4.1 and their formation rules in Section 4.2, let us now discuss the respective meaning of these additional constructs.

We proceed by building on the semantics defined for basic type theory, as specified in Section 3. Firstly, we introduce the preliminary notions of *syntactic congruence* (see Definition 7) and *simplification function* (see Definition 8). We then prove the well-definedness property of the latter (see Theorem 4.3.2) and conclude the section with a summary of the interpretation rules for term reflection (see Table 4.4), together with explanations.

To give a preparatory intuition, we remark on the fact that quoted terms are interpreted through *classes*, while their type is interpreted through *quotients*. These quotient constructions are taken with respect to the equivalence relation given by (syntactic) equality (see Section 3.2). In the following, we specify the congruence relation this relation forms on the set of all terms definable via the grammar.

Definition 7. Syntactic Congruence.

Consider the set of all terms t that are well-formed with respect to a theory stack frame W:

$$Terms = \{t \mid W \vdash^G t : A\} \tag{4.44}$$

Let us define *syntactic congruence* as the binary relation $\cong \subseteq Terms \times Terms$ on $Terms$, such that:

$$t_1 \cong t_2 \text{ iff } W \vdash^G t_1 = t_2 \tag{4.45}$$

As is generally the case, this congruence $\cong$ has a corresponding quotient structure $Terms/\cong$ comprising of equivalence classes.

In addition to this *syntactic congruence* relation, due to similar reasons as those given in Section 4.1 to introduce the auxiliary morphism construction, we require a *simplification function* $(M\alpha)^n$, as defined below. Particularly, this function helps us monitor the balance between potential nested quotations and evaluations inside a term and to adequately give this term meaning through a simplification of its form.

Definition 8. Simplification Function.

Let us define the following simplification function, which maps terms $W, (S, \cdot) \vdash^G t : A$ to closed terms $(S, \cdot) \vdash^G (M\alpha)^n(t) : (M\alpha)^n(A)$.

$$
\begin{array}{rcl}
(M\alpha)^n(W \vdash^G c) & = & c \\
(M\alpha)^n(W \vdash^G x) & = & x \\
(M\alpha)^n(W \vdash^G \texttt{type}) & = & \texttt{type} \\
(M\alpha)^n(W \vdash^G \uparrow_S A) & = & \uparrow_S (M\alpha)^{n+1}(W, (S, \cdot) \vdash^G A) \\
(M\alpha)^n(W \vdash^G \lceil t \rceil_S) & = & \lceil (M\alpha)^{n+1}(W, (S, \cdot) \vdash^G t) \rceil_S \\
(M\alpha)^{n+1}(W, (S, \cdot) \vdash^G \lfloor t \rfloor_S) & = & \lfloor (M\alpha)^n(W \vdash^G t) \rfloor_S \\
(M\alpha)^1((S, \cdot) \vdash^G \lfloor t \rfloor_S) & = & r, \text{ where } r \in [\![(S, \cdot) \vdash^G t]\!]^{M,\alpha} \\
(M\alpha)^n(W \vdash^G t^\tau) & = & (M\alpha)^n(W \vdash^G t)^{(M\alpha)^n(W \vdash^G \tau)}
\end{array}
\tag{4.46}
$$

As before, the index n keeps track on the stack of frames. Explicitly, when the simplification function σ^n is applied to the quotation $\lceil t \rceil_S$ of a term, the index count is incremented by a unit and, conversely, when it is applied to the evaluation $\lfloor t \rfloor_S$ of a term, its index count is decremented by a unit. Note that, if the index count is one, applying the simplification function $(M\alpha)^1$ to a term t coincides with taking a representative r from the equivalence class corresponding to the semantics of this term (see Table 4.4 below).

For index values greater than one, the application of the simplification function preserves constants c, variables x, as well as the `type` base type. Also, looking at the elimination t^τ of a term t by a morphism τ, $(M\alpha)^n$ is applied compositionally to both the term and the morphism, for all values of n.

Analogous to the type preservation property for auxiliary morphisms, it holds that the well-definedness of $(M\alpha)^n$ can be stated as an invariant. Before formulating and proving the respective Theorem 4.3, let us state below the needed auxiliary commutativity Lemma 4.3.1.

Lemma 4.3.1. ***Commutativity.***

$$
\frac{S, R_1, \cdots, R_n, T \vdash^G A : \textit{type} \quad S, R_1, \cdots, R_n \vdash^G \tau : T}{R_1, \cdots, R_n \vdash^G (M\alpha)^n(\tau(A)) = (M\alpha)^n(\tau)(\uparrow_T (M\alpha)^{n+1}(A))}
\tag{4.47}
$$

According to this commutativity result, simplifying the application $\tau(A)$ of a morphism τ to a term A is the same as applying the simplification $(M\alpha)^n(\tau)$ to the simplification $(M\alpha)^n(\uparrow_S A)$ of the type $\uparrow_S A)$ of all reflected terms of type A. The result is provable by induction on n.

We can now assert and verify the well-definedness of the simplification function.

Theorem 4.3.2. ***Well-Definedness of*** $(M\alpha)^n$.

Given the function $(M\alpha)^n$ defined above, the following invariant is preserved:

$$\frac{S, R_1, \cdots, R_n \ \vdash^G \ t : A \quad (M\alpha) \ \in [\![S]\!]}{R_1, \cdots, R_n \ \vdash^G \ (M\alpha)^n(t) : (M\alpha)^n(A)} \tag{4.48}$$

Remark:
In particular, for $n = 0$, we can think of $(M\alpha^0)$ as coinciding with the set-theoretical interpretation function and of typing as coinciding with set membership.

$$\frac{S \ \vdash^G \ t : A \quad (M\alpha) \ \in [\![S]\!]}{(M\alpha)^0(t) \in (M\alpha)^0(A)} \tag{4.49}$$

Proof. The proof proceed by structural induction on the term t.

- $t ::= c$

We know:

$$S, R_1, \cdots, R_n \ \vdash^G \ c : A \tag{4.50}$$

$$(M\alpha) \in [\![S]\!] \tag{4.51}$$

The theories on the frame stack adhere to the following ordering:

$$R_n <^G \cdots <^G R_1 <^G S \tag{4.52}$$

Hence, by a similar argument as previously, A cannot use symbols from S and, consequently, $(M\alpha)(A) = A$. Also, from the definition of $(M\alpha)(c)$, it follows trivially that:

$$R_1, \cdots, R_n \ \vdash^G \ c : A \tag{4.53}$$

- $t ::= x$

Analogous to the previous case.

- $t ::= \texttt{type}$

Analogous to the previous case.

- $t ::= \uparrow_T A$

We know:

$$S, R_1, \cdots, R_n \ \vdash^G \ \uparrow_T A : \texttt{type} \tag{4.54}$$

Applying the $\uparrow_{typ}$ **Type Formation Rule**:

$$S, R_1, \cdots, R_n, T \ \vdash^G \ A : \texttt{type} \tag{4.55}$$

The Induction Hypothesis and the fact that $(M\alpha)^{n+1}(\texttt{type}) = \texttt{type}$ yield:

$$R_1, \cdots, R_n, T \ \vdash^G \ (M\alpha)^{n+1}(A) : \texttt{type} \tag{4.56}$$

The $\uparrow_{typ}$ **Type Formation Rule** leads to:

$$R_1, \cdots, R_n \ \vdash^G \ \uparrow_T (M\alpha)^{n+1}(\texttt{A}) : \texttt{type} \tag{4.57}$$

Applying the definition of $\uparrow_T (M\alpha)^{n+1}(\texttt{A})$:

$$R_1, \cdots, R_n \ \vdash^G \ (M\alpha)^n(\uparrow_T A) : \texttt{type} \tag{4.58}$$

Hence:

$$\frac{S, R_1, \cdots, R_n \ \vdash^G \ \uparrow_T A : \texttt{type} \quad (M\alpha) \ \in [\![S]\!]}{R_1, \cdots, R_n \ \vdash^G \ (M\alpha)^n(\uparrow_T A) : \sigma^n(\texttt{type})} \tag{4.59}$$

- $t ::= \lceil t \rceil_T$

We know:

$$S, R_1, \cdots, R_n \ \vdash^G \ \lceil t \rceil_T : \ \uparrow_T A \tag{4.60}$$

Applying the $\uparrow_I$ **Introduction Rule**:

$$S, R_1, \cdots, R_n, T \ \vdash^G \ t : A \tag{4.61}$$

From the Induction Hypothesis:

$$R_1, \cdots, R_n, T \ \vdash^G \ (M\alpha)^{n+1}(t) : (M\alpha)^{n+1}(A) \tag{4.62}$$

Using the $\uparrow_I$ **Introduction Rule** again:

$$R_1, \cdots, R_n \ \vdash^G \ \lceil (M\alpha)^{n+1}(t) \rceil_T : \ \uparrow_T (M\alpha)^{n+1}(A) \tag{4.63}$$

Substituting the corresponding definitions for $\lceil (M\alpha)^{n+1}(t) \rceil_T$ and $\uparrow_T (M\alpha)^{n+1}(A)$:

$$R_1, \cdots, R_n \ \vdash^G \ (M\alpha)^n(\lceil t \rceil_T) : (M\alpha)^n(\uparrow_T A) \tag{4.64}$$

- $t ::= \lfloor t \rfloor_T$

We know:

$$S, R_1, \cdots, R_n, T \vdash^G \lfloor t \rfloor_T : A \tag{4.65}$$

Applying the $\uparrow_{ev}$ **Evaluation Rule**:

$$S, R_1, \cdots, R_n \vdash^G t : \uparrow_T A \tag{4.66}$$

From the Induction Hypothesis:

$$R_1, \cdots, R_n \vdash^G (M\alpha)^n(t) : (M\alpha)^n(\uparrow_T A) \tag{4.67}$$

Substituting the corresponding definition for $(M\alpha)^n(\uparrow_T A)$:

$$R_1, \cdots, R_n \vdash^G (M\alpha)^n(t) : (M\alpha)^n(\uparrow_T A) \tag{4.68}$$

Applying the $\uparrow_{ev}$ **Evaluation Rule** again:

$$R_1, \cdots, R_n, T \vdash^G \lfloor (M\alpha)^n(t) \rfloor_T : (M\alpha)^n(\uparrow_T A) \tag{4.69}$$

Substituting the corresponding definition for $\lfloor (M\alpha)^n(t) \rfloor_T$ and $(M\alpha)^n(\uparrow_T A)$:

$$R_1, \cdots, R_n, T \vdash^G (M\alpha)^{n+1}(\lfloor t \rfloor_T) : \uparrow_T (M\alpha)^{n+1}(A) \tag{4.70}$$

- $t ::= t^\tau$

We know:

$$S, R_1, \cdots, R_n \vdash^G t^\tau : \tau(A) \tag{4.71}$$

$$S, R_1, \cdots, R_n \vdash^G \tau : T \tag{4.72}$$

From the Induction Hypothesis on 4.71 and the definition for $(M\alpha)^n(\uparrow_T A)$:

$$R_1, \cdots, R_n \vdash^G (M\alpha)^n(t) : \uparrow_T (M\alpha^{n+1})(A) \tag{4.73}$$

Similarly, for 4.72, using $(M\alpha)^n(T) = T$, since T is just a theory name:

$$R_1, \cdots, R_n \vdash^G (M\alpha)^n(\tau) : T \tag{4.74}$$

Applying the $\uparrow_E$ **Elimination Rule** to 4.73 and 4.74:

$$R_1, \cdots, R_n \vdash^G (M\alpha)(t)^{(M\alpha)(\tau)} : (M\alpha)^n(\tau)(\uparrow_T (M\alpha)^{n+1}(A)) \tag{4.75}$$

The definition for $(M\alpha)(t)^{(M\alpha)(\tau)}$ and the **Commutativity Lemma** 4.47 yield:

$$R_1, \cdots, R_n \vdash^G (M\alpha)(t^\tau) : (M\alpha)^n(\tau(A)) \tag{4.76}$$

□

We can now inspect the semantic rules for term reflection primitives, as seen in Table 4.4. In formulating these rules, we make a case distinction depending on the number of theories on the frame stack W with respect to which the constructs are defined. This differentiation was foreshadowed by the way in which we specified the simplification function $(M\alpha)^n$ (see Definition 8). Unless otherwise specified, in the following we concede that the frame stack contains a single theory T together with its context Γ.

To start with, note that theory graphs G are intepreted as classes, containing signature models, while the interpretation of stacks W is given by sets of pairings, collecting signature and context models.

The reflection type $\uparrow_S A$ is interpreted as the $Terms/\cong$ quotient construction of the set $Terms$, collecting all well-formed terms of type A, with respect to the congruence relation given by syntactic equality.

The semantics of a $\lceil t \rceil_S$ quoted term t consists of the congruence class, whose representative is $(M\alpha)^1(t)$. Intuitively, a reflected term is, thus, interpreted as the set of all terms that are syntactically equivalent with the "reduction" of t through $(M\alpha)^1$.

Considering the evaluation $\lfloor q \rfloor_S$ of a term q, the interpretation function picks a representative from the semantics of q. Note that this implies that the set-theory used to specify the semantics of our reflective type theory has to contain the axiom of choice.

Finally, for the elimination q^σ of a reflected term q via a morphism σ, we define the semantics as the model-application. Specifically, a model of σ is applied to the result obtained by interpretating a representative r from the semantics of q.

Looking at these cases we see that, with the exception of the last one for term elimination, all the other ones can be unified into a single rule. In particular, we can state that the interpretation of a term t with respect to an arbitrary frame stack of size n is given by the equivalence class represented by $(M\alpha)^n$.

Syntax			Semantics
	$\vdash$	G `TGph`	$[\![\vdash G]\!] \in$ `Class`
	$\vdash^G$	W `Stack`	$[\![\vdash^G W]\!] \in$ `Set`
	$\vdash^G$	Σ `Sig`	$[\![\vdash^G \Sigma]\!] \in$ `Class`
W	$\vdash^G$	$\sigma : S$	$[\![W \vdash^G \sigma]\!] : [\![\vdash^G W]\!] \to [\![\vdash^G S]\!]$
G	$::=$	$\cdot \mid G, S = \{\Sigma\}$	$[\![\vdash \cdot]\!] = \emptyset \mid [\![\vdash G, S = \{\Sigma\}\ \texttt{TGph}]\!] = [\![\vdash G]\!] \cup \{(S, [\![\vdash^G \Sigma]\!])\}$
W	$::=$	$\cdot \mid W, (S, \Gamma)$	$[\![\vdash^G \cdot]\!] = \emptyset \mid [\![\vdash^G W, (S, \Gamma)\ \texttt{Stack}]\!] = \{(M, \alpha) \mid M \in [\![\vdash^G \Sigma]\!], \alpha \in [\![\vdash^G \Gamma]\!]^M\}$
Σ	$::=$	$\cdot \mid \Sigma, c : A$	$[\![\vdash^G \cdot]\!] = \{\emptyset\} \mid [\![\vdash^G \Sigma, c : A\ \texttt{Sig}]\!] = \{M \cup \{(c, u)\} \mid , M \in [\![\vdash^G \Sigma]\!], u \in [\![\vdash^G A]\!]^M\}$
σ	$::=$	$\cdot \mid \sigma, c/E$	$[\![W \vdash^G \cdot]\!] : (\emptyset, \emptyset \mapsto \emptyset) \mapsto \emptyset \mid [\![W \vdash^G \sigma, c/E]\!] : (M, \alpha) \mapsto [\![W \vdash^G \sigma]\!] (M, \alpha) \cup \{(c, [\![W \vdash^G E]\!])\}$
For a **fixed** $W = (S, \cdot)$, where $S = \{\Sigma\}$, and for a **fixed** model $M \in [\![\vdash^G \Sigma]\!]$, s.t $(M, \cdot) \in [\![\vdash^G W]\!]$:			
W	$\vdash^G$	Γ `Ctx`	$[\![W \vdash^G \Gamma]\!]^M \in$ `Set`
W	$\vdash^G$	$\gamma : \Gamma$	$[\![W \vdash^G \gamma]\!]^M : \{\emptyset\} \to [\![W \vdash^G \Gamma]\!]^M$
Γ	$::=$	$\cdot \mid \Gamma, x : A$	$[\![W \vdash^G \cdot]\!]^M = \{\emptyset\} \mid [\![W \vdash^G \Gamma, x : A]\!]^M = \{\alpha \cup \{(x, u)\} \mid \alpha \in [\![W \vdash^G \Gamma]\!]^M, u \in [\![W \vdash^G A]\!]^M\}$
γ	$::=$	$\cdot \mid \gamma, x/t$	$[\![W \vdash^G \cdot]\!]^M : \emptyset \mapsto \emptyset \mid [\![W \vdash^G \gamma, x/t]\!]^M : \alpha \mapsto ([\![W \vdash^G \gamma]\!]^M (\alpha) \cup \{(x, [\![W \vdash^G t]\!]^{M,\alpha})\})$
For a **fixed** $W = W_0, (S, \Gamma)$, where $S = \{\Sigma\}$ and Γ `Ctx`, and for a **fixed** assignment $\alpha \in [\![W \vdash^G \Gamma]\!]^M$, s.t $(M, \alpha) \in [\![\vdash^G W]\!]$:			
W	$\vdash^G$	$t : A$	$[\![W \vdash^G t]\!]^{M,\alpha} \in [\![W \vdash^G A]\!]^{M,\alpha}$
E	$::=$	$c \mid x \mid$ `type`	$[\![W \vdash^G c]\!]^{M,\alpha} = M(c) \mid [\![W \vdash^G x]\!]^{M,\alpha} = \alpha(x) \mid [\![W \vdash^G \texttt{type}]\!]^{M,\alpha} =$ `Set`
	$\mid$	$\uparrow_S A$	$[\![(T, \Gamma) \vdash^G \uparrow_S A]\!]^{M,\alpha} = Terms/\cong$
	$\mid$	$\lceil t \rceil_S$	$[\![(T, \Gamma) \vdash^G \lceil t \rceil_S]\!]^{M,\alpha} = [(M\alpha)^1(t)]^\cong$
	$\mid$	$\lfloor q \rfloor_S$	$[\![(T, \Gamma) \vdash^G \lfloor q \rfloor_S]\!]^{M,\alpha} = r$, where $r \in [\![(S, \cdot) \vdash^G q]\!]^{M,\alpha}$
	$\mid$	q^σ	$[\![(T, \Gamma) \vdash^G q^\sigma]\!]^{M,\alpha} = [\![(T, \Gamma) \vdash^G r]\!]^{[\![(S, \cdot) \vdash^G \sigma]\!]^{M,\alpha}}$, where $r \in [\![(S, \cdot) \vdash^G q]\!]^{M,\alpha}$
			$[\![(T, \Gamma), W \vdash^G t]\!]^{M,\alpha} = [(M\alpha)^n(t)]^\cong$, where $\lvert W \rvert = n$

Table 4.4: Interpretation Function with Term Reflection

4.4 Soundness

In this section, we prove the *Soundness Theorem* 4.4.3 for the semantics defined in Section 4.3. As auxiliary results, we establish the *Substitution Lemma* 4.4.1 and *Semantic Well-Definedness* 4.4.2.

Lemma 4.4.1 relates the value of obtained by applying a morphism model to the interpretation of a term with that obtained by interpreting the morphism application to the term.

Lemma 4.4.1. *Substitution Lemma*

Let us assume $\vdash^G W$, $(M, \alpha) \in [\![\vdash^G W]\!]$ *and* $W >^G S$. *Then, it holds that:*

$$[\![(S, \cdot) \vdash^G t]\!]^{[\![W \vdash^G \sigma : S]\!]^{M,\alpha}_{,\emptyset}} = [\![W \vdash^G \sigma(t)]\!]^{M,\alpha} \tag{4.77}$$

Proof. The proof follows by structural induction on the term t. We only prove the case for term reflection, the other ones being analogous.

- $t ::= \lceil t \rceil_S$

$$\begin{aligned}
[\![W \vdash^G t^\sigma]\!] &= [\![W \vdash^G \sigma(t)]\!] \\
&= [\![W \vdash^G \lceil t' \rceil_S^\sigma]\!] \\
&= [\![W \vdash^G r^\sigma]\!] \text{, where } r \in [\![W \vdash^G \lceil t' \rceil]\!] \\
&= [\![W \vdash^G r^\sigma]\!] \text{, where } r \in [t']^{\cong} \\
&= [\![W \vdash^G t'^\sigma]\!] \\
&= [\![W \vdash^G \sigma(t')]\!] \text{, by applying the Induction Hypothesis}
\end{aligned} \tag{4.78}$$

□

Next, Lemma 4.4.2 states that the *well-formedness* of a term t of type A with respect to a frame stack W is the same as the membership of the interpretation t in the class given by the interpretation of A, i.e *inhabitation* of the type's interpretation.

Lemma 4.4.2. *Semantic Well-Definedness.*

Let us assume $\vdash^G W$ *and* $(M, \alpha) \in [\![\vdash^G W]\!]$. *If:*

$$W \vdash^G t : A \tag{4.79}$$

Then:

$$[\![W \vdash^G t]\!]^{M,\alpha} \in [\![W \vdash^G A]\!]^{M,\alpha} \tag{4.80}$$

Proof. The proof proceeds by case analysis on the term t.

- $t ::= \uparrow_S A$

From the corresponding definitions:

$$\begin{aligned} [\![W \vdash^G t]\!]^{M,\alpha} &::= [\![W \vdash^G \uparrow_S A]\!]^{M,\alpha} = \{t \mid (S,\cdot) \vdash^G t : A\}/\cong \\ [\![W \vdash^G A]\!]^{M,\alpha} &::= [\![W \vdash^G \texttt{type}]\!]^{M,\alpha} = \texttt{Set} \end{aligned} \tag{4.81}$$

Since $\{t \mid (S,\cdot) \vdash^G t : A\}/\cong \in \texttt{Set}$, it follows that:

$$[\![W \vdash^G t]\!]^{M,\alpha} \in [\![W \vdash^G A]\!]^{M,\alpha}$$

- $t ::= \lceil t \rceil_S$

We know, from the $\uparrow_I$ **Introduction Rule**:

$$W, (S,\cdot) \vdash^G t : A \tag{4.82}$$

$$W >^G S \tag{4.83}$$

Applying the Well-Definedness Theorem for $(M\alpha)^n$ (see Theorem 4.3) to 4.82:

$$(S,\cdot) \vdash^G (M\alpha)^1(t) : (M\alpha)^1(A)$$

Looking at the theory ordering in 4.83, we see that theory S cannot quote anything from W. Since A only uses symbols from S and, since it does not contain any evaluations, it remains invariant with respect to applying $(M\alpha)^n$: $(M\alpha)^1(A)$.

Hence, $(S,\cdot) \vdash^G (M\alpha)^1(t) : A$ and $(M\alpha)^1(t) \in \{t \mid (S,\cdot) \vdash^G t : A\}$. Trivially:

$$[(M\alpha)^1(t)]^{\cong} \in \{t \mid (S,\cdot) \vdash^G t : A\}/\cong$$

We can conclude:

$$[\![W \vdash^G t]\!]^{M,\alpha} \in [\![W \vdash^G A]\!]^{M,\alpha}$$

- $t ::= \lfloor t \rfloor_S$

$$[\![(T,\Gamma) \vdash^G t]\!]^{M,\alpha} ::= [\![(T,\Gamma) \vdash^G \lfloor t \rfloor_S]\!]^{M,\alpha} = r, \text{ where } r \in [\![(S,\cdot) \vdash^G t]\!]$$

- $t ::= t^\tau$

$$\begin{aligned} [\![W \vdash^G t]\!]^{M,\alpha} &::= [\![W \vdash^G t^\tau]\!]^{M,\alpha} \\ &= [\![W \vdash^G r]\!]^{[\![(S,\cdot) \vdash^G \sigma : S]\!]^{M,\alpha},\emptyset}, r \in [\![(S,\cdot) \vdash^G q]\!]^{M,\alpha} \\ [\![W \vdash^G A]\!]^{M,\alpha} &::= [\![W \vdash^G \sigma(A)]\!]^{M,\alpha} \end{aligned}$$

□

Having verified the needed complementary results, we can prove our main result, i.e *soundness*. Accordingly, the syntactic equality of two terms t_1 and t_2 implies the "semantic" equality of their corresponding interpretations $W \vdash^G t_1$ and $W \vdash^G t_2$.

Theorem 4.4.3. *Soundness Lemma*

Let us assume $\vdash^G W$ *and* $(M, \alpha) \in [\![\vdash^G W]\!]$. *Then, if:*

$$W \vdash^G t_1 = t_2 \tag{4.84}$$

It holds that we have:

$$[\![W \vdash^G t_1]\!]^{M,\alpha} = [\![W \vdash^G t_2]\!]^{M,\alpha} \tag{4.85}$$

Proof. Let us proceed by case distinction on the length of the frame stack W.

- $W = W_0, (T, \Gamma)$, where $|W_0| > 0$

According to the definition of the semantics:

$$\begin{aligned} [\![W \vdash^G t_1]\!]^{M,\alpha} &= [t_1]^{\cong} \\ [\![W \vdash^G t_2]\!]^{M,\alpha} &= [t_2]^{\cong} \end{aligned} \tag{4.86}$$

Since $W \vdash^G t_1 = t_2$, then $t_1 \cong t_2$. Consequently, $[t_1]^{\cong} = [t_2]^{\cong}$ and, from 4.86, we have that: $[\![W \vdash^G t_1]\!]^{M,\alpha} = [\![W \vdash^G t_2]\!]^{M,\alpha}$. Hence, soundness holds, in the case of having a non-trivial frame stack.

- $W = (T, \Gamma)$

We proceed by nested structural inductions on the theory graph G and on the derivation of $W \vdash^G t_1 = t_2$.

Graph Base Case: $G = \cdot$

$$[\![W \vdash^G t_1]\!]^{M,\alpha} = \emptyset \tag{4.87}$$

$$[\![W \vdash^G t_2]\!]^{M,\alpha} = \emptyset \tag{4.88}$$

Trivially, $[\![W \vdash^G t_1]\!]^{M,\alpha} = [\![W \vdash^G t_2]\!]^{M,\alpha}$.

Graph Induction Hypothesis: For a well-formed theory graph G_0, soundness holds.

Graph Step Case: $G = G_0, T = \{\Phi\}$

- $U \neq T$. Then, $T = S, G_1$ and $G = G_0, S, G_1$. This can be reduced to the previous case, since the semantics induced by theory graph G_1 is independent of that for G_0, S.
- $U = T$. Then, $G = G_0, S$.

As a result, we continue by structural induction on the terms t_1 and t_2, with respect to the theory graph $G = G_0, S$.

In the following, let us denote $W = W_0, (S, \cdot)$.

- **Congruence for Introduction**, i.e $t_1 ::= \lceil t \rceil_S$ and $t_2 ::= \lceil t' \rceil_S$

  Applying the $=_I^t$ equality rule to:

$$W \vdash^G \lceil t \rceil_S = \lceil t' \rceil_S \tag{4.89}$$

  we have that:

$$W, (S, \cdot) \vdash^{G_0} t = t' \tag{4.90}$$

  Hence

$$(S, \cdot) \vdash^{G_0} (M\alpha)^1(t) = (M\alpha)^1(t') \tag{4.91}$$

  and

$$(M\alpha)^1(t) \cong (M\alpha)^1(t') \tag{4.92}$$

  which leads to:

$$[(M\alpha)^1(t)]^{\cong} = [(M\alpha)^1(t')]^{\cong} \tag{4.93}$$

  The **Graph Induction Hypothesis** together with equality 4.90 leads to:

$$[\![W, (S, \cdot) \vdash^{G_0} t]\!]^{M,\alpha} = [\![W, (S, \cdot) \vdash^{G_0} t']\!]^{M,\alpha} \tag{4.94}$$

  The corresponding semantics for terms t_1 and t_2 yields:

$$[\![W \vdash^G t_1]\!]^{M,\alpha} = [\![W \vdash^G \lceil t \rceil_S]\!]^{M,\alpha} \overset{Def}{=} [(M\alpha)^1(t)]^{\cong} \tag{4.95}$$

$$[\![W \vdash^G t_2]\!]^{M,\alpha} = [\![W \vdash^G \lceil t' \rceil_S]\!]^{M,\alpha} \overset{Def}{=} [(M\alpha)^1(t')]^{\cong} \tag{4.96}$$

  From equalities 4.95 and 4.96, as well as 4.93, we can conclude that 4.85 holds.

- **Congruence for Elimination**, i.e $t_1 ::= q^\sigma$ and $t_2 ::= q'^\sigma$

  Applying the $=^t_{elim}$ (see Table **??**) to:

$$W \vdash^G q^\sigma = q'^\sigma \tag{4.97}$$

  we have that:

$$W \vdash^G q = q' \tag{4.98}$$

  According to the **Term Induction Hypothesis**, we get:

$$[\![W \vdash^G q]\!]^{M,\alpha} = [\![W \vdash^G q']\!]^{M,\alpha} \tag{4.99}$$

  Also, from the corresponding interpretation function rule (see Table 4.4):

$$\begin{aligned} [\![W \vdash^G t_1]\!]^{M,\alpha} &= [\![W \vdash^G q^\sigma]\!]^{M,\alpha} \\ &= [\![W \vdash^G r]\!]^{[\![W \vdash^G \sigma : S]\!]^{M,\alpha}, \emptyset}, \text{ where } r \in [\![W \vdash^G q]\!]^{M,\alpha} \end{aligned} \tag{4.100}$$

$$\begin{aligned} [\![W \vdash^G t_2]\!]^{M,\alpha} &= [\![W \vdash^G q'^\sigma]\!]^{M,\alpha} \\ &= [\![W \vdash^G r']\!]^{[\![W \vdash^G \sigma : S]\!]^{M,\alpha}, \emptyset}, \text{ where } r' \in [\![W \vdash^G q']\!]^{M,\alpha} \\ &= [\![W \vdash^G r']\!]^{[\![W \vdash^G \sigma : S]\!]^{M,\alpha}, \emptyset}, \text{ where } r' \in [\![W \vdash^G q]\!]^{M,\alpha} \text{ (4.99)} \\ &= [\![W \vdash^G r]\!]^{[\![W \vdash^G \sigma : S]\!]^{M,\alpha}, \emptyset}, \text{ by picking } r' = r \end{aligned} \tag{4.101}$$

  Given 4.101, we can conclude that equality 4.85 holds.

- **Computation**, i.e $t_1 ::= \lceil t \rceil^\sigma_S$ and $t_2 ::= \sigma(t')$

  Applying the $=^t_{comp}$ equality rule to $W \vdash^G \lceil t \rceil^\sigma_S = \sigma(t')$, we obtain:

$$W \vdash^G t = t' \tag{4.102}$$

  Hence:

$$W \vdash^G (M\alpha)^1(t) = (M\alpha)^1(t') \tag{4.103}$$

  Also, given the **Term Induction Hypothesis**, we get:

$$[\![W \vdash^G t]\!]^{M,\alpha} = [\![W \vdash^G t']\!]^{M,\alpha} \tag{4.104}$$

$$
\begin{aligned}
\llbracket W \vdash^G t_1 \rrbracket^{M,\alpha} &= \llbracket W \vdash^G \lceil t \rceil_S^{\sigma} \rrbracket^{M,\alpha} \\
&= \llbracket (S,\cdot) \vdash^G r \rrbracket^{M,\alpha} \left[\!\left[W \vdash^G \sigma : S \right]\!\right]^{M,\alpha}_{,\emptyset}, \text{ for some } r \in \llbracket W \vdash^G \lceil t \rceil_S \rrbracket^{M,\alpha} \\
&= \llbracket (S,\cdot) \vdash^G r \rrbracket^{M,\alpha} \left[\!\left[W \vdash^G \sigma : S \right]\!\right]^{M,\alpha}_{,\emptyset}, \text{ for some } r \in [(M\alpha)^1(t)]^{\cong} \\
&= \llbracket (S,\cdot) \vdash^G (M\alpha)^1(t) \rrbracket^{M,\alpha} \left[\!\left[W \vdash^G \sigma : S \right]\!\right]^{M,\alpha}_{,\emptyset}, \text{ by chosing } r = (M\alpha)^1(t) \\
&= \llbracket (S,\cdot) \vdash^G \sigma((M\alpha)^1(t)) \rrbracket^{M,\alpha}, \text{ by applying the Substitution Lemma}
\end{aligned} \tag{4.105}
$$

$$
\begin{aligned}
\llbracket W \vdash^G t_2 \rrbracket^{M,\alpha} &= \llbracket W \vdash^G \sigma(t') \rrbracket^{M,\alpha} \\
&= \llbracket W \vdash^G \sigma((M\alpha)^1(t')) \rrbracket^{M,\alpha} \\
&= \llbracket W \vdash^G \sigma((M\alpha)^1(t)) \rrbracket^{M,\alpha}, \text{ from 4.103}
\end{aligned} \tag{4.106}
$$

Applying the **Substitution Lemma** to 4.106, it follows that 4.85 holds.

- **Soundness**, i.e $t_1 ::= \lfloor \lceil t \rceil_S \rfloor_S$ and $t_2 ::= t'$

$$
\begin{aligned}
\llbracket W \vdash^G t_1 \rrbracket^{M,\alpha} &= \llbracket W \vdash^G \lfloor \lceil t \rceil_S \rfloor_S \rrbracket^{M,\alpha} \\
&\overset{Def}{=} \llbracket W \vdash^G \lceil t \rceil_S \rrbracket^{M,\alpha} \\
&= [(M\alpha)^1(t)]^{\cong}
\end{aligned} \tag{4.107}
$$

$$
\llbracket W \vdash^G t_2 \rrbracket^{M,\alpha} = \llbracket W \vdash^G t \rrbracket^{M,\alpha} \tag{4.108}
$$

- **Completeness**, i.e $t_1 ::= \lceil \lfloor q \rfloor_S \rceil_S$ and $t_2 ::= q'$

Applying the $=^t_{compl}$ equality rule to:

$$
W \vdash^G \lceil \lfloor q \rfloor_S \rceil_S = q' \tag{4.109}
$$

we get:

$$
W \vdash^G q = q' \tag{4.110}
$$

Hence, according to the **Term Induction Hypothesis**:

$$
\llbracket W \vdash^G q \rrbracket^{M,\alpha} = \llbracket W \vdash^G q' \rrbracket^{M,\alpha} \tag{4.111}
$$

$$
\begin{aligned}
\llbracket W \vdash^G t_1 \rrbracket^{M,\alpha} &= \llbracket W \vdash^G \lceil \lfloor q \rfloor_S \rceil_S \rrbracket^{M,\alpha} \\
&\overset{Def}{=} [(M\alpha)^1(\lfloor q \rfloor_S)]^{\cong} \\
&\overset{Def}{=} [r]^{\cong}, \text{ where } r \in \llbracket W \vdash^G q \rrbracket^{M,\alpha} \\
&= \llbracket W \vdash^G q \rrbracket^{M,\alpha} \\
&= \llbracket W \vdash^G q' \rrbracket^{M,\alpha}, \text{ from equality 4.111} \\
&= \llbracket W \vdash^G t_2 \rrbracket^{M,\alpha}
\end{aligned}
\tag{4.112}
$$

- **Extensionality**, i.e $t_1 ::= q$ and $t_2 ::= q'$

  Applying the $=^t_{ext}$ equality rule (see Table **??**) to

  $$W \vdash^G q = q' \tag{4.113}$$

  we get:

  $$W, (S, \cdot) \vdash^{G_0} \lfloor q \rfloor_S = \lfloor q' \rfloor_S \tag{4.114}$$

  Using the **Graph Induction Hypothesis** together with 4.114, it follows that:

  $$\llbracket W, (S, \cdot) \vdash^{G_0} \lfloor q \rfloor_S \rrbracket^{M,\alpha} = \llbracket W, (S, \cdot) \vdash^{G_0} \lfloor q' \rfloor_S \rrbracket^{M,\alpha} \tag{4.115}$$

  We know that:

  $$
  \begin{aligned}
  \llbracket W, (S, \cdot) \vdash^{G_0} \lfloor q \rfloor_S \rrbracket^{M,\alpha} &= \llbracket W \vdash^G q \rrbracket^{M,\alpha} \\
  \llbracket W, (S, \cdot) \vdash^{G_0} \lfloor q' \rfloor_S \rrbracket^{M,\alpha} &= \llbracket W \vdash^G q' \rrbracket^{M,\alpha}
  \end{aligned}
  \tag{4.116}
  $$

  From 4.114 and 4.116, it can be derived that $\llbracket W \vdash^G q \rrbracket = \llbracket W \vdash^G q' \rrbracket$.
  Hence, $\llbracket W \vdash^G t_1 \rrbracket^{M,\alpha} = \llbracket W \vdash^G t_2 \rrbracket^{M,\alpha}$, QED.

□

Chapter 5

Towards a General Theory of Reflection

5.1 Reflecting Morphisms

In keeping with the type theoretical extension for term reflection, let us now consider the case for morphism reflection. As previously, we begin by identifying the needed primitives of our extended language.

At the object-level, additional expressions are obtained through the *type formation* of a reflected type $\downarrow S$ inhabited by all reflected reflected morphisms mapping out of a named signature S, through the *quotation* $\lceil\sigma\rceil_S$ of morphism σ with domain S, by the *evaluation* $\lfloor m\rfloor_S$ of a reflected morphism m from S and through the *elimination* t^m of a term t via a quoted morphism m (see Table 5.1).

	Grammar	Typing Judgment	Equality Judgment
Theory graphs (G)	$G ::= \cdot \mid G,\ \text{sig}\ S = \{\Sigma\}$	$\vdash\ G\ \texttt{TGph}$	$\vdash\ G = G'$
Frame stacks (W)	$W ::= \cdot \mid W,\ (S, \Gamma)$	$\vdash^G W\ \texttt{Stack}$	$\vdash^G W = W'$
Signatures (Σ)	$\Sigma ::= \cdot \mid \Sigma,\ c : E \mid S$	$\vdash^G \Sigma\ \texttt{Sig}$	$\vdash^G \Sigma = \Sigma'$
Morphisms (σ)	$\sigma ::= \cdot \mid \sigma,\ c/E$	$W \vdash^G \sigma : S$	$W \vdash^G \sigma = \sigma' : S$
Contexts (Γ)	$\Gamma ::= \cdot \mid \Gamma,\ x : E$	$W \vdash^G \Gamma\ \texttt{Ctx}$	$W \vdash^G \Gamma = \Gamma'$
Substitutions (γ)	$\gamma ::= \cdot \mid \gamma,\ x/E$	$W \vdash^G \gamma : \Gamma$	$W \vdash^G \gamma = \gamma' : \Gamma$
Expressions (E, A, t, σ, m) Reflected morphisms (m)	$E ::= c \mid x \mid \texttt{type} \mid E \to E \mid E\ E \mid [E]\ E \mid \downarrow S \mid \lceil\sigma\rceil_S \mid \lfloor m\rfloor_S \mid t^m$	$W \vdash^G E : E'$	$W \vdash^G E = E'$

Table 5.1: Grammar, Typing and Equality Judgments for a Type Theory with Morphism Reflection

We proceed to briefly discuss the well-formedness inference rules corresponding to the new constructs introduced above. The *typing rules* for morphism reflection are summarized in Table 5.2, while an overview of the *equality rules* is given in Table 5.3.

First, let us give an account of the *typing rules*, starting from the $\downarrow_{typ}$ rule for the *type formation* of the reflected type corresponding to quoted morphisms mapping from a

theory S. This $\downarrow S$ type is well-formed with respect to a stack W and of base type `type`, if the theory S belongs to the theory graph G.

Next, we have the *introduction* rule $\downarrow_I$ for the type $\downarrow S$ of reflected morphisms mapping from S. The rule states that the quotation $\lceil\sigma\rceil_S$ of a morphism σ over a named signature S is well-formed with respect to a stack W (and of reflected type $\downarrow S$), if the morphism σ itself is well-formed with respect to the extended stack $W, (S, \cdot)$ and maps from S.

Conversely, the *evaluation* rule $\downarrow_{ev}$ states that the evaluation of a quoted morphism m under a named signature S is well-formed with respect to the extended stack $W, (S, \cdot)$ and has domain S, if m is well-formed under W and has the reflected type $\downarrow S$. Note that, when type checking evaluations, the top-level frame recording the quoted theory and its context is popped from the stack. Thus, in the implementation, it is necessary to perform an additional check against the frame stack being empty (see Section 6).

Finally, in the $\downarrow_E$ rule, we have that the term t^m obtained by eliminating a term t by a morphism m is well-formed with respect to a frame stack W, if the morphism m is well-formed and of reflection type and if the term t is well-formed of type A in signature A. The corresponding type for the elimination term is given by the applying the evaluation of m to type A.

Type formation	$\dfrac{S \in \mathrm{dom}(G)}{W \vdash^{\mathrm{G}} \downarrow S : \mathtt{type}} \downarrow_{typ}$
Introduction (reflection)	$\dfrac{W, (S, \cdot) \vdash^G \sigma : S}{W \vdash^G \lceil\sigma\rceil_S : \downarrow S} \downarrow_I$
Evaluation	$\dfrac{W \vdash^G m : \downarrow S}{W, (S, \cdot) \vdash^G \lfloor m \rfloor_S : S} \downarrow_{ev}$
Elimination (induction)	$\dfrac{W \vdash^G m : \downarrow S \quad \vdash_S t : A}{W \vdash^G t^m : \lfloor m \rfloor_S(A)} \downarrow_E$

Table 5.2: Inference System for Basic Type Theory with Morphism Reflection

From Table 5.2, we can notice the $=^m_I$ *introduction* and $=^m_E$ *elimination* rules for reflected morphisms express the preservation of morphism equality under the application of quotation and, respectively, elimination via a morphism.

The $=^m_C$ *computation* rule states that the term $t^{\lceil\sigma\rceil_S}$, obtained by eliminating a term t with the quotation $\lceil\sigma\rceil_S$ of a morphism σ, is equal to that obtained by applying the meta-level morphism σ to the term t, with respect to the frame W, given that, in turn, σ is well-formed with respect to W.

Next, the $=^t_{sound}$ rule states that a morphism is preserved via quoting and then evaluating the result over the same theory S, if the morphism is well-formed and maps from S. In other words, according to this soundness rule, quotation and evaluation are partially inverse operations.

Analogous to the soundness rule, the $=^t_{compl}$ *completeness* rule asserts that, by first evaluating a reflected morphism m under a theory S and then reflecting it over the same theory, the morphism is preserved. Together with the previous result, this introduces quotation and evaluation as inverse to each other.

Finally, conforming to the $=^t_{ext}$ *extensionality* reflection rule, equality between two well-formed quoted morphisms m and m', with respect to a frame W is preserved by evaluation under a theory S with respect to the extended frame $W, (S, \cdot)$.

Reflecting...	...morphisms
Introduction	$\dfrac{W, (S,\cdot) \vdash^{G_0} \sigma = \sigma' : S}{W \vdash^{G} \lceil\sigma\rceil_S = \lceil\sigma'\rceil_S} =^m_I$
Elimination	$\dfrac{W \vdash^{G} t = t'}{W \vdash^{G} t^m = t'^m} =^m_E$
Computation	$\dfrac{W \vdash^{G} \sigma : S}{W \vdash^{G} t^{\lceil\sigma\rceil_S} = \sigma(t)} =^m_C$
Soundness	$\dfrac{W, (S,\cdot) \vdash^{G_0} \sigma : S}{W, (S,\cdot) \vdash^{G_0} \lfloor\lceil\sigma\rceil_S\rfloor_S = \sigma} =^m_{sound}$
Completeness	$\dfrac{W \vdash^{G} m : S}{W \vdash^{G} \lceil\lfloor m\rfloor_S\rceil_S = m} =^m_{compl}$
Extensionality	$\dfrac{W, (S,\cdot) \vdash^{G_0} \lfloor m\rfloor_S = \lfloor m'\rfloor_S : S}{W \vdash^{G} m = m'} =^m_{ext}$

Table 5.3: Equality Inference Rules with Morphism Reflection

Example 4. Ring of Polynomials.

The formalization of the type of polynomial rings `polyType` provides us with a very interesting example, in that it combines the use of both inductive types (reflected terms over a closed signature) and dependent record types (reflected morphisms out of the type of rings). Moreover, it displays an instance where one actually reflects not over a signature, but over the result of applying an operation on it, namely evaluation.

We encode the type of rings using the named signature `Ring`, which contains the generic universe `ring_univ` for rings, the neutral elements `0` and `1`, binary operations corresponding to addition (`+`), multiplication (`*`) and the unary inverse operation (`-`).

```
Ring = {

  ring_univ : type.
            0 : ring_univ.
            1 : ring_univ.
            + : ring_univ → ring_univ → ring_univ.
            * : ring_univ → ring_univ → ring_univ.
            - : ring_univ → ring_univ.
```

```
}.
```

One method of encoding polynomials relies on their Horn normal form, as given in the signature Σ. Consequently, starting from a‘ universe type of polynomials `poly_univ`, we code them inductively using the constructors `Pc`, for constant polynomials, and `Px`, for univariate polynomials. The former transforms constants from the ring universe `r.ring_univ` into constant polynomials in `poly_univ`. The latter takes a polynomial P in `poly_univ`, a constant c from the ring universe `r.ring_univ` and outputs a univariate polynomial $P * X + c$ in `poly_univ`. Next, in order to obtain the signature for polynomial rings, we reflect Σ and parameterize it over ring models `r`, which are reflected morphisms of type $\downarrow$`Ring`. The corresponding type of `polySig` is that of a model functor going from the type $\downarrow$`Ring` of models of `Ring` to an extension $\vec{.}$ of the empty signature.

```
polySig : ↓Ring → .⃗ = [ r :↓Ring ] ⌈ Σ ⌉
(where Σ = poly_univ : type.                                    % poly universe
           Pc : r.ring_univ → poly_univ.                        % constant poly
           Px : poly_univ → r.ring_univ → poly_univ. % univariate
                poly
           ).

polyType = [ r :↓Ring ] ↑⌊polySig r⌋ poly_univ.
```

The type of polynomial rings `polyType` [1] is intuitively the type of the reflected terms of type `poly_univ` over the evaluation ⌊`polySig` r⌋ of the signature `polySig`. In order to allow instantiations and, hence, a higher level of generality, this type is explicitly made dependent on a reflected morphism of type $\downarrow$`Ring`, the type corresponding to the reflected morphisms out of `Ring`. Due to this ***parameterization***, one can substitute the underlying ring with any particular model of `Ring`. A potential drawback of this design choice is that it makes type reconstruction harder.

Another method for specifying the type of polynomials is to consider the actual terms and not their normalization. To this extent, we build a signature Σ', by including the declarations from the signature `Ring`, a constructor for constant polynomials `Pc` as before, and by making the variables `X` primitive constants in `poly_univ`. This matches to the way in which mathematicians would think of variables in this setting. Indeed, since these variables are globally declared and not bound, they behave similarly to constants. Notice that, in this case, the signature Σ' is an extension of that of `Ring`.

```

polySig : ↓Ring → ⟶Ring = [ r :↓Ring ] ⌈ Σ′ ⌉
                  (where Σ′ = ⌊ Ring ⌋
                         Pc : r.ring_univ → poly_univ. % constant poly
                         X  : poly_univ.               % constant var
                         ).

polyType = [ r :↓Ring ] ↑⌊polySig r⌋ poly_univ.
```

[1] This is not really the case, as one has to actually take the quotient type over the congruences generated.

The type of `polyType` is the same as above, the intuition now being that one takes the set of polynomials extending that of rings, adds a special construct for variables and then reflects the terms of type polynomial `poly_univ` over the the evaluation $\lfloor$`polySig r`$\rfloor$ of `polySig`.

At this point we emphasize that one has to comet to either treating variables as constants, as has been done in the previous example, or as proper variables, which we argue is a more elegant solution, since it facilitates α-renaming. According to the latter viewpoint, we get a signature Σ for polynomials that no longer has variables as primitives. Also, in the encoding of the type of polynomial rings, instead of reflecting polynomial terms of type `poly_univ`, we reflect polynomial terms with free variables of type Πx : `poly_univ.poly_univ`.

```
1
  polySig : ↓Ring → ⟶Ring = [ r :↓Ring ] ⌈ Σ'' ⌉
                    (where Σ'' = ⌊ Ring ⌋
                           Pc : r.ring_univ → poly_univ.).

6 polyType = λr : ↓Ring. ↑⌊polySig r⌋ ({x : poly_univ} poly_univ).
```

Example 5. Model Theory.

As mentioned in the introductory part, **models** and **model functors** appear often as part of routine mathematical practice. The key observation is that, taking a signature S, which can represent, for instance, a mathematical or logical theory, models of S can be seen as *formed from reflected morphisms* out of the signature and as *related via model functors*.

In the following, we exemplify these two concepts based on the algebraic formalizations of monoids, groups, rings and polynomials.

Models. We begin by examining the monoid specification `Monoid`, containing a set `i`, the generic (universe) type, which closed under a binary operation (composition) $\circ$ with identity (neutral element) `e`.

```
    Monoid = {
3
        i : type.
        e : i.
        ∘ : i → i → i.

8         }.
```

Next, we illustrate the formalization of a monoidal model, formed as a free object, essentially a "generic" algebraic structure, generated by the one-element set `X`. The free monoid model `Monoid(X)`, parameterized by `X`, is encoded via a reflected morphism σ mapping out of `Monoid`. This morphism assigns to the universe type `i`, the type $\uparrow_{\texttt{Monoid}}$ ({`x : i`} `i`) of reflected terms with a free variable, type which we will abbreviate with `u`. The neutral element in the free construction is obtained by reflecting the neutral element built using the variable `x`. The composition in `Monoid(X)` takes two elements of the free monoid and outputs the reflected term obtained by evaluating the elements terms back to `Monoid(X)`, constructing corresponding ones through the variable `x` and using $\circ$ to combine them.

```
Monoid(X) : ↓Monoid
          = ⌈ σ : Monoid ⌉.
( where σ = { i ↦ u = ↑Monoid ({x : i} i).
              e ↦ ⌈ [x : i] e ⌉Monoid.
              ∘ ↦ [a : u] [b : u] ⌈ [x : i] (⌊a⌋Monoid x) ∘ (⌊b⌋Monoid x)) ⌉Monoid
            }.)
```

One can draw an analogy with the previous example and view `Monoid(X)` as the type of polynomials over a monoid.

Model Functors. In addition to being able to form models out of signatures as we have seen above, one can also formalize the various connections between them through model functors. Moreover, in this context, it is interesting to explore how one can handle the meta-theories or foundations of our models and their relations.

Note that one can base an encoding `S` on a given foundation `F`, by including the latter in `S`, essentially considering the signature an extension of `F`, whose type is $\vec{\mathtt{F}}$ (see Section5.2). As shown in Figure 5.1, the signature morphism $\sigma : \mathtt{S} \to \mathtt{F}$, whose reflection $\lceil\sigma\rceil : \downarrow \mathtt{S}$ forms a model of `S`, is basically a retraction (left inverse) of the inclusion morphism from the foundation into the signature.

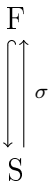


Figure 5.1: Relating Signatures and their Foundation

As mentioned in the introduction (see Section 1), ZFC is the standard set-theoretical foundation for formalized mathematics. We model the theory as a signature that contains the declarations for sets, propositions and truth judgements, as well as corresponding ones for union, intersection, pairs, products, implication, with predicates for set membership and pairings. Typing of sets can be realized by applying the `Elem` operator and elements of a given set are represented as pairs containing the element and a proof of its membership in the set. These can be extracted using the projections `which` and `why`. Consequently, the type of elements of a set is a dependent sum type. We also give ourselves application and lambda operators.

```
ZFC = {

  set : type.
  prop : type.
  ded : prop → type.
  ∈    : set → set → prop.
  ⋃    : set → set → set = ...
  ⋂    : set → set → set = ...
  pair : set → set → set = ...
  Π1   : set → set = ...
```

```
    Π₂    : set → set = ...
  isPair: set → set → set → prop = ...
   prod : set → set → set = ...
   ⇒    :  set → set → set = ...

   Elem : set → type.
   elem : {x : set} ded x ∈ A → Elem A.
   which: Elem A → set.
   why : {x : Elem A} ded x (which x) ∈ A.
   @    : Elem (A ⇒ B) → Elem A → Elem B = ...
   λ    : (Elem A → Elem B) → Elem (A ⇒ B) = ...

}
```

Having ZFC as a foundation, we can encode monoids and groups in a straightforward manner.

```
Monoid = {
      ZFC.
      e : Elem univ.
      ∘ : Elem univ → Elem univ → Elem univ.
      }.

Group = {
      ZFC.
      e : Elem univ.
      inv : Elem univ → Elem univ.
      ∘ : Elem univ → Elem univ → Elem univ.
      }.
```

A model functor between models of `Monoid` and `Group` is given by `unitgroup`, as shown in Figure 5.2.

Starting from a model `m` of `Monoid`, one can define the functor as a reflected morphism σ out of Group. The morphism maps the type of group elements to that of the elements of a set containing all invertible monoid elements `u`. For ease of readability, we abbreviate that set with `q`. Next, the neutral element `e` is mapped to the type obtained through the application of the operator `elem` to the corresponding monoid neutral element `m.e` and to a proof `proof_e` that `m.e` belongs to `q`. We will omit the details of such proof. The type value for `circ` arises by taking two objects in `Monoid`, projecting the corresponding elements to get to the term (`which x`) `m.∘` (`which y`) and applying `elem` to it and to the proof that it is part of q (closure under composition). Given an element in `Monoid`, we obtain the type value for the inverse in `Group`, by applying `elem` to the monoid element satisfying the inverse property and to a proof it belongs to `q`.

```
unitgroup : ↓Monoid → ↓Group
          = [m :↓Monoid] ⌈ σ : Group ⌉
( where σ =
      { i ↦ Elem q.
        e ↦ elem m.e pf_e.
        ∘ ↦ [x : Elem q] [y : Elem q] elem ((which x) m.∘ (which y)) pf_comp
          .
      inv ↦ [x : Elem q]
```

```
            elem (the (x' : m.i) (x' m.∘ x = m.e) ∧ (x m.∘ x' = m.e)) pf_inv
              .
        }.
and q = {u : m.i | ∃ v : m.i. (u m.∘ v) = m.e} )
```

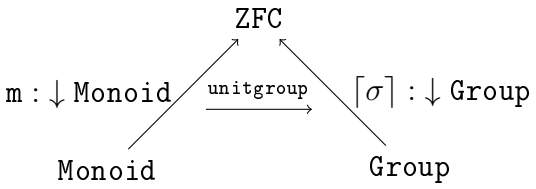
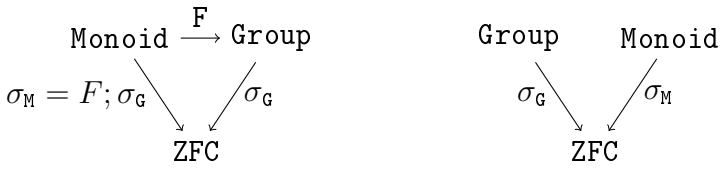


Figure 5.2: Monoid to Group Model Functor

As we have shown above, models can essentially be seen as theory (signature) morphisms. However, we remark that, while every signature morphism can induce a "model morphism" by compositionality, the inverse does not apply. Indeed, considering a signature morphism M : `Monoid` $\to$ `Group` and a model σ_G of `Group` in `ZFC`, one can obtain the model σ_M of `Monoid` in `ZFC`, by composing M and σ_G (see Figure 5.3a). Nonetheless, just by taking two ZFC-models σ_G and σ_M of `Group` and `Monoid`, one cannot determine if there is, in fact, a signature morphism between them. Thus, the need arises to further investigate the ***abstract adequacy properties*** between the syntax and semantics of our types (see Figure 5.3b).



(a) Theory Morphism (b) Model Morphisms

Figure 5.3: Theory and Model Morphisms

5.2 Reflection Perspectives

In building a type theory with reflection, we follow a *constructivist approach*. Namely, starting from a basic ("empty") type theory (see Section 3), in the sense that it contains a bare minimum of type constructors, we seek to iteratively add, as orthogonal features, the *reflection* of its meta-judgements.

An exhaustive theory of reflection in this setting would support six such reflected meta-judgements: three for *signature-based reflection* (the well-formedness of reflected signatures and, respectively, that of terms and morphisms with respect to a signature) and three for *context-based reflection* (the well-formedness of reflected contexts, that of terms with respect to a context and that of substitutions).

Reflecting Signatures Hence, in the setting of a general signature-based reflective theory, apart from reflecting terms and morphisms, we can also think of reflecting signatures. To this end, and in line with the previous syntactical extensions to the basic type theory in Section 3, we introduce the following signature reflection primitives (see Table 5.4).

At the object level, we have the type $\vec{S}$ of extensions of a named signature S by a set of declarations Σ, and the corresponding type of morphisms out of such extensions. One can view an extended signature $T = \Sigma + S$ as containing an inclusion morphism, which copies all declaration from S into the target signature. This signature arises as a pushout construction, as illustrated in Figure 5.4.

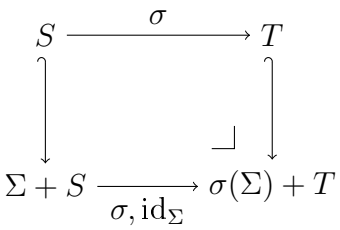


Figure 5.4: Signature Extensions

Next, $\lceil \Sigma \rceil_S$ introduces the reflection of the signature declaration Σ and, conversely, $\lfloor u \rfloor_S$ introduces the evaluation of the named signature declaration S. Note that by allowing the reflection of signatures, we can have signatures which reflect themselves or that contain objects whose type is given by a reflected signature. This may cause our theory to become *impredicative* and, hence, special restriction have to be made, which constitute an open question.

	Grammar	Typing Judgment	Equality Judgment
Theory graphs (G)	$G ::= \cdot \mid G,\ \text{sig}\ S = \{\Sigma\}$	$\vdash\ G\ \ \texttt{TGph}$	$\vdash\ G = G'$
Frame stacks (W)	$W ::= \cdot \mid W,\ (S, \Gamma)$	$\vdash^G W\ \ \texttt{Stack}$	$\vdash^G W = W'$
Signatures (Σ)	$\Sigma ::= \cdot \mid \Sigma,\ c : E \mid S$	$\vdash^G \Sigma\ \ \texttt{Sig}$	$\vdash^G \Sigma = \Sigma'$
Morphisms (σ)	$\sigma ::= \cdot \mid \sigma,\ c/E$	$W \vdash^G \sigma : S$	$W \vdash^G \sigma = \sigma' : S$
Contexts (Γ)	$\Gamma ::= \cdot \mid \Gamma,\ x : E$	$W \vdash^G \Gamma\ \ \texttt{Ctx}$	$W \vdash^G \Gamma = \Gamma'$
Substitutions (γ)	$\gamma ::= \cdot \mid \gamma,\ x/E$	$W \vdash^G \gamma : \Gamma$	$W \vdash^G \gamma = \gamma' : \Gamma$
Expressions (E, u) Reflected signatures (u)	$E ::= c \mid x \mid \texttt{type} \mid E \to E \mid E\ E \mid [E]\ E \mid \vec{S} \mid \lceil \Sigma \rceil_S \mid \lfloor u \rfloor_S \mid u^\sigma$	$W \vdash^G E : E'$	$W \vdash^G E = E'$

Table 5.4: Grammar, Typing and Equality Judgments for a Type Theory with Signature Reflection

Let us provide a succinct description of the inference system for signature reflection described in Table 5.5.

We give an account of the *typing rules*, starting from the $\vec{typ}$ rule for the *type formation* of the reflected type corresponding to the reflection of declarations over a named signature S. This $\vec{S}$ type is well-formed with respect to a stack W and of base type `type`, if S belongs to the theory graph G.

Next, we have the *introduction* rule $\vec{I}$ for the type $\vec{S}$ corresponding to the reflection of declarations with respect to a named signature S. The rule states that the quotation $\lceil\Sigma\rceil_S$ of a list of declarations Σ over a named signature S is well-formed with respect to a stack W (and of reflected type $\vec{S}$), if the signature $S, \lfloor u \rfloor_S$, constructed by adding the declarations in Σ to the list of declarations from S, is well-formed with respect to the theory stack W.

Conversely, the *evaluation* rule $\vec{ev}$ states the required constraint for the well-formedness with respect to a stack W of a signature $S, \lfloor u \rfloor_S$, built through the extension of a named signature S with the evaluation of a quoted list of declarations u under S. In particular, such an extended signature is well-formed, if u is well-formed under W and of reflected type $\vec{S}$.

The $\vec{E}$ rule specifies the needed conditions such that the term u^σ, obtained by eliminating a quoted list of declarations u by a morphism σ, is well-formed with respect to a frame stack W and is of reflected type $\vec{T}$. Specifically, the morphism σ has to be well-formed with respect to $W, (T, \cdot)$, the frame stack extended with the quoted theory T and has to map from the theory S. Also, u has to be well-formed with respect to W and of reflected type $\vec{S}$.

Type formation	$\dfrac{S \in \mathrm{dom}(G)}{W \vdash^G \vec{S} : \mathtt{type}}\ \vec{typ}$
Introduction (reflection)	$\dfrac{W \vdash^G S, \Sigma\ \mathtt{Sig}}{W \vdash^G \lceil\Sigma\rceil_S : \vec{S}}\ \vec{I}$
Evaluation	$\dfrac{W \vdash^G u : \vec{S}}{W \vdash^G S, \lfloor u \rfloor_S\ \mathtt{Sig}}\ \vec{ev}$
Elimination (induction)	$\dfrac{W \vdash^G u : \vec{S} \quad W, (T, \cdot) \vdash^G \sigma : S}{W \vdash^G u^\sigma : \vec{T}}\ \vec{E}$

Table 5.5: Inference System for Basic Type Theory with Signature Reflection

A Unified Theory for Reflection. To conclude, we give an overview of the inference systems for term, morphism and signature reflection in Table 5.6 and remark on the symmetrical form of the rules.

Type formation	$\dfrac{W,(S,\cdot) \vdash^G A : \texttt{type} \quad W >^G S}{W \vdash^G \uparrow_S A : \texttt{type}} \uparrow_{typ}$	$\dfrac{S \in \mathrm{dom}(G)}{W \vdash^G \downarrow S : \texttt{type}} \downarrow_{typ}$	$\dfrac{S \in \mathrm{dom}(G)}{W \vdash^G \vec{S} : \texttt{type}} \vec{typ}$
Introduction (reflection)	$\dfrac{W,(S,\cdot) \vdash^G t : A \quad W >^G S}{W \vdash^G \lceil t \rceil_S : \uparrow_S A} \uparrow_I$	$\dfrac{W \vdash^G \sigma : S}{W \vdash^G \lceil \sigma \rceil_S : \downarrow S} \downarrow_I$	$\dfrac{W \vdash^G S, \Sigma\ \texttt{Sig}}{W \vdash^G \lceil \Sigma \rceil_S : \vec{S}} \vec{I}$
Evaluation	$\dfrac{W \vdash^G q : \uparrow_S A}{W,(S,\cdot) \vdash^G \lfloor q \rfloor_S : A} \uparrow_{ev}$	$\dfrac{W \vdash^G m : \downarrow S}{W \vdash^G \lfloor m \rfloor_S : S} \downarrow_{ev}$	$\dfrac{W \vdash^G u : \vec{S}}{W \vdash^G S, \lfloor u \rfloor_S\ \texttt{Sig}} \vec{ev}$
Elimination (induction)	$\dfrac{W \vdash^G q : \uparrow_S A \quad W \vdash \sigma : S}{W \vdash^G q^\sigma : \sigma(A)} \uparrow_E$	$\dfrac{W \vdash^G m : \downarrow S \quad \vdash_S t : A}{W \vdash^G t^m : \lfloor m \rfloor_S(A)} \downarrow_E$	$\dfrac{W \vdash^G u : \vec{S} \quad W,(T,\cdot) \vdash^G \sigma : S}{W \vdash^G u^\sigma : \vec{T}} \vec{E}$

Table 5.6: Inference System for Basic Type Theory with Reflection

Chapter 6

Implementation

We base the implementation of our generic type theory with reflection on the existing Scala implementation of the MMT framework. The core of the API is formed by data structures modelling the MMT language, whose simplified grammar is given below, for the sake of completeness. Also, we can remark on its similarity with that for the basic type theory described in Section 3.

Specifically, signatures can be represented as *MMT theories*, containing *symbol declarations*, such as constants, functions and predicates. Also, analogous to the case for signatures, MMT theories are related via *theory morphisms*. However, unlike the generic type theory previously presented, MMT is enriched with a module system, allowing one to build large theories and morphisms, through reuse and inheritance. Moreover, at the object-level, entities are represented as OpenMath objects, that can be truth-preservingly translated between theories via theory morphisms.

Theory Graph	$\mathcal{G}$	$::=$	Mod^*
Module Declaration	Mod	$::=$	$\%\texttt{sig}\, T =^{[T]} \{Sym^*\}$
Symbol Declaration	Sym	$::=$	$\%\texttt{include}\, T \mid c[:\omega][=\omega]$
Term	ω	$::=$	$T?c \mid x \mid \omega\, \omega^+ \mid \omega X.\omega \mid \texttt{String}$
Variable Context	X	$::=$	$\cdot \mid X,\, x[:\omega][=\omega]$
Theory Identifier	T	$::=$	`MMT URI`
Local Declaration Name	c	$::=$	`MMT Name`

Figure 6.1: Simplified MMT Grammar

Delayed Constraint Satisfaction Particularly relevant to our extensions is the existing framework for constraint satisfaction, which makes use of a delay mechanism to find solutions for all the unknown variables occurring in a given problem instance. We give a a rough overview of the structures involved in Figure 6.

The framework's input scenario comprises of a *system of constraints*, given as a set of judgments (Judgement 1, ..., Judgement n), which may contain *unknowns* and which have to be solved within a given MMT *theory*. This solution is a `Substitution` operating on the variable declaration list of a `Context`, in order to provide a closed `Term` for every unknown variable. A `Solver` class takes as arguments a individual judgment from the constraints, the MMT-theory and the set of typing rules stored in a `Controller`. These

rules are used by the `Solver` to decompose the judgment, to collect partial solutions and to delay unsolvable judgments, until further simplifications can lend it otherwise.

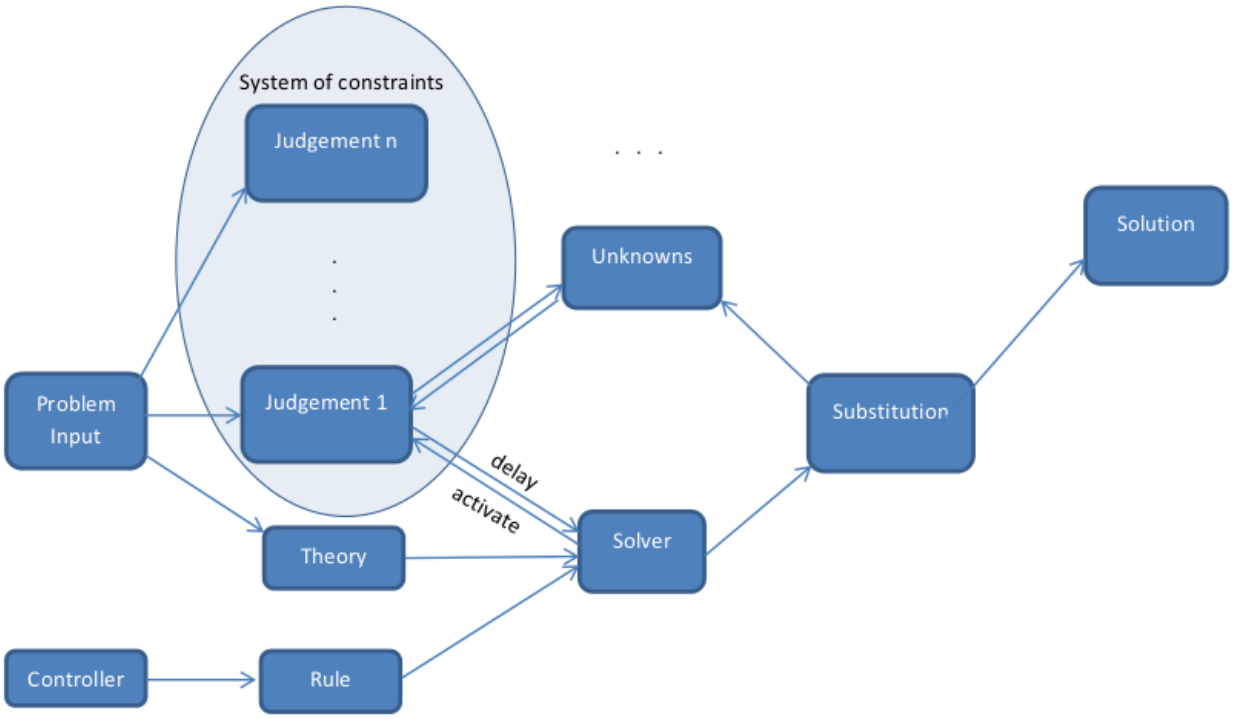


Figure 6.2: Delayed Constraint Satisfaction Workflow Outline

MMT with Reflected Constructs In line with the theoretical extensions to the MMT language, we extend the implementation with the notion of a `Frame`, formed by taking the pairing of a theory and a context. This extension serves the purpose of enabling the bookkeeping of quoted theories and of their corresponding contexts. Indeed, these are stored in a `Stack`, implemented as a list of frames and containing the newest quoted theory along with its context, as the top-level entry. The idea of introducing a stack of frames stems from programming languages, which pervasively use *execution stacks* to store information about the subroutines of a program, i.e the address to which each active subroutine should return control upon completing execution, local data storage or parameter values. In our setting, by using quotation, every theory and term of that theory can be used as a function. Thus, the *quotation* operator acts as a *function call* that is not committed to any particular function, but that can instead take any term from a quoted theory as one. Such a function is not parametrically constrained; instead, whenever we want a value we can draw back to it by *evaluation*, by looking into what arguments were passed.

The reflection of terms and that of morphisms are implemented orthogonally, thus facilitating future work on large-scale case studies. In the case of the first feature, that of *term reflection*, we construct a separate object containing the MMT document level

path (`base`), the module level path of the theory "TermReflection" (`theory`), as well as a method for defining constants of this new theory (`constant`). The primitive constants are given by that for the base type (`ttype`) and by the `intro`, `elim`, `rtype` and `eval` constants pertaining to each primitive term reflection construct, which we will proceed to explaining in detail bellow.

```
object Reflection {
   val base = new DPath(utils.URI("http", "cds.omdoc.org") / "foundations"
      / "reflection" / "termReflection.omdoc")
   val theory = base ? "TermReflection"
   def constant(name : String) : GlobalName = theory ? name
   val intro = constant("introduction")
   val elim = constant("elimination")
   val rtype = constant("refltype")
   val eval = constant("evaluation")
   val ttype = constant("type")
}
```

6.1 MMT with Term Reflection

6.1.1 Syntax

To start with, from an implementation perspective, in our new theory for term reflection, new expressions are formed by reflecting terms over signatures (`TermRefl`), evaluating reflected terms (`TermEval`), forming the reflected type of all reflected terms of a given term (`ReflType`) and applying a morphism to the quotation of a term (`TermElim`). These primitive constructs are implemented as separate objects containing apply and extractor methods for the Scala constructor and pattern matcher.

The `TermRefl` object, for term reflection introduction, takes as arguments two terms, namely the `term` to be quoted and the `theory` corresponding to the signature over which the term is quoted. It is formed by applying the `intro` constant (`OMS(Reflection.intro)`) to the list of arguments. Making this qualification will serve a later purpose, when pattern-matching on a term, to identify the used constructors and, thus, be able to determine which inference rule to apply to further simplify it. Conversely, the object is destructed into the option pair consisting of the theory and the reflected term.

```
object TermRefl {
   def apply(theory: Term, term: Term) = OMA(OMS(Reflection.intro), List(
      theory, term))
   def unapply(t : Term) : Option[(Term,Term)] = t match {
         case OMA(OMS(Reflection.intro), List(thy, tm)) => Some((thy, tm)
            )
         case _ => None
   }
}
```

The implementation for the `TermEval` term evaluation object is similar. In order to obtain the `ReflType` reflected type and the `TermElim` elimination form, their corresponding objects take as arguments the type of the reflected terms and the theory over which

they were quoted and, respectively, a reflected term and the morphism to be applied to it. The rest is analogous.

Example 6. *Encoding the Natural Numbers.*

As a test case for our implementation, we encode in MMT the natural number example presented above. The meta-level theory `Nat` is introduced as a new declared LF-theory and is added to the controller. The constructors `nat`, `zero` and `succ` are given as named, undefined, constants within the parent theory `OMID(Nat.path)`. Their type is formed using the corresponding LF type constructors `LF.ktype` and `Arrow`, as seen in the signature presentation of `Nat`.

```
//the meta-level theory of natural numbers
val Nat = new DeclaredTheory(testbasenat, LocalPath(List("Nat")),
   Some (LF.lftheory))
controller.add(Nat)

//generic type of natural numbers in the theory Nat
val nat = new Constant(OMID(Nat.path),
  LocalName("nat"),
    Some(LF.ktype), None, None, None)
controller.add(nat)

//zero constructor in the theory Nat
val zero = new Constant(OMID(Nat.path),
  LocalName("zero"),
    Some(OMID(nat.path)), None, None, None)
controller.add(zero)

//successor constructor in the theory Nat
val succ = new Constant(OMID(Nat.path),
  LocalName("succ"),
    Some(Arrow(OMID(nat.path),OMID(nat.path))), None, None, None)
controller.add(succ)
```

We declare `NatR` as the previously stated natural numbers theory with reflection and define its constructors as follows. The reflection type `nat_refl` is introduced as constant of kind type `LF.ktype`, whose definens is a MMT-term obtained by calling the `ReflType` constructor with the parent theory `OMMOD(NatR.path)` and the constant term `OMID(nat.path)` as arguments. Likewise, the 0 zero constructor `zero` is given by the reflected term `TermRefl(OMMOD(NatR.path), OMID(zero.path))` formed from the constant `OMID(zero.path)` being quoted over the parent theory `OMMOD(NatR.path)`. Lastly, the s successor constructor has function type `Arrow(OMID(nat_refl.path),OMID(nat_refl.path))` and is defined as the reflection of `OMID(succ.path)` over the parent theory `OMMOD(NatR.path)`.

```
//the reflected theory of natural numbers
val NatR = new DeclaredTheory(testbasenat2, LocalPath(List("NatR")),
   Some (LF.lftheory))
controller.add(NatR)

//reflected type of natural numbers in the theory NatR
val nat_refl = new Constant(OMID(NatR.path),
  LocalName("NN"),
```

```
        Some(LF.ktype), Some(ReflType(OMMOD(NatR.path), OMID(nat.path))),
            None, None)
    controller.add(nat_refl)

    //zero constructor reflected from the theory Nat down to the theory
        NatR
    val zero_refl = new Constant(OMID(NatR.path),
      LocalName("0"),
        Some(OMID(nat_refl.path)), Some(TermRefl(OMMOD(NatR.path), OMID(
            zero.path))), None, None)
    controller.add(zero_refl)

    //successor constructor reflected from the theory Nat down to the
        theory NatR
    val succ_refl = new Constant(OMID(NatR.path),
      LocalName("s"),
        Some(Arrow(OMID(nat_refl.path),OMID(nat_refl.path))), Some(
            TermRefl(OMMOD(NatR.path), OMID(succ.path))), None, None)
    controller.add(succ_refl)
```

6.1.2 Proof Theory

In the following, we present the implementation of the inference system for term reflection described in Section 4.2. Note that the theory graph G collecting all declared theories is never used in the implementation and is stated for completeness purposes. Also not present in the implementation is a theory well-ordering check, for the $<^G$ relation, occurring in the statement of the type formation and introduction rules (see Table 4.2).

Typing Rules. As seen in Section 6.1.1, type inference (well-formedness) is dealt with using a constraint-based approach, in that the bottom-up type inference rules collect type constraints that later need to be solved.

Let us give a more detailed account of the rules, starting from that for the *type formation* $\uparrow_{typ}$ of the reflected type corresponding to quoted terms with type A, under a named signature (theory) S (see Table 4.2). Its implementation is realized via the `TypeReflectionRule` object extending the `InferenceRule` class, designed to infer the type of an expression. The apply method takes as arguments a `solver`, which consists of a controller, a specific theory and a set of unknown meta-variables, as well as a term `tm` and an implicit `stack` of frames. This procedure is called on a typing judgement for the term `tm`, whose type is unknown. Once the term has been matched to `ReflType(s,a)`, the reflected type of quoted terms of type `a` over a signature `s`, there is a callback to the type inference method `inferType` with `a` as argument.

```
object TypeReflectionRule extends InferenceRule(Reflection.rtype) {
  def apply(solver: Solver)(tm: Term)(implicit stack: Stack): Option[Term]
      =
    tm match {
      case ReflType(s,a) => solver.inferType(a)
      case _ => None
    }
}
```

Next, we have the introduction rule $\uparrow_I$ for the type $\uparrow_S A$ of reflected terms over a named signature S. Analogous to the previous case, the implementation `TermReflectionRule` mirrors the theoretical formulation of the rule (see Section 4.2), with `inferType` being called on the term `t`, once the `apply` method pattern-matches the argument `tm` to `TermRefl(s,t)`, a quoted term `t` over a signature `s`.

```
object TermReflectionRule extends InferenceRule(Reflection.intro) {
 def apply(solver: Solver)(tm: Term)(implicit stack: Stack): Option[Term]
    =
   tm match {
      case TermRefl(s,t) => solver.inferType(t)
      case _ => None
    }
}
```

The evaluation rule $\uparrow_{ev}$ states that the evaluation of a quoted term q under a named signature S is well-formed with respect to the extended stack $W, (S, \cdot)$ and has type A, if q is well-formed under W and has the reflected type $\uparrow_S A$ over S. Note that, when type checking evaluations, the top-level frame recording the quoted theory and its context is popped from the stack. Thus, it is necessary to perform an additional check against the frame stack being empty, as seen below.

```
object ReflTermEvalRule extends InferenceRule(Reflection.eval){
 def apply(solver: Solver)(tm: Term)(implicit stack: Stack): Option[Term]
    =
   tm match {
      case TermEval(s,q) =>
        if(!stack.frames.isEmpty) {
        solver.inferType(q)
        }
        else None
      case _ => None
    }
}
```

The final rule $\uparrow_E$, namely that for morphism elimination is the most involved. In particular, the elimination of a quoted term q through a morphism σ is well-formed with respect to the stack W (and has type $\sigma(A)$), if two constraints are satisfied. Firstly, q has to be well-formed with respect to W and have the reflected type $\uparrow_S A$ of quoted terms of type A over S. Secondly, the morphism σ has to be well-formed with respect to W and translate terms from the theory S.

```
object ElimReflectionRule extends InferenceRule(Reflection.elim){
 implicit def pCont(p:Path){}
 def apply(solver: Solver)(tm: Term)(implicit stack : Stack) : Option[Term
    ] = {
   tm match {
      case Elim(q,mor) =>
        if(!stack.frames.isEmpty) {
          solver.inferType(q) match {
            case None => None
            case Some(ReflType(s,a)) => Some(solver.controller.checker.
                checkMorphism(mor,s,stack.theory))
            case _ => None
```

```
            }
          }
          else None
        case _ => None
      }
    }
  }
```

Equality Rules. We first remark on the fact that the introduction and elimination rules for reflected terms are straight-forwardly stating that quotation and morphism application preserve term equality. This is given to us for free within the framework of the system and needs no implementation.

The computation $=^t_I$, soundness $=^t_{sound}$ and completeness $=^t_{compl}$ rules are implemented as instances of the `ComputationRule` class, aimed at simplifying expressions operating at the toplevel of the term.

According to the computation rule, the term $\lceil t \rceil^\sigma_S$ obtained by applying a morphism σ to a reflected term $\lceil t \rceil_S$ is equal to that obtained by applying the meta-level morphism σ to the term t with respect to the frame W, given that, in turn, σ is well-formed with respect to W.

The apply procedure of the corresponding `ComputationReflectionRule` matches on the term argument `tm` for the elimination form `Elim(TermRefl(s,t),mor)` of a reflected term `t` through a morphism `mor`. If the stack frame is non-empty and if the morphism the term is eliminated with is well-formed, the method returns the application `OMA(mor,List(t))` of the morphism `mor` to the singleton list containing `t`.

```
object ComputationReflectionRule extends ComputationRule(Reflection.elim){
 implicit def pCont(p:Path){}
 implicit val stack = Stack(List())
 def apply(solver: Solver)(tm: Term)(implicit stack : Stack) : Option[Term]
     = {
   tm match {
     case Elim(TermRefl(s,t),mor) =>
       if(!stack.frames.isEmpty) {
        try {
          solver.checkMorphism(mor,s)
        }
        catch {
         case e:Error => false
        }
       Some(OMA(mor, List(t)))
       }
       else None
     case _ => None
 }
 }
}
```

Next, the soundness rule states that a well-formed term is preserved via quoting and then evaluating the result over the same theory S, i.e that quotation and evaluation are partially inverse operations. The implementation of the rule follows analogously with a equality check for the theories `s1`, `s2`, over which one is quoting and, respectively, evaluating.

```
object SoundnessReflectionRule extends ComputationRule(Reflection.eval){
  def apply(solver: Solver)(tm: Term)(implicit stack: Stack) : Option[Term]
      = {
    tm match {
      case TermEval(s1,TermRefl(s2,t)) =>
        if (s1 == s2)
          Some(t)
        else
          None
      case _ => None
    }
  }
}
```

Similarly to the soundness rule, the completeness rule asserts that, by first evaluating a reflected term q under a theory S and then reflecting it over the same theory, the result q' is equal to q under a frame stack W, if equality holds between q and q'. Together with the previous result, this introduces quotation and evaluation as inverse to each other.

As before, the apply method of the `CompletenessReflectionRule` matches the argument term `tm` against the evaluation-quotation term `TermRefl(s1,TermEval(s2,t))`, checking for the equality of theories `s1` and `s2`. Given that the two are equal, we have to infer the reflected type `ReflType(s,a)`, in order to extract the theory `s` that has to match the previous ones. Thus, we make sure that the inverse reflection operation applications maintain the term they are apply to in the same theory.

```
object CompletenessReflectionRule extends ComputationRule(Reflection.intro)
    {
  def apply(solver: Solver)(tm: Term)(implicit stack: Stack) : Option[Term]
      = {
    tm match {
      case TermRefl(s1,TermEval(s2,t)) =>
        if (s1 == s2) {
          solver.inferType(tm) match {
            case Some(ReflType(s,a)) => s == s2
            case _ => false
        }
        Some(t)
        }
        else None
      case _ => None
    }
  }
}
```

Finally, conforming to the extensionality reflection rule, equality between two well-formed quoted terms q and q', with respect to a frame W is preserved by evaluation under a theory S with respect to the extended frame $W,(S,\cdot)$. The implementation matches the rule exactly, save for the additional check for equality between the theories `s1` and `s2` the terms are quoted over. While the previous rules in this section extended the `ComputationRule` class, extensionality is realized via a `EqualityRule`, that checks expression equality, returning a boolean. Also, trivially note that, since we can only evaluate previously quoted terms, there is no need for an analogous complementary rule stating

the extensionality of quotation application; this holds, however, due to completeness.

```
object ExtensionalityReflectionRule extends EqualityRule(Reflection.intro){
  def apply(solver: Solver)(tm1: Term, tm2: Term, tp: Term)(implicit stack:
      Stack) : Boolean = {
    (tm1,tm2) match {
      case (TermRefl(s1,t1),TermRefl(s2,t2)) =>
        if (s1 == s2) {
          solver.checkEquality(TermEval(s1,t1),TermEval(s2,t2),None)
        }
        else false
      case (_,_) => false
    }
  }
}
```

In addition to the theoretical rules stated in Table 4.3, we have implemented solution rules `SolveEvalReflectionRule` and `SolveReflReflectionRule`, to determine the value of unknowns situated inside evaluations and, respectively, inside quotations. Indeed, a term `tm2` is equal to the evaluation `TermEval(s,x)` of an unknown term x under a theory `s`, if this term is equal to the reflection of `tm2` over s, as established through the `checkEquality` method. The rule `SolveReflReflectionRule` for solving the value of a reflected unknown over a given theory proceeds in the same manner, using the invertibility of quotation with respect to evaluation.

```
object SolveEvalReflectionRule extends SolutionRule(Reflection.eval){
  def apply(solver: Solver)(tm1: Term, tm2: Term)(implicit stack : Stack) :
      Boolean = {
    tm1 match {
      case TermEval(s,x) => solver.checkEquality(x,TermRefl(s,tm2),None)
      case _ => false
    }
  }
}
```

6.2 MMT with Morphism Reflection

Syntax. In practice, the objects used to represent the morphism reflection constructs are very similar to those used in the previous section. As a result, we will only discuss the case for morphism elimination, modelled by `MorphElim`.

To this purpose, let us first look at how we encode morphisms. To start with, the contents of a morphism are represented using the class `Record`, as lists of pairings containing *constants* (given relative to a module through a `LocalName`) and *terms* (`Term`) that are assigned to these constants. Specifically, a morphism's body is coded to be a `OMREC` object, while masking the low-level details of using lists in the implementation. This is done by relating `Record` to OpenMath's `OMATTR`, which takes as arguments a term without attribution, a key (constant) and a corresponding value (constant assignment).

```
case class Record(fields: List[(LocalName,Term)]){
}
```

```
object OMREC {
  val recordkey = utils.mmt.mmtbase ? "recordlabel"
  val recordSymbol = utils.mmt.mmtbase ? "records" ? "record"
  def apply(r : Record) : Term =  r match {
    case Record(fields) =>
      fields match {
        case (k,v)::l => OMATTR(apply(Record(l)), OMID(recordkey ? k), v)
        case Nil => OMID(recordSymbol)
      }
  }
  def unapply(m : Term) : Option[Record] = m match {
    case OMATTR(arg, key, value) => key.path match {
      case GlobalName(OMMOD(this.recordkey), name) =>
        unapply(arg) match {
          case Some(Record(fields)) => Some(Record((name, value)::fields))
          case None => None // should be illegal
        }
      case _ => None
    }
    case OMID(this.recordSymbol) => Some(Record(Nil))
    case _ => None
  }
}
```

Finally, morphisms are encoded through `ExplicitMorph`, comprising of methods for application and extraction. In particular, an explicit morphism is formed by specifying its contents (`rec`) and the theory (`dom`) it maps from. These arguments are passed to the `apply` method, which returns a term is constructed by applying the `OMID(mmt.explmorph)` constant formed through the MMT-reference to a list containing the target theory `dom` and the morphism's assignments. The destructor method extracts these components from the term denoting an explicit morphism.

```
object ExplicitMorph {
  def apply(rec : Record, dom : Term) : Term = OMA(OMID(mmt.explmorph),
     List(dom, OMREC(rec)))
  def unapply(m : Term) : Option[(Record, Term)] = m match {
    case OMA(OMID(mmt.explmorph), List(dom, OMREC(rec))) => Some((rec, dom)
       )
    case _ => None
  }
}
```

As previously mentioned, the t^m construct (see Table 5.4) for elimination is implemented via `MorphElim`, analogously to the constructs for term reflection. Specifically, given a term `t` and a reflected morphism `mor`, the apply method matches `mor` to be an explicit morphism and t to be a constant `OMID(path)`. If the constant is found among the record's fields, its corresponding value is returned. If `t` does not match a constant the procedure returns the application of the `OMS(ReflectionMorph.elim)` symbolic reference to a list comprising of `t` and `mor`. The unapply procedure is straight-forward.

```
 object MorphElim {
  def apply(t: Term, mor: Term) = mor match {
    case ExplicitMorph(rec,dom) =>  t match {
```

```
        case OMID(path) => rec.fields.find(localName => Some(localName._1) ==
            path.toTriple._3) match {
          case Some(pair) => pair._2
        }
      }
      case _ => OMA(OMS(ReflectionMorph.elim), List(t, mor))
    }
    def unapply(t : Term) : Option[(Term,Term)] = t match {
      case OMA(OMS(ReflectionMorph.elim), List(t, mor)) => Some((t, mor))
      case _ => None
    }
}
```

Proof Theory. The implementation of the inference rules matches their theoretical descriptions given in Section 5.1. As the corresponding objects used to model them are very similiar to each other, we limit ourselves to discussing only the reflection *typing rule* $\downarrow_E$ for morphism elimination. The apply procedure of the encoded inference rule takes a `solver`, and a term *tm* defined with respect to a frame `stack`. If the term matches the reflection `MorphElim` construct for elimination, we proceed to further checking its type. If its inferred type matches the reflected morphism type, then we continue to checking the well-formedness of the morphism `mor` via the *checkMorphism* method.

```
object MorphElimReflectionRule extends InferenceRule(ReflectionMorph.elim)
    {
  implicit def pCont(p:Path){}
  def apply(solver: Solver)(tm: Term)(implicit stack : Stack) : Option[Term
      ] = {
    tm match {
      case MorphElim(q,mor) =>
        if(!stack.frames.isEmpty) {
          solver.inferType(q) match {
            case None => None
            case Some(MorphReflType(s,a)) => Some(solver.controller.checker
                .checkMorphism(mor,s,stack.theory))
            case _ => None
          }
        }
        else None
      case _ => None
    }
  }
}
```

Let us examine the details of how the morphism check is implemented. The method takes as arguments a morphism term `mor`, its domain theory `from` and the frame `stack`, which stores, at the top-level, the morphism's codomain theory. Due to the module system, theories are not stored directly as lists of declarations. However, these theories are denoted via a specific MMT term, namely `OMMOD(p)`, where the path `p` is used to retrieve the actual theory object. Looking at the corresponding object for the morphism's target theory, we investigate the two existing possibilities: that it is a defined theory (`thdf`) and that it is a declared theory `thd` containing a list of symbols: constants or imports. In the former case, we take the definiens and recurs. In the latter, we extract the `clist` list of constants, by filtering out the symbols corresponding to imports, since they are not

relevant for our purposes. Matching the morphism `mor` to be an explicit morphism with body `rec` and source theory `dom`, we first check that `dom` indeed matches the argument `from` passed as an initial argument. We then order the list of constants in the morphism `mor`, thus obtaining `ocassig`. For every constant in the domain theory, we test whether the local names match those in the morphism and, if such is the case, we continue by type checking the constants in the usual way.

```
    def checkMorphism(mor: Term, from : Term)(implicit stack: Stack) :
       Boolean = {
      log("typing: " + stack.theory + " |- " + mor + " : " + from)
      report.indent
      val res : Boolean = from match {
        case OMMOD(p) => controller.globalLookup.getTheory(p) match {
          case thdf : DefinedTheory =>  checkMorphism(mor, thdf.df)
          case thd : DeclaredTheory =>
            val clist : List[Declaration] = thd.valueList filter (p => !p.
               isInstanceOf[Structure])
            mor match {
              case ExplicitMorph(rec, dom) =>
                if (from == dom)
                {
                  val ocassig = rec.fields.sortWith((x,y) => x._1.toString
                     () <=  y._1.toString())
                  if (clist.map(_.name) == ocassig.map(_._1)) {
                    (clist zip ocassig) forall (pair =>
                     {
                      inferType(OMID(pair._1.path)) match {
                        case Some(tp) => checkTyping(pair._2._2,OMM(tp,mor
                           ))
                        case None => true
                      }
                    })
                  }
                  else false
                }
                else false
              case _ => false
            }
          case _ => false
        }
        case _ => false
      }
      report.unindent
      res
    }
```

The *equality rules* for reflected morphisms are analogous to those for reflected terms presented in Section 6.1 and are summarized in Table 5.2. We refrain from explaining the implementation of all the rules and limit ourselves to just the $=^m_C$ computation rule. The corresponding `ComputationMorphReflectionRule` object takes a solver, a term `tm` defined with respect to a frame `stack` and tries to simplify *tm* according to the rule described above. Concretely, if `tm` matches the construct `MorphElim(t, MorphRefl(s,m))`, formed by eliminating a term `t` with the reflection `MorphRefl(s,m)` of a morphism `m` mapping from `s`, the term can be reduced to the application `OMA(MorphRefl(s,m),List(t)`

of `MorphRefl(s,m)` to a list containing `t`. In line with the statement for $=^m_C$, this simplification can be made only if the reflected morphism `MorphRefl(s,m)` is well-formed with respect to the theory `s`.

```
object ComputationMorphReflectionRule extends ComputationRule(
    ReflectionMorph.elim){
  implicit def pCont(p:Path){}
  implicit val stack = Stack(List())
  def apply(solver: Solver)(tm: Term)(implicit stack : Stack) : Option[Term
      ] = {
    tm match {
      case MorphElim(t, MorphRefl(s,m)) =>
        if(!stack.frames.isEmpty) {
          try {
            solver.checkMorphism(MorphRefl(s,m),s)
          }
          catch {
            case e:Error => false
          }
          Some(OMA(MorphRefl(s,m),List(t)))
        }
        else None
      case _ => None
    }
  }
}
```

Chapter 7

Conclusion

As we have emphasized in the introductory sections, the concept of reflection is a fundamental one and, although it is largely present in computer science, it is not as well-understood foundationally. The benefits of conceptually grasping reflection in a formal setting are two-fold. On the one hand, the impact on the development and design of type theories in the large is substantial, if we have in mind advanced meta-reasoning features that can be supported through having reflection. On the other hand, being able to deal with meta-theoretical foundations and to conflate meta-levels leads to increased *expressivity*, *elegance* and *flexibility* in terms of implementation and system integration.

To the best of the author's knowledge, reflection has been mostly studied from the perspective of enabling its addition on top of existing type theories. The approach we undertake differs in that, starting from reflection as a central property, we seek to recover *type theoretical features*. In doing so, we focus on *reflecting terms*, which leads to obtaining inductive datatypes, like the one for natural numbers, and on *reflecting morphisms*, which leads to obtaining recursive datatypes.

In this project we have developed a MMT-extension for signature-based reflection, supporting as features the reflection of terms and that of types. We have implemented their corresponding syntax and proof theory and tested it on a case-study aimed at encoding the theory of natural numbers with reflection. The example presented validates the claim that mathematical operations can be expressed naturally using reflection and paves the way for future large-scale investigation.

A primary future goal is to realize this potential through developing the theoretical framework to a sufficient breadth of coverage, as to enable it to handle complex hierarchical mathematical and logical theories and morphisms. Particularly, such a reflective system should be able to support dependent inductive types, parameterized model functors, quotient types, self-reflecting signatures and to satisfy abstract adequacy. More elaborate case studies could be developed to identify and test more desirable properties that can facilitate the formalization of a wider scope of mathematical and logical constructs, while preserving compactness and naturality.

An exhaustive theory of reflection would support six such features (reflected meta-judgements): three for *signature-based reflection* (the well-formedness of reflected signatures and, respectively, that of terms and morphisms with respect to a signature) and three for *context-based reflection* (the well-formedness of reflected contexts, that of terms

with respect to a context and that of substitutions). Along similar lines as previously, when reflecting in context, once can obtain function and, respectively, product types. Future work could be aimed at providing a theoretical and implementational account of context-based reflection.

Moreover, an interesting direction for future research would be that of exploiting the benefits of working within a module system, an aspect which has been disregarded in our work, for simplification purposes. As includes between reflected signatures lead to subtyped relations, we would thus obtain subtypes. Also, such a reflective module system would give a way to, for example, extend inductive datatypes and to extend inductive definitions with cases.

Bibliography